THÈSE

Pour obtenir le grade de

DOCTEUR DE L’UNIVERSITÉ DE GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Hosein NAZARPOUR

Thèse dirigée par Saddek BENSÁLEM
et codirigée par Yliès FALCONE

préparée au sein du laboratoire VERIMAG
dans l’École Doctorale Mathématiques, Sciences et Technologies de l’Information, Informatique

Surveillance de Systèmes à Composants Multi-Threads et Distribués

Monitoring Multi-Threaded and Distributed (Component-Based) Systems

Thèse soutenue publiquement le 26 Juin 2017,
devant le jury composé de :

Monsieur Martin LEUCKER
PROFESSEUR, UNIVERSITÉ DE LÜBECK, Rapporteur

Monsieur Dominique MÉRY
PROFESSEUR, UNIVERSITÉ DE LORRAINE, Rapporteur

Madame Olga KOUCHNARENKO
PROFESSEUR, UNIVERSITÉ DE FRANCHE-COMTÉ, Examinateur

Monsieur Leonardo MARIANI
PROFESSEUR, UNIVERSITÉ DE MILAN-BICOCCA, Examinateur

Monsieur Gwen SALAÜN
PROFESSEUR, GRENOBLE INP, Examinateur

Monsieur Saddek BENSÁLEM
PROFESSEUR, UNIVERSITÉ GRENOBLE ALPES, Directeur de thèse

Monsieur Yliès FALCONE
DOCTEUR, UNIVERSITÉ GRENOBLE ALPES, Co-Directeur de thèse
This thesis is dedicated to my parents for their love, endless support and encouragement.
Abstract

Context. Component-based design is the process leading from given requirements and a set of predefined components to a system meeting the requirements [86, 6, 16]. Components are abstract building blocks encapsulating behavior. They can be composed in order to build composite components. Their composition should be rigorously defined so that it is possible to infer the behavior of composite components from the behavior of their constituents as well as global properties from the properties of individual components. It is, however, generally not possible to ensure or verify the desired property using static verification techniques such as model-checking or static analysis, either because of the state-space explosion problem or because the property can only be decided with information available at runtime (e.g., from the user or the environment). Runtime Verification (RV) [54, 61, 36, 87, 3] is an umbrella term denoting the languages, techniques, and tools for the dynamic verification of system executions against formally-specified behavioral properties. In this context, a run of the system under scrutiny is analyzed using a decision procedure: a monitor. Generally, the monitor may be generated from a user-provided specification (e.g., a temporal-logic formula, an automaton), performs a step-by-step analysis of an execution captured as a sequence of system states, and produces a sequence of verdicts (truth-values taken from a truth-domain) indicating specification satisfaction or violation.

Contributions. This thesis addresses the problem of runtime monitoring multi-threaded and distributed component-based systems with multi-party interactions (CBSs). Although, neither the exact model nor the behavior of the system are known (black box system), the semantic of such CBSs can be modeled with labeled transition systems (LTSs). Inspiring from conformance testing theory, we refer to this as the monitoring hypothesis. Our monitoring hypothesis makes our approach oblivious of (i) the behavior of the CBSs, and (ii) how this behavior is obtained. We consider a general abstract semantic model of CBSs consisting of a set of intrinsically independent components whose interactions are managed by several schedulers. Using such an abstract model, one can obtain systems with different degrees of parallelism, such as sequential, multi-threaded and distributed systems. When monitoring concurrent (multi-threaded
and distributed) CBSs, the problem that arises is that a global state of the system is not available at runtime, since the schedulers execute interactions even by knowing the partial state of the system. Moreover, in distributed systems the total ordering of the execution of the interaction is not observable. A naïve solution to these problems would be to plug in a monitor which would however force the system to synchronize in order to obtain the sequence of global states as well as the total ordering of the executions at runtime. Such a solution would defeat the whole purpose of having concurrent executions and distributed systems.

We define two approaches for the monitoring of multi-threaded and distributed CBSs. In both approaches, we instrument the system to retrieve the local events of the schedulers. Local events are sent to an online monitor which reconstructs on-the-fly the set of global traces that are i) compatible with the local traces of the schedulers, and ii) suitable for monitoring purposes, in a concurrency-preserving fashion.

**Monitoring multi-threaded CBSs.** In the multi-threaded setting the interactions are executed concurrently with a centralized scheduler. In this setting, we address the problem of online monitoring of any logic-independent linear-time user-provided properties in multi-threaded CBSs described in the BIP (Behavior, Interaction, Priority) framework. BIP is an expressive framework for the formal construction of heterogeneous systems. Our technique reconstructs on-the-fly the global states by accumulating the partial states traversed by the system at runtime. We define transformations of components that preserve their semantics and concurrency and, at the same time, allow to monitor global-state properties. We present RVMT-BIP, a prototype tool implementing the transformations for monitoring multi-threaded BIP systems. Our experiments on several multi-threaded BIP systems show that RVMT-BIP induces a cheap runtime overhead.

This part of the thesis has been published in [72, 71].

**Monitoring distributed CBSs.** In the distributed setting the interactions are executed concurrently with several schedulers. Moreover, simultaneous execution of interactions by several schedulers is possible. In this setting, we address the problem of runtime monitoring of distributed CBSs against user-provided properties expressed in linear-temporal logic and referring to global states. In this context, the reconstruction of the global traces is done on-the-fly using a lattice of partial states encoding the global traces compatible with the locally-observed traces. We implemented our monitoring approach in a prototype tool called RVDIST. RVDIST executes in parallel with the distributed system and takes as input the events generated from each scheduler and outputs the evaluated computation lattice. Our experiments show that, thanks to the optimization applied in the online monitoring algorithm, (i) the size of the constructed computation lattice is insensitive to the number of received events, (ii) the lattice size is kept reasonable and (iii) the overhead
of the monitoring process is cheap.

This part of the thesis is under consideration for publication.
Résumé

Contexte. La conception à base de composants est le processus qui permet à partir d’exigences et un ensemble de composants prédéfinis d’aboutir à un système respectant les exigences. Les composants sont des blocs de construction encapsulant du comportement. Ils peuvent être composés afin de former des composants composites. Leur composition doit être rigoureusement définie de manière à pouvoir i) inférer le comportement des composants composites à partir de leurs constituants, ii) déduire des propriétés globales à partir des propriétés des composants individuels. Cependant, il est généralement impossible d’assurer ou de vérifier les propriétés souhaitées en utilisant des techniques de vérification statiques telles que la vérification de modèles ou l’analyse statique. Ceci est du au problème de l’explosion d’espace d’états et au fait que la propriété est souvent décidable uniquement avec de l’information disponible durant l’exécution (par exemple, provenant de l’utilisateur ou de l’environnement).

La vérification à l’exécution (Runtime Verification) désigne les langages, les techniques, et les outils pour la vérification dynamique des exécutions des systèmes par rapport à des propriétés spécifiant formellement leur comportement. En vérification à l’exécution, une exécution du système vérifiée est analysée en utilisant une procédure de décision : un moniteur. Un moniteur peut être généré à partir d’une spécification écrite par l’utilisateur (par exemple une formule de logique temporelle, un automate) et a pour but de détecter les satisfactions ou les violations par rapport à la spécification. Généralement, le moniteur est une procédure de décision réalisant une analyse pas à pas de l’exécution capturée comme une séquence d’états du système, et produisant une séquence de verdicts (valeur de vérité prise dans un domaine de vérité) indiquant la satisfaction ou la violation de la spécification.

Contribution. Cette thèse s’intéresse au problème de la vérification de systèmes à composants multi-thread et distribués. Nous considérons un modèle général de la sémantique et système à composants avec interactions multi-parties: les composants intrinsèquement indépendants et leur interactions sont partitionnées sur plusieurs ordonnanceurs. Dans ce contexte, il est possible d’obtenir des modèles avec différents degrés de parallelisme, des systèmes séquentiels, multi-thread, et distribués. Cependant, ni le modèle ex-

**Monitoring des systèmes à composants multi-thread.** Tout d’abord, nous nous intéressons aux problèmes du monitoring en ligne de propriétés temporelles linéaire et indépendantes de leur logique de spécification. Nous considérons des systèmes à composants multi-threads décrits dans le framework BIP (Behavior Interaction Priority), dans lequel les interactions peuvent être exécutées de manière concurrentes avec un ordonnanceur central pour les interactions multi-partie. BIP est un cadre expressif pour la construction formelle de systèmes hétérogènes. Notre technique reconstruit à la volée les états globaux par accumulation des états partiels traversés par le système lors de son exécution. Nous définissons des transformations de composants qui préservent leur sémantique et leur concurrence, et qui à la fois permet de vérifier des propriétés sur les états globaux. Nous présentons RVMT-BIP, un outil prototype implémentant les transformations pour le monitoring de système BIP. Nos expérimentation sur plusieurs systèmes BIP démontrent que RVMT-BIP n’induit qu’une faible dégradation des performances. Cette partie de la thèse a été publié dans [72, 71]

**Monitoring des systèmes à composants distribués.** Deuxièmement, nous nous intéressons au problème du monitoring des systèmes à composants avec interaction multi-partie pour des propriétés exprimées en logique linéaire temporelle qui font référence aux états globaux. Dans ce contexte, la reconstruction des traces globales est faite à la volée en utilisant un treillis des états partiels encodant les traces globales qui sont compatibles avec les traces observés localement. Nous avons implémenté notre approche de monitor-
ing dans un outil prototype appelé RVDIST. RVDIST s’exécute en parallèle avec le modèle distribué et prend en entrée les événements générés par chacun des ordonnanceurs et produit le treillis des exécutions. Nos expérimentations démontrent que, grâce à l’optimisation appliquée dans l’algorithme de monitoring en ligne, (i) la taille du treillis construit ne dépend pas du nombre d’événements reçus, (ii) la taille du tri est raisonnable, (iii) l’impact en terme de performance du processus de monitoring est faible. Cette partie de la thèse est en cours de soumission et d’évaluation.
# Contents

1 Introduction .................................................. 1  
1.1 Runtime Verification .................................. 1  
1.2 Component-Based Systems with Multi-Party Interactions (CBSs) .................................. 2  
1.3 Challenges of Monitoring Multi-threaded and Distributed CBSs .................................. 4  
1.4 Approach Overview ................................ 6  
1.5 Thesis Organization ................................ 10  

2 Monitoring User-Provided Specifications on Concurrent Systems .................................. 13  
2.1 Monitoring .................................................. 13  
2.1.1 Online Versus Offline .................................. 14  
2.1.2 Synchronous Versus Asynchronous Distributed Systems .................................. 14  
2.1.3 Centralized Monitor Versus Decentralized Monitor .................................. 14  
2.2 Monitoring Distributed Systems .................................. 16  
2.2.1 Summary .................................................. 22  

3 Preliminaries and Notations .................................. 25  

4 Monitoring Component-Based Systems with Multi-Party Interactions (CBSs) .................................. 31  

4 Abstract Semantic Model of Distributed, Multi-threaded and Sequential CBSs .................. 33  
4.1 Semantics of Distributed CBSs .................................. 34  
4.2 Semantics of Multi-Threaded CBSs .................................. 40  
4.3 Semantics of Sequential CBSs .................................. 41  

5 Observing a CBSs at Runtime .................................. 43  
5.1 Observable trace .................................................. 43
CHAPTER

Introduction

1.1 Runtime Verification

Runtime Verification (RV) [77, 54, 61, 36, 87, 9, 34, 5] is a lightweight and effective technique to ensure the correctness of a system at runtime, that is whether or not the system respects or meets a desirable behavior. It can be used in numerous application domains, and more particularly when integrating together unreliable software components. Runtime verification complements exhaustive verification methods such as model checking [22, 76], and theorem proving [55], as well as incomplete solutions such as testing [12] and debugging [85]. In RV, a run of the system under inspection is analyzed incrementally using a decision procedure: a monitor. This monitor may be generated from a user-provided high level specification (e.g., a temporal formula, an automaton). This monitor aims to detect violation or satisfaction w.r.t. the given specification. Generally, it is a state machine processing an execution sequence (step by step) of the monitored program, and producing a sequence of verdicts (truth-values taken from a truth-domain) indicating specification fulfillment or violation. For a monitor to be able to observe the runs of the system, the system should be instrumented in such a way that at runtime, the program sends relevant events that are consumed by the monitor. Usually, one of the main challenges when designing an RV framework is its performance. That is, adding a monitor in the system should not deteriorate executions of the initial system, time and memory wise.
1.2 Component-Based Systems with Multi-Party Interactions (CBSs)

Component-based design consists in constructing complex systems from given requirements using a set of predefined components [86]. Components are abstract building blocks encapsulating behavior. Each component is defined as an atomic entity with some actions and interfaces. Components communicate and interact with each other through their interfaces. They can be composed in order to build composite components. Their composition should be rigorously defined so that it is possible to infer the behavior of composite components from the behavior of their constituents as well as global properties from the properties of individual components and the interactions between them. Each multi-party interaction is a set of simultaneously-executed actions of the existing components [16].

The execution of a CBS with multi-party interactions is carried on using schedulers (also known as processes or engines) managing the interactions. The architecture of schedulers and components provides various types of CBSs in terms of functionality and different degrees of parallelism in the execution of the interactions. The sequential setting is the simplest execution procedure, in the sense that the interactions are carried out by the sequential execution of the participating components on a single thread of execution. For performance reasons, in the multi-threaded setting, the execution of components is parallelized, so that the components can run in multiple threads concurrently. Moreover, the highest level of parallelism is obtained in the distributed setting, which is mainly considered for efficiency reasons. To achieve more performance, doubling the frequency of processors consumes twice as much energy as doubling the number of processors in the system which is efficiently distributed. Another reason for considering distributed systems is the geographical location of components of the system. Processing local data and managing the local executions requires dedicated computing units in specific locations.

According to its degree of parallelism, a CBS can be categorized as follows:

- Sequential (centralized or single-threaded): In a Sequential CBS, one scheduler is in charge of the execution of the interactions of the system. Components notify the scheduler of their current states. Then, the scheduler computes the possible interactions, selects one, and then sequentially executes the actions of each component involved in the interaction. When components finish their executions, they notify the scheduler of their new states, and the aforementioned steps are repeated. In this setting, the execution of an interaction is not possible when some component is performing a computation. Moreover, the global state of the system, which is the union of the states of the individual components, is always defined. In this case, a global trace is represented by the sequence of global states obtained after the execution of interactions. Indeed, the scheduler is aware of the global trace of the system.
Multi-threaded (decentralized with a centralized scheduler): In the multi-threaded setting, similar to the centralized setting, one scheduler is in charge of the execution of the interactions, with the difference that each component executes on a separate thread and the scheduler is in charge of coordination. The components can perform their corresponding actions independently after synchronization, and the interactions become non-atomic. That is, after starting an interaction, and before this interaction completes (meaning that certain components are still performing internal computations), the scheduler can start another interaction among ready components. In this setting, the global state of the system is not always defined since the scheduler executes interactions even by knowing the partial state of the system, that is a snapshot of the system where at least one component is busy with its internal computation. The scheduler is aware of the partial trace which is a sequence of partial states.

Distributed (fully decentralized): In the distributed setting, the execution of interactions of a CBS is distributed among several independent schedulers. In an implemented distributed CBS, schedulers and components are interconnected (e.g., networked physical locations) and work together as a whole unit to meet some requirements. The execution of a multi-party interaction is then achieved by sending/receiving messages between the scheduler in charge of the execution of the interaction and the components involved in the interaction [5]. In this setting, each scheduler along with its associated components can be seen as a multi-threaded system, so that the computations of the components in the scope of the scheduler are done concurrently. Moreover, the simultaneous execution of several interactions managed by several schedulers is possible. Thus, the execution trace of a distributed system is a partial trace. Each scheduler is aware of its execution trace, that is the locally observed partial-trace consisting of a sequence of the partial states of components in the scope of the scheduler. A set of the local partial-traces of the schedulers represents the execution trace of the system.

Example 1.1. Figure 1.1 depicts an abstract component-based system of four components $A, B, C, D$ and a set of multi-party interactions $\{ab, bc, cd\}$, that are synchronization of the actions of the components. From a global state, an interaction can execute if all the participants are ready to execute that interaction. For instance, executing interaction $ab$ requires that components $A$ and $B$ are ready to execute their corresponding actions allowing interaction $ab$. In this case, interaction $ab$ is enabled. Let $(A^0, B^0, C^0, D^0)$ be the initial state of the system, where all the components are ready and all the interactions are enabled.

![Figure 1.1: Depiction of an abstract CBS with four components and three interactions](image-url)
– In the sequential setting, interactions are executed atomically and sequentially by the central scheduler. The state of all participants in the interaction is changed in a single execution step. The states of other components are not modified. From the initial state, the execution of interaction \( ab \) results in the following global trace: \((A^0, B^0, C^0, D^0) \cdot ab \cdot (A^1, B^1, C^0, D^0)\), where \( A^1 \) and \( B^1 \) are the next global states of components \( A \) and \( B \) respectively.

– In the multi-threaded setting, for instance after triggering interaction \( ab \), components \( A \) and \( B \) are considered as busy components until they finish their corresponding computations. The associated partial trace is: \((A^0, B^0, C^0, D^0) \cdot ab \cdot (A^\perp, B^\perp, C^0, D^0)\) where \( X^\perp \) denotes the busy state of component \( X \). From the partial state \((A^\perp, B^\perp, C^0, D^0)\), the scheduler can either execute interaction \( cd \), or wait for the busy components \( A \) or \( B \) to finish their internal computations.

– If the scheduler executes interaction \( cd \), the corresponding partial trace is: \((A^0, B^0, C^0, D^0) \cdot ab \cdot (A^\perp, B^\perp, C^0, D^0) \cdot cd \cdot (A^\perp, B^\perp, C^\perp, D^\perp)\).

– If component \( B \) finishes its computation and notifies the scheduler about its ready state, the system moves to state \((A^\perp, B^1, C^0, D^0)\), which is a partial state. From this state, the scheduler can execute \( bc \) reaching \((A^\perp, B^1, C^\perp, D^0)\) or \( cd \) reaching \((A^\perp, B^1, C^\perp, D^\perp)\), or wait for the ready state of component \( A \) to reach the global state \((A^1, B^1, C^0, D^0)\).

– In the distributed setting, assuming the following partitioning of interactions \{\( ab, bc \}\} and \{\( cd \}\), such that the execution of each partition is managed by a corresponding scheduler, if two interactions \( ab \) and \( cd \) are enabled, they can be executed concurrently by the two schedulers. Then the local observable partial traces are as follows:

  – Local partial trace of the first scheduler: \((A^0, B^0, C^0, D^0) \cdot ab \cdot (A^\perp, B^\perp, ?, ?)\),

  – Local partial trace of the second scheduler: \((A^0, B^0, C^0, D^0) \cdot cd \cdot (?, ?, C^\perp, D^\perp)\),

where ? denotes the unknown state.

### 1.3 Challenges of Monitoring Multi-threaded and Distributed CBSs

However, it is generally not possible to ensure or verify a desired behavior of such systems using static verification techniques such as model-checking or static analysis, either because of the state-space explosion problem or because the property can only be decided with information available at runtime (e.g., from the user or the environment). In this thesis, we are interested in complementary verification techniques for CBSs such as runtime verification. To this end, we propose techniques to runtime verify a component-based
system against properties referring to the global state of the system. This implies in particular that properties
can not be projected and checked on individual components. In the following we point out the problems that
one encounters when monitoring CBSs at runtime.

Runtime verification of sequential CBSs requires an instrumentation technique to derive the sequence of
(existing) global states at runtime to the monitor. Such an instrumentation must ensure that the performance
of the instrumented system is close to the performance of the original system. Moreover, the instrumentation
should not alter the behavior of the initial system. One possible solution is to add the monitor as a new
component, which is allowed to observe the system by adding interactions. Such interactions should be
inserted carefully because they could modify the existing interactions and/or modify the behavior of the
components. Verification of sequential CBSs has been studied in [27, 38, 39], and is not the focus of this
thesis.

In the multi-threaded and distributed settings we deal with concurrent executions and partial states.

– Although, in the multi-threaded setting, components execute with a centralized scheduler and there
is a global clock and communication considered to be instantaneous and atomic, the global state of
the system (where all components are ready to perform an interaction) may never exist at runtime.
This causes the main problem when monitoring multi-threaded CBSs against properties referring to
the global state of the system.

Example 1.2. We consider the CBS presented in Example 1.1 (p. 3) to illustrate the monitoring chal-
 lenges for a multi-threaded CBS. Although in the partial trace \((A^0, B^0, C^0, D^0) \cdot ab \cdot (A^\perp, B^\perp, C^0, D^0) \cdot
cd \cdot (A^\perp, B^\perp, C^\perp, D^\perp)\) the total order of the executions is defined and observable by the central sched-
uler, the global trace of the system does not exist.

– Furthermore, the runtime monitoring of an asynchronous distributed system is a much more difficult
task, because in the distributed setting, (i) the execution of the system is more dynamic and parallel,
in the sense that each scheduler executes its associated actions concurrently and we have a set of par-
allel executions, (ii) neither a global clock nor a shared memory is used, hence, schedulers can have
different processing speeds and can suffer from clock drifts, and (iii) since the execution of interac-
tions is based on sending/receiving messages and delays in the reception of messages in asynchronous
communications are inevitable, the runtime monitor does not receive the events with the same order
as they are actually occurred. Therefore, events cannot be ordered based on time. The absence of
ordering between the execution of the interactions in different schedulers causes the main problem in
the distributed setting that is (i) the global state of the system does not exist, and (ii) the actual partial
trace of the system is not observable.
1. INTRODUCTION

Example 1.3. We consider again Example 1.1 (p. 3) to illustrate the monitoring challenges on a distributed CBS. The two local partial-traces \( (A^0, B^0, C^0, D^0) \cdot ab \cdot (A^\perp, B^\perp, ?, ?) \) and \( (A^0, B^0, C^0, D^0) \cdot cd \cdot (?, ?, C^\perp, D^\perp) \) are observable locally by their corresponding schedulers. The actual ordering between \( ab \) and \( cd \) is not observable by any of the schedulers or components. Moreover, sending these local partial-traces to a central monitor through the associated events does not guarantee the reception of the events in the same order as they have actually occurred. Therefore, the monitor has to take into account the following possible partial traces:

- \( ab \) happened before \( cd \): \( (A^0, B^0, C^0, D^0) \cdot ab \cdot (A^\perp, B^\perp, C^0, D^0) \cdot cd \cdot (A^\perp, B^\perp, C^\perp, D^\perp) \)

- \( cd \) happened before \( ab \): \( (A^0, B^0, C^0, D^0) \cdot cd \cdot (A^0, B^0, C^\perp, D^\perp) \cdot ab \cdot (A^\perp, B^\perp, C^\perp, D^\perp) \)

- \( ab \) and \( cd \) happened concurrently: \( (A^0, B^0, C^0, D^0) \cdot ab, cd \cdot (A^\perp, B^\perp, C^\perp, D^\perp) \)

Consequently, the runtime monitor must i) find the total order of the events, ii) similarly to the multi-threaded setting, reconstruct the global trace associated to each partial trace, and iii) evaluate the possible global traces on-the-fly.

For monitoring such systems, we avoid synchronization to take global snapshots, which would go against the parallelism of the verified system. The monitoring problem is even more complicated because no component of the system can be aware of the global trace and the monitor needs to reconstruct the global trace from the events emitted by schedulers at runtime, and then reason about their correctness. Our goal is to provide methods that can be used for the verification of such CBSs by applying instrumentation techniques to observe the global behavior of the systems while preserving their performance and initial behavior. Consequently, the designed instrumentation technique should be defined formally and its correctness formally proved.

1.4 Approach Overview

We define a monitoring hypothesis based on the definition of an abstract semantic model of CBS. The abstract semantic system is composed of a non-empty set of components \( B \) and their joint actions which are managed by a non-empty set of scheduler \( S \). Each component \( B \in B \) is endowed with a set of actions \( Act_B \). Joint actions of component, aka multi-party interactions, involve the execution of actions on several components. An interaction is a non-empty subset of \( \{ Act_B \mid B \in B \} \), such that at most one action of each component is involved in an interaction. In addition, to model concurrent behavior, each atomic component \( B \in B \) has internal actions which we model as a unique action \( \beta \), such that each action of \( B \) is followed by
the internal action $\beta$. The set of interactions in the system is distributed among a non-empty set of schedulers $S$. Schedulers coordinate the execution of interactions and ensure that each multi-party interaction is jointly executed. Our monitoring hypothesis is that the behavior of the monitored system complies to this model. We argue that this model is abstract enough to encompass a variety of (component-based) systems, and serves the purpose of describing the knowledge needed on the verified system and later guides their instrumentation. A property $\varphi$ specifies the desired runtime behavior of the system (referring to the global state of the system) which has to be evaluated while the system is running. In this context, each scheduler is only aware of its local partial-trace, that is a set of ordered local events (i.e., actions which change the state of the system). Moreover, events from different schedulers are not totally ordered. In order to evaluate the global behavior of the system consisting of several schedulers, it is necessary to find i) a set of possible ordering among the events of all schedulers, that is, the set of compatible partial traces that could possibly happen in the system, and ii) the set of global traces corresponding to the compatible partial traces.

Figure 1.2 (p. 9) presents an overview of our approach. Intuitively, our method consists in two steps, (i) instrumentation of the abstract CBS to obtain system events and send them to the monitor, and (ii) reconstruction of the compatible global trace(s) of the system and evaluate them on-the-fly by the monitor. The instrumentation is done as follows:

- Each scheduler $S \in S$ is composed with a controller. Each controller is in charge of detecting and sending the events that occurred in the corresponding scheduler.

- A central monitor component (global observer) is added to the system. This new module receives the events which occurred in schedulers and are sent by their associated controllers. The monitor works in parallel with the system and applies an online monitoring algorithm upon the reception of each event.

- In the multi-threaded setting the controller of the scheduler sends the events to the monitor with the same order as they are occurred (we assume that communication channels are reliable). Although the partial trace of the system exists and can be obtained through the events, the sequence of global states of the system is not available.

- In the distributed setting where (i) we have more than one schedulers, (ii) we have possibly some shared components (i.e., components in the scope of more than one scheduler), and (iii) schedulers do not communicate together and only communicate with their own associated components, we compose each shared component with a controller. The controller of a shared component only communicates with the controllers of the schedulers whenever the shared components and the schedulers communicate. Indeed, in our abstract model, what makes the events of different schedulers to be causally
related is only the shared components which are involved in several multi-party interactions managed by different schedulers. In other words, the executions of two actions managed by two schedulers and involving a shared component are definitely causally related, because each execution requires the termination of the other execution in order to release the shared component. To take into account these existing causalities among the events, in the distributed setting, we employ vector clocks to define the ordering of events. The controller of a shared component is used to resolve the ordering among the events involving the shared component. Each event associated to the execution of a multi-party interaction is labeled by a vector clock. Ordering of such events are defined based on their vector clocks. The monitor receives the partially-ordered events representing the local partial-traces.

We propose two online monitoring methods for multi-threaded and distributed systems as follows:

- In the multi-threaded setting, the monitor is aware of the partial trace of a system, because the received events are totally ordered and represent the actual execution of the system. We define the notion of witness trace, which is intuitively the unique compatible global trace corresponding to the partial trace of the multi-threaded CBS as if this CBS was executed on a single thread (sequential setting) with the same sequence of the executed interactions. We introduce an online algorithm to reconstruct the corresponding witness trace of the current execution which allows the monitor to verify any logic-independent linear-time user-provided properties. Our method does not introduce any delay in the detection of verdicts since it always reconstructs the maximal (information-wise) prefix of the witness trace. Moreover, we show that our method is correct in the sense that it always produces the correct witness trace (Theorem 7.14, p. 68). Note that our approach allows one to monitor any linear-time property. Moreover, how the property is defined is irrelevant as one can use the approaches in [9, 35] to synthesize a monitor which emits verdicts in a 4-valued domain. Our approach directly uses the definition of a monitor as input and is thus compatible with the various approaches compatible with the ones in [9, 35].

We introduce RVMT-BIP, a tool integrated in the BIP tool suite for runtime verification of multi-threaded BIP systems.\textsuperscript{1} BIP (Behavior, Interaction, Priority) framework is a powerful and expressive component framework for the formal construction of heterogeneous systems. BIP offers two powerful mechanisms for composing components by using multiparty interactions and priorities. The combination of interactions and priorities is expressive enough to express usual composition operators of other languages as shown in [15]. A BIP model is layered. The lowest layer contains atomic components whose behavior is described by state machines with data and functions described in the

\textsuperscript{1}RVMT-BIP is available for download at [70].
C language. As in process algebras, atomic components can communicate by using ports. The second layer contains interactions which are relations between communication ports of individual components. Priorities are used to express scheduling policies by selecting among the enabled interactions of the layer underneath. RVMT-BIP takes as input a BIP model and a monitor description which expresses a property $\varphi$, and outputs a new BIP system whose behavior is monitored against $\varphi$ while running concurrently. We experiment with RVMT-BIP on several systems, where each system is monitored against dedicated properties. Our experimental results show that RVMT-BIP induces a cheap runtime overhead [72, 71].

- In the distributed setting, the monitor is aware of the local partial-traces of the schedulers. The monitor computes all the compatible partial traces of the system with respect to the partial ordering of the
received events. Each compatible partial trace could possibly happen in the system and would produce
the same events. We introduce an online algorithm to reconstruct the corresponding global trace for
each partial trace (using the similar technique used for reconstruction of the witness trace in the
multi-threaded setting). To represent the set of reconstructed compatible global traces we use the
genral notion of computation lattice. A computation lattice has $|S|$ orthogonal axes, with one axis
for each scheduler. The direction of each axis represents the system state evolution with respect to the
execution of interactions managed by the associated scheduler. Each path in the lattice represents a
compatible global trace of the system. We define a novel on-the-fly monitoring technique to evaluate
any Linear Temporal Logic (LTL) properties over the computation lattice. To this end, we define a
new structure of the computation lattice in which each node $\eta$ of the lattice is augmented by a set of
formulas representing the evaluation of all the possible global traces from the initial node of the lattice
(i.e., initial state of the system) up to node $\eta$. We show that the constructed lattice is correct in the sense
that it encompasses all the compatible global traces (Proposition 7.38 p. 84, and Proposition 7.39
p. 84). The given formula is monitored by progression over the constructed lattice, so that the frontier
node of the lattice contains a set of formulas, each of which corresponding to the evaluation of a
compatible global trace (Theorem 7.47 p. 92 and Theorem 7.48 p. 92). Furthermore, we introduce an
optimization algorithm to keep the size of the constructed lattice small by removing the unnecessary
nodes. We show that such an optimization on the one hand does not affect the evaluation of the system
and on the other hand increases the performance of the monitoring process.

We present an implementation of our monitoring approach in a tool called RVDIST. RVDIST is a
prototype tool written in the C++ programming language. RVDIST takes as input an LTL formula
and a sequence of events, then constructs and evaluate the computation lattice against the given LTL
property. Moreover, we present the evaluation of our monitoring approach on several distributed
systems carried out with RVDIST. Our experiments show that, thanks to the optimization applied in
the online monitoring algorithm, (i) the size of the constructed computation lattice is insensitive to
the the number of received events, (ii) the lattice size is kept reasonable and (iii) the overhead of the
monitoring process is cheap.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2 (p. 13) a state-of-the-art on monitoring
is presented. Chapter 3 (p. 25) introduces some preliminary concepts. The thesis is partitioned into three
parts.
1.5. THESIS ORGANIZATION

– Part I is the main theoretical part of this thesis. Chapter 4 (p. 33) defines an original abstract semantic model of CBSs, suitable for monitoring purposes, and allowing to define a monitoring hypothesis for the runtime verification of distributed CBSs. In Chapter 5 (p. 43) we define the observable trace of the abstract model and explain the runtime monitoring problem based on the observable trace. Chapter 6 (p. 49) describes the instrumentation of the abstract model to generate the events of the system. In Chapter 7 (p. 61) we introduce methods to reconstruct the corresponding global trace using the events of the system. Such a global trace is suitable for online monitoring.

– Part II presents the implementation and evaluation of the theories introduced in Part I. In Chapter 8 (p. 97) we present the BIP framework. Chapter 9 (p. 105) describes RVMT-BIP, an implementation of the monitoring approach on multi-threaded BIP models. Chapter 10 (p. 119) describes RVDIST, an implementation of the monitoring approach for distributed CBSs. The tools are evaluated in Chapter 11 (p. 123) using several examples.

– Part III consists in Chapter 12 (p. 141) presenting related work and their comparison with our approaches and Chapter 13 (p. 147) in which we conclude and present future work.

Complete proofs related to the correctness of the approach are given in Appendix A (p. 151). In Appendix B (p. 171), we present the user guides of RVMT-BIP and RVDIST.

The part of this thesis associated to the monitoring multi-threaded CBS has been published in iFM 2016, the 12th international conference on integrated Formal Methods [72] and in Formal Aspects of Computing (FAC), a Springer journal [71]. The part of the thesis associated to the monitoring of distributed CBSs is under consideration for publication.
Monitoring User-Provided Specifications on Concurrent Systems

Chapter abstract
In this chapter, we briefly introduce monitoring and we describe the different parameters that affect the design of systems monitoring. We overview the work done on monitoring user-provided specification on concurrent systems. In Chapter 12 (p. 141), we compare our approach to the related approaches described in this chapter.

2.1 Monitoring
Monitoring or runtime verification (RV) is a dynamic analysis method aiming at checking whether a run of the system under scrutiny satisfies a given property. The inputs to an RV system are: (i) a system to be verified, and (ii) a set of properties to be checked against the system execution. The properties can be expressed in a formal specification language (e.g., automata-based or logic-based formalism), or even as a program in a general-purpose programming language. A runtime verification process typically consists of the following three stages. First, from a property is generated a monitor, that is a decision procedure for the property. This step is often referred to as monitor synthesis. The monitor is capable of consuming events produced by a running system and emits verdicts according to the current satisfaction of the property based on the history of received events. Second, the system under scrutiny is instrumented. The purpose of this stage is to be able to generate the relevant events to be fed to the monitor. This step is often referred to
as system instrumentation. Third, the execution of the system is analyzed by the monitor. This analysis can occur either during the execution in a lock-step manner (online monitoring), or after the execution has finished assuming that events have been written to a log (offline monitoring). This step is often referred to as execution analysis.

### 2.1.1 Online Versus Offline

In online monitoring, the monitor is running alongside the monitored system and is expected to detect satisfaction/violation to the desired properties as soon as they occur. The online monitor can either be inline (internal) where it is included in the code of the system, or outline (external) where it exists as an external entity [36]. Offline monitoring takes place after the system has terminated. Offline monitoring can be used for instance for testing purposes, where the system is executed many times with different input parameters and then the execution traces are evaluated by the monitor. Decentralized offline monitors may be used for parallelizing the evaluation process of large traces [53, 4].

### 2.1.2 Synchronous Versus Asynchronous Distributed Systems

The design of distributed systems affects greatly the design of the monitoring system. Synchronous distributed systems that depend on global clock are relatively easier to monitor than asynchronous distributed systems, where each scheduler (process) has a local clock. In a synchronous distributed system, the monitor can easily order the events generated by different schedulers and process them sequentially to evaluate the desired property. However, in an asynchronous distributed systems clock drifts are inevitable and therefore, the monitor can not use the timestamps of events from different schedulers to order the events. Instead, the monitor uses the partial order induced by the communication between the distributed entities (schedulers or processes) or indirectly through the shared elements (shared components) to order the events. Note that the total order of concurrent events can not be determined in an asynchronous setting, therefore the monitor has to consider all the possible total orders that could happened in the actual run.

### 2.1.3 Centralized Monitor Versus Decentralized Monitor

The runtime monitor can be either centralized as a single process or decentralized as a set of multiple processes such that the monitoring tasks are distributed among them.
Centralized Monitor

A centralized monitor receives the events from all schedulers, evaluates the desired property and emits the corresponding verdict about the violation or satisfaction of the property. The central monitor is also responsible for ordering the events it receives from the schedulers, which is a harder job if the system is asynchronous. The central monitor resides on either, a new distributed entity dedicated to monitoring, or on one of the existing schedulers. In [8], the authors present a framework for detecting and analyzing synchronous distributed systems faults in a centralized manner using LTL properties. In [64], the authors present a monitoring technique that uses symbolic composition of events with the monitor to detect satisfaction/violation of LTL properties in an asynchronous distributed system.

Decentralized Monitor

Decentralized monitoring aims at decentralizing the monitoring load from one node to several monitoring nodes [30, 29]. Each monitoring node is attached to one scheduler and receives the events from the scheduler as soon as they happen. The decentralized design offers many advantages compared to the centralized design such as the absence of single point of failure, faster notification of failures or violations, distributed memory and computation overload among the monitoring nodes. However, decentralized monitors are required to communicate together to evaluate the correctness property leading to a complicated design. In [83], the authors introduced a method for decentralized monitoring safety properties in distributed systems using the past-time linear temporal logic (PLTL). The monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. In [13], the authors introduce decentralized monitoring algorithms for runtime verification of sequential programs. In [11], the authors presented a decentralized monitoring algorithm to verify LTL formulas for synchronous distributed systems. Lattice-theoretic centralized and decentralized online monitoring has been studied in [21, 68]. Migration and choreography are two design approaches of decentralized monitor, described after.

Migration  In the migrating monitors design, the monitor process migrates from a program process to another with the objective of minimizing computation and communication overhead. In [11], the authors present a runtime verification algorithm of LTL specifications for synchronous distributed systems, where processes share a single global clock. The LTL formulas transferred across subsystems to gather local information.

Choreography  In the choreography design presented first in [23], the authors present a technique where the LTL formula is divided into subformulas and the monitor nodes are arranged in a tree-like structure,
such that each leaf node is responsible for evaluating local propositions, while the intermediate nodes aggregate the results and forwards it upwards until the formula is evaluated. In [43], dynamic choreography is proposed. The latter is similar to state choreography presented in [23], but rearranges the network of monitors during execution, allowing monitoring dynamic properties such as the ones created or that has evolved during runtime.

2.2 Monitoring Distributed Systems

In the following, we briefly describe some research efforts on monitoring distributed systems.

**Distributed snapshots: determining global states of distributed systems.** In [20], Chandy and Lamport introduced the notion of global snapshot, which is the basis for almost all subsequent research on detecting the truth value of stable predicates. A predicate is stable if it does not turn false once it becomes true. They presented an algorithm by which a process in a distributed system determines a global state (consistent cut) of the system during a computation.

**Repeated snapshot** Bouge [17] and Spezialetti and Kearns [88] have extended global snapshot method [20] for repeated snapshot. By taking the global snapshot periodically, the truth value of a stable property can be detected. In their work, processes can take several snapshots of the system in such a way that successive phases of the algorithm do not interfere, and taking snapshots does not overflood the system. Nevertheless, this approach does not work for unstable predicates which may be true only between two snapshots and not at the time the snapshot is taken.

**Detection of weak unstable predicates in distributed programs.** In [49], Garg and Waldechel define a class of unstable predicates called weak conjunctive predicates. A weak conjunctive predicate consists of a conjunction of local predicates such as \((p_1 \land p_2 \land \cdots \land p_n)\), where \(p_i\) is evaluated in process \(i\). It is defined to be true in an execution if there exists a global state in the execution such that the expression evaluates to true. In that paper, an algorithm for detecting such predicated is presented.

**Detection of unstable predicates in distributed programs.** In [48], Garg and Waldechel also define strong conjunctive predicates which, like their weak counterparts, consist of a conjunction of local predicates. The predicate evaluates to true (for a particular execution) when every sequence of global states consistent with the execution contains a global state which satisfies the conjunctive boolean expression. In other words, a strong conjunctive predicate is true if and only if the system will always reach a global state
such that all local predicates hold. In that paper, an algorithm for detecting strong conjunctive predicates is presented.

**Detecting conjunctive channel predicates in a distributed programming environment.** In [46], Garg and Chase introduce the concept of a channel predicate and extend weak conjunctive predicates to include predicates on the state of message channels. A channel is a uni-directional connection between any two processes through which messages can be passed. A channel predicate is any boolean function of the state of the channel. The authors restrict the channel predicates to a class called dynamically monotonic predicates which can behave in some states as a send-monotonic predicate which can not be made true by sending more messages along the same channel, and in other states as a receive-monotonic predicate which is analogous, so that when it is false, it can not be made true by only receiving more messages.

**Monitoring functions on global states of distributed programs.** A global function is a function whose domain is the set of all global states of a given execution. In [90], Tomlinson and Garg show how to monitor the value of a global function in a distributed program while the program is executing. The authors discuss relational global predicates of the form \((x_1 + x_2 \geq k)\), where \(x_i\) is defined in process \(i\). They proposed a fully decentralized algorithm and generalized it to monitor the system in order to determine if there exists a global state in which predicate \(x_1 + x_2 + \cdots + x_n > k\) holds. They examined the cases where \(x_i\) is an integer variable and where \(x_i\) is a boolean variable.

**Consistent global states of distributed systems: fundamental concepts and mechanisms.** Babaoglu and Marzullo [1] define two strategies for solving Global Predicate Evaluation in asynchronous distributed systems by construction of the computation lattice. The first strategy is such that an active monitor queries the rest of the system in order to construct the global state. The second strategy is such that a passive monitor observes the system in order to construct its global states. Whenever processes execute an event, they notify the passive monitor by sending the event.

**Specification and verification of behavioral patterns in distributed computations.** In [2], Babaoglu and Raynal introduce the syntactic classes simple sequence and interval-constrained sequence of global predicates that define sets of global states related through the notion of causality-preserving sequencing. These classes admit Boolean expression over global states as building blocks and include temporal specifications through causality-preserving sequencing and interval negation. They develop an online algorithm for verifying the satisfaction of such formulas. The verification relies on a monitor which is internal to the system and concurrently executing with the actual computation.
A general approach to trace-checking in distributed computing systems. In [57], Jard and Jeron propose to check a distributed computation against regular properties which are described by finite-state automata. They consider the lattice of global states representing all possible observation of the distributed computation. Some satisfaction is claimed when at least one path of the lattice is recognized by the automaton whereas every satisfaction is claimed when each path of the lattice is recognized by the automaton. They proposed an approach to check properties on traces, based on partial order theory.

Breakpoints and halting in distributed programs. In [66], Miller and Choi extend Chandy and Lamport’s algorithm for recording global state in [20] by introducing linked predicates, which describes a causal sequence of local states where each state in the sequence satisfies a specific local predicate. Linked predicates are used with hardware-based debugging tools such as logic-state analyzer. The behavior "an occurrence of local predicate $p$ is causally followed by an occurrence of local predicate $q$" is an example of a linked predicate. The satisfaction of these predicates corresponds to interesting point in the execution of the distributed program, which is called breakpoints. The authors present a halting algorithm to halt the computation at the breakpoints.

Detecting atomic sequences of predicates in distributed computations. In [56], Hurfin and Plouzeau generalized linked predicates to a broader class called atomic sequences of predicates. In this class, the occurrences of local predicates can be forbidden between adjacent predicates in a linked predicate. In other words, they describe global properties by causal composition of local predicates augmented with atomicity constraints. The behavior "an occurrence of local predicate $p$ is causally followed by an occurrence of local predicate $q$" could be expanded to include "$q$ follows $p$ and $r$ never occurs in between" which is an example of such predicates, where $p$, $q$ and $r$ could be evaluated in different processes.

On the fly testing of regular patterns in distributed computations. In [45], Formentin and Raynal introduce regular pattern which is a class of properties of distributed computations and is based upon regular expressions and includes linked predicates and atomic sequences. This class of properties allows the user to specify an expected (or unwanted) behavior of a computation as sequences of relevant events or as sequence of local predicates that must be successively verified. These sequences are defined by a finite-state automaton. For example $pq^*r$ is true in a computation if there exists a sequence of consecutive local states $(s_1, s_2, \cdots, s_n)$ such that $p$ is true in $s_1$, $q$ is true in $s_2, \cdots, s_{n-1}$, and $r$ is true in $s_n$. In that paper, an algorithm is defined, so that a computation verifies the property if and only if one of its causal path matches a sequence. The detection algorithm works on the fly without building a complex and expensive data structure such as a lattice.
An efficient decentralized algorithm for detecting properties of distributed computation. In [50], Garg and Tomlinson extend the results of [45] to introduce a class of behavior which includes regular patterns as a special case. They designed a logic (called LRDAG) for expressing these properties and presented an efficient decentralized algorithm for detecting formulas in the logic. They also defined a class of algorithms called Efficient Passive Detection Algorithm (EPDA) to verify the LRDAG properties of a distributed computation. They used the term passive because the algorithms can only observe the computation, such that they can not initiate sending or receiving of messages and then can not alter the original behavior of the observed computation.

Consistent detection of global predicates. In [24], Cooper and Marzullo present three algorithms for detecting global predicates based on the construction of the lattice associated with a distributed execution. The algorithms traverse the lattice of global states in an online manner. The first algorithm determines that the predicate was possibly true at some point in the past; the second algorithm determines that the predicate was definitely true in the past; while the third algorithm establishes that the predicate is currently true, but to do so it may delay the execution of certain processes.

Reachability analysis on distributed executions. In [26], Diehl, Jard and Rampon present a verification technique based on the execution trace of a distributed program, and they called it trace checking. The authors introduce an algorithm for trace checking by building the lattice of all reachable states of the distributed system under test, based on the on-the-fly observation of the partial order of message causality. Actually this work addresses the shortcoming of the work of Cooper and Marzullo [24] (the first to perform a reachability analysis on the state space associated to a distributed execution), that is, the execution of some processes must be delayed while waiting for an event.

Techniques and applications of computation slicing. In [68], Mittal and Garg developed computation slicing, which was first introduced in [47] as an abstraction technique for analyzing traces of distributed programs. Intuitively, a slice of a trace with respect to a specification $p$ is a subtrace that contains all the states of the trace that satisfy $p$. Note that the set of states that satisfy $p$ may be large, so one could not simply enumerate all of the states efficiently either in space or time. A slice contains all of the states that satisfy $p$ such that it is computed efficiently (without traversing the state space) and represented concisely (without explicit representation of individual states). In that paper, a centralized offline algorithm for slicing based regular predicate detection is presented.
A distributed abstraction algorithm for online predicate detection. In [21], authors used the computation slicing approach for abstracting distributed computations with respect to a given regular predicate. Computation slicing is an abstraction technique for efficiently finding the slice, without explicitly enumerating all such global states. A slice contains the least number of global states of a distributed computation that satisfy a given global predicate. The slice is updated incrementally with the arrival of every new relevant event. In this work, a decentralized online monitoring algorithm is presented.

Detecting temporal logic predicates on distributed computations. In [73], Ogale and Garg present a technique that allows efficient detection of a class of predicates which they call Basic Temporal Logic (BTL). An example of a BTL predicate would be a property based on local predicates and arbitrarily-placed negations, disjunctions and conjunctions along with $F$ (eventually) and $G$ (globally) temporal operators. They introduce the concept of basis, a compact representation of the subset of the computational lattice containing exactly those global states (or cuts) that satisfy the predicate. In that paper, an offline algorithm to compute a basis of a computation given any BTL predicates is presented.

Model-based runtime analysis of distributed reactive systems. In [8], Bauer et al. present a framework for detecting and analyzing synchronous distributed systems faults in a centralized manner using local LTL properties that require only a trace of the execution at the local node. Each node checks that specific safety properties hold and if violated, sends a report to a centralized diagnosis engine that attempts to ascertain the source of the problem and to steer the distributed system to a safe state.

Decentralized LTL monitoring In [11], the authors present an algorithm for distributing and monitoring LTL formulas for synchronous distributed systems such that satisfaction or violation of the property can be detected locally. In this setting, there exist multiple local monitors in the system. Each local monitor sees only a distinct part of the global behavior, and observes a subset of some global event trace. Given an LTL property $\varphi$, their goal is to create sound formula derived from $\varphi$ that can be monitored on each local trace, while minimizing inter-component communication.

Efficient decentralized monitoring of safety in distributed systems. In [83], Sen and Vardhan design a method for monitoring safety properties in distributed systems using the past-time linear temporal logic (PLTL). However, their algorithm is not sound, meaning that the evaluation of some predicates and properties may not be observed. This is due to the fact that monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. That is, if processes rarely communicate, then monitors exchange very little information and, hence, some violations of properties
may remain undetected. In that paper, a tool called DIANA (distributed analysis) is introduced in order to implement the proposed monitoring method.

**Efficient online monitoring of LTL properties for asynchronous distributed systems.** In [64], Massart and Meuter define an online method to monitor the execution of asynchronous distributed systems. The online monitor collects the trace and checks on-the-fly whether it satisfies a requirement, given by any LTL property defined over finite sequences. Their method explores the possible configurations symbolically, as it handles sets of configurations. Their proposed monitor is not always sensitive to all events, which results in the reduction in the number of interleavings to explore. In their approach, each configuration separates both kinds of events: optional events i.e., events that do not take part in a monitor move, and mandatory events, i.e., events that take part in a monitor move.

**Three-valued asynchronous distributed runtime verification.** In [78], Scheffel and Schmitz studied runtime verification of distributed asynchronous systems against properties expressed in Distributed Temporal Logic (DTL). DTL combines three-valued Linear Temporal Logic (LTL$_3$) [10] with past-time Distributed Temporal Logic (ptDTL). In that paper, a distributed system is modeled as multiple agents and each agent has a local monitor. These monitors work together to check a property, but they only communicate by adding some data to the messages already sent by the agents. They can not force their agent to send a message or even communicate on their own.

**Decentralized runtime verification of LTL specifications in distributed systems.** In [69], a decentralized algorithm for runtime verification of distributed programs is proposed. The algorithm is dedicated to the 3-valued semantics of the linear temporal logic (LTL$_3$) [10]. In that paper, they adapt the distributed computation slicing algorithm for distributed online detection of conjunctive predicates, and also the lattice-theoretic technique is adapted for detecting global-state predicates at run time.

**Detecting temporal logic predicates on the happened-before model.** In [82], Sen and Garg use Computation Tree Logic (CTL), for specifying properties of distributed computation. CTL properties are expressed over a tree of all possible executions of the system and was first presented in [31]. Their method interprets the property on a finite lattice of global states and checks that a predicate is satisfied for an observed single execution trace of the program.

**Detecting temporal logic predicates in distributed programs using computation slicing.** In [80], Sen and Garg used computation slicing for offline predicate detection in the subset of CTL with the following
three properties; (i) temporal operators, (ii) atomic propositions are regular predicates and (iii) negation operator is pushed onto atomic propositions. They called this logic Regular CTL plus (RCTL+), where plus indicates that the disjunction and negation operators are included in the logic. In that paper, the authors gave the formal definition of RCTL+. They implemented their predicate detection algorithms in a prototype tool called Partial Order Trace Analyzer (POTA). Moreover, the authors developed this work in [81], by presenting the central online algorithm with respect to properties expressed in RCTL+.

### 2.2.1 Summary

Table 2.1 (p. 23) summarizes these research efforts on monitoring distributed systems. In this table, we categorized the research efforts based on the monitoring methods which are either online or offline and the type of behavioral predicates they verify.

Behavioral predicates (i.e., properties) can be partitioned into two main categories: global-state based and non-global-state base. A global-state based predicate is predicate whose satisfaction or violation is evaluated on the set of traversed global states of the system in the execution. For each global state, the global-state predicate is evaluated to true or false. The global-state predicates can be further divided into stable and unstable predicates. A predicate is said to be stable if it stays true once it becomes true for all reachable states, e.g., deadlock, termination. An unstable predicate can become true and then later become false.

Non-global-state based predicates can not be evaluated on a global state. The evaluation of a non-global-state based predicates requires the global trace of the system, i.e., a sequence of causally related global states with respect to the global behavior of the system. Variety of non-global-state based predicates are defined so far, such as regular predicates, sequential predicates, regular pattern, labeled root directed acyclic graph (LRDAG), and temporal predicates. However, temporal logic predicates are gaining increasing interests in several application areas e.g., formal verification, model checking.

The work presented in this thesis falls into the online monitoring of non-global-state predicates.
Table 2.1: State-of-the-art monitoring distributed systems

<table>
<thead>
<tr>
<th>Behavioral Predicate</th>
<th>Global-State Based</th>
<th>Non-Global-State Based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stable</td>
<td>Unstable</td>
</tr>
<tr>
<td>Conjugate Predicates</td>
<td>Channel Predicates</td>
<td>Relational Predicates</td>
</tr>
<tr>
<td>Global Snapshot</td>
<td>[20]</td>
<td></td>
</tr>
<tr>
<td>Repeated Snapshot</td>
<td>[17], [88]</td>
<td></td>
</tr>
<tr>
<td>Predicate Exploiting</td>
<td>[49], [48], [46], [90]</td>
<td></td>
</tr>
<tr>
<td>Lattice Construction</td>
<td>[21], [20], [1], [2], [57]</td>
<td></td>
</tr>
<tr>
<td>Piggybacking</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computation Sclicing</td>
<td>[21]</td>
<td></td>
</tr>
<tr>
<td>DIANA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbolic Monitoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Offline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computation Sclicing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computing Basis</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter abstract

In this chapter, we introduce some preliminary concepts and notations used throughout this thesis. We present the notions of functions, pattern matching, sequences and map operator (which applies a function to a sequence). We recall labeled transition systems (LTSs), which are used to define the semantics of component-based systems. We present the bi-simulation relation between two LTSs. We define notations related to distributed systems such as vector clock, happened-before relation and computation lattice.

Finally, we recall linear temporal logic which we use to formalize the requirement of systems.

Functions. For two domains of elements $E$ and $F$, we note $E \rightarrow F$ the set of functions from $E$ to $F$. The fact that function $f$ belongs to $E \rightarrow F$ is denoted $f : E \rightarrow F$. For two functions $v : X \rightarrow Y$ and $v' : X' \rightarrow Y'$, the function obtained by overriding $v$ images by $v'$ images is denoted by $v \backslash v'$, where $v \backslash v' : X \cup X' \rightarrow Y \cup Y'$, and is defined as follows:

$$v \backslash v'(x) = \begin{cases} v'(x) & \text{if } x \in X', \\ v(x) & \text{otherwise.} \end{cases}$$

Example 3.1 (Functions). For two functions $v_n : \mathbb{N} \rightarrow \mathbb{N}$ and $v_r : \mathbb{R} \rightarrow \mathbb{R}$ such that $v_n(x) = x + 1$ and $v_r(x) = x \div 2$, the overriding function $v_r \backslash v_n(4.2) = 2.1$ whereas $v_r \backslash v_n(4) = 5$.

Pattern-matching. We shall use the mechanism of pattern-matching to concisely define some functions. We recall an intuitive definition for the sake of completeness. Evaluating the expression:
match expression with
| pattern_1 → expression_1
| pattern_2 → expression_2
... | pattern_n → expression_n

consists in comparing successively expression with the patterns pattern_1, ..., pattern_n in order. When a pattern pattern_i fits expression, then the associated expression_i is returned.

Sequences. For a finite set of elements $E$, a sequence $s$ containing elements of $E$ is formally defined by a total function $s : I \to E$ where $I$ is either the integer interval $[0..n]$ for some $n \in \mathbb{N}$, or $\mathbb{N}$ itself (the set of natural numbers). Given a set of elements $E$, $e_1 \cdot e_2 \cdots e_n$ is a sequence or a list of length $n$ over $E$, where $\forall i \in [1..n]. e_i \in E$. The empty sequence is noted $\epsilon$ or $[]$, depending on the context. The set of (finite) sequences over $E$ is noted $E^*$. $E^+$ is defined as $E^* \setminus \{\epsilon\}$. The length of a sequence $s$ is noted $\text{length}(s)$. We define $s(i)$ as the $i^{\text{th}}$ element of $s$ and $s(i\cdots j)$ as the factor of $s$ from the $i^{\text{th}}$ to the $j^{\text{th}}$ element. $s(i\cdots j) = \epsilon$ if $i > j$. We also note $\text{pref}(s)$, the set of non-empty prefixes of $s$, i.e., $\text{pref}(s) = \{s(1\cdots k) | 1 \leq k \leq \text{length}(s)\}$. Operator $\text{pref}$ is naturally extended to sets of sequences. Function $\text{max}_\preceq$ (resp. $\text{min}_\preceq$) returns the maximal (resp. minimal) sequence w.r.t. prefix ordering of a set of sequences. We define function $\text{last} : E^+ \to E$ as $\text{last}(s) = s(\text{length}(s))$. For an infinite sequence $s = e_1 \cdot e_2 \cdot e_3 \cdots$, we define $s(i\cdots) = e_i \cdot e_{i+1} \cdots$ as the suffix of sequence $s$ from index $i$ on.

Example 3.2 (Sequences). For a sequence of natural numbers $s = 3 \cdot 18 \cdot 7 \cdot 30 \cdot 24 \cdot 1 \cdot 12$, the length of the sequence is $\text{length}(s) = 7$, $s(4) = 30$, $s(2\cdots 5) = 18 \cdot 7 \cdot 30 \cdot 24$, $p_1 = 3 \cdot 18 \cdot 7$ and $p_2 = 3 \cdot 18 \cdot 7 \cdot 30 \cdot 24 \cdot 1$ are two elements of $\text{pref}(s)$, last($s$) = 12 and $s(4\cdots) = 30 \cdot 24 \cdot 1 \cdot 12$.

Map operator: applying a function to a sequence. For a sequence $e = e_1 \cdot e_2 \cdots e_n$ of elements over $E$ of some length $n \in \mathbb{N}$, and a function $f : E \to F$, map $f \cdot e$ is the sequence of elements of $F$ defined as $f(e_1) \cdot f(e_2) \cdots f(e_n)$ where $\forall i \in [1..n]. f(e_i) \in F$.

Example 3.3 (Map operator). For a sequence of natural numbers $s = 3 \cdot 18 \cdot 7 \cdot 30 \cdot 24 \cdot 1 \cdot 12$ and a function $v_n : \mathbb{N} \to \mathbb{N}$ such that $v_n(x) = x + 1$, map $v_n \cdot s$ = $4 \cdot 19 \cdot 8 \cdot 31 \cdot 25 \cdot 2 \cdot 13$.

Tuples. An $n$-tuple is an ordered list of $n$ elements, where $n$ is a strictly positive integer. By $t[i]$ we denote the $i^{\text{th}}$ element of tuple $t$.

**Labeled transition systems.** Labeled Transition Systems (LTSs) are used to define the semantics of CBSs. An LTS is defined over an alphabet Σ and is a 3-tuple (State, Lab, Trans) where State is a non-empty set of states, Lab is a set of labels, and Trans ⊆ State × Lab × State is the transition relation. A transition \((q, a, q') \in \text{Trans}\) means that the LTS can move from state \(q\) to state \(q'\) by consuming label \(a\). We abbreviate \((q, a, q') \in \text{Trans}\) by \(q \xrightarrow{a} q'\) or by \(q \xrightarrow{a} q'\) when clear from context. Moreover, relation Trans is extended to its reflexive and transitive closure in the usual way and we allow for regular expressions over Lab to label moves between states: if \(\text{expr}\) is a regular expression over Lab (i.e., \(\text{expr}\) denotes a subset of \(\text{Lab}^\ast\)), \(q \xrightarrow{\text{expr}} q'\) means that there exists one sequence of labels in Lab matching \(\text{expr}\) such that the system can move from \(q\) to \(q'\).

**Observational equivalence and bi-simulation.** The observational equivalence of two transition systems is based on the usual definition of weak bisimilarity [67], where \(\theta\)-transitions are considered to be unobservable. Given two transition systems \(S_1 = (\text{Sta}_1, \text{Lab} \cup \{\theta\}, \rightarrow_1)\) and \(S_2 = (\text{Sta}_2, \text{Lab} \cup \{\theta\}, \rightarrow_2)\), system \(S_1\) weakly simulates system \(S_2\), if there exists a relation \(R \subseteq \text{Sta}_1 \times \text{Sta}_2\) that contains the 2-tuple made of the initial states of \(S_1\) et \(S_2\) and such that the two following conditions hold:

1. \(\forall (q_1, q_2) \in R, \forall a \in \text{Lab} . q_1 \xrightarrow{a} q'_1 \implies \exists q'_2 \in \text{Sta}_2 . \big( (q'_1, q'_2) \in R \land q_2 \xrightarrow{\theta^* a \theta^*} q'_2 \big)\), and

2. \(\forall (q_1, q_2) \in R . \big( \exists q'_1 \in \text{Sta}_1 . q_1 \xrightarrow{\theta} q'_1 \big) \implies \exists q'_2 \in \text{Sta}_2 . \big( (q'_1, q'_2) \in R \land q_2 \xrightarrow{\theta^*} q'_2 \big)\).

Equation 1. states that if a state \(q_1\) simulates a state \(q_2\) and if it is possible to perform \(a\) from \(q_1\) to end in a state \(q'_1\), then there exists a state \(q'_2\) simulated by \(q'_1\) such that it is possible to go from \(q_2\) to \(q'_2\) by performing some unobservable actions, the action \(a\), and then some unobservable actions. Equation 2. states that if a state \(q_1\) simulates a state \(q_2\) and it is possible to perform an unobservable action from \(q_1\) to reach a state \(q'_1\), then it is possible to reach a state \(q'_2\) by a sequence of unobservable actions such that \(q'_1\) simulates \(q'_2\). In that case, we say that relation \(R\) is a weak simulation over \(S_1\) and \(S_2\) or equivalently that the states of \(S_1\) are (weakly) similar to the states of \(S_2\). Similarly, a weak bi-simulation over \(S_1\) and \(S_2\) is a relation \(R\) such that \(R\) and \(R^{-1}\) are both weak simulations. In this latter case, we say that \(S_1\) and \(S_2\) are observationally equivalent and we write \(S_1 \sim S_2\) to express this formally.

**Vector clock.** Lamport introduced logical clocks as a device to substitute for the global real time clock [60]. Logical clocks are used to order events based on their relative logical dependencies rather than on a “time” in the common sense. Mattern and Fidge’s vector clocks [42] [65] are a more powerful extension (i.e., strongly consistent with the ordering of events) of Lamport’s scalar logical clocks. In a distributed system with a set of schedulers \(\{S_1, \ldots, S_m\}\), \(VC = \{(c_1, \ldots, c_m) \mid j \in [1..m] \land c_j \in \mathbb{N}\}\) is the set of vector
clocks, such that vector clock $vc \in VC$ is a tuple of $m$ scalar (initially zero) values $c_1, \ldots, c_m$ locally stored in each scheduler $S_j \in \{S_1, \ldots, S_m\}$ where $\forall k \in [1..m] \cdot vc[k] = c_k$ holds the latest (scalar) clock value scheduler $S_j$ knows about scheduler $S_k \in \{S_1, \ldots, S_m\}$. Each event in the system is associated to a unique vector clock. For two vector clocks $vc_1$ and $vc_2$, $\max(vc_1, vc_2)$ is a vector clock $vc_3$ such that $\forall k \in [1..m] \cdot vc_3[k] = \max(vc_1[k], vc_2[k])$. $\min(vc_1, vc_2)$ is defined in similar way. Moreover two vector clocks can be compared together such that $vc_1 < vc_2 \iff \forall k \in [1..m] \cdot vc_1[k] \leq vc_2[k] \land \exists z \in [1..m]\cdot vc_1[z] < vc_2[z]$.

**Example 3.5** (Vector clock). For a distributed system consisting of three schedulers $\{S_1, S_2, S_3\}$, if the value of the vector clock of scheduler $S_2$ is $(2, 4, 3)$, it means that 4 events have happened in scheduler $S_2$ and also scheduler $S_2$ is aware of the occurrence of 2 first events of scheduler $S_1$ and 3 first events of scheduler $S_3$. For two vector clocks $vc_1 = (2, 4, 3)$ and $vc_2 = (3, 2, 5)$, $\max(vc_1, vc_2) = (3, 4, 5)$. For two vector clocks $vc_1 = (2, 4, 3)$ and $vc_2 = (4, 7, 4)$, we have $vc_1 < vc_2$.

**Happened-before relation** [60]. The relation $\rightarrow$ on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If $a$ and $b$ are events in the same scheduler, and $a$ comes before $b$, then $a \rightarrow b$. (2) If $a$ is the sending of a message by one scheduler and $b$ is the reception of the same message by another scheduler, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$. Two distinct events $a$ and $b$ are said to be concurrent if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Vector clocks are strongly consistent with happened-before relation. That is, for two events $a$ and $b$ with associated vector clocks $vc_a$ and $vc_b$, respectively, $vc_a < vc_b \iff a \rightarrow b$.

**Example 3.6** (Happened-before relation). For three events $e_1$ with the associated vector clock $vc_1 = (2, 4, 3)$, $e_2$ with the associated vector clock $vc_2 = (4, 7, 4)$, and $e_3$ with the associated vector clock $vc_3 = (3, 2, 5)$, we say that event $e_1$ is happened before event $e_2$ (denoted by $e_1 \rightarrow e_2$), because $vc_1 < vc_2$. Moreover, event $e_3$ is concurrent with events $e_1$ and $e_2$.

**Computation lattice** [65]. The computation lattice of a distributed system is represented in the form of a directed graph with $m$ (i.e., number of schedulers that are executed in distributed manner) orthogonal axes. Each axis is dedicated to the state evolution of a specific scheduler. A computation lattice expresses all the possible traces in a distributed system. Each path in the lattice represents a global trace of the system that could possibly have happened. A computation lattice $\mathcal{L}$ is a pair $(N, \rightarrow)$, where $N$ is the set of nodes (i.e., global states) and $\rightarrow$ is the set of happened-before relations among the nodes.

**Example 3.7** (Computation lattice). Let us consider a distributed system with two schedulers $S_1$ and $S_2$. One event is occurred in scheduler $S_1$ and two events in scheduler $S_2$. The first event of schedulers $S_1$ is
concurrent with the first event of schedulers $S_2$, whereas the second event of schedulers $S_2$ is happened after them. The occurrence of each event extends the lattice in the dedicated direction of the scheduler executed the event. Figure 3.1 depicts the two dimensions computation lattice of such a system. The set of nodes is $\{\eta_0, \eta_1, \eta_2, \eta_3, \eta_4\}$ and the set of happened-before relations is $\{\eta_0 \rightarrow \eta_1, \eta_0 \rightarrow \eta_2, \eta_0 \rightarrow \eta_3, \eta_1 \rightarrow \eta_3, \eta_2 \rightarrow \eta_3, \eta_3 \rightarrow \eta_4\}$. Node $\eta_0$ represents the initial state of the system. Node $\eta_1$ represents the global state of the system after the occurrence of the first event of scheduler $S_1$ and before the occurrence of the two events of scheduler $S_2$. Node $\eta_3$ represents the global state of the system after the occurrence of first events of the schedulers and before the second event of scheduler $S_2$. Such a computation lattice has three paths: (i) $\eta_0 \cdot \eta_1 \cdot \eta_3 \cdot \eta_4$, (ii) $\eta_0 \cdot \eta_2 \cdot \eta_3 \cdot \eta_4$, and (iii) $\eta_0 \cdot \eta_3 \cdot \eta_4$. The path from node $\eta_0$ to $\eta_3$ represents the state evolution of the system in case of the simultaneous occurrence of two concurrent events (that is the first events of the schedulers).

**Linear Temporal Logic (LTL)** [75]. Linear temporal logic (LTL) is a formalism for specifying properties of systems. An LTL formula is built over a set of atomic propositions $AP$. LTL formulas are written with the following grammar:

$$\varphi ::= p \mid \neg \varphi \mid \varphi_1 \lor \varphi_2 \mid X \varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $p \in AP$ is an atomic proposition.

Let $\sigma = q_0 \cdot q_1 \cdot q_2 \cdots$ be an infinite sequence of states and $|$ denotes the satisfaction relation. The semantics of LTL is defined inductively as follows:

- $\sigma \models p$ if and only if $q_0 \models p$ (i.e., $p \in q_0$), for any $p \in AP$
- $\sigma \models \neg \varphi$ if and only if $\sigma \not\models \varphi$
- $\sigma \models \varphi_1 \lor \varphi_2$ if and only if $\sigma \models \varphi_1 \lor \sigma \models \varphi_2$
- $\sigma \models \text{X} \varphi \iff \sigma(1 \cdots) \models \varphi$
- $\sigma \models \varphi_1 \text{U} \varphi_2 \iff \exists j \geq 0. \sigma(j \cdots) \models \varphi_2 \land \sigma(i \cdots) \models \varphi_1, 0 \leq i < j$

An atomic proposition $p$ is satisfied by $\sigma$ when it is member of the first state of $\sigma$. $\sigma$ satisfies formula $\neg \varphi$ when it does not satisfy $\varphi$. Disjunction of $\varphi_1$ and $\varphi_2$ is satisfied when either $\varphi_1$ or $\varphi_2$ is satisfied by $\sigma$. $\sigma$ satisfies formula $\text{X} \varphi$ when the sequence of states starting from the next state of $\sigma$, that is, $q_1$ satisfies $\varphi$. $\varphi_1 \text{U} \varphi_2$ is satisfied when $\varphi_2$ is satisfied at some point and $\varphi_1$ is satisfied until that point.

Note that we use only the $\text{X}$ and $\text{U}$ modalities for defining the valid formulas in LTL. The other modalities such as $\text{F}$ (eventually), $\text{G}$ (globally), $\text{R}$ (release), etc. in LTL can be defined using the $\text{X}$ and $\text{U}$ modalities such that for example:

- $\varphi_1 \text{R} \varphi_2 = \neg (\neg \varphi_1 \text{U} \neg \varphi_2). \varphi_2$ remains true until and including once $\varphi_1$ becomes true. If $\varphi_1$ never become true, $\varphi_2$ must remain true forever.
- $\text{F} \varphi = \text{T} \text{F} \varphi$. Eventually $\varphi$ becomes true.
- $\text{G} \varphi = \text{F} \text{R} \varphi = \neg \text{F} \neg \varphi$. $\varphi$ always remains true.

**Example 3.8 (LTL).** The requirement stating that “Once the traffic light is green, it cannot become red immediately” can be formalized in LTL as: $\text{G}(\text{green} \Rightarrow \neg \text{red})$. The requirement stating that “Once red, the traffic light becomes green eventually” can be formalized in LTL as: $\text{G}(\text{red} \Rightarrow \text{F} \text{green})$. The requirement stating that “Once green, the traffic light becomes red eventually, after being yellow for some time in between” can be formalized in LTL as: $\text{G}(\text{green} \Rightarrow (\text{green U yellow}) \text{U red})$. Suppose $\varphi_x$ denotes “process $x$ is in critical state”. The requirement stating that “process $1$ and $2$ are never both in their critical state at the same time (mutual exclusion)” can be formalized in LTL as: $\text{G}(\neg \varphi_1 \lor \neg \varphi_2)$. 


Part I

Monitoring Component-Based Systems with Multi-Party Interactions (CBSs)
Abstract Semantic Model of Distributed, Multi-threaded and Sequential CBSs

Chapter abstract
In this chapter, we define an abstract semantic model of component-based systems with multi-party interactions, referred to as CBS. Since our model only rely on the semantics of CBS, which is given in terms of LTSs, our abstract model is thus compatible with CBS frameworks and systems that have their semantics that can be modeled with LTSs. Therefore, this model is abstract enough to encompass a variety of (component-based) systems, and serves the purpose of describing the knowledge needed on the verified system and later guides their instrumentation. In the following, we describe our assumptions on the considered CBSs. To this end, we assume a general semantics to define the behavior of the system under scrutiny in order to make our monitoring approach as general as possible. However, neither the exact model nor the behavior of the system are known. Usually in runtime verification, the model of the system is a black box whose behavior can be modeled as an LTS over some set of concrete actions of the system. This model is generally unknown, except for predictive RV. In our context, we also do not know the exact model but we have more information on the structure since we have a CBS. Inspiring from conformance-testing theory [91], we refer to this hypothesis as the monitoring hypothesis. Consequently, our monitoring approach can be applied to (component-based) systems whose behavior can be modeled as described in the sequel.
4.1 Semantics of Distributed CBSs

In the following, we present the architecture of our abstract semantic CBS which is used throughout this thesis.

Architecture of the system. The system under scrutiny \( \mathcal{M} \) is composed of components in a non-empty set \( \mathcal{B} = \{ B_1, \ldots, B_{|\mathcal{B}|} \} \) and schedulers in a non-empty set \( \mathcal{S} = \{ S_1, \ldots, S_{|\mathcal{S}|} \} \). Each component \( B_i \) is endowed with a set of actions \( \mathcal{A}_{B_i} \). Joint actions of component, aka multi-party interactions, involve the execution of actions on several components. An interaction is a non-empty subset of \( \bigcup_{i=1}^{|\mathcal{B}|} \mathcal{A}_{B_i} \) and we denote by \( \mathit{Int} \) the set of interactions in the system. At most one action of each component is involved in an interaction: \( \forall a \in \mathit{Int} \, |a \cap \mathcal{A}_{B_i}| \leq 1 \). In addition, to model concurrent behavior, each atomic component \( B_i \) has internal actions which we model as a unique action \( \beta_i \), such that each action of \( B_i \) is followed by the internal action \( \beta_i \). Schedulers coordinate the execution of interactions and ensure that each multi-party interaction is jointly executed (see Definition 4.3, p. 35).

Let us assume some auxiliary functions obtained from the architecture of the system.

- Function \( \mathit{involved} : \mathit{Int} \to 2^{\mathcal{B}} \setminus \{\emptyset\} \) indicates the components involved in an interaction. Moreover, we extend function \( \mathit{involved} \) to internal actions by setting \( \mathit{involved}(\beta_i) = i \), for any \( \beta_i \in \{\beta_1, \ldots, \beta_{|\mathcal{B}|}\} \). Interaction \( a \in \mathit{Int} \) is a joint action if and only if \( |\mathit{involved}(a)| \geq 2 \).

- Function \( \mathit{managed} : \mathit{Int} \to \mathcal{S} \) indicates the scheduler managing an interaction: for an interaction \( a \in \mathit{Int} \), \( \mathit{managed}(a) = S_j \) if \( a \) is managed by scheduler \( S_j \).

- Function \( \mathit{scope} : \mathcal{S} \to 2^{\mathcal{B}} \setminus \{\emptyset\} \) indicates the set of components in the scope of a scheduler such that \( \mathit{scope}(S_j) = \bigcup_{a \in \mathit{Int} \mid \mathit{managed}(a) = S_j} \mathit{involved}(a') \).

In the remainder, we describe the behavior of components, schedulers, and their composition.

Components. The behavior of an individual component is defined as follows.

**Definition 4.1** (Behavior of a component). The behavior of a component \( B \) is defined as an LTS \( (Q_B, \mathcal{A}_{B} \cup \{\beta_B\}, \rightarrow_B) \) such that:

- \( Q_B = Q_B^r \cup Q_B^b \) is the set of states, where \( Q_B^r \) (resp. \( Q_B^b \)) is the so-called set of \emph{ready} (resp. \emph{busy}) states,

- \( \mathcal{A}_{B} \) is the set of actions, and \( \beta_B \) is the internal action,

- \( \rightarrow_B \subseteq (Q_B^r \times \mathcal{A}_{B} \times Q_B^b) \cup (Q_B^b \times \{\beta_B\} \times Q_B^r) \) is the set of transitions.
4.1. Semantics of Distributed CBSS

Moreover, \( Q_B \) has a partition \( \{ Q^r_B, Q^b_B \} \).

Intuitively, the set of ready (resp. busy) states \( Q^r_B \) (resp. \( Q^b_B \)) is the set of states such that the component is ready (resp. not ready) to perform an action. Component \( B \) (i) has actions in set \( Act_B \) which are possibly synchronous with the actions of some of the other components, (ii) has an internal action \( \beta_B \) such that \( \beta_B \notin Act_B \) which models internal computations of component \( B \), and (iii) alternates moving from a ready state to a busy state and from a busy state to a ready state, that is component \( B \) does not have busy to busy or ready to ready move (as defined in the transition relation above).

**Example 4.2** (Component). Figure 4.1 depicts a component \( Tank \) whose behavior is defined by the LTS \( (Q^r \cup Q^b, Act \cup \{ \beta \}, \rightarrow) \) such that:

- \( Q^r = \{ d, f \} \) is the set of ready states and \( Q^b = \{ d^{-}, f^{-} \} \) is the set of busy states,
- \( Act = \{ Drain, Fill \} \) is the set of actions and \( \beta \) is the internal action,
- \( \rightarrow = \{(d, Drain, d^{-}), (f, Drain, f^{-}), (d^{-}, \beta, f), (f^{-}, \beta, d)\} \) is the set of transitions.

On the border, each \( \bullet \) represents an action and provides an interface for the component to synchronize with actions of other components in case of joint actions.

In the following, we assume that each component \( B_i \in B \) is defined by the LTS \( (Q_{B_i}, Act_{B_i} \cup \{ \beta_{B_i} \}, \rightarrow_{B_i}) \) where \( Q_{B_i} \) has a partition \( \{ Q^r_{B_i}, Q^b_{B_i} \} \) of ready and busy states; as per Definition 4.1 (p. 34).

**Schedulers.** The behavior of a scheduler is defined as follows.

**Definition 4.3** (Behavior of a scheduler). The behavior of a scheduler \( S \) is an LTS \( (Q_S, Act_S, \rightarrow_S) \) such that:

- \( Q_S \) is the set of states,
- \( Act_S = Act^\gamma_S \cup Act^\beta_S \) is the set of actions, where \( Act^\gamma_S = \{ a \in Int \mid managed(a) = S \} \) and \( Act^\beta_S = \{ \beta_i \mid B_i \in \text{scope}(S) \} \),
4. ABSOLUTE SEMANTIC MODEL OF DISTRIBUTED, MULTI-THREADS, AND SEQUENTIAL CBSS

\[ \text{Scheduler } S_i \]

\[ \text{Scheduler } S_j \]

\[ \text{Drain}_{1} \quad \beta_1 \quad \text{Fill}_{1} \quad \beta_2 \quad \text{Drain}_{2} \]

\[ \text{Tank } 1 \]

\[ \beta_1 \quad \text{Drain}_{1} \quad \beta_2 \quad \text{Fill}_{12} \quad \beta_2 \quad \text{Drain}_{2} \]

\[ \text{Tank } 2 \]

\[ \beta_3 \quad \text{Fill}_{3} \quad \beta_2 \quad \text{Drain}_{23} \]

\[ \text{Tank } 3 \]

\[ \text{Figure 4.2: Abstract representation of a CBS} \]

- \( \to_S \subseteq Q_S \times \mathbb{S} \times Q_S \) is the set of transitions.

\( \mathbb{S} \subseteq \text{Int} \) is the set of interactions managed by \( S \), and \( \mathbb{S} \subseteq \text{Int} \) is the set of internal actions of the components involved in an action managed by \( S \).

In the following, we assume that each scheduler \( S \in \mathbb{S} \) is defined by the LTS \((Q_S, \mathbb{S}, \to_S)\) where \( \mathbb{S} = \mathbb{S} \cup \mathbb{S} \), as per Definition 4.3. The coordination of interactions of the system i.e., the interactions in \( \text{Int} \), is distributed among schedulers. Actions of schedulers consist of interactions of the system. Nevertheless, each interaction of the system is associated to exactly one scheduler \((\forall a \in \text{Int}, \exists ! S \in \mathbb{S}. a \in \mathbb{S})\). Consequently, schedulers manage disjoint sets of interactions \((\forall S_i, S_j \in \mathbb{S}. S_i \neq S_j \implies \mathbb{S} \cap \mathbb{S} = \emptyset)\). Intuitively, when a scheduler executes an interaction, it triggers the execution of the associated actions on the involved components. Moreover, when a component executes an interaction, it triggers the execution of the corresponding action on the associated schedulers, that is, the component sends a message including its current state to the associated schedulers. Note, we assume that, by construction, schedulers are always ready to receive such a state update.

**Remark 4.4.** Since components send their updated states to the associated schedulers, we assume that the current state of a scheduler contains the last state of each component in its scope.

**Example 4.5 (Scheduler).** To illustrate the behavior of scheduler, we give a CBS, depicted in Figure 4.2 consisting of three components each of which is an instance of the component in Figure 4.1 (p. 35). The set of multi-party interactions is \( \text{Int} = \{\{\text{Drain}_1\}, \text{Fill}_{12}, \text{Drain}_{23}, \{\text{Fill}_3\}\} \) where \( \text{Fill}_{12} = \{\text{Fill}_1, \text{Fill}_2\} \) and \( \text{Drain}_{23} = \{\text{Drain}_2, \text{Drain}_3\} \) are joint actions. Two schedulers \( S_1 \) and \( S_2 \) coordinate the execution of
multi-party interactions such that managed(\{\text{Drain}_1\}) = \text{managed}(\text{Fill}_{12}) = S_1 and managed(\{\text{Fill}_3\}) = \text{managed}(\text{Drain}_{23}) = S_2. For \( j \in [1..2] \), scheduler \( S_j \) is defined as \((Q_{S_j}, \text{Act}_{S_j}, \rightarrow_{S_j})\) with:

- \( Q_{S_j} = \{l_0, l_1, l_2, l_3\} \),
- \( \text{Act}^\gamma_{S_1} = \{\text{Drain}_1, \text{Fill}_{12}\}, \text{Act}^\beta_{S_1} = \{\beta_1, \beta_2\} \),
- \( \text{Act}^\gamma_{S_2} = \{\text{Drain}_{23}, \text{Fill}_3\}, \text{Act}^\beta_{S_2} = \{\beta_2, \beta_3\} \),
- \( \rightarrow_{S_1} = \{(l_0, \beta_2, l_0), (l_1, \beta_2, l_1), (l_1, \beta_1, l_0), (l_2, \beta_2, l_1), (l_3, \beta_2, l_0), (l_2, \beta_1, l_3), (l_3, \text{Drain}_1, l_2), (l_0, \text{Drain}_1, l_1), (l_0, \text{Fill}_{12}, l_2)\} \),
- \( \rightarrow_{S_2} = \{(l_0, \beta_2, l_0), (l_1, \beta_2, l_1), (l_1, \beta_3, l_0), (l_2, \beta_2, l_1), (l_3, \beta_2, l_0), (l_2, \beta_3, l_3), (l_3, \text{Fill}_3, l_2), (l_0, \text{Fill}_3, l_1), (l_0, \text{Drain}_{23}, l_2)\} \).

**Definition 4.6 (Shared component).** The set of shared components is defined as

\[
B_s = \{ B \in B \mid |\{|S \in S \mid B \in \text{scope}(S)\}| \geq 2 \} .
\]

A shared component \( B \in B_s \) is a component in the scope of more than one scheduler, and thus, the execution of the actions of \( B \) are managed by more than one scheduler.

**Example 4.7 (Shared component).** In Figure 4.2 (p. 36), component \( \text{Tank}_2 \) is a shared component because interaction \( \text{Fill}_{12} \), which is a joint action of \( \text{Fill}_1 \) and \( \text{Fill}_2 \), is coordinated by scheduler \( S_1 \) and interaction \( \text{Drain}_{23} \), which is a joint action of \( \text{Drain}_2 \) and \( \text{Drain}_3 \), is coordinated by scheduler \( S_2 \).

The global execution of the system can be described as the parallel execution of interactions managed by the schedulers.

**Definition 4.8 (Global behavior).** The global behavior of system \( M \) is the LTS \((Q, \text{GAct}, \rightarrow)\) where:

- \( Q \subseteq \bigotimes_{i=1}^{\left|B\right|} Q_i \times \bigotimes_{j=1}^{\left|S\right|} Q_{S_j} \) is the set of states consisting of the states of schedulers and components,
- \( \text{GAct} \subseteq 2^\text{Int} \cup \bigcup_{i=1}^{\left|B\right|} \{\beta_i\} \setminus \emptyset \) is the set of possible global actions of the system consisting of either several interactions and/or several internal actions (several interactions can be executed concurrently by the system),
- \( \rightarrow \subseteq Q \times \text{GAct} \times Q \) is the transition relation defined as the smallest set abiding to the following rule.

A transition is a move from state \((q_1, \ldots, q_{\left|B\right|}, q_{s_1}, \ldots, q_{s_{\left|S\right|}})\) to state \((q'_1, \ldots, q'_{\left|B\right|}, q'_{s_1}, \ldots, q'_{s_{\left|S\right|}})\) on global actions in set \( \alpha \cup \beta \) by \( \alpha \subseteq \text{Int} \) and \( \beta \subseteq \bigcup_{i=1}^{\left|B\right|} \{\beta_i\} \), noted \((q_1, \ldots, q_{\left|B\right|}, q_{s_1}, \ldots, q_{s_{\left|S\right|}}) \xrightarrow{\alpha \cup \beta} (q'_1, \ldots, q'_{\left|B\right|}, q'_{s_1}, \ldots, q'_{s_{\left|S\right|}})\), whenever the following conditions hold:
\[ C_1: \forall i \in [1, |B|]. |(\alpha \cap Act_i) \cup (\{\beta_i\} \cap \beta)| \leq 1, \]
\[ C_2: \forall j \in [1, |S|]. |(\alpha \cup \beta) \cap Act_S_j| \leq 1, \]
\[ C_3: \forall a \in \alpha . (\exists S_j \in S . managed(a) = S_j) \Rightarrow (q_{s_j}^{a} \xrightarrow{a} S_j q_{s_j}^{a} \wedge \forall B_i \in involved(a) . q_i \xrightarrow{a \cap Act_i} B_i q_i^{a}), \]
\[ C_4: \forall \beta_i \in \beta . q_i \xrightarrow{\beta_i} B_i q_i^{\beta_i} \wedge \forall S_j \in S . B_i \in scope(S_j) . q_{s_j}^{\beta_i} \xrightarrow{S_j} S_j q_{s_j}^{\beta_i}, \]
\[ C_5: \forall B_i \in B \setminus involved(\alpha \cup \beta) . q_i = q_i^{e}, \]
\[ C_6: \forall S_j \in S \setminus managed(\alpha) . q_{s_j} = q_{s_j}^{e}. \]

where functions involved and managed are extended to sets of interactions and internal actions in the usual way.

The above rule allows the components of the system to execute independently according to the decisions of the schedulers. It can intuitively be understood as follows:

- **Condition** \( C_1 \) states that a component can perform at most one execution step at a time. The executed global actions \((\alpha \cup \beta)\) contains at most one interaction involving each component of the system.

- **Condition** \( C_2 \) states that a scheduler can perform at most one execution step at a time. The executed global actions \((\alpha \cup \beta)\) contains at most one action concerning each scheduler of the system.

- **Condition** \( C_3 \) states that whenever an interaction \( a \) managed by scheduler \( S_j \) is executed, \( S_j \) and all components involved in this multi-party interaction must be ready to execute it.

- **Condition** \( C_4 \) states that internal actions are executed whenever the corresponding components are ready to execute them. Moreover, schedulers are aware of internal actions of components in their scope. Note that, the awareness of internal actions of a component results in transferring the updated state of the component to the schedulers.

- **Conditions** \( C_5 \) and \( C_6 \) state that the components and the schedulers not involved in an interaction remain in the same state.

An example illustrating the global behavior of the system depicted in Figure 4.2 (p. 36) is provided later and described in terms of execution traces (see Example 4.10, p. 39).

**Trace.** At runtime, the execution of a CBS produces a trace. Intuitively, a trace is the sequence of traversed states of the system, from some initial state and following the transition relation of the LTS of the system. For the sake of simplicity and for our monitoring purposes, the states of schedulers are irrelevant in the trace (since the desired property refers to the global states of the components in \( B \)), and thus we restrict the states of the system to states of the components in \( B \).
To define the trace, we consider system $M$ consisting of a set $B$ of components (as per Definition 4.1, p. 34) and a set $S$ of schedulers (as per Definition 4.3, p. 35) with the global behavior as per Definition 4.8 (p. 37).

**Definition 4.9** (Trace of a CBS). A trace of system $M$ is a continuously-growing sequence $(q_1^0, \ldots, q_{|B|}^0) \cdot (\alpha^0 \cup \beta^0) \cdot (q_1^1, \ldots, q_{|B|}^1) \cdot (q_1^2, \ldots, q_{|B|}^2) \cdot \ldots$, such that $(q_1^0, \ldots, q_{|B|}^0) = \text{init}$ is the initial state of the system where $q_i^0$, $\ldots$, $q_{|B|}^0$ are the initial states of components $B_1, \ldots, B_{|B|}$ respectively and $\forall i \in [0 \ldots k - 1], (q_1^i, \ldots, q_{|B|}^i) \xrightarrow{\alpha^i \cup \beta^i} (q_1^{i+1}, \ldots, q_{|B|}^{i+1})$, where $\rightarrow$ is the transition relation of the global behavior of the system and the states of schedulers are discarded.

The set of traces of system $M$ is denoted by $\text{Tr}(M)$. Since trace $t \in \text{Tr}(M)$ has partial states where at least one component is busy with its internal computation, trace $t$ is referred to as a *partial trace*.

**Example 4.10** (Trace). Two possible partial traces of the system in Example 4.5 (p. 36) (depicted in Figure 4.2, p. 36) are:

- $t_1 = (d_1, d_2, d_3) \cdot \{\text{Fill}_{12}\} \cdot (\perp, \perp, d_3) \cdot \{\beta_1\} \cdot (f_1, \perp, d_3) \cdot \{\{\text{Drain}_1\}, \{\text{Fill}_3\}\} \cdot (\perp, \perp, \perp) \cdot \{\beta_2\} \cdot (\perp, f_2, \perp)$,

- $t_2 = (d_1, d_2, d_3) \cdot \{\text{Fill}_{12}, \{\text{Fill}_3\}\} \cdot (\perp, \perp, \perp) \cdot \{\beta_3\} \cdot (\perp, \perp, f_3) \cdot \{\beta_2\} \cdot (\perp, f_2, f_3) \cdot \{\{\text{Drain}_{23}\}, \beta_1\} \cdot (f_1, \perp, \perp)$.

Traces $t_1$ and $t_2$ are obtained following the global behavior of the system (Definition 4.8, p. 37).

- In trace $t_1$, the execution of interaction $\text{Fill}_{12}$ represents the simultaneous execution of (i) action $\text{Fill}_{12}$ in scheduler $S_1$, (ii) action $\text{Fill}_1$ in component $\text{Tank}_1$, and (iii) action $\text{Fill}_2$ in component $\text{Tank}_2$. After interaction $\text{Fill}_{12}$, component $\text{Tank}_1$ and $\text{Tank}_2$ move to their busy state whereas the state of component $\text{Tank}_3$ remains unchanged. Moreover, the execution of internal action $\beta_2$ in trace $t_1$ represents the simultaneous execution of (i) internal action $\beta_2$ in component $\text{Tank}_2$, (ii) action $\beta_2$ in scheduler $S_1$ and (iii) action $\beta_2$ in scheduler $S_2$. After the internal action $\beta_2$, component $\text{Tank}_2$ goes to ready state $f_2$.

- In trace $t_2$, the execution of global action $\{\text{Fill}_{12}, \{\text{Fill}_3\}\}$ represents the simultaneous execution of two interactions $\text{Fill}_{12}$ and $\{\text{Fill}_3\}$, that is the simultaneous executions of (i) action $\text{Fill}_{12}$ in scheduler $S_1$, (ii) action $\text{Fill}_3$ in scheduler $S_2$, (iii) action $\text{Fill}_1$ in component $\text{Tank}_1$, (iv) action $\text{Fill}_2$ in component $\text{Tank}_2$, and (v) action $\text{Fill}_3$ in component $\text{Tank}_3$. Trace $t_2$ ends up with the simultaneous execution of interaction $\text{Drain}_{23}$ and the internal action of component $\text{Tank}_1$.

---

1To facilitate the description of the trace, we represent each busy state as $\perp$. 

---

4.1. SEMANTICS OF DISTRIBUTED CBS
Remark 4.11. The operational description of a CBS is usually more detailed. For instance, the execution of conflicting interactions in schedulers needs first to be authorized by a conflict-resolution module which guarantees that two conflicting interactions are not executed at the same time. Moreover, schedulers follow the (possible) priority rules among the interactions, that is, in the case of two or more enabled interactions (interactions which are ready to be executed by schedulers), those with higher priority are allowed to be executed. Since we only deal with the execution traces of a distributed system, we assume that the obtained traces are correct with respect to the conflicts and priorities. Therefore, defining the other modules is out of the scope of this work.

Definition 4.12 (Monitoring hypothesis). The behavior of the CBS under scrutiny can be modeled as an LTS as per Definition 4.8 (p. 37).

4.2 Semantics of Multi-Threaded CBSs

The abstract system $M$, defined in the previous section, consists of a set $B$ of components (as per Definition 4.1, p. 34) and a set $S$ of schedulers (as per Definition 4.3, p. 35). One can obtain a multi-threaded CBS by limiting the number of schedulers to 1 (i.e., $|S| = 1$) and taking $Act_S = Int$. The multi-threaded system is denoted by $M^\perp$. In this setting, all the actions are managed by one scheduler $S$, and according to Definition 4.6 (p. 37) none of the components is shared, that is $B_s = \emptyset$.

Therefore, according to the semantics of the abstract CBS (see Definition 4.8 (p. 37), condition $C_2$), the global execution of $M^\perp$ is restricted to a single execution of either a multi-party interaction or an internal action of a component.

In the multi-threaded system $M^\perp$, since all the interactions are managed by one scheduler, the trace is defined based on the sequence of actions locally executed by the scheduler.

Definition 4.13 (Trace of multi-threaded CBS). A trace of system $M^\perp$ is a continuously-growing sequence

$$(q_0^0, \ldots, q_0^{|B|}) \cdot a^0 \cdot (q_1^0, \ldots, q_1^{|B|}) \cdot \ldots \cdot (q_k^0, \ldots, q_k^{|B|}) \cdot \ldots,$$

such that $(q_0^0, \ldots, q_0^{|B|}) = init$ and $\forall i \in [0 \ldots k - 1], (q_i^0, \ldots, q_i^{|B|}) \xrightarrow{a^i} (q_i^{i+1}, \ldots, q_i^{i+1}) \land a^i \in Act_S$, where $\rightarrow$ is the transition relation of the global behavior of the system as per Definition 4.8 (p. 37) and the state of scheduler $S$ is discarded.

Example 4.14 (Trace of multi-threaded CBS). One possible partial traces of the composite component depicted in Figure 4.2 (p. 36) in multi-threaded setting is:

$$t = (d_1, d_2, d_3) \cdot \{Fill_{12}\} \cdot (\bot, \bot, d_3) \cdot \{\beta_1\} \cdot (f_1, \bot, d_3) \cdot \{\{Drain\}_1\} \cdot (\bot, \bot, d_3) \cdot \{\{Fill_3\}\} \cdot (\bot, \bot, \bot) \cdot \{\beta_2\} \cdot (f_2, \bot, \bot).$$
Although in the partial state \((f_1, \bot, d_3)\) two interactions \(\text{Fill}_3\) and \(\text{Drain}_1\) are enabled, the central scheduler can only execute one at each execution step.

### 4.3 Semantics of Sequential CBSs

The general semantic system and the multi-threaded system are partial-state semantics of CBSs in which concurrent executions is possible in the sense that schedulers are able to execute actions involving ready components. To model sequential CBSs, we refer to the global-states semantics of atomic component in which (i) transitions are atomic, (ii) interactions are executed sequentially by one scheduler, and (iii) the execution of an interaction is not possible when some component is performing a computation. Similar to the multi-threaded setting, in the sequential setting one scheduler is in charge of the execution of the multi-party interactions. The sequential system is denoted by \(M^s\).

**Definition 4.15** (Behavior of a component with global-state semantics). For component \(B\) in partial-state semantics with behavior \((Q^r_B \cup Q^b_B, \text{Act}_B \cup \{\beta_B\}, \rightarrow_B)\) as per Definition 4.1 (p. 34), the associated global-state semantics version of \(B\), denoted by \(B\), is defined as an LTS \((Q, \text{Act}, \rightarrow)\) such that the set of transitions \(\rightarrow \subseteq (Q^r_B \times \text{Act}_B \times Q^r_B)\) is a set of moves from a ready state to another ready state.

**Example 4.16** (Atomic component with global state). Figure 4.3 depicts component \textit{Tank}, the global-state semantics version of component \textit{Tank} depicted in Example 4.1 (p. 35). The behavior of component \textit{Tank} is LTS \((Q, \text{Act}, \rightarrow)\), such that:

- \(Q = \{d, f\}\) is the set of states,
- \(\text{Act} = \{\text{Drain}, \text{Fill}\}\) is the set of actions,
- \(\rightarrow = \{(f, \text{Drain}, d), (d, \text{Fill}, f)\}\) is the set of transitions.

**Definition 4.17** (Global behavior of sequential CBS). The behavior of sequential system \(M^s\) is similarly defined based on the behavior of our abstract system \(M\) as per Definition 4.8 (p. 37), where \(|S| = 1, Q^1_S\) and \(\beta_i\) for \(i \in [1, |B|]\) and \(\text{Act}^\beta_S\) are empty sets.
Accordingly, the trace of a sequential system $M_s$ is a continuously-growing sequence $(q_0^1, \ldots, q_0^{|B|}) \cdot a^0 \cdot (q_1^1, \ldots, q_1^{|B|}) \cdot a^1 \cdot \cdots (q^k_1, \ldots, q^k_1) \cdots$ such that $(q_0^0, \ldots, q_0^{|B|}) = \text{init}$, $q_i^j+1 \in Q_i$ and $a^j \in \text{Int}$ for all $i \in [1 \ldots |B|], j \in [0 \ldots k]$. Given a trace of a sequential system $t = (q_0^1, \ldots, q_0^{|B|}) \cdot a^0 \cdot (q_1^1, \ldots, q_1^{|B|}) \cdot a^1 \cdot (q^k_1, \ldots, q^k_1)$, the sequence of interactions is defined as $\text{interactions}(t) = a^0 \cdot a^1 \cdots a^{k-1}$.

In the sequential system, an execution trace is defined over the global states of the system whose components states are ready, therefore the trace is referred to as global trace.

**Example 4.18 (Sequential CBS).** Figure 4.4 depicts the associated sequential composite component of system Tank (see Figure 4.2, p. 36) consists of a set of components $B = \{\text{Tank}_1, \text{Tank}_2, \text{Tank}_3\}$ where each $\text{Tank}_i$ is identical to the component in Figure 4.3 (p. 41). One of the possible global traces of such a system is: $(d_1, d_2, d_3) \cdot \{\text{Fill}_{12}\} \cdot (f_1, f_2, d_3) \cdot \{\{\text{Drain}_1\}\} \cdot (d_1, f_2, f_3) \cdot \{\{\text{Fill}_3\}\}$, such that from the initial state $(d_1, d_2, d_3)$, where tanks are at state $d$, interaction $\text{Fill}_{12}$ is fired and $\text{Tank}_1$ and $\text{Tank}_2$ move to state $f$. Then, $\text{Tank}_1$ moves to state $d$ by executing interaction $\text{Drain}_1$, and finally by executing interaction $\text{Fill}_3$, $\text{Tank}_3$ moves to state $f$.

**Summary:** In this chapter, we define the execution trace of the abstract semantic model in different settings. The problem that arises is that, in the absence of a global clock, the actual execution trace of such a model is not observable. In the next chapter, we investigate the observability issue of the execution trace at runtime.
Observing a CBSs at Runtime

Chapter abstract
In the previous chapter we defined the execution trace of the abstract semantic model of CBS in different settings. However, in the distributed and multi-threaded settings, generally the execution trace is either not fully-observable or not informative enough to be used for online monitoring. In this chapter, we define the observable execution trace of such systems at runtime. The observable trace of the system exists in the set of schedulers. Moreover, we illustrate the problem of runtime monitoring based on the observable traces.

In the following, we consider a system $M$ consisting of a set $B$ of components as per Definition 4.1 (p. 34), and a set $S$ of schedulers as per Definition 4.3 (p. 35), with the global behavior as per Definition 4.8 (p. 37). At runtime, the execution of such a system produces a partial trace as per definition Definition 4.9 (p. 39).

5.1 Observable trace
Although the partial trace of system $M$ exists, it is not observable because it would require a perfect observer having simultaneous access to the states of the components and knowing the order of the executed interactions in several schedulers. Introducing such an observer (able to observe global states) in the system would require all components to synchronize, and would defeat the purpose of building a distributed system. Instead of introducing such an observer, first, we shall instrument the system (see Chapter 6, p. 49) to obtain the locally observable partial-trace through the schedulers. Then, we reconstruct the global trace using the local partial-traces of the system (see Chapter 7, p. 61).
In the following we consider a partial trace \( t = (q_1^0, \ldots, q_{|B|}^0) \cdot (\alpha^0 \cup \beta^0) \cdot (q_1^1, \ldots, q_{|B|}^1) \cdots \), as per Definition 4.9 (p. 39). Each scheduler \( S_j \in S \) for \( j \in [1..|S|] \), observes a local partial-trace \( s_j(t) \) which consists in the sequence of the states of the components in its scope and actions it manages.

**Definition 5.1** (Locally observed partial-trace). The local partial-trace \( s_j(t) \) observed by scheduler \( S_j \) is inductively defined on the partial trace \( t \) as follows:

- \( s_j((q_1^0, \ldots, q_{|B|}^0)) = (q_1^0, \ldots, q_{|B|}^0) \), and

- \( s_j(t \cdot (\alpha \cup \beta) \cdot q) = \begin{cases} t & \text{if } S_j \not\in \text{managed}(\alpha) \land \text{involved}(\beta) \cap \text{scope}(S_j) = \emptyset \\ t \cdot \theta \cdot q' & \text{otherwise} \end{cases} \)

where

- \( q = (q_1, \ldots, q_{|B|}) \),

- \( \theta = (\alpha \cap \{a \in \text{Int} | \text{managed}(a) = S_j\}) \cup (\beta \cap \{\beta_i \mid B_i \in \text{scope}(S_j)\}) \),

- \( q' = (q_1, \ldots, q_{|B|}) \) with

\[
q' = \begin{cases} \text{last}(s_j(t))[i] & \text{if } B_i \in \overline{\text{involved}(\theta)} \cap \text{scope}(S_j), \\ q_i & \text{if } B_i \in \text{involved}(\theta) \cap \text{scope}(S_j), \\ ? & \text{otherwise (} B_i \not\in \text{scope}(S_j) \text{).} \end{cases}
\]

We assume that the initial state of the system, that is \( \text{init} = (q_1^0, \ldots, q_{|B|}^0) \), is observable by all schedulers. An interaction \( a \in \text{Int} \) is observable by scheduler \( S_j \) if \( S_j \) manages the interaction (i.e., \( S_j \in \text{managed}(a) \)). Moreover, an internal action \( \beta_i \), with \( i \in [1..|B|] \), is observable by scheduler \( S_j \) if \( B_i \) is in the scope of \( S_j \) (i.e., \( B_i \in \text{scope}(S_j) \)). The state observed after an observable interaction or internal action consists of the states of components in the scope of \( S_j \), that is a state \( (q_1, \ldots, q_{|B|}) \) where \( q_i \) is the new state of component \( B_i \) if \( B_i \in \text{scope}(S_j) \) and ? otherwise.

**Example 5.2** (Locally observed partial-trace). The associated locally observed partial-trace of \( t_1 \) and \( t_2 \) of Example 4.10 (p. 39) are:

- \( s_1(t_1) = (d_1, d_2, d_3) \cdot \{\text{Fill12}\} \cdot (\perp, \perp, ?) \cdot \{\beta_1\} \cdot (f_1, \perp, ?) \cdot \{\text{Drain}_1\} \cdot (\perp, \perp, ?) \cdot \{\beta_2\} \cdot (f_2, \perp, ?) \),

- \( s_2(t_1) = (d_1, d_2, d_3) \cdot \{\text{Fill3}\} \cdot (\perp, d_2, \perp) \cdot \{\beta_2\} \cdot (f_2, \perp, ?) \),

- \( s_1(t_2) = (d_1, d_2, d_3) \cdot \{\text{Fill12}\} \cdot (\perp, \perp, ?) \cdot \{\beta_2\} \cdot (f_1, f_2, ?) \),

- \( s_2(t_2) = (d_1, d_2, d_3) \cdot \{\text{Fill3}\} \cdot (\perp, d_2, \perp) \cdot \{\beta_3\} \cdot (f_1, f_2, ?) \cdot \{\text{Drain}_{23}\} \cdot (\perp, \perp, ?) \).

For instance, the local partial-trace \( s_1(t_2) \) shows that scheduler \( S_1 \) is aware of the execution of interaction \( \text{Fill12} \) but it is not aware of the occurrence of internal action \( \beta_3 \) because component \( \text{Tank}_3 \) is not in the
5.2. PROBLEM STATEMENT: MONITORING THE TRACE OF A CBS

scope of scheduler $S_1$ and consequently the state of component $Tank_3$ in the local partial-trace of scheduler $S_1$ is denoted by $\emptyset$ (except for the initial state). Moreover, scheduler $S_1$ is aware of the occurrences of internal actions $\beta_2$ and $\beta_1$ but it is not aware of action $Drain_{23}$ because scheduler $S_1$ does not manage action $Drain_{23}$.

5.1.1 Observable Trace of a Multi-Threaded CBS

In multi-threaded system $M_\perp$, since all the executions are managed by one scheduler $S$, the actual execution partial trace $t$ is the local partial-trace of the scheduler, that is $s(t) = t$. Hence, the partial trace of system $M_\perp$ is observable by scheduler $S$.

5.1.2 Observable Trace of a Sequential CBS

Similar to the multi-threaded setting, the execution trace of the sequential system $M^s$ is observable by the central scheduler. Moreover, the global state of $M^s$ is always defined, so that the execution trace $t$ is a global trace. Therefore, one can directly use the sequence of the global states of the global trace $t$ for runtime monitoring [39]. The sequence of these states represents the evolution of the system at runtime. Runtime monitor consumes these global states as input and outputs the corresponding verdicts.

5.2 Problem Statement: Monitoring the Trace of a CBS

Our aim is to runtime verify a CBS against properties referring to its global states. Although in our abstract model defined in Section 4.1 (p. 34) each scheduler has its local clock, we use neither a global clock nor a shared memory. On the one hand, this makes the execution of the system more dynamic and parallel because we do not reconstruct global states with ready state of components at runtime. Thus, we avoid synchronization to take global snapshots, which would go against the parallel execution of the verified system. On the other hand, it complicates the monitoring problem because no component of the system can be aware of the global trace. Since the execution of interactions in the implemented distributed system is based on sending/receiving messages, communications are asynchronous and delays in the reception of messages are inevitable. Moreover, the absence of ordering between the execution of the interactions in several schedulers causes the main problem in this case: the actual global trace of the system is not observable.

Example 5.3 (Monitoring problem in the distributed setting). Given the associated local partial-traces of the partial trace $t_1$ presented in Example 5.2 (p. 45), the actual ordering between the execution of interactions $Fill_{12}$ conducted by scheduler $S_1$, and $\{Fill_3\}$ conducted by scheduler $S_2$, is not determined.
Although in the multi-threaded setting, components execute with a centralized scheduler, where i) there is a global clock (which is the local clock of the scheduler), ii) communication is instantaneous and atomic, and iii) the ordering between the executions is known, the global state of the system (where all components are ready to perform an interaction) may never exist at runtime (see the partial trace in Example 4.14, p. 40).

Example 5.4 (Monitoring problem in the multi-threaded setting). To illustrate the monitoring problem in the multi-threaded setting, let us consider the partial trace $t$ presented in Example 4.14 (p. 40), that is $t = (d_1, d_2, d_3) \cdot \{\text{Fill}_{12}\} \cdot (\perp, \perp, d_3) \cdot \{\beta_1\} \cdot (f_1, \perp, d_3) \cdot \{\text{Drain}_{1}\} \cdot (\perp, \perp, \perp) \cdot \{\beta_2\} \cdot (\perp, f_2, \perp)$. The centralized scheduler is locally aware of such partial trace. The sequence of observed states is $(d_1, d_2, d_3) \cdot (\perp, \perp, d_3) \cdot (f_1, \perp, d_3) \cdot (\perp, \perp, d_3) \cdot (\perp, \perp, \perp) \cdot (\perp, f_2, \perp)$. Based on our assumption, the desired property is defined over the global state of the system, and therefore, such a sequence of states is not informative enough to be fed to the runtime monitor.

Our goal is to allow for the verification of our system $M$ (as per Definition 4.8, p. 37) and the multi-threaded version of $M$, that is $M^\perp$ (see Section 4.2, p. 40), by formally instrumenting them to observe their global behavior while preserving their performance and initial behavior.

First, in Chapter 6 (p. 49) we present an instrumentation for system $M$ to obtain the observable partial trace(s) and the associated events. Second, in Chapter 7, p. 61 according to the events of the system, we reconstruct the global trace(s) of the system. Finally, the global trace(s) is(are) evaluated at runtime, so that the reconstruction and verification are conducted simultaneously.

For a multi-threaded system $M^\perp$, we propose an online monitoring algorithm accumulating the partial states traversed by the system at runtime to reconstruct the associated global trace with global states. Such a reconstructed global trace has the same execution ordering of the interactions and can be interpreted as the equivalent global trace of the corresponding sequential system $M^s$ (see Section 4.3, p. 41).

For a distributed system $M$ with $|S| > 1$, we deal with a set of locally observed partial-trace. We propose to build the computation lattice in which all the possible ordering of the executions are taken into account. The computation lattice consists of a set of partially connected nodes. Each node represents a global states of the system. A sequence of connected nodes in the lattice (a path of the lattice) represents a compatible global trace of the system. The computation lattice is expanded as system $M$ evolves at runtime. Of course keeping all the nodes of the computation lattice from the initial state until the end of system’s run, would impose a huge overhead to the system. Instead, we propose an online monitoring algorithm to evaluate the lattice nodes as soon as they are generated. Moreover, we only keep those nodes of the lattice that are needed for the possible future extension of the lattice, which results a relatively low overhead on the system’s computation process.
Summary: In this chapter, we defined what is observable in the abstract semantic model. In the next chapter, we instrument the model so that executing the instrumented model produces the events associated to the observable trace.
Chapter abstract

In this chapter, we propose an instrumentation method for the abstract semantic model of CBS. The execution of the instrumented system generates the events associated to the partial trace of the model. Events are sent to a central observer. We prove that the instrumentation is correct in the sense that the behavior of the original system is preserved.

We define an instrumentation on the abstract CBS defined in Section 4.1 (p. 34), so that it can be applied on both multi-threaded and distributed systems. In our abstract model, since schedulers do not interact directly together by sending/receiving messages, the execution of an interaction by one scheduler seems to be concurrent with the execution of all interactions by other schedulers. Nevertheless, if scheduler $S_j$ manages interaction $a$ and scheduler $S_k$ manages interaction $b$ such that a shared component $B_i \in B_s$ is involved in $a$ and $b$, i.e., $B_i \in \text{involved}(a) \cap \text{involved}(b)$, as a matter of fact, the execution of interactions $a$ and $b$ are causally related. In other words, there exists only one possible ordering of $a$ and $b$ and they could not have been executed concurrently. Ignoring the actual ordering of $a$ and $b$ would result in retrieving inconsistent global states (i.e., states that do not belong to the original system). Our instrumentation must detect and leverage such a causality among the executions. To this end, we employ vector clocks to define the global order of executions. Moreover, to each scheduler and each shared component, we add a corresponding component, its controller. The controller of a scheduler extracts and notifies the events of the scheduler. Any state update of the system is known as an event. An event is either triggering an interaction which changes the state of the involving components, or occurrence of the busy action of a component. The events of each scheduler represent the observable local partial-trace of the scheduler. The controllers of schedulers and
the controllers of shared components interact whenever the scheduler and the shared components interact in order to interchange vector clocks. Generated events are sent to a central online monitoring unit, that is the observe. Whenever a scheduler executes an interaction, the associated controller of the scheduler attaches the correct vector clock to this execution and notifies the observer about this event. The observer runs in parallel with the system, collects the local events of the schedulers, and reconstructs the set of possible global traces compatible with the locally observed partial-traces (Figure 6.1). The observer is always ready to receive the events. Moreover, The observer does not force the system to send data and thus does not modify the execution of the monitored system (it is a passive observer). For monitoring purposes, the observer should be able to order the execution of interactions with respect to the received local events appropriately. We shall prove that such observer does not violate the semantics nor the behavior of the distributed system, that is, the observed system is observationally equivalent (see Section 3, p. 25) to the initial system (see Property 6.14, p. 59).

In the following, we provide details of our instrumentation for an abstract semantic CBS to let schedulers send their local events to an observer through their controllers.

### 6.1 Composing Schedulers and Shared Components with Controllers

We consider a CBS consisting of a set of components \( B = \{B_1, \ldots, B_{|B|}\} \) (as per Definition 4.1, p. 34) and a set of schedulers \( S = \{S_1, \ldots, S_{|S|}\} \) where scheduler \( S_j = (Q_{S_j}, Act_{S_j}, \rightarrow_{S_j}) \) manages the interactions in \( Act^\beta_{S_j} \) and is notified by internal actions in \( Act^\alpha_{S_j} \), for \( S_j \) (as per Definition 4.3, p. 35). We attach to \( S_j \) a local controller \( C^j \) in charge of

- sending the state updates (events) of \( S_j \) to the central observer, that corresponds to the local partial-trace of \( S_j \).
computing the vector clock using shared components.

Moreover, for each shared component \( B_i \in S \), we attach a local controller \( C^0_{B_i} \) to communicate with the controllers of the schedulers that have \( B_i \) in their scope.

In the following, we define the controllers (instrumentation code) and the composition \( \otimes \) as instrumentation process.

### 6.1.1 Controllers of Schedulers

Controller \( C_{S_j} \) is in charge of computing the correct vector clock of scheduler \( S_j \) (Definition 6.1). It does so through the data exchange with the controllers of shared components, i.e., the controllers in the set \( \{ C^0_{B_i} \mid S_j \in S \land B_i \in \text{scope}(S_j) \} \), which are later defined in Definition 6.4 (p. 54).

**Definition 6.1 (Controller of scheduler).** Controller \( C_{S_j} \) is an LTS \( (Q_{C_{S_j}}, R_{C_{S_j}}, \rightarrow_{C_{S_j}}) \) such that:

- \( Q_{C_{S_j}} = 2^{[1..|B|]} \times VC \) is the set of states where \( 2^{[1..|B|]} \) is the set of subsets of component indexes and \( VC \) is the set of vector clocks;
- \( R_{C_{S_j}} = \{ (\beta_i, \emptyset) \mid B_i \in \text{scope}(S_j) \} \cup \{ (-, \{ \text{rev}_i | vc \}) \mid B_i \in \text{scope}(S_j) \cap B_s \land \text{vc} \in VC \} \cup \{ (a [\text{vc}], \text{send}) \mid a \in \text{Act}_{\gamma S_j} \land \text{send} \subseteq \{ \text{send}_i | \text{vc} \mid B_i \in \text{scope}(S_j) \cap B_s \land \text{vc} \in VC \} \} \) is the set of actions;
- \( \rightarrow_{C_{S_j}} \subseteq Q_{C_{S_j}} \times R_{C_{S_j}} \times Q_{C_{S_j}} \) is the transition relation defined as:

\[
\begin{align*}
\left\{ (I, vc) \xrightarrow{(a [\text{vc}], \{ \text{send}_i | \text{vc} \} \mid i \in \text{involved}(a) \land B_i \in B_s)} C_{S_j} \mid a \in \text{Act}_{\gamma S_j} \land \text{vc'} = \text{inc}(vc, j) \right\} \\
\cup \left\{ (I, vc) \xrightarrow{(-, \{ \text{rev}_i | \text{vc} \})} C_{S_j} \mid \beta_i \in \text{Act}_{\beta S_j} \land B_i \in B_s \right\} \\
\cup \left\{ (I, vc) \xrightarrow{(a [\text{vc}], \{ \text{send}_i | \text{vc} \} \mid i \in \text{involved}(a) \land B_i \in B_s)} C_{S_j} \mid a \in \text{Act}_{\gamma S_j} \land \text{vc'} = \text{inc}(vc, j) \right\}
\end{align*}
\]

where \( \text{inc}(vc, j) \) increments the \( j \)th element of vector clock \( vc \).

When the controller \( C_{S_j} \) is in state \( (I, vc) \), it means that (i) \( I \) is the set of busy components in the scope
of scheduler $S_j$, (ii) the execution of their latest action has been managed by scheduler $S_j$, and (iii) $vc$ is the current value of the vector clock of scheduler $S_j$.

An action in $R_{C_j}$ is a pair $(x, y)$ where $x$ is associated to the actions which send information from the controller to the observer and $y$ is associated to the actions in which the controller sends/receives information to/from the controllers of shared components, such that

$$x \in \{ a[vc] \mid a \in Act^7_{S_j} \land vc \in VC \} \cup \{ \tilde{\beta}_i \mid i \in scope(j) \} \cup \{-\}, \text{ and}$$

$$y \subseteq \{ send_i[vc], rcv_i[vc] \mid B_i \in scope(S_j) \cap B_s \land vc \in VC \}$$
can be intuitively understood as follows,

- action $a[vc]$ consists in notifying the observer about the execution of interaction $a$ with vector clock $vc$ attached.
- action $\tilde{\beta}_i$ consists in notifying the observer about the internal action of component $B_i$. The last state of component $B_i$ is also transmitted to the observer.
- action $-$ is used in the case when the controller does not interact with the observer,
- action $send_i[vc]$ consists in sending the value of the vector clock $vc$ of the scheduler to the shared component $B_i$,
- action $rcv_i[vc]$ consists in receiving the value of the vector clock $vc$ stored in the shared component $B_i$.

The set of transitions is obtained as the union of three sets which can be intuitively understood as follows:

- For each interaction $a \in Act^7_{S_j}$ managed by scheduler $S_j$, we include a transition with action

$$\left( a[vc'], \{ send_i[vc'] \mid B_i \in involved(a) \cap B_s \} \right),$$

where $a[vc']$ is a notification to the observer about the execution of interaction $a$ along with the value of vector clock $vc'$, and actions in set $\{ send_i[vc'] \mid B_i \in involved(a) \cap B_s \}$ send the value of the vector clock $vc'$ to the shared components involved in interaction $a$. Moreover, the set of indexes of the components involved in interaction $a$ (i.e., in involved($a$)) is added to the set of busy components; and the current value of the vector clock is incremented.

- For each action associated to the notification of the internal action of component $B_i$ (that is, $\beta_i$), we include a transition labeled with action $(\tilde{\beta}_i, \emptyset)$ in the controller to send the updated state to the observer. Moreover, this transition removes index $i$ from the set of busy components.

- For each action associated to the notification of internal action of a shared component $B_i \in B_s$, we include a transition labeled with action $(-, \{ rcv_i[vc'] \})$ in the controller to receive the value of
Note that to each shared component $B_i \in B_s$, we also attach a controller in order to exchange the vector clock among schedulers in the set \( \{ S_j \in S \mid B_i \in \text{scope}(S_j) \} \); see Definition 6.4 (p. 54).

Below, we define how a scheduler is composed with its controller. Intuitively, the controller of a scheduler ensures sending/receiving information among the scheduler, the associated vector clock stored in the scheduler and the received vector clock from the shared component.

### Definition 6.2 (Semantics of $S_j \otimes_s C^*_j$)

The composition of scheduler $S_j$ and controller $C^*_j$, denoted by $S_j \otimes_s C^*_j$, is the LTS $(Q_{S_j} \times Q_{C^*_j}, Act_{S_j} \times R_{C^*_j}, \rightarrow_{scj})$ where the transition relation $\rightarrow_{scj} \subseteq (Q_{S_j} \times Q_{C^*_j}) \times (Act_{S_j} \times R_{C^*_j}) \times (Q_{S_j} \times Q_{C^*_j})$ is defined by the semantic rules in Figure 6.2.

The semantic rules in Figure 6.2 can be intuitively be understood as follows:

- **Rule CONT-SCH1.** When the scheduler executes an interaction $a \in Int$, the controller (i) updates the vector clock by increasing its local clock, (ii) updates the set of busy components, (iii) notifies the observer of the execution of $a$ along with the associated vector clock $vc'$, and (iv) sends vector clock $vc'$ to the shared components involved in $a$.

- **Rule CONT-SCH2.** When the scheduler is notified of an internal action of component $B_i$ where $i \in I$ (that is, the scheduler has managed the latest action of component $B_i$) through action $\beta_i$, the controller transfers the updated state of component $B_i$ to the observer through action $\beta_i$. 

![Figure 6.2: Semantic rules defining the composition controller / scheduler](image)
– Rule CONTR-SCH3. When the scheduler is notified of an internal action of the shared component $B_i$ where $i \not\in I$ (that is, the scheduler has not managed the latest action of component $B_i$), the controller receives the vector clock stored in component $B_i$ and updates its vector clock.

Example 6.3 (Controller of scheduler). Figure 6.3 depicts the controller of scheduler $S_1$ (initially presented in Figure 4.2, p. 36). Actions $(\overline{\beta_1}, \emptyset)$ and $(\overline{\beta_2}, \emptyset)$ consist in sending the updated state to the observer. Actions $(\overline{\text{Drain}_1}, \emptyset)$ and $(\overline{\text{Fill}_{12}}, \{\text{send}_{2}\})$ consist in notifying the observer about the occurrence of interactions managed by the scheduler. Moreover, $\text{send}_2$ sends the vector clock to the shared component $\text{Tank}_2$. The controller receives the vector clock stored in the shared component $\text{Tank}_2$ through action $(\overline{\text{recv}_2}, \emptyset)$ and updates its vector clock. For the sake of simplicity, the variables attached to the transition labels are not shown.

6.1.2 Controllers of Shared Components

Below, we define the controllers attached to shared components. Intuitively, the controller of a shared component ensures data exchange among the shared component and the corresponding schedulers. A scheduler sets its current clock in the controller of a shared component which can be used later by another scheduler.

Definition 6.4 (Controller of shared component). Local controller $C_{bi}^t$ for a shared component $B_i \in B_s$ with the behavior $(Q_i, \text{Act}_i \cup \{\beta_i\}, \rightarrow_i)$ is the LTS $(Q_{bi}^t, \mathcal{R}_{bi}^t, \rightarrow_{C_{bi}^t})$, where

\begin{itemize}
  \item $Q_{bi}^t = VC$ is the set of states,
  \item $\mathcal{R}_{bi}^t \subseteq \{\text{send}_j \llbracket vc \rrbracket, \text{recv}_j \llbracket vc \rrbracket \mid S_j \in S \land B_i \in \text{scope}(S_j) \land vc \in VC\}$ is the set of actions,
  \item $\rightarrow_{C_{bi}^t} \subseteq Q_{bi}^t \times \mathcal{R}_{bi}^t \times Q_{bi}^t$ is the transition relation defined as

$$\begin{align*}
\{ & vc \xrightarrow{\text{recv}_j \llbracket vc' \rrbracket}^C_{bi} \max(vc, vc') \mid a \in \text{Int} \land a \cap \text{Act}_i \neq \emptyset \land \text{managed}(a) = S_j \} \\
\cup & \{ & vc \xrightarrow{\text{send}_j \llbracket vc \rrbracket}^C_{bi} \mid S_j \in S \land B_i \in \text{scope}(S_j) \} .
\end{align*}$$
\end{itemize}
The state of the controller $C^b_i$ is represented by its vector clock. Controller $C^b_i$ has two types of actions:

- action $rcv_j \{vc'\}$ consists in receiving the vector clock $vc'$ of scheduler $S_j$,

- action $send_j \{vc\}$ consists in sending the vector clock $vc$ stored in the controller $C^b_i$ to scheduler $S_j$.

The two types of transitions can be understood as follow:

- For each action of component $B_i$, which is managed by scheduler $S_j$, we include a transition executing action $rcv_j \{vc'\}$ to receive the vector clock $vc'$ of scheduler $S_j$ and to update the vector clock stored in controller $C^b_i$.

- We include a transition with a set of actions for all the schedulers that have component $B_i$ in their scope, that is $\{S_j \in S \mid B_i \in \text{scope}(S_j)\}$, to send the stored vector clock of controller $C^b_i$ to the controllers of the corresponding schedulers, that is $\{C^b_j \mid S_j \in S \land B_i \in \text{scope}(S_j)\}$.

**Definition 6.5** (Semantics of $B_i \otimes_b C^b_i$). The composition of shared component $B_i$ and controller $C^b_i$, denoted by $B_i \otimes_b C^b_i$, is the LTS $(Q_i \times Q_c, (Act_i \cup \{\beta_i\}) \times R_{bc}, \rightarrow_{bc})$ where the transition relation $\rightarrow_{bc} \subseteq (Q_i \times Q_c) \times (\{\text{Act}_i \cup \{\beta_i\}\} \times R_{bc}) \times (Q_i \times Q_c)$ is defined by the semantics rules in Figure 6.4.

The semantic rules in Figure 6.4 can be intuitively understood as follows:

- **Rule Cont-Sha1.** applies when the scheduler notifies the shared component to execute an action part of an interaction. Controller $C^b_i$ receives the value of the vector clock of scheduler $S_j$ from the associated controller $C^b_j$ in order to update the value of the vector clock stored in controller $C^b_i$.

- **Rule Cont-Sha2.** applies when the shared component $B_i$ finishes its computation by executing $\beta_i$, and controller $C^b_i$ notifies the controllers of the schedulers that have component $B_i$ in their scope, through actions $send_j$, for $j \in J$, where $J$ is the set of indexes of schedulers which have the shared component $B_i$ in their scope. Actions $send_j$ sends the vector clock stored in controller $C^b_i$ to controllers $C^b_j$. 

**Figure 6.4:** Semantic rules defining the composition controller / shared component
Example 6.6 (Controller of shared component). Figure 6.5 depicts the controller of the shared component Tank₂ (initially presented in Figure 4.2, p. 36). Action \textit{rcv₁} (resp. \textit{rcv₂}) consists in the reception and storage of the vector clock from scheduler \( S₁ \) (resp. \( S₂ \)) upon the execution of interaction \textit{Fill₁₂} (resp. \textit{Drain₂₃}). Action \{\textit{send₁}, \textit{send₂}\} sends the stored vector clock to the schedulers \( S₁ \) and \( S₂ \) when the component Tank₂ performs its internal action \( β₂ \).

6.1.3 Instrumentation of Multi-threaded CBSs

Following the instrumentation defined in Section 6.1.1 (p. 51), for the systems with a centralized scheduler, since there is no shared component, the instrumentation is simply defined by only adding a controller to the scheduler. The controller is in charge of sending the events corresponding to the local partial-trace (that is, the partial trace of the system) to the central observer. The vector clock is discarded since the order of the execution of interactions is known by the central scheduler. The events are sent to the observer in the same order as they have occurred (we assume that communication channels are reliable).

Definition 6.7 (Controller of the scheduler of a multi-threaded CBS). Controller \( C^s \) is defined as an LTS \((Q_{C^s}, R_{C^s}, →_{C^s})\) such that:

- \( Q_{C^s} = \{q_c\} \) is the singleton set of states;
- \( R_{C^s} = \{\hat{β}_i \mid B_i ∈ B\} \cup \{\hat{a} \mid a ∈ Int\} \) is the set of actions;
- \( →_{C^s} ⊆ Q_{C^s} × R_{C^s} × Q_{C^s} \) is the transition relation defined as:

\[
\{q_c →_{C^s} q_c \mid a ∈ Int\} \cup \{q_c →_{C^s} q_c \mid B_i ∈ B\}
\]

Controller \( C^s \) has two actions. Action \( \hat{a} \) consists in notifying the observer about the execution of interaction \( a \). Action \( \hat{β}_i \) consists in notifying the observer about the internal action of component \( B_i \). The last state of component \( B_i \) is also transmitted to the observer.

Definition 6.8 (Semantics of \( S⊗_s C^s \)). The composition of scheduler \( S \) and controller \( C^s \), denoted by \( S⊗_s C^s \),
6.2 Event Extraction from the Local Partial-Traces of the Instrumented System

is the LTS \((Q_S \times Q_{C^*}, Int \times R_{C^*}, \rightarrow_{sc})\) where the transition relation \(\rightarrow \subseteq (Q_S \times Q_{C^*}) \times (Int \times R_{C^*}) \times (Q_S \times Q_{C^*})\) is defined by the semantic rules in Figure 6.6.

**Rule Cont-Sch1.** When the scheduler executes an interaction \(a \in Int\), the controller notifies the observer of the execution of \(a\).

**Rule Cont-Sch2.** When the scheduler is notified of an internal action of component \(B_i \in B\) through action \(\beta_i\), the controller transfers the updated state of component \(B_i\) to the observer through action \(\widehat{\beta}_i\).

6.2 Event Extraction from the Local Partial-Traces of the Instrumented System

According to Definition 6.1 (p. 51) and Definition 6.4 (p. 54), the first action in the semantic rules of a controlled scheduler or shared component corresponds to an interaction of the initial system. Thus, the notion of trace is extended in the natural way by considering the additional semantics rules. Elements of a trace are updated by including the new configurations and actions of controlled schedulers and shared components.

**Example 6.9** (Local partial-traces of the instrumented system). Consider Example 5.2 (p. 45), the local partial-traces of the instrumented system for two partial traces \(t_1\) and \(t_2\) are:

- \(s_1(t_1) = (d_1, d_2, d_3) \cdot \left(\text{Fill}_{12}, \left(\text{Fill}_{12}[1,0], \text{send}_{2}[1,0]\right)\right) \cdot (\bot, \bot, ?) \cdot (\{\beta_1\}, (\{\beta_1\}, \emptyset)) \cdot (f_1, \bot, ?) \cdot (\{\text{Drain}_1\}, (\{\text{Drain}_1\}[2,0])) \cdot (\bot, \bot, ?) \cdot (\{\beta_2\}, (\{\beta_2\}, \emptyset)) \cdot (\bot, f_2, ?),

- \(s_2(t_1) = (d_1, d_2, d_3) \cdot (\{\text{Fill}_3\}, (\{\text{Fill}_3\}[0,1])) \cdot (\bot, d_2, \bot) \cdot (\{\beta_2\}, (\{\beta_2\}, \emptyset)) \cdot (\bot, f_2, ?),

- \(s_1(t_2) = (d_1, d_2, d_3) \cdot (\text{Fill}_{12}, \left(\text{Fill}_{12}[1,0], \text{send}_{2}[1,0]\right)\) \cdot (\bot, \bot, ?) \cdot (\{\beta_2\}, (\{\beta_2\}, \emptyset)) \cdot (\bot, f_2, ?) \cdot (\{\beta_1\}, (\{\beta_1\}, \emptyset)) \cdot (f_1, f_2, ?),

- \(s_2(t_2) = (d_1, d_2, d_3) \cdot (\{\text{Fill}_3\}, (\{\text{Fill}_3\}[0,1])) \cdot (\bot, d_2, \bot) \cdot (\{\beta_3\}, (\{\beta_3\}, \emptyset)) \cdot (\bot, d_2, f_3) \cdot (\{\beta_2\}, (\{\beta_2\}, \emptyset)) \cdot (\{\beta_2\}, (\{\beta_2\}, \emptyset)) \cdot (\{\beta_2\}, \emptyset)) \cdot (\beta_2, d_2, f_3) \cdot \left(\text{Drain}_{23}, \left(\text{Drain}_{23}[1,0], \text{send}_{2}[1,2]\right)\right) \cdot (\bot, \bot, \bot).\)
For both traces $t_1$ and $t_2$, scheduler $S_2$ is notified of the state update of component $Tank_2$ (that is $\beta_2$), but scheduler $S_2$ does not sent the associated event to the observer. Indeed, following the semantics rules of composition of a scheduler and its controller (Definition 6.2, p. 53), a scheduler only sends the received state from a component only if the execution of the latest action on this component has been managed by this scheduler.

**Definition 6.10 (Sequence of events).** Let $t$ be the partial trace of the system $M$ and $s_j(t) = q_0 \cdot \gamma_1 \cdot q_1 \cdot \gamma_2 \cdot q_2 \cdot \gamma_3 \cdot q_3 \cdot \gamma_4 \cdot q_4$, for $j \in [1 \ldots |S|]$, be the local partial-trace of scheduler $S_j$ (as per Definition 5.1, p. 44). The sequence of events of $s_j(t)$ is inductively defined as follows:

- event($s_j(t)$) = $\epsilon$,

- event ($s_j(t) \cdot \gamma \cdot q$) =
  \[
  \begin{cases}
  \text{event}(s_j(t)) \cdot (a, vc) & \text{if } \gamma \text{ is of the form } (*, (a[vc], *)) , \\
  \text{event}(s_j(t)) \cdot \beta_i & \text{if } \gamma \text{ is of the form } (*, (\beta_i, *)) , \\
  \text{event}(s_j(t)) & \text{otherwise}.
  \end{cases}
  \]

According to the semantic rules of composition $S_j \otimes C_j$ (see Definition 6.2, p. 53), controller $C_j$ sends information to the observer (actions denoted by ^ over them) when scheduler $S_j$ (i) executes an interaction $a \in \text{Act}$, or (ii) is notified by the internal action of a component which the execution of its latest action has been managed by scheduler $S_j$.

**Example 6.11 (Sequence of events).** The sequences of events of local partial-traces in Example 6.9 (p. 57) are:

- event($s_1(t_1)$) : ($\text{Fill}_{12}, (1, 0)) \cdot \beta_1 \cdot (\text{Drain}_{12}, (2, 0)) \cdot \beta_2$,

- event($s_2(t_1)$) : ($\text{Fill}_{3}, (0, 1))$,

- event($s_1(t_2)$) : ($\text{Fill}_{12}, (1, 0)) \cdot \beta_2 \cdot \beta_1$,

- event($s_2(t_2)$) : ($\text{Fill}_{3}, (0, 1)) \cdot \beta_3 \cdot (\text{Drain}_{23}, (1, 2))$.

**Events of instrumented multi-threaded CBS.** Sequence of events in a multi-threaded system is defined in a similar way where the vector clocks are discarded.

**Definition 6.12 (Sequence of events of multi-threaded CBS).** Let $t = q_0 \cdot a_1 \cdot q_1 \cdot \ldots \cdot a_{k-1} \cdot q_{k-1} \cdot a_k \cdot q_k$ be the partial trace of a multi-threaded CBS as per Definition 4.13 (p. 40). The sequence of events of $t$ is event($t$) = $a_1 \cdot \ldots \cdot a_{k-1} \cdot a_k \in \left(\text{Int} \cup \bigcup_{i=1}^{\left|\beta_i\right|} \{\beta_i\}\right)^\star$.

In the multi-threaded setting the actual partial trace of the system is the local partial-trace of the central scheduler, so that all the events are generated from a unique scheduler, therefore the sequence of events is totally ordered.
6.3 Correctness of Instrumentation

Following the instrumentation defined in Section 6.1 (p. 50), one can obtain a transformed system whose execution at runtime generates the associated events. Furthermore, we shall prove that such an instrumentation does not modify the initial behavior of the original system.

**Definition 6.13** (Instrumented system). For a CBS consisting of a set of components $B = \{B_1, \ldots, B_{|B|}\}$ where $B_i = (Q_i, \text{Act}_i \cup \{\beta_i\}, \rightarrow_i)$ (as per Definition 4.1, p. 34), a set of schedulers $S = \{S_1, \ldots, S_{|S|}\}$ where $S_j = (Q_{S_j}, \text{Act}_{S_j}, \rightarrow_{S_j})$ (as per Definition 4.3, p. 35), and with the global behavior $(Q, \text{GAct}, \rightarrow)$ (as per Definition 4.8, p. 37), the instrumented CBS is the set of components $\{S_1 \otimes C_{S_1}, \ldots, S_{|S|} \otimes C_{S_{|S|}}, B'_1, \ldots, B'_{|B|}\}$ where $B'_i = B_i \otimes b_i$ if $B_i \in B_s$ and $B'_i = B_i$ otherwise. We define the behavior of the instrumented CBS as an LTS $(Q_c, \text{GAct}_c, \rightarrow_c)$ where:

- $Q_c \subseteq \bigotimes_{i=1}^{|B|} Q'_i \times \bigotimes_{j=1}^{|S|} (Q_{S_j} \times Q_{C_{S_j}})$ is the set of states consisting of the states of schedulers and components with their controllers where $Q'_i = Q_i \times Q_{C_{S_i}}$ if $B_i \in B_s$ and $Q'_i = Q_i$ otherwise.

- $\text{GAct}_c = \text{GAct} \times \{R_{C_{S_j}}, R_{C_{b_i}} \mid S_j \in S \land B_i \in B_s\}$ is the set of actions,

- $\rightarrow_c \subseteq Q_c \times \text{GAct}_c \times Q_c$ is the transition relation.

Component $C_{S_j}$ is the controller of scheduler $S_j$ for $j \in [1..|S|]$ as per Definition 6.1 (p. 51), and component $C_{b_i}$ is the controller of shared component and $B_i$ as per Definition 6.4 (p. 54). An action of the instrumented system consists of two synchronous actions; an action of the original system and the action of the associated controllers to notify the observer about the occurrence of the action and/or the action of the controllers to exchange vector clocks.

**Proposition 6.14.** $(Q, \text{GAct}, \rightarrow) \sim (Q_c, \text{GAct}_c, \rightarrow_c)$.

Proposition 6.14 states that the LTS of the instrumented CBS (see Definition 6.13) is weakly bi-similar to the LTS of initial CBS. Thus the composition of a set of controllers with schedulers and shared components defined in Section 6.1 (p. 50) does not affect the semantics of the initial system.

**Proof.** The proof of Proposition 6.14 is in Appendix A.2.2 (p. 163). □

**Example 6.15** (Bisimulation between instrumented CBS and original CBS). Intuitively, by comparing the associated local partial-traces of the partial trace $t_1$ of the instrumented system presented in Example 6.9 (p. 57) and the corresponding local partial-traces of the original system presented in Example 5.2 (p. 44), we have:
6. INSTRUMENTING CBSS

By filtering out the actions of the controllers, we obtain the local partial-traces of the original system.

\(- s_1(t_1) = (d_1, d_2, d_3) \cdot (\text{Fill}_{12}, (\text{Fill}_{12}[(1,0)], \text{send}_{2}[(1,0)])) \cdot (\bot, \bot, ?) \cdot (\{\beta_1\}, (\{\beta_1\}, \emptyset)) \cdot (f_1, \bot, ?) \cdot (\{\text{Drain}_1\}, (\{\text{Drain}_1\}[(2,0)])) \cdot (\bot, \bot, ?) \cdot (\{\beta_2\}, (\{\beta_2\}, \emptyset)) \cdot (\bot, f_2, ?),\)

\(- s_2(t_1) = (d_1, d_2, d_3) \cdot (\{\text{Fill}_3\}, (\{\text{Fill}_3\}[(0,1)])) \cdot (\bot, \bot, ?) \cdot (\{\beta_2\}, (\{\beta_2\}, \emptyset)) \cdot (\bot, f_2, ?)\).

The sequence of interactions and partial states (consisting of the states of component \(\text{Tank}_1\), \(\text{Tank}_2\) and \(\text{Tank}_3\)) are similar in both systems. The actions of the controllers do not interfere the actions of the original system. Each action of the instrumented system is followed by a simultaneous action of the corresponding controller to generate and send the associated event.

**Summary:** In this chapter, we instrumented the model with abstract semantics to obtain the execution events at runtime. In the next chapter, we use such events to monitor the desired property by reconstructing the global trace of the system.
Reconstructing and Monitoring the Global Trace

Chapter abstract
In this chapter, we present how to reconstruct the global trace of the system using system’s events. The reconstructed global trace is needed for online monitoring. We introduce two online global-trace reconstruction methods. The first method is applicable for the multi-threaded CBS, where the generated events are totally ordered. In this method, upon the reception of any event, we build, on-the-fly, the unique equivalent global trace with the global state of the components, that is the witness trace. Witness trace is given to an online monitor to evaluate the global states step by step. The second method is more general and is applicable for the distributed CBS where the events are not globally ordered. In this method, all the possible combinations of the occurrence of the events are taken into account with respect to the vector clocks of the events. Thus, we provide a set of compatible global traces. Each compatible global trace is the witness trace of a partial trace of the system that could have been the actual partial trace of the system. We represent the set of reconstructed global trace as a computation lattice. Each node of the lattice represents a global state of the system. A path in the lattice is a sequence of causally-related nodes, started from the initial node (i.e., the initial state of the system). Each path of the lattice represents a compatible global trace of the system. Moreover, we introduce a novel online LTL monitoring technique on the constructed computation lattice, so that each nodes carries a set of formula evaluating the set of paths end up with the node.
In the previous chapter, we define how to instrument a CBS to have controllers generating events (i.e., the state updates corresponding to the local partial-traces) sent to a central observer. The observer receives two sorts of events: events related to the execution of an interaction in \( \text{Int} \), referred to as action events, and events related to internal actions in \( \bigcup_{i \in [1..B]} (\{\beta_i\} \times Q_i) \), referred to as update events. (Recall that internal actions carry the state of the component that has performed the action – the state is transmitted to the observer by the controller that is notified of this action; see Chapter 4, p. 33). Hence, the set of action events is defined as \( E_a = \text{Int} \times VC \) (Recall that in multi-threaded setting \( E_a = \text{Int} \)), and the set of update events is defined as \( E_\beta = \bigcup_{i \in [1..B]} (\{\beta_i\} \times Q_i) \). The set of all events is denoted by \( E = E_\beta \cup E_a \).

Two cases are considered when reconstructing the global trace:

1. If the system is a multi-threaded CBS with one scheduler, the observer constructs the witness trace of the partial trace of the system, which allows monitoring the system against the properties referring to the global state of the system.

2. If the system has more than one scheduler (i.e., distributed system), the observer constructs a computation lattice representing the set of compatible global traces of the system.

In the following, we present details about each above-mentioned case separately.

### 7.1 Construction of the Witness Trace of Multi-threaded CBS

It is possible to show that a multi-threaded system with partial-state semantics \( M^\perp \) is (weakly) bisimilar to the corresponding sequential system with global-state semantics \( M^* \) (see [5], Theorem 1). Therefore, any partial trace in multi-threaded system \( M^\perp \) is related to a global trace of the corresponding sequential system \( M^* \). A weak bisimulation relation \( R \) is defined between the set of states of the system in global-state semantics (i.e., \( Q \subseteq \bigotimes_{i=1}^B Q_i^r \)) and the set of states of its partial-state system (i.e., \( Q^\perp \subseteq \bigotimes_{i=1}^B (Q_i^r \cup Q_i^b) \)), such that \( R = \{(q, r) \in Q \times Q^\perp \mid r \xrightarrow{\beta^*} q \} \). Any global state in partial-state semantics system is equivalent to the corresponding global state in global-state semantics system, and any partial state (i.e., a global state with at least one busy state) in partial-state semantics system is equivalent to the successor global state obtained after stabilizing the system by executing busy interactions (which take place independently).

In the sequel, we consider a CBS with global-state semantics \( M^* \) and its partial-states semantics version \( M^\perp \). Intuitively, from any partial trace of \( M^\perp \), we want to reconstruct on-the-fly the corresponding global trace in \( M^* \) and evaluate a property which is defined over global states of \( M^* \).
7.1. CONSTRUCTION OF THE WITNESS TRACE OF MULTI-THREADED CBS

7.1.1 Witness Relation and Witness Trace

We define the notion of witness relation between traces in global-state semantics and traces in partial-state semantics, based on the bisimulation between $M^s$ and $M^\perp$. Any trace of $M^\perp$ is related to a trace of $M^s$, i.e., its witness. The witness trace allows one to monitor the system in partial-state semantics (thus benefiting from the parallelism) against properties referring to the global state of the system.

**Definition 7.1** (Witness relation and witness trace). Given the bisimulation $R$ between $M^s$ and $M^\perp$, the witness relation $W \subseteq \text{Tr}(M^s) \times \text{Tr}(M^\perp)$ is the smallest set that contains $(\text{init}, \text{init})$ and satisfies the following rules:

- $(t_1 \cdot a \cdot q_1, t_2 \cdot a \cdot q_2) \in W$, if $a \in \text{Int}$ and $(q_1, q_2) \in R$,
- $(t_1, t_2 \cdot \beta \cdot q_2) \in W$, if $(\text{last}(t_1), q_2) \in R$;

whenever $(t_1, t_2) \in W$.

If $(t_1, t_2) \in W$, we say that $t_1$ is a witness trace of $t_2$.

Suppose that the witness relation relates a trace in partial-state semantics $t_2$ to a trace in global-state semantics $t_1$. The states obtained after executing the same interaction in the two systems are bisimilar. Moreover, any move through a busy interaction in $M^\perp$ preserves the bisimulation between the state of $t_2$ followed by the busy interaction in $M^\perp$ and the last state of $t_1$ in $M^s$.

**Example 7.2** (Witness relation). Figure 7.1 illustrates the witness relation. State $q_0$ is the initial state of $M^s$ and $M^\perp$. In the trace of $M^\perp$, gray circles after each interaction represent partial states which are bisimilar to the global state that comes after the corresponding trace of $M^s$.

**Example 7.3** (Witness trace). Let us consider $t_2$ as a partial trace of system Tank with partial-state semantics depicted in Figure 7.2 (p. 64), where $t_2 = (d_1, d_2, d_3) \cdot \{\text{Fill}_{12}\} \cdot (\bot, \bot, d_3) \cdot \beta_1 \cdot (f_1, \bot, d_3) \cdot \{\{\text{Drain}_{1}\}\} \cdot (\bot$.
7. RECONSTRUCTING AND MONITORING THE GLOBAL TRACE

\[(d_1, d_2, d_3) \xrightarrow{\text{Fill}_{12}} (\bot, \bot, d_3) \xrightarrow{\beta_1} (f_1, \bot, d_3) \xrightarrow{\text{Drain}_1} (\bot, \bot, d_3)\]

(R, \_)

\[(d_1, d_2, d_3) \xrightarrow{\text{Fill}_{12}} (f_1, f_2, d_3) \xrightarrow{\text{Drain}_1} (d_1, f_2, d_3)\]

Figure 7.2: An example of witness trace in system Tank

\[\bot, \bot, d_3). \text{The witness trace corresponding to trace } \tau_2 \text{ is } (d_1, d_2, d_3) \cdot \{\text{Fill}_{12}\} \cdot (f_1, f_2, d_3) \cdot \{\{\text{Drain}_1\}\} \cdot (d_1, f_2, d_3).\]

The following property states that any trace in partial-state semantics and its witness trace have the same sequence of interactions.

**Property 7.4.** \(\forall (\tau_1, \tau_2) \in W, \text{interactions}(\tau_1) = \text{interactions}(\tau_2).\)

**Proof.** The proof is done by induction on the length of the sequence of interactions and follows from the definitions of the witness relation and witness trace. The proof of this property can be found in Appendix A.1.1 (p. 152).

The next property states that any trace in the partial-state semantics has a unique witness trace in the global-state semantics.

**Property 7.5.** \(\forall \tau_2 \in \text{Tr}(M^{\bot}), \exists! \tau_1 \in \text{Tr}(M^s) . (\tau_1, \tau_2) \in W.\)

**Proof.** This proof is done by contradiction. The proof of this property is given in Appendix A.1.2 (p. 152).

We note \(W(\tau_2) = \tau_1\) when \((\tau_1, \tau_2) \in W.\)

Note that when running a system in partial-state semantics, the global state of the witness trace after an interaction \(a\) is not known until all the components involved in \(a\) have reached their ready locations after the execution of \(a.\) Nevertheless, even in non-deterministic systems, after a deterministic execution, this global state is uniquely defined and consequently there is always a unique witness trace (that is, non-determinism is resolved at runtime).

### 7.1.2 Construction of the Witness Trace

Given a trace in partial-state semantics, the witness trace is computed using function \(\text{RGT} (\text{Reconstructor of Global Trace}).\) The global states (of the trace in the global-state semantics) are reconstructed from partial states. We define a function to reconstruct global states from partial states.
**Definition 7.6 (Function RGT - Reconstructor of Global Trace).** Function \( RGT : \text{Tr}(\mathbb{M}^\perp) \rightarrow \text{pref}(\text{Tr}(\mathbb{M}^*)) \) is defined as:

\[
RGT(t) = \text{discriminant}(\text{acc}(\text{event}(t))),
\]

where:

1. \( \text{acc} : E \rightarrow Q \cdot (\text{Int} \cdot Q)^* \cdot (\text{Int} \cdot (Q^\perp \setminus Q))^* \) is defined as:
   - \( \text{acc}(\text{event}(\text{init})) = \text{acc}(\epsilon) = \text{init} \),
   - \( \text{acc}(\zeta \cdot e) = \begin{cases} \text{acc}(\zeta) \cdot e \cdot q & \text{if } e \in E_a \\ \text{map } [x \mapsto \text{upd}(q', x)] (\text{acc}(\zeta)) & \text{otherwise } (e = (\beta, q') \in E_\beta), \end{cases} \)
   
   where \( q = (q_1, \ldots, q_B), \forall i \in [1..|B|], q_i = \begin{cases} \bot & \text{if } i \in \text{involved}(e) \\ \text{last}(\text{acc}(\zeta))[i] & \text{otherwise}. \end{cases} \)

2. \( \text{discriminant} : Q \cdot (\text{Int} \cdot Q)^* \cdot (\text{Int} \cdot (Q^\perp \setminus Q))^* \rightarrow \text{pref}(\text{Tr}(\mathbb{M}^*)) \) is defined as:

\[
\text{discriminant}(t) = \max_{\preceq} \{ t' \in \text{pref}(t) \mid \text{last}(t') \in Q \}
\]

with \( \text{upd} : \{Q_i^1\}_{i=1}^{|B|} \times (Q^\perp \cup \text{Int}) \rightarrow Q^\perp \cup \text{Int} \) defined as:

- \( \text{upd}(q_i, a) = a, \text{for } a \in \text{Int}, \)
- \( \text{upd}(q_i, (q'_1, \ldots, q'_n)) = (q''_1, \ldots, q''_n), \)

where \( \forall k \in [1..|B|], q''_k = \begin{cases} q_k & \text{if } k = i \wedge q'_{k} \in Q^\beta_k \\ q'_k & \text{otherwise}. \end{cases} \)

Function RGT uses helper functions \( \text{acc} \) and \( \text{discriminant} \). First, function \( \text{acc} \) is an *accumulator* function which takes as input a sequence of events of a trace in partial-state semantics \( t \) (as per Definition 6.10, p.58) to construct the global trace. Function \( \text{acc} \) uses i) action events (i.e., \( e \in E_a \)) to extend the constructed trace, and ii) the (information in the) update event (i.e., \( e \in E_\beta \)) in order to update the partial states (i.e., states with busy states of components) using function \( \text{upd} \). Then, function \( \text{discriminant} \) returns the longest prefix of the result of \( \text{acc} \) corresponding to a trace in global-state semantics.

Note that because of the inductive definition of function \( \text{acc} \), the events of the input trace can be processed step by step by function RGT and allows to generate the witness incrementally. Moreover, such definition allows to apply the function RGT to a running system by online monitoring execution of interactions and partial states of components. Finally, we note that function RGT is monotonic (w.r.t. prefix ordering on sequences).

Such an online computation is illustrated in the following example.
Table 7.1: Values of function RGT for a sample input

<table>
<thead>
<tr>
<th>Step</th>
<th>Input trace in partial semantics</th>
<th>Corresponding event</th>
<th>Intermediate step</th>
<th>Output trace in global semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(d₁, d₂, d₃)</td>
<td>ε</td>
<td>(d₁, d₂, d₃)</td>
<td>(d₁, d₂, d₃)</td>
</tr>
<tr>
<td>1</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>Fill₁₂</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
<tr>
<td></td>
<td>(⊥, ⊥, d₃). β₁.</td>
<td></td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
<tr>
<td>2</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>Drain₁</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
<tr>
<td></td>
<td>(f₁, ⊥, d₃). β₁.</td>
<td></td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
<tr>
<td>3</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td></td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
<tr>
<td></td>
<td>(⊥, ⊥, d₃). β₁.</td>
<td></td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
<tr>
<td>4</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td></td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
<tr>
<td></td>
<td>(f₁, ⊥, d₃). β₂.</td>
<td></td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
<td>(d₁, d₂, d₃). Fill₁₂.</td>
</tr>
</tbody>
</table>

Example 7.7 (Applying function RGT). Table 7.1 (p. 66) illustrates Definition 7.6 (p. 65) on one trace of system Tank with initial state (d₁, d₂, d₃) followed by interactions Fill₁₂, β₁, Drain₁ and β₂. We comment on certain steps illustrated in the table. At step 0, the outputs of functions acc and discriminant are equal to the initial state. At step 1, the execution of interaction Fill₁₂ generates an action event which adds two elements Fill₁₂ · (⊥, ⊥, d₃) to traces t and acc(event(t)). At step 2, execution of β₁ generates an update event that has fresh information on component Tank₂ which is used to update the existing partial states, so that (⊥, ⊥, d₃) is updated to (f₁, ⊥, d₃). At step 4, Tank₂ becomes ready after β₂, and the partial state (f₁, ⊥, d₃) in the intermediate step is updated to the global state (f₁, f₂, d₃), therefore it appears in the output trace.

7.1.3 Properties of Global-trace Reconstruction

We state some properties of global-trace reconstruction based on function RGT, namely the soundness and maximality (information-wise) of the reconstructed global trace. To do so, we first start by stating some intermediate lemmas on the computation performed by function RGT.

Lemma 7.8. ∀(t₁, t₂) ∈ W, |acc(event(t₂))| = |t₁| = 2s + 1, where s = |interactions(t₁)|, acc is the accumulator used in the definition of function RGT (Definition 7.6 p. 65), and function interactions (defined in Section 4.3 p. 41) returns the sequence of interactions in a trace (removing β).

Lemma 7.8 states that, for a given trace in partial-state semantics t₂, the length of acc(event(t₂)) is equal to the length of the witness of t₂ (i.e., t₁).
Lemma 7.9. \( \forall t \in \text{Tr}(M^\perp) \). let \( \text{acc}(\text{event}(t)) = (q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s) \) in

\[ \exists k \in [1 \ldots s] . q_k \in Q \implies \forall z \in [1 \ldots k] . q_z \in Q \land q_{z-1} \xrightarrow{a_z} q_z. \]

Lemma 7.9 states that, for a given trace in partial-state semantics \( t \), if there exists a global state \( q_k \) in sequence \( \text{acc}(\text{event}(t)) \) such that \( q_k \in Q, k \in [1 \ldots s] \), then all the states occurring before \( q_k \) in sequence \( \text{acc}(\text{event}(t)) \) are global states.

The next proposition states that the sequence of global states produced by function \( \text{RGT} \) (which is the composition of functions \( \text{discriminant} \) and \( \text{acc} \)) follows the global-state semantics.

Proposition 7.10. \( \forall t \in \text{Tr}(M^\perp) \).

\[ |\text{discriminant}(\text{acc}(\text{event}(t)))| \leq |\text{acc}(\text{event}(t))| \]

\[ \land \text{discriminant}(\text{acc}(\text{event}(t))) = q_0 \cdot a_1 \cdot q_1 \cdots a_d \cdot q_d \implies \forall i \in [1 \ldots d] . q_{i-1} \xrightarrow{a_i} q_i, \]

where \( \text{acc} \) (resp. \( \text{discriminant} \)) is the accumulator (resp. discriminant) function used in the definition of function \( \text{RGT} \) (Definition 7.6, p. 65) such that \( \text{RGT}(t) = \text{discriminant}(\text{acc}(\text{event}(t))) \).

Proposition 7.10 states that, for any trace in partial-state semantics \( t \), 1) the length of the output trace of function \( \text{RGT} \) (i.e., \( \text{discriminant}(\text{acc}(\text{event}(t))) \)) is shorter than or equal to the length of the output of function \( \text{acc} \) (i.e., \( \text{acc}(\text{event}(t)) \)), and 2) the output trace of function \( \text{RGT} \) is a trace in global-state semantics.

Example 7.11 (Illustration of Proposition 7.10). We illustrate Proposition 7.10 based on the execution trace in Table 7.1 (p. 66). At step 0, the length of \( \text{RGT}(t) \) is equal to the length of \( \text{acc}(\text{event}(t)) \), whereas for the next steps the length of \( \text{acc}(\text{event}(t)) \) is longer than the length of \( \text{RGT}(t) \). At step 4, the output trace follow the global-state semantics, that is \( (d_1, d_2, d_3) \xrightarrow{\text{Fill}_{12}} (f_1, f_2, d_3) \).

Moreover, the last element of a given trace in partial-state semantics \( t \) is always the same as the last element of output of \( \text{acc}(\text{event}(t)) \), as stated by the following lemma.

Lemma 7.12. \( \forall t \in \text{Tr}(M^\perp) \). \( \text{last}(\text{acc}(\text{event}(t))) = \text{last}(t) \).

Finally, any trace in partial-state semantics and its image through functions \( \text{event} \) and \( \text{acc} \) have the same sequence of interactions, as stated by the following lemma.

Lemma 7.13. \( \forall t \in \text{Tr}(M^\perp) \). \( \text{interactions(acc}(\text{event}(t))) = \text{interactions}(t) \).

Based on the above lemmas, we have the following theorem which states the soundness and maximality of the reconstructed global trace. That is, applying function \( \text{RGT} \) on a trace in partial-state semantics...
produces the longest possible prefix of the corresponding witness trace with respect to the current trace of the partial-state semantics system.

**Theorem 7.14 (On the reconstructed global trace with function RGT).** \( \forall t \in \text{Tr}(M^\perp) \).

\[
\begin{align*}
\text{last}(t) \in Q & \implies \text{RGT}(t) = W(t) \\
\text{AND} \text{last}(t) \notin Q & \implies \text{RGT}(t) = W(t') \cdot a, \text{with} \\
& \quad \text{min}_p \{ t_p \in \text{Tr}(M^\perp) \mid \exists a \in \text{Int}, \exists t'' \in \text{Tr}(M^\perp). t = t_p \cdot a \cdot t'' \wedge \exists i \in [1..|B|]. \\
& \quad (B_i.P \cap a \neq \emptyset) \wedge (\forall j \in [1..\text{length}(t'')]. \beta_i \neq \sigma''(j)) \}
\end{align*}
\]

Theorem 7.14 distinguishes two cases:

- When the last state of a system is a global state (i.e., \( \text{last}(t) \in Q \)), none of the components are in a busy location. Moreover, function RGT has sufficient information to build the corresponding witness trace (\( \text{RGT}(t) = W(t) \)).

- When the last state of a system is a partial state, at least one component is in a busy location and function RGT can not build a complete witness trace because it lacks information on the current state of such components. It is possible to decompose the input sequence \( t \) into two parts \( t' \) and \( t'' \) separated by an interaction \( a \). The separation is made on the interaction \( a \) occurring in trace \( t \) such that, for the interactions occurring after \( a \) (i.e., in \( t'' \)), at least one component involved in \( a \) has not executed any \( \beta \) transition (which means that this component is still in a busy location). Note that it may be possible to split \( t \) in several manners with the above description. In such a case, function RGT computes the witness for the smallest sequence \( t' \) (w.r.t. prefix ordering) as above because it is the only sequence for which it has information regarding global states. Note also that such splitting of \( t \) is always possible as \( \text{last}(t) \notin Q \) implies that \( t \) is not empty, and \( t' \) can be chosen to be \( \epsilon \).

In both cases, because of its inductive definition and monotonicity, RGT returns the maximal prefix of the corresponding witness trace that can be built with the information contained in the partial states observed so far.

The above explanation can be extended to a full proof which is given in Appendix A.1.4 (p. 156).

**Example 7.15 (Illustration of Theorem 7.14).** We illustrate the correctness of Theorem 7.14 based on the execution trace in Table 7.1 (p. 66). At step 0, since the last element in the trace is the initial state we can see that the output of function RGT is equal to the witness trace which is the initial state as well. At step 5, the output of function RGT is a sequence which consists of the witness of sequence \( (d_1, d_2, d_3) \cdot \text{Fill}_{12} \cdot (\perp, \perp, d_3) \cdot \beta_1 \cdot (f_1, \perp, d_3) \) (i.e., \( (d_1, d_2, d_3) \cdot \text{Fill}_{12} \cdot (f_1, f_2, d_3) \)) followed by \( \text{Drain}_{1} \). At this step,
function RGT can not process partial states following interaction $\text{Drain}_1$, because the component involved in $\text{Drain}_1$ is still busy.

### 7.1.4 Monitoring

One can reuse the results in [39] to monitor a multi-threaded system with partial-state semantics. One just has to instrument this system by adding the controller (see Chapter 6, p. 49) to generate the corresponding execution events at runtime and apply function RGT over the sequence of events and plug a monitor for a property on the global states of the system. At runtime, such monitor will (i) receive the sequence of reconstructed global states (with ready state of components) corresponding to the witness trace, (ii) preserve the concurrency of the system, and iii) state verdicts on the witness trace.

### 7.2 Construction of the Computation Lattice of Distributed CBSs

Contrarily the multi-threaded setting, in the distributed setting several schedulers execute actions concurrently so that it is not possible to extract the actual partial trace of the system by only observing the local partial-traces. One can find a set of compatible partial traces, each of which can be considered as the actual partial trace of the system with respect to the observed local partial-traces. Intuitively, the projection of each compatible partial trace on a specific scheduler results the observed local partial-trace of the scheduler. A naive monitoring solution is to construct the witness trace of each compatible partial-trace using the technique for the multi-threaded setting. Such a solution would result a huge computation process overhead, because of the enormous number of compatible partial traces, specially for the distributed system with less number of shared component and more concurrent events. Instead, we apply the global trace reconstruction on a computation lattice, that is a multi-dimensional execution trace, and introduce a novel technique for the monitoring of the computation lattice on-the-fly (see Section 7.2.5, p. 85). Such a computation lattice encompasses all the compatible global trace of the system.

**Computation Lattice**  
The computation lattice is represented implicitly using vector clocks. The construction of the lattice mainly performs the two following operations: (i) creations of new nodes and (ii) updates of existing nodes in the lattice. Action events lead to the creation of new nodes in the direction of the scheduler emitting the event while update events complete the information in the nodes of the lattice related to the state of the component related to the event. The state of a node initially (after creation) is a partial state and by updating the lattice it becomes a global state. Since the received events are not totally ordered (because of potential communication delay), we construct the computation lattice based on the vector clocks attached
to the received events. Note that we assume the events received from a scheduler are totally ordered.

We first extend the notion of computation lattice.

**Definition 7.16 (Extended Computation lattice).** An extended computation lattice $L$ is a tuple $(N, Int, \rightarrow)$, where

- $N \subseteq Q^l \times VC$ is the set of nodes, with $Q^l = \bigotimes_{i=1}^{|B|} \left( Q_i^l \cup \left\{ \perp_i \mid S_j \in S \land B_i \in \text{scope}(S_j) \right\} \right)$ and $VC$ is the set of vector clocks,
- $Int$ is the set of multi-party interactions as defined in Section 4.1 (p. 34),
- $\rightarrow = \{ (\eta, a, \eta') \in N \times Int \times N \mid a \in Int \land \eta \rightarrow \eta' \land \eta.\text{state} \xrightarrow{a} \eta'.\text{state} \}$,

where $\rightarrow$ is the extended presentation of happened-before relation which is labeled by the set of multi-party interactions and $\eta.\text{state}$ referring to the state of node $\eta$.

We simply refer to extended computation lattice as computation lattice. Intuitively, a computation lattice consists of a set of partially connected nodes, where each node is a pair, consisting of a state of the system and a vector clock. A system state consists in the states of all components. The state of a component is either a ready state or a busy state (as per Definition 4.1, p. 34). In this context we represent a busy state of component $B_i \in \mathcal{B}$, by $\perp_i$ which shows that component $B_i$ is busy to finish the computation process of its latest action which has been managed by scheduler $S_j \in \mathcal{S}$. A computation lattice $L$ initially consists of an initial node $\text{init}_L = (\text{init}, (0, \ldots, 0))$, where $\text{init}$ is the initial state of the system and $(0, \ldots, 0)$ is a vector clock where all the clocks associated to the schedulers are zero. The set of nodes of computation lattice $L$ is denoted by $L.\text{nodes}$, and for a node $\eta = (q, vc) \in L.\text{nodes}$, $\eta.\text{state}$ denotes $q$ and $\eta.\text{clock}$ denotes $vc$. If (i) the event of node $\eta$ happened before the events of node $\eta'$, that is $\eta'.\text{clock} > \eta.\text{clock}$ and $\eta \rightarrow \eta'$, and (ii) the states of $\eta$ and $\eta'$ follow the global behavior of the system (as per Definition 4.8, p. 37) in the sense that the execution of an interaction $a \in Int$ from the state of $\eta$ brings the system to the state of $\eta'$, that is $\eta.\text{state} \xrightarrow{a} \eta'.\text{state}$, then in the computation lattice it is denoted by $\eta \xrightarrow{a} \eta'$ or by $\eta \rightarrow \eta'$ when clear from context.

Two nodes $\eta$ and $\eta'$ of the computation lattice $L$ are said to be concurrent if neither $\eta.\text{clock} > \eta'.\text{clock}$ nor $\eta'.\text{clock} > \eta.\text{clock}$. For two concurrent nodes $\eta$ and $\eta'$ if there exists a node $\eta''$ such that $\eta'' \rightarrow \eta$ and $\eta'' \rightarrow \eta'$, then node $\eta''$ is said to be the meet of $\eta$ and $\eta'$ denoted by $\text{meet}(\eta, \eta', L) = \eta''$.

The rest of this section is structured as follows. In Section 7.2.1 some intermediate notions are defined in order to introduce our algorithm to construct the computation lattice in Section 7.2.2 (p. 75). In Section 7.2.3 (p. 80) we discuss the correctness of the algorithm.
7.2.1 Intermediate Operations for the Construction of the Computation Lattice

In the reminder, we consider a computation lattice \( L \) as per Definition 7.16 (p. 70). The reception of a new event either modifies \( L \) or is kept in a queue to be used later. Action events extend \( L \) using operator \( \text{extend} \) (Definition 7.17, p. 71), and update events update the existing nodes of \( L \) by adding the missing state information into them using operator \( \text{update} \) (Definition 7.24, p. 73). By extending the lattice with new nodes, one needs to further extend the lattice by computing joints of the created nodes (Definition 7.22, p. 72) with the existing ones so as to complete the set of possible global traces.

Extension of the lattice. We define a function to extend a node of the lattice with an action event which takes as input a node of the lattice and an action event and outputs a new node.

**Definition 7.17 (Node extension).** Function \( \text{extend} : (Q^l \times VC) \times E_a \rightarrow Q^l \times VC \) is defined as follows. For a node \( \eta = ((q_1, \ldots, q_{|B|}), vc) \in Q^l \times VC \) and an action event \( e = (a, vc') \in E_a \),

\[
\text{extend}(\eta, e) = \begin{cases} 
((q_1', \ldots, q_{|B|}', vc')) & \text{if } \exists j \in [1..|S|]. \\
\text{undefined} & \text{otherwise}; 
\end{cases}
\]

with \( \forall i \in [1..|B|]. q_i' = \begin{cases} 
q_i & \text{if } B_i \in \text{involved}(a), \\
\bot_k & \text{otherwise}. 
\end{cases} \)

where \( k = \text{managed}(a).\text{index}. \)

Node \( \eta \) said to be extendable by event \( e \) if \( \text{extend}(\eta, e) \) is defined. Intuitively, node \( \eta = (q, vc) \) represents a global state of the system and extensibility of \( \eta \) by action event \( e = (a, vc') \) means that from the global state \( q \), scheduler \( S_j = \text{managed}(a) \), could execute interaction \( a \). State \( \bot_k \) indicates that component \( B_i \) is busy and being involved in a global action which has been executed (managed) by scheduler \( S_k \) for \( k \in [1..|S|] \).

We say that computation lattice \( L \) is extendable by action event \( e \) if there exists a node \( \eta \in L.\text{nodes} \) such that \( \text{extend}(\eta, e) \) is defined.

**Property 7.18.** \( \forall e \in E_a. |\{ \eta \in L.\text{nodes} | \exists \eta' \in Q^l \times VC. \eta' = \text{extend}(\eta, e) \}| \leq 1. \)

Property 7.18 states that for any update event \( e \), there exists at most one node in the lattice for which function \( \text{extend} \) is defined (meaning that \( L \) can be extended by event \( e \) from that node).

**Example 7.19 (Node extension).** Considering the local partial-traces described in Example 6.11 (p. 58), initially, the computation lattice consists of the initial node which has the initial state \( \text{init} \), with an associated...
vector clock \((0, 0)\), i.e., \(\text{init}_L = ((d_1, d_2, d_3), (0, 0))\). As for the sequence of events in trace \(t_1\), node \(((d_1, d_2, d_3), (0, 0))\) is extendable by event \((\text{Fill}_{12}, (1, 0))\) because, according to Definition 7.17, we have:

\[
\text{extend}(((d_1, d_2, d_3), (0, 0)), (\text{Fill}_{12}, (1, 0))) = ((\perp_1, \perp_2, d_3), (1, 0)).
\]

Furthermore, to illustrate Property 7.18, let us consider the extended lattice after event \((\text{Fill}_{12}, (1, 0))\) which consists of two nodes, initial node \(((d_1, d_2, d_3), (0, 0))\) and node \(((\perp_1, \perp_2, d_3), (1, 0))\). When action event \((\text{Fill}_3, (0, 1))\) is received, we have \(\text{extend}((\text{init}_L), (\text{Fill}_3, (0, 1))) = ((d_1, d_2, \perp_3), (0, 1))\) whereas extend\(((\perp_1, \perp_2, d_3), (1, 0)), (\text{Fill}_5, (0, 1))\) is not defined which shows that Property 7.18 holds on the lattice.

We define a relation between two vector clocks to distinguish the concurrent execution of two interactions such that both could happen from a specific global state of the system.

**Definition 7.20** (Relation \(\mathcal{J}_L\)). Relation \(\mathcal{J}_L \subseteq VC \times VC\) is defined between two vector clocks as follows:

\[
\mathcal{J}_L = \{(vc, vc') \in VC \times VC \mid \exists! k \in [1 \ldots |S|]. vc[k] = vc'[k] + 1 \wedge \exists! l \in [1 \ldots |S|]. vc'[l] = vc[l] + 1 \wedge \forall j \in [1 \ldots |S|] \setminus \{k, l\}. vc[j] = vc'[j]\}.
\]

For two vector clocks \(vc\) and \(vc'\) to be in relation \(\mathcal{J}_L\), \(vc\) and \(vc'\) should agree on all but two clocks values related to two schedulers of indexes \(k\) and \(l\). On one of these indexes, the value of one vector clock is equal to the value of the other vector clock plus 1, and the converse on the other index. Intuitively, \((\eta.\text{clock}, \eta'.\text{clock}) \in \mathcal{J}_L\) means that nodes \(\eta\) and \(\eta'\) are associated to two concurrent events (caused by the execution of two interactions managed by two different schedulers) that both could happen from a unique global state of the system which is the meet of \(\eta\) and \(\eta'\) (see Property 7.21). Example 7.23 illustrates relation \(\mathcal{J}_L\).

**Property 7.21.** \(\forall \eta, \eta' \in L.\text{nodes} \cdot (\eta.\text{clock}, \eta'.\text{clock}) \in \mathcal{J}_L \implies \text{meet}(\eta, \eta', L) \in L.\text{nodes}\).

Property 7.21 states that for two nodes \(\eta\) and \(\eta'\) in lattice \(L\) such that \((\eta.\text{clock}, \eta'.\text{clock}) \in \mathcal{J}_L\), there exists a node in lattice \(L\) as the meet of \(\eta\) and \(\eta'\), that is meet\((\eta, \eta', L) \in L.\text{nodes}\).

The joint node of \(\eta\) and \(\eta'\) is defined as follows.

**Definition 7.22** (Joint node). For two nodes \(\eta, \eta' \in L.\text{nodes}\) such that \((\eta.\text{clock}, \eta'.\text{clock}) \in \mathcal{J}_L\), the joint node of \(\eta\) and \(\eta'\), denoted by \(\text{joint}(\eta, \eta', L) = \eta''\), is defined as follows:

\[
- \forall i \in [1 \ldots |B|]. \eta''.\text{state}[i] = \begin{cases} \eta.\text{state}[i] & \text{if } \eta.\text{state}[i] \neq \eta_m.\text{state}[i], \\ \eta'.\text{state}[i] & \text{otherwise} \end{cases};
- \eta''.\text{clock} = \text{max}(\eta.\text{clock}, \eta'.\text{clock});
\]

where \(\eta_m = \text{meet}(\eta, \eta', L)\).
According to Property 7.21, for two nodes $\eta$ and $\eta'$ in relation $\mathcal{J}_L$, their meet node exists in the lattice. The state of the joint node of $\eta$ and $\eta'$ is defined by comparing their states and the state of their meet. Since two nodes in relation $\mathcal{J}_L$ are concurrent, the state of component $B_i$ for $i \in [1..|\mathcal{B}|]$ in nodes $\eta$ and $\eta'$ is either equal to the state of component $B_i$ in their meet, or only one of the nodes $\eta$ and $\eta'$ has a different state than their meet (components can not be both involved in two concurrent executions). The joint node of two nodes $\eta$ and $\eta'$ takes into account the latest changes of the state of the nodes $\eta$ and $\eta'$ compared to their meet. Note that $\text{joint}(\eta, \eta', \mathcal{L}) = \text{joint}(\eta', \eta, \mathcal{L})$, because joint is defined for nodes whose clocks are in relation $\mathcal{J}_L$.

**Example 7.23 (Relation $\mathcal{J}_L$ and joint node).** To continue Example 7.19 (p. 71), the reception of action event $(\text{Fill}_3, (0, 1))$ extends the lattice in the direction of scheduler $S_2$ because function $\text{extend}$ is defined, that is:

$$\text{extend}(((d_1, d_2, d_3), (0, 0)), (\text{Fill}_3, (0, 1))) = ((d_1, d_2, \bot_3^2), (0, 1)).$$

After this extension, the lattice has three nodes which are $((d_1, d_2, d_3), (0, 0)), ((\bot_1^1, \bot_2^1, \bot_3^3), (1, 0))$ and $((d_1, d_2, \bot_3^3), (0, 1))$. According to Definition 7.20 (p. 72), the vector clocks of the two nodes of the lattice $((\bot_1^1, \bot_2^1, d_3), (1, 0))$ and $((d_1, d_2, \bot_3^3), (0, 1))$ are in relation $\mathcal{J}_L$ (i.e., $((1, 0), (0, 1)) \in \mathcal{J}_L$). Therefore, following Definition 7.22 (p. 72), the joint node of the two above nodes is $((\bot_1^1, \bot_2^1, \bot_3^3), (1, 1))$, and their meet is $((d_1, d_2, d_3), (0, 0))$.

**Update of the lattice.** We define a function to update a node of the lattice which takes as input a node of the lattice and an update event and outputs the updated version of the input node.

**Definition 7.24 (Node update).** Function $\text{update} : (Q^l \times VC) \times E_\beta \rightarrow Q^l \times VC$ is defined as follows. For a node $\eta = ((q_1, \ldots, q_{|\mathcal{B}|}), vc)$ and an update event $e = (\beta_i, q_i') \in E_\beta$ with $i \in [1..|\mathcal{B}|]$ which is sent by scheduler $S_k$ with $k \in [1..|\mathcal{S}|]$:

$$\text{update}(\eta, e) = ((q_1, \ldots, q_{i-1}, q_i'', q_{i+1}, \ldots, q_{|\mathcal{B}|}), vc),$$

with $q_i'' = \begin{cases} q_i' & \text{if } q_i = \bot_1^k, \\ q_i & \text{otherwise.} \end{cases}$

An update event $(\beta_i, q_i')$ contains an updated state of some component $B_i$. By updating a node $\eta$ in the lattice with an update event which is sent from scheduler $S_k$, we update the partial state associated to $\eta$ by adding the state information of that component, if the state of component $B_i$ associated to node $\eta$ is $\bot_1^k$. Intuitively means that a busy state which is caused by an execution of an action managed by scheduler $S_k$ can only be replace by a ready state sent by the same scheduler $S_k$. Updating node $\eta$ does not modify the associated vector clock $vc$. 

---
Example 7.25 (Node update). To continue Example 7.23, let us consider node \(((\bot_1, \bot_1, d_3), (1, 0))\) whose state is a partial state (because of the lack of the state information of \(Tank_1\) and \(Tank_2\)), and update event \((\beta_1, f_1)\) sent by scheduler \(S_1\). To obtain the updated node, we apply function update over the node and the update event. We have: 

\[
\text{update(((\bot_1, \bot_1, d_3), (1, 0)), (\beta_1, f_1)) = ((d_1, d_2, d_3), (0, 0)).}
\]

Buffering events. The reception of an action event or an update event might not always lead to extending or updating the current computation lattice. Due to communication delay, an event which has happened before another event might be received later by the observer. It is necessary for the construction of the computation lattice to use events in a specific order. Such events must be kept in a waiting queue to be used later. For example, such a situation occurs when receiving action event \(e\) such that function extend is not defined over \(e\) and none of the existing nodes of the lattice. In this case event \(e\) must be kept in the queue until obtaining another configuration of the lattice in which function extend is defined. Moreover, an update event \(e'\) referring to an internal action of component \(B_i\) is kept in the queue if there exists an action event \(e''\) in the queue such that component \(B_i\) is involved in \(e''\), because we can not update the nodes of the lattice with an update event associated to an execution which is not yet taken into account in the lattice.

Definition 7.26 (Queue \(\kappa\)). A queue of events is a finite sequence of events in \(E\). Moreover, for a non-empty queue \(\kappa = e_1 \cdot e_2 \cdots e_r\), remove\((\kappa, e) = \kappa(1 \cdots z - 1) \cdot \kappa(z + 1 \cdots r)\) with \(e = e_z \in \{e_1, e_2, \ldots, e_r\}\).

Queue \(\kappa\) is initialized to an empty sequence. Function remove takes as input queue \(\kappa\) and an event in the queue and outputs the version of \(\kappa\) in which the given event is removed from the queue.

Example 7.27 (Event storage in the queue). Let us consider trace \(t_2\) in Example 6.11 (p. 58), such that all the events of scheduler \(S_2\) are received by the observer earlier than the events of scheduler \(S_1\). After the reception of action event \((\text{Fill}_3, (0, 1))\), since extend\(((d_1, d_2, d_3), (0, 0)), (\text{Fill}_3, (0, 1))\) is defined, the lattice is extended in the direction of scheduler \(S_2\) and the new node \(((d_1, d_2, \bot_3^2)(0, 1))\) is created. The reception of update event \((\beta_3, f_3)\) updates the newly created node \(((d_1, d_2, \bot_3^2)(0, 1))\) to \(((d_1, d_2, f_3)(0, 1))\). After the reception of action event \((\text{Drain}_{23}, (1, 2))\), since there is no node in the lattice where function extend is defined over, event \((\text{Drain}_{23}, (1, 2))\) must be stored in the queue, therefore \(\kappa = (\text{Drain}_{23}, (1, 2))\).
Algorithm 7.1 MAKE

Global variables: $L$ initialized to $\text{init}_L$, $\kappa$ initialized to $\epsilon$, $V$ initialized to $(0, \ldots, 0)$.

1: procedure MAKE($e$, from-the-queue)
2: if $e \in E_a$ then $\triangleright$ if $e$ is an action event.
3: ACTIONEVENT($e$, from-the-queue)
4: else if $e \in E_\beta$ then $\triangleright$ if $e$ is an update event.
5: UPDATEEVENT($e$, from-the-queue)
6: end if
7: end procedure

7.2.2 Algorithm Constructing the Computation Lattice

In the following, we define an algorithm based on the above definitions to construct the computation lattice based on the events received by the global observer.

The algorithm consists of a main procedure (see Algorithm 7.1 p. 75) and several sub-procedures using global variables lattice $L$ (Definition 7.16 p. 70) and queue $\kappa$ (Definition 7.26).

For an action event $e \in E_a$ with $e = (a, vc)$, $e$.action denotes interaction $a$ and $e$.clock denotes vector clock $vc$. For an update event $e \in E_\beta$ with $e = (\beta_i, q_i)$, $e$.index denotes index $i$.

Initially, after the reception of event $e$ from a controller of a scheduler, the observer calls the main procedure MAKE($e$, false). In the following, we describe each procedure in detail.

MAKE (Algorithm 7.1): Procedure MAKE takes two parameters as input: an event $e$ and a boolean variable from-the-queue. Parameters $e$ and from-the-queue vary based on the type of event $e$. Boolean variable from-the-queue is true when the input event $e$ is picked up from the queue and false otherwise (i.e., event $e$ is received from a controller of a scheduler). Procedure MAKE uses two sub-procedures, ACTIONEVENT and UPDATEEVENT. If the input event is an action event, sub-procedure ACTIONEVENT is called, and if the input event is an update event, sub-procedure UPDATEEVENT is called. Procedure MAKE updates the global variables.

ACTIONEVENT (Algorithm 7.2): Procedure ACTIONEVENT is associated to the reception of action events and takes as input an action event $e$ and a boolean parameter from-the-queue, which is false when event $e$ is received from a controller of a scheduler and true when event $e$ is picked up from the queue. Procedure ACTIONEVENT modifies global variables $L$ and $\kappa$.

Procedure ACTIONEVENT has a local boolean variable lattice-extend which is true when an input action event could extend the lattice (i.e., the current computation lattice is extendable by the input action
Algorithm 7.2 ACTIONEVENT

1: procedure ACTIONEVENT(e, from-the-queue)
2:     lattice-extend ← false
3:     for all η ∈ L.nodes do
4:         if ∃η′ ∈ Q × VC . η′ = extend(η, e) then
5:             L.nodes ← L.nodes ∪ {η′} ▷ extend the lattice with the new node.
6:             MODIFYQUEUE(e, from-the-queue, true) ▷ event e is removed from the queue if it was
7:                 picked up from the queue.
8:         break
9:     end if
10: end for
11: if ¬ lattice-extend then
12:     MODIFYQUEUE(e, from-the-queue, false)  ▷ event e is added to the queue if it was not
13:         picked up from the queue.
14:     return
15: end if
16:  JOINTS()  ▷ extend the lattice with joint nodes.
17:  REMOVEEXTRANODES()  ▷ lattice size reduction.
18: if ¬ from-the-queue then
19:     CHECKQUEUE()  ▷ recall the events stored in the queue.
20: end if
21: end procedure

event) and false otherwise.

By iterating over the existing nodes of lattice L, ACTIONEVENT checks if there exists a node η in
L.nodes such that function extend is defined over event e and node η (Definition 7.17, p. 71). If such
a node η is found, ACTIONEVENT creates the new node extend(η, e), adds it to the set of the nodes of
the lattice, invokes procedure MODIFYQUEUE, and stops iteration. Otherwise, ACTIONEVENT invokes
procedure MODIFYQUEUE and terminates.

In the case of extending the lattice by a new node, it is necessary to create the (possible) joint nodes. To
this end, in Line[15] procedure JOINTS is called to evaluate the current lattice and create the joint nodes. For
optimization purposes, after making the joint nodes procedure REMOVEEXTRANODES is called to eliminate
unnecessary nodes to optimize the lattice size.

After making the joint nodes and (possibly) reducing the size of the lattice, if the input action event
is not picked from the queue, ACTIONEVENT invokes procedure CHECKQUEUE in Line[18] otherwise it
terminates.
Algorithm 7.3 UPDATEEVENT

1: procedure UPDATEEVENT(e, from-the-queue)
2:     for all e′ ∈ κ do
3:         if e′ ∈ Ea ∧ e.index ∈ involved(e′.action) then \ Circled{check if there exists an action event in the queue concerning component B_e.index.}
4:             MODIFYQUEUE(e, from-the-queue, false) \ Circled{event e is added to the queue if it was not picked up from the queue.}
5:         return
6:     end if
7:     end for
8:     for all η ∈ L.nodes do
9:         η ← update(η, e) \ Circled{update nodes according to Definition 7.24 (p. 73).}
10:    end for
11:   MODIFYQUEUE(e, from-the-queue, true)
12: end procedure

Algorithm 7.4 MODIFYQUEUE

1: procedure MODIFYQUEUE(e, from-the-queue, event-is-used)
2:     if from-the-queue ∧ event-is-used then
3:         κ ← remove(κ, e) \ Circled{event e is removed from the queue if it is picked from queue and used.}
4:     else if ∼from-the-queue ∧ ∼event-is-used then
5:         κ ← κ · e \ Circled{event e is added to the queue if it is not picked from queue and could not be used.}
6:     end if
7: end procedure

UPDATEEVENT (Algorithm 7.3): Procedure UPDATEEVENT is associated to the reception of update events. Recall that an update event e contains the state update of some component B_i with i ∈ [1,\|B\|] (e.index = i). Procedure UPDATEEVENT takes as input an update event e and a boolean value associated to parameter from-the-queue. Procedure UPDATEEVENT modifies global variables L and κ.

First, UPDATEEVENT checks the events in the queue. If there exists an action event e′ in the queue such that component B_i is involved in e′.action, UPDATEEVENT adds update event e to the queue using MODIFYQUEUE and terminates. Indeed, one can not update the nodes of the lattice with an update event associated to an execution which is not yet taken into account in the lattice.

If no action event in the queue concerned component B_i, UPDATEEVENT updates all the nodes of the lattice (Lines 8\text{10}) according to Definition 7.24 (p. 73).

Finally, the input update event is removed from the queue if it is picked from the queue, using MODIFYQUEUE.
Algorithm 7.5 JOINTS

1: procedure JOINTS
2: \[ J_L \leftarrow J\text{COMPUTE} \quad \triangleright \text{compute the pairs of the vector clocks of the nodes which are in } J_L. \]
3: \[ \text{while } J_L \neq \emptyset \text{ do} \]
4: \[ \text{for all } \eta, \eta' \in L.\text{nodes such that } (\eta.\text{clock}, \eta'.\text{clock}) \in J_L \text{ do} \]
5: \[ L.\text{nodes} \leftarrow L.\text{nodes} \cup \{\text{joint}(\eta, \eta', L)\} \quad \triangleright \text{extend the lattice with the new joint node.} \]
6: \[ J_L \leftarrow J_L \setminus \{(\eta.\text{clock}, \eta'.\text{clock})\} \]
7: \[ \text{end for} \]
8: \[ J_L \leftarrow J\text{COMPUTE} \]
9: \[ \text{end while} \]
10: end procedure

MODIFYQUEUE (Algorithm 7.4): Procedure MODIFYQUEUE takes as input an event \( e \) and boolean variables \( \text{from-the-queue} \) and \( \text{event-is-used} \). Procedure MODIFYQUEUE adds (resp. removes) event \( e \) to (resp. from) queue \( \kappa \) according to the following conditions. If event \( e \) is picked up from the queue (i.e., \( \text{from-the-queue} = \text{true} \)) and \( e \) is used in the algorithm to extend or update the lattice (i.e., \( \text{event-is-used} = \text{true} \)), event \( e \) is removed from the queue (Line 3). Moreover, if event \( e \) is not picked up from the queue and it is not used in the algorithm, event \( e \) is stored in the queue (Line 5).

JOINTS (Algorithm 7.5): Procedure JOINTS extends lattice \( L \) in such a way that all the possible joints have been created. First, procedure JCOMPUTE is invoked to compute relation \( J_L \) (Definition 7.20, p. 72) among the existing nodes of the lattice and then creates the joint nodes and adds them to the set of the nodes of the lattice. Then, after the creation of the joint node of two nodes \( \eta \) and \( \eta' \), \((\eta.\text{clock}, \eta'.\text{clock})\) is removed from relation \( J_L \). It is necessary to compute relation \( J_L \) again after the creation of joint nodes, because new nodes can be in relation \( J_L \). This process terminates when \( J_L \) is empty.

Algorithm 7.6 JCOMPUTE

1: procedure JCOMPUTE
2: \[ \text{for all } \eta, \eta', \eta'' \in L.\text{nodes do} \]
3: \[ \quad \text{if } \eta'' \rightarrow \eta \land \eta'' \rightarrow \eta' \text{ then} \quad \triangleright \text{if } \eta \text{ and } \eta' \text{ are associated to two concurrent events.} \]
4: \[ \quad J_L = J_L \cup \{(\eta.\text{clock}, \eta'.\text{clock})\} \quad \triangleright \text{\( \eta.\text{clock} \) and \( \eta'.\text{clock} \) are added to relation } J_L. \]
5: \[ \quad \text{end if} \]
6: \[ \text{end for} \]
7: \[ \text{return } J_L \]
8: end procedure

JCOMPUTE (Algorithm 7.6): Procedure JCOMPUTE computes relation \( J_L \) by pairwise iteration over all the nodes of the lattice and checks if the vector clocks of any two nodes satisfy the conditions in Defini-
Algorithm 7.7 CheckQueue
1: \textbf{procedure} CheckQueue
2: \hspace{1em} \textbf{while} true \textbf{do}
3: \hspace{2em} \kappa' \leftarrow \kappa
4: \hspace{2em} \textbf{for all} \ z \in [1 \ldots \text{length}(\kappa)] \ \textbf{do}
5: \hspace{3em} \text{MAKE}(\kappa(z), \text{true}) \triangleright \text{recall the events of the queue.}
6: \hspace{2em} \textbf{end for}
7: \hspace{2em} \textbf{if} \ \kappa = \kappa' \ \textbf{then}
8: \hspace{3em} \text{break} \triangleright \text{break if none of the events in the queue is used.}
9: \hspace{3em} \textbf{end if}
10: \hspace{2em} \textbf{end while}
11: \textbf{end procedure}

CheckQueue (Algorithm 7.7): Procedure CheckQueue recalls the events stored in the queue \( e \in \kappa \) and executes \text{MAKE}(e, \text{true})\), to check whether the conditions for taking them into account to update the lattice hold.

Procedure CheckQueue checks the events in the queue until none of the events in the queue can be used either to extend or to update the lattice. To this end, before checking queue \( \kappa \), in Line 3 a copy of queue \( \kappa \) is stored in \( \kappa' \), and after iterating all the events in queue \( \kappa \), the algorithm checks the equality of current queue and the copy of the queue before checking. If the current queue \( \kappa \) and copied queue \( \kappa' \) have the same events, it means that none of the events in queue \( \kappa \) has been used (thus removed), therefore the algorithm stops checking the queue again by breaking the loop in Line 8.

Note, when the algorithm is iterating over the events in the queue, i.e., when the value of variable \text{from-the-queue} is true, it is not necessary to iterate over the queue again (Algorithm 7.2, p. 76, Line 17). Moreover, events in the queue are picked up in the same order as they have been stored in the queue (FIFO queue).

Algorithm 7.8 RemoveExtraNodes
1: \textbf{procedure} RemoveExtraNodes
2: \hspace{1em} \textbf{for all} \ \eta \in \mathcal{L}.\text{nodes} \ \textbf{do}
3: \hspace{2em} \textbf{if} \ \forall j \in [1 \ldots |\mathcal{S}|], \exists \eta' \in \mathcal{L}.\text{nodes}.\eta'.\text{clock}[j] > \eta.\text{clock}[j] \ \textbf{then} \triangleright \text{if there exists a node with a strictly greater clocks in the vector clock.}
4: \hspace{3em} \text{remove}(\mathcal{L}.\text{nodes}, \eta) \triangleright \text{the node with the smaller vector clock is removed.}
5: \hspace{2em} \textbf{end if}
6: \hspace{2em} \textbf{end for}
7: \textbf{end procedure}
7. RECONSTRUCTING AND MONITORING THE GLOBAL TRACE

**Figure 7.3**: Computation lattice associated to trace $t_2$ in Example 5.2

**REMOVEEXTRA_NODES (Algorithm 7.8)**: For optimization reasons, after extending the lattice by an action event, procedure REMOVEEXTRA_NODES is called to eliminate some (possibly existing) nodes of the lattice. A node in the lattice can be removed if the lattice no longer can be extended from that node. Having two nodes of the lattice $\eta$ and $\eta'$ such that every clock in the vector clock of $\eta'$ is strictly greater than the respective clock of $\eta$, one can remove node $\eta$. This is due to the fact that the algorithm never receives an action event which could have extended the lattice from $\eta$ where the lattice has already took into account an occurrence of event which has greater clocks stamp than $\eta$.clock.

**Remark 7.28.** The reason to remove the extra nodes of the lattice can be explained as following. First, our online algorithm is used for runtime monitoring purposes, and second, each node $n$ represents the evaluation of system execution up to node $n$. Hence, the nodes which reflect the state of the system in the past are not valuable for the runtime monitor.

**Example 7.29 (Lattice construction).** Figure 7.3a depicts the computation lattice according to the received sequence of events concerning trace $t_2$ of Example 6.11 (p. 58). Node $((d_1, d_2, f_3), (0, 1))$ is associated to event $(\text{Fill}_{12}, (1, 0))$ and node $((f_1, f_2, d_3), (1, 0))$ is associated to event $(\text{Fill}_3, (0, 1))$. Since these two events are concurrent, joint node $((f_1, f_2, f_3), (1, 1))$ is made. Node $((f_1, \perp_{22}, \perp_{23}), (1, 2))$ is associated to event $(\text{Drain}_{23}, (1, 2))$. Due to vector clock update technique, the node with vector clock of $(0, 2)$ is not created.

### 7.2.3 Insensibility to Communication Delay

Algorithm MAKE can be defined over a sequence of events received by the observer $\zeta = e_1 \cdot e_2 \cdot e_3 \cdots e_z \in E^*$ in the sense that one can apply MAKE sequentially from $e_1$ to $e_z$ initialized by taking event $e_1$, the initial lattice $\text{init}_L$ and an empty queue.
7.2. CONSTRUCTION OF THE COMPUTATION LATTICE OF DISTRIBUTED CBSS

**Proposition 7.30** (Insensitivity to the reception order). For any two sequences of events $\zeta, \zeta' \in E^*$, we have $\left( \forall S_j \in S : \zeta \downarrow S_j = \zeta' \downarrow S_j \right) \implies \text{MAKE}(\zeta) = \text{MAKE}(\zeta')$, where $\zeta \downarrow S_j$ is the projection of $\zeta$ on scheduler $S_j$ which results the sequence of events generated by $S_j$.

Proposition 7.30 states that different ordering of the events does not affect the output result of Algorithm MAKE. Note, this proposition assumes that all events in $\zeta$ and $\zeta'$ can be distinguished. For a sequence of events $\zeta \in E^*$, $\text{MAKE}(\zeta).\text{lattice}$ denotes the constructed computation lattice $L$ by algorithm MAKE.

7.2.4 Correctness of Lattice Construction

Computation lattice $L$ has an initial node $\text{init}_L$ which is the node with the smallest vector clock, and a frontier node which is the node with the greatest vector clock. A path of the constructed computation lattice $L$ is a sequence of causally-related nodes of the lattice, starting from the initial node and ending up in the frontier node.

**Definition 7.31** (Set of the paths of a lattice). The set of the paths of a constructed computation lattice $L$ is $\Pi(L) = \{ q_0 \cdot \alpha_1 \cdot \eta_1 \cdot \alpha_2 \cdot \eta_2 \cdot \alpha_z \cdot \eta_z \mid \eta_0 = \text{init}_L \land \forall r \in [1..z] . \left( \eta_{r-1} \overset{\alpha_r}{\Rightarrow} \eta_r \land (\exists N \subseteq L.\text{nodes} . \eta_{r-1} = \text{meet}(N, L) \land \eta_r = \text{joint}(N, L) \land \forall \eta \in N . \eta_{r-1} \overset{\alpha_{r-1}}{\Rightarrow} \eta \land \alpha_r = \bigcup_{\eta \in N} \alpha_{\eta}) \right) \}$, where the notions of meet and joint are naturally extended over a set of nodes.

A path is a sequence of nodes such that for each pair of adjacent nodes either (i) the prior node is in relation with the next node or (ii) the prior and the next node are the meet and the joint of a set of existing nodes respectively. A path from a meet node to the associated joint node represents an execution of a set of concurrent joint actions.

**Example 7.32** (Set of the paths of a lattice). In the computation lattice $L$ depicted in Figure 7.3a (p. 80), there are three distinct paths that begin from the initial node $\{(d_1, d_2, d_3), (0, 0)\}$ and end up to the frontier node $\{(f_1, \perp^2, \perp^3), (1, 2)\}$. The set of paths is $\Pi(L) = \{ \pi_1, \pi_2, \pi_3 \}$, where:

$\pi_1 = \{(d_1, d_2, d_3), (0, 0)\}, \{\text{Fill}_{12}\}, \{(f_1, f_2, d_3), (1, 0)\}, \{\{\text{Fill}_3\}\}, \{(f_1, f_2, f_3), (1, 1)\}, \{\text{Drain}_{23}\}, \{(f_1, \perp^2, \perp^3), (1, 2)\}$

$\pi_2 = \{(d_1, d_2, d_3), (0, 0)\}, \{\{\text{Fill}_3\}\}, \{(d_1, d_2, f_3), (0, 1)\}, \{\{\text{Fill}_{12}\}\}, \{(f_1, f_2, f_3), (1, 1)\}, \{\text{Drain}_{23}\}, \{(f_1, \perp^2, \perp^3), (1, 2)\}$

$\pi_3 = \{(d_1, d_2, d_3), (0, 0)\}, \{\text{Fill}_{12}\}, \{\{\text{Fill}_3\}\}, \{(f_1, f_2, f_3), (1, 1)\}, \{\text{Drain}_{23}\}, \{(f_1, \perp^2, \perp^3), (1, 2)\}$

Let us consider system $M$ with the global behavior $(Q, GAct, \rightarrow)$ as per Definition 4.8 (p. 37). At runtime, the execution of such a system produces a global trace $t = q^0 \cdot (\alpha_1 \cup \beta_1) \cdot q^1 \cdot (\alpha_2 \cup \beta_2) \cdots (\alpha_k \cup \beta_k) \cdot q^k$.
as per Definition 4.9 (p. 39). Since the actual partial trace \( t \) is not observable due to the occurrence of simultaneous and concurrent interactions and internal actions, partial trace \( t \) can be represented as a set of compatible partial traces, which could have happened in the system at runtime.

**Definition 7.33** (Compatible partial traces of a partial trace). The set of all compatible partial traces of partial trace \( t \) is \( \mathcal{P}(t) = \{ t' \in Q \cdot (GAct \cdot Q)^* \mid \forall j \in [1..|S|] \cdot t' \downarrow_{S_j} = t \downarrow_{S_j} = s_j(t) \} \).

Partial trace \( t' \) is compatible with the partial trace \( t \) if the projection of both \( t \) and \( t' \) on scheduler \( S_j \), for \( j \in [1..|S|] \), results the local partial-trace of scheduler \( S_j \). In a partial trace, for each global action which consists of several concurrent interactions and internal actions of different schedulers, one can define different ordering of those concurrent interactions, each of which represents a possible execution of that global action. Consequently, several compatible partial traces can be encoded from a partial trace of the distributed system \( M \).

Note that two compatible traces with only difference in the ordering of their internal actions are considered as a unique compatible trace. What matters in the compatible traces of a partial trace is the different ordering of interactions.

**Example 7.34** (The set of compatible partial traces). Let us consider the partial trace \( t_1 \) described in Example 4.10 (p. 39), that is \( t_1 = (d_1, d_2, d_3) \cdot \{ \text{Fill}_{12} \} \cdot (\bot, \bot, d_3) \cdot \{ \beta_1 \} \cdot (f_1, \bot, d_3) \cdot \{ \{ \text{Drain}_1 \} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_2 \} \cdot (\bot, f_2, \bot) \). The projection of \( t_1 \) on each scheduler is represented as follow:

\[
\begin{align*}
- t_1 \downarrow_{S_1} &= (d_1, d_2, d_3) \cdot \{ \text{Fill}_{12} \} \cdot (\bot, \bot, d_3) \cdot \{ \beta_1 \} \cdot (f_1, \bot, d_3) \cdot \{ \{ \text{Drain}_1 \} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_2 \} \cdot (\bot, f_2, \bot), \\
- t_1 \downarrow_{S_2} &= (d_1, d_2, d_3) \cdot \{ \text{Fill}_3 \} \cdot (\bot, , d_2, \bot).
\end{align*}
\]

The set of compatible partial traces is \( \mathcal{P}(t_1) = \{ t_1^1, t_1^2, t_1^3, t_1^4, t_1^5 \} \) where:

\[
\begin{align*}
- t_1^1 &= (d_1, d_2, d_3) \cdot \{ \text{Fill}_{12} \} \cdot (\bot, \bot, d_3) \cdot \{ \beta_1 \} \cdot (f_1, \bot, d_3) \cdot \{ \{ \text{Drain}_1 \} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_2 \} \cdot (\bot, f_2, \bot), \\
- t_1^2 &= (d_1, d_2, d_3) \cdot \{ \text{Fill}_{12} \} \cdot (\bot, \bot, d_3) \cdot \{ \beta_1 \} \cdot (f_1, \bot, d_3) \cdot \{ \{ \text{Drain}_1 \} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_2 \} \cdot (\bot, f_2, \bot), \\
- t_1^3 &= (d_1, d_2, d_3) \cdot \{ \text{Fill}_{12} \} \cdot (\bot, \bot, d_3) \cdot \{ \beta_1 \} \cdot (f_1, \bot, d_3) \cdot \{ \{ \text{Drain}_1 \} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_2 \} \cdot (\bot, f_2, \bot), \\
- t_1^4 &= (d_1, d_2, d_3) \cdot \{ \text{Fill}_{12} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_1 \} \cdot (f_1, \bot, \bot) \cdot \{ \{ \text{Drain}_1 \} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_2 \} \cdot (\bot, f_2, \bot), \\
- t_1^5 &= (d_1, d_2, d_3) \cdot \{ \{ \text{Fill}_3 \} \} \cdot (d_1, d_2, \bot) \cdot \{ \text{Fill}_{12} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_1 \} \cdot (f_1, \bot, \bot) \cdot \{ \{ \text{Drain}_1 \} , \{ \text{Fill}_3 \} \} \cdot (\bot, \bot, \bot) \cdot \{ \beta_2 \} \cdot (\bot, f_2, \bot).
\end{align*}
\]
Since the desired property is defined over the global states, for monitoring purposes it is necessary to obtain the global trace of the system with a sequence of global states. To this end, by inspiring the technique introduced in Section 7.1.2 (p. 64) to reconstruct the witness trace, we define a function which takes as input a partial trace of the distributed system (i.e., a sequence of partial states) and outputs an equivalent global trace in which all the internal actions ($\beta$) are removed from the trace and instead the updated state after each internal action is used to complete the states of the global trace.

Definition 7.35 (Function refine $R_\beta$). Function $R_\beta : Q \cdot (GAct \cdot Q)^* \rightarrow Q \cdot (Int \cdot Q)^*$ is defined as:

- $R_\beta(init) = init$,
- $R_\beta(\sigma \cdot (\alpha \cup \beta) \cdot q) = \begin{cases} R_\beta(\sigma) \cdot \alpha \cdot q & \text{if } \beta = \emptyset, \\ \text{map}[x \mapsto \text{upd}(q, x)](R_\beta(\sigma)) & \text{if } \alpha = \emptyset, \\ \text{map}[x \mapsto \text{upd}(q, x)](R_\beta(\sigma) \cdot \alpha \cdot q) & \text{otherwise}; \end{cases}$

with $\text{upd} : Q \times (Q \cup 2^{Int}) \rightarrow Q \cup 2^{Int}$ defined as:

- $\text{upd}((q_1, \ldots, q_{|B|}), \alpha) = \alpha$,
- $\text{upd}((q_1, \ldots, q_{|B|}, \alpha), (q_1', \ldots, q_{|B'|})) = (q_1'', \ldots, q_{|B''|})$.

where $\forall k \in [1..|B|], q''_k =\begin{cases} q_k & \text{if } (q_k \notin Q^b_k) \land (q'_k \in Q^b_k) \\ q'_k & \text{otherwise}. \end{cases}$

Function $R_\beta$ uses the (information in the) state after internal actions in order to update the partial states using function $\text{upd}$.

By applying function $R_\beta$ over the set of compatible partial traces $P(t)$, we obtain a new set of compatible global traces which is (i) equivalent to $P(t)$ (according to Definition 7.1, p. 63), (ii) internal actions are discarded in the presentation of each global trace and (iii) contains maximal global states that can be built with the information contained in the partial states observed so far.

A refined global trace $R_\beta(t)$ is said to be equal with a path $\eta_0 \cdot \alpha_1 \cdot \eta_1 \cdot \alpha_2 \cdot \eta_2 \cdot \ldots \cdot \alpha_z \cdot \eta_z$ if $R_\beta(t) = (\eta_0.state) \cdot \alpha_1 \cdot (\eta_1.state) \cdot \alpha_2 \cdot (\eta_2.state) \cdot \ldots \cdot \alpha_z \cdot (\eta_z.state)$.

Example 7.36 (Applying function $R_\beta$). By applying function $R_\beta$ over the set of compatible partial traces in Example 7.34 (p. 82) we have the refined traces (compatible global traces):

- $R_\beta(t^1_1) = (d_1, d_2, d_3) \cdot \{\text{Fill12}\} \cdot (f_1, f_2, d_3) \cdot \{\text{Drain1}\} \cdot \{\text{Fill3}\} \cdot (\perp, f_2, \perp)$,
- $R_\beta(t^2_1) = (d_1, d_2, d_3) \cdot \{\text{Fill12}\} \cdot (f_1, f_2, d_3) \cdot \{\text{Drain1}\} \cdot (\perp, f_2, d_3) \cdot \{\text{Fill3}\} \cdot (\perp, f_2, \perp)$,
- $R_\beta(t^3_1) = (d_1, d_2, d_3) \cdot \{\text{Fill12}\} \cdot (f_1, f_2, d_3) \cdot \{\text{Fill3}\} \cdot (\perp, f_2, \perp) \cdot \{\text{Drain1}\} \cdot (\perp, f_2, \perp)$,
7. RECONSTRUCTING AND MONITORING THE GLOBAL TRACE

- \( R_\beta(t_4) = (d_1, d_2, d_3) \cdot \{ Fill_{12}, \{ Fill_3 \} \} \cdot (f_1, f_2, \bot) \cdot \{ \{ Drain_1 \} \} \cdot (\bot, f_2, \bot), \)
- \( R_\beta(t_5) = (d_1, d_2, d_3) \cdot \{ \{ Fill_3 \} \} \cdot (d_1, d_2, \bot) \cdot \{ Fill_{12} \} \cdot (f_1, f_2, \bot) \cdot \{ \{ Drain_1 \} \} \cdot (\bot, f_2, \bot). \)

In Section 5.1 (Definition 5.1, p. 44) we defined \( \{ s_1(t), \ldots, s_m(t) \} \), the set of observable local partial-traces of the schedulers obtained from partial trace \( t \). According to Definition 6.10 (p. 58), from each local partial-trace we can obtain the sequences of events generated by the controller of each scheduler, such that the set of all the sequences of the events is \( \{ \text{event}(s_1(t)), \ldots, \text{event}(s_m(t)) \} \) with \( \text{event}(s_j(t)) \in E^* \) for \( j \in [1..|S|] \).

In the following, we define the set of all possible sequences of events that could be received by the observer.

**Definition 7.37** (Events order). Considering partial trace \( t \), the set of all possible sequences of events that could be received by the observer is \( \Theta(t) = \{ \zeta \in E^* | \forall j \in [1..|S|], \zeta \downarrow_{S_j} = \text{event}(s_j(t)) \} \).

Events are received by the observer in any order just under a condition in which the ordering of the local events of a scheduler is preserved.

**Proposition 7.38** (Soundness). \( \forall \zeta \in \Theta(t), \forall \pi \in \Pi(\text{MAKE}(\zeta) . \text{lattice}), \forall j \in [1..|S|], \pi \downarrow_{S_j} = R_\beta(s_j(t)). \)

Proposition 7.38 states that the projection of all paths in the lattice on a scheduler \( S_j \) for \( j \in [1..|S|] \) results in the refined local partial-trace of scheduler \( S_j \).

The following proposition states the correctness of the construction in the sense that applying Algorithm \text{MAKE} over a sequence of observed events (i.e., \( \zeta \in \Theta \)) at runtime, results in a computation lattice which encodes a set of the sequences of global states, such that each sequence represents a global trace of the system.

**Proposition 7.39** (Completeness). Given a partial trace \( t \) as per Definition 4.9 (p. 39), we have

\( \forall \zeta \in \Theta(t), \forall t' \in \mathcal{P}(t), \exists! \pi \in \Pi(\text{MAKE}(\zeta) . \text{lattice}) . \pi = R_\beta(t'). \)

\( \pi \) said to be the associated path of the compatible partial-trace \( t' \).

Applying algorithm \text{MAKE} over any of the sequence of events, constructs a computation lattice whose set of paths consists on all the compatible global traces.

**Example 7.40** (Existence of the set of compatible global traces in the constructed lattice). Let us consider partial trace \( t_1 \) presented in Example 4.10 (p. 39) and the set of all associated event of \( t_1 \) that is presented in Example 6.11 (p. 58). Events are received by the observer in order to make the lattice. Figure 7.4 illustrates the associated constructed computation lattice using algorithm \text{MAKE} consists of 5 paths \( \pi_1 \) to \( \pi_5 \). The set of
compatible global traces (presented in Example 7.36, p. 83) can be extracted from the reconstructed lattice, where $\pi_k = \mathcal{R}_\beta(t^k_1)$ for $k \in [1..5]$. Paths $\pi_1$ to $\pi_5$ are associated paths of the compatible partial-traces $t^1_1$ to $t^5_1$ respectively.

### 7.2.5 Monitoring

In this section, we address the problem of monitoring an LTL formula specifying the desired global behavior of the system.

In the usual case, evaluating whether an LTL formula holds requires the monitoring procedure to have access to the global state of the system. In the previous section we introduced how to construct the computation lattice using the partially-ordered events. Although by stabilizing the system to have the ready state of all components we could obtain a complete computation lattice using algorithm MAKE (see Section 7.2.2, p. 75), that is the state of each node is a global state, instead, we propose an on-the-fly verification of an LTL property during the construction of computation lattice.

There are many approaches to monitor LTL formulas based on various finite-trace semantics (see 19). One way of looking at the monitoring problem for some LTL formula $\varphi$ is described in 11 based on formula rewriting, which is also known as formula progression, or just progression. Progression splits a formula into (i) a formula expressing what needs to be satisfied by the observed events so far and (ii) a new
formula (referred to a future goal), which has to be satisfied by the trace in the future. We apply progression over a set of finite partial-traces, where each trace consists in a sequence of (possibly) partial states, encoded from the constructed computation lattice. An important advantage of this technique is that it often detects when a formula is violated or validated before the end of the execution trace, that is, when the constructed lattice is not complete, so it is suitable for online monitoring.

To monitor the execution of a distributed CBS with respect to an LTL property $\varphi$, we introduce a more informative computation lattice by attaching to each node of lattice $L$ a set of formulas. Given a computation lattice $L = (N, \rightarrow)$ (as per Definition 7.16, p. 70), we define an augmented computation lattice $L^\varphi$ as follow.

**Definition 7.41** (Computation lattice augmentation). $L^\varphi$ is a pair $(N^\varphi, \rightarrow)$, where $N^\varphi \subseteq Q^l \times VC \times 2^{LTL}$ is the set of nodes augmented by $2^{LTL}$, that is the set of LTL formulas. The initial node is $\text{init}^\varphi_L = (\text{init}, (0, \ldots, 0), \{\varphi\})$ with $\varphi \in LTL$ the global desired property.

In the newly defined computation lattice, a set of LTL formulas is attached to each node. The set of formulas attached to a node represents the different evaluation of the property $\varphi$ with respect to different possible paths from the initial node to the node. The state and the vector clock associated to each node and the happened-before relation are defined similar to the initial definition of computation lattice (see Definition 7.16, p. 70).

The construction of the augmented computation lattice requires some modifications to algorithm MAKE:

- Lattice $L^\varphi$ initially has node $\text{init}^\varphi_L = (\text{init}, (0, \ldots, 0), \{\varphi\})$.

- The creation of a new node $\eta$ in the lattice with $\eta.\text{state} = q$ and $\eta.\text{clock} = vc$, calculates the set of formulas $\Sigma$ associated to $\eta$ using the progression function (see Definition 7.42). The augmented node is $\eta = (q, vc, \Sigma)$, where $\Sigma = \{\text{prog}(LTL', q) \mid LTL' \in \eta'.\Sigma \land (\eta' \rightarrow \eta) \lor \exists N \subseteq L^{\varphi}.\text{nodes} . \eta' = \text{meet}(N, L) \land \eta = \text{joint}(N, L)\}$. We denote the set of formulas of node $\eta \in L^{\varphi}.\text{nodes}$ by $\eta.\Sigma$.

- Updating node $\eta = (q, vc, \Sigma)$ by update event $e = (\beta_i, q_i) \in E_\beta$, $i \in [1..|B|]$ which is sent by scheduler $S_j$, $j \in [1..|S|]$ updates all associated formulas $\Sigma$ to $\Sigma'$ using the update function (see Definition 7.43, p. 87), where $\Sigma' = \{\text{upd}_\varphi(LTL, q_i, j) \mid LTL \in \Sigma\}$.

**Definition 7.42** (Progression function). $\text{prog} : LTL \times Q^l \rightarrow LTL$ is defined using a pattern-matching with
\( p \in AP_i \in [1..|B|] \) and \( q = (q_1, \ldots, q_{|B|}) \in Q^i \).

\[
\text{prog}(\varphi, q) = \text{match}(\varphi) \begin{cases} 
T & \text{if } q_i \in Q^i_i \land p \in q_i \\
F & \text{if } q_i \in Q^i_i \land p \notin q_i \\
X^k p & \text{otherwise } (q_i = \perp^k, k \in [1..|S|])
\end{cases}
\]

We define a new modality \( X^k \) such that \( X^k p \) for \( p \in AP_i \in [1..|B|] \) and \( k \in [1..|S|] \) means that atomic proposition \( p \) has to hold at next ready state of component \( B_i \) which is sent by scheduler \( S_k \). For a sequence of partial states obtained at runtime \( \sigma = q_0 \cdot q_1 \cdot q_2 \cdot \cdots \) such that \( \sigma_j = q_j \), we have \( \sigma_j \models X^k p \iff z \models p \) where \( z = \min \left\{ r > j \mid (\sigma_{r-1} \downarrow S_k) \overset{\beta_{r, k}}{\rightarrow} (\sigma_r \downarrow S_k) \right\} \).

The truth value of the progression of an atomic proposition \( p \in AP_i \) for \( i \in [1..|B|] \) with a partial state \( q = (q_1, \ldots, q_{|B|}) \) is evaluated by true (resp. false) if the state of component \( B_i \) (that is \( q_i \)) is a ready state and satisfies (resp. does not satisfy) the atomic proposition \( p \). If the state of component \( B_i \) is not a ready state, the evaluation of the atomic proposition \( p \) is postponed to the next ready state of component \( B_i \).

**Definition 7.43** (Formula update function). \( \text{upd}_p : LTL \times \{Q^i\}_{i=1}^{|B|} \times [1..|S|] \rightarrow LTL \) is defined using a
pattern-matching with \( q_i \in Q_i^t \) for \( i \in [1..|B|] \).

\[
\text{upd}_\varphi(\varphi, q_i, j) = \text{match}(\varphi) \begin{cases} T & \text{if } p \in AP_i \cap q_i \land k = j \\ F & \text{if } p \in AP_i \cap \overline{q_i} \land k = j \\ X^k_{\beta p} & \text{otherwise (} p \notin AP_i \lor k \neq j \) \end{cases}
\]

\[
\text{X}^k_{\beta p} = \begin{cases} \varphi_1 \lor \varphi_2 \rightarrow \text{upd}_\varphi(\varphi_1, q_i) \lor \text{upd}_\varphi(\varphi_2, q_i) \\ \varphi_1 \land \varphi_2 \rightarrow \text{upd}_\varphi(\varphi_1, q_i) \land \text{upd}_\varphi(\varphi_2, q_i) \\ \varphi_1 U \varphi_2 \rightarrow \text{upd}_\varphi(\varphi_1, q_i) U \text{upd}_\varphi(\varphi_2, q_i) \\ G \varphi \rightarrow G \text{upd}_\varphi(\varphi, q_i) \\ F \varphi \rightarrow F \text{upd}_\varphi(\varphi, q_i) \\ X \varphi \rightarrow X \text{upd}_\varphi(\varphi, q_i) \\ \neg \varphi \rightarrow \neg \text{upd}_\varphi(\varphi, q_i) \\ T \rightarrow T \\ p \in AP_i \rightarrow p \end{cases}
\]

Update function updates a progressed LTL formula with respect to a ready state of a component. Intuitively, a formula consists in an atomic proposition whose truth or falsity depends on the next ready state of component \( B_i \) sent by scheduler \( S_k \), that is \( X^k_{\beta p} \) where \( p \in AP_i \), can be evaluated using update function by taking the first ready state of component \( B_i \) received from scheduler \( S_k \) after the formula rewrote to \( X^k_{\beta p} \).

**Example 7.44** (Formula progression and formula update over an augmented computation lattice). Let consider the system presented in Example 5.2 (p. 45) with partial trace \( t_2 \) and the associated sequence of events presented in Example 6.11 (p. 58) and desired property \( \varphi = G(d_3 \lor f_1) \). \( \mathcal{E}^\varphi \) initially has node \( init_\mathcal{E}^\varphi = ((d_1, d_2, d_3), (0, 0), \{\varphi\}) \). By observing the action event \((Fill_{12}, (1, 0))\), algorithm MAKE creates new node \( \eta_1 = ((\perp, \perp, d_3), (1, 0), \{\varphi\}) \), because \( \text{prog}(G(d_3 \lor f_1), (\perp, \perp, d_3)) = \text{prog}((d_3 \lor f_1), (\perp, \perp, d_3)) \land G(d_3 \lor f_1) = T \land G(d_3 \lor f_1) = G(d_3 \lor f_1) = \varphi \).

By observing the action event \((Fill_3, (0, 1))\), new node \( \eta_2 = ((d_1, d_2, \perp), (0, 1), \{X_\beta d_3 \land \varphi\}) \) is created, because \( \text{prog}(G(d_3 \lor f_1), (d_1, d_2, \perp)) = \text{prog}((d_3 \lor f_1), (d_1, d_2, \perp)) \land G(d_3 \lor f_1) = X_\beta d_3 \land G(d_3 \lor f_1) = X_\beta d_3 \land \varphi \). Formula \( X_\beta d_3 \) means that the evaluation of partial state \((d_1, d_2, \perp)\) with respect to formula \((d_3 \lor f_1)\) is on hold until the next ready state of component \( Tank_3 \).

Consequently joint node \( \eta_3 = ((\perp, \perp, \perp), (1, 1), \{X_\beta d_3 \land (X_\beta d_3 \lor X_\beta f_1) \land \varphi, (X_\beta d_3 \lor X_\beta f_1) \land \varphi, (X_\beta d_3 \lor X_\beta f_1) \land \varphi \}) \) is made. The update event \((\beta_3, f_3)\) updates both state and the set of formulas associated to each node as follows. Although \( init_\mathcal{E}^\varphi \) and \( \eta_1 \) remain intact, but node \( \eta_2 \) is updated.
Table 7.2: On-the-fly construction and verification of computation lattice

<table>
<thead>
<tr>
<th>step</th>
<th>event</th>
<th>constructed lattice</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\epsilon$</td>
<td>$((d_1, d_2, d_3), (0, 0), {x})$</td>
</tr>
<tr>
<td></td>
<td>- receiving event $\text{Fill}_{12}(1, 0)$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>$((d_1, d_2, d_3), (1, 0), {x})$</td>
</tr>
<tr>
<td></td>
<td>- extending by making a new node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- the formula projection of new node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- updating the formulas of existing nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- updating states of the existing nodes</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$((d_1, d_2, d_3), (0, 0), {x})$</td>
</tr>
<tr>
<td></td>
<td>- receiving event $\text{Fill}_{13}(0, 1)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- extending by making a new node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- extending by making the joint node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- the formula projection of new nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- three formulas attached to the frontier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- extending by making a new node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- the node with vector clock $(0, 0)$ can be removed</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$\beta_3, f_3$</td>
<td>$((d_1, d_2, d_3), (1, 0), {x})$</td>
</tr>
<tr>
<td></td>
<td>- updating states of the existing nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- updating the formulas of existing nodes</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$\beta_2, f_2$</td>
<td>$((d_1, d_2, d_3), (0, 0), {x})$</td>
</tr>
<tr>
<td></td>
<td>- updating states of the existing nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- updating the formulas of existing nodes</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$\text{Drain}_{23}(1, 2)$</td>
<td>$((d_1, d_2, d_3), (1, 0), {x})$</td>
</tr>
<tr>
<td></td>
<td>- extending by making a new node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- the formula projection of new node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- the node with vector clock $(0, 1)$ can be removed</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$\beta_1, f_1$</td>
<td>$((d_1, d_2, d_3), (0, 0), {x})$</td>
</tr>
<tr>
<td></td>
<td>- updating states of the existing nodes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- updating the formulas of existing nodes</td>
<td></td>
</tr>
</tbody>
</table>
7. RECONSTRUCTING AND MONITORING THE GLOBAL TRACE

to \(((d_1, d_2, f_3), (0, 1), \{F\})\) because \(\text{upd}_\varphi(X_\beta d_2 \land \varphi, f_3) = F\). Moreover, node \(\eta_3\) is updated to \(\eta_3 = (((\bot, f_3), (1, 1), \{F, (X_\beta f_1) \land \varphi, (X_\beta f_1) \land \varphi\})\).

The update event \((\beta_2, f_2)\) updates nodes \(\eta_1\) and \(\eta_2\) such that \(\eta_1 = (((\bot, f_2, d_3), (1, 0), \{\varphi\})\) and \(\eta_3 = (((\bot, f_2, f_3), (1, 1), \{F, (X_\beta f_1) \land \varphi, (X_\beta f_1) \land \varphi\})\).

By observing the action event \((\text{Drain}_{23}, (1, 2))\), the new node \(\eta_4 = (((\bot, \bot, \bot), (1, 2), \{F, (X_\beta d_3 \lor X_\beta f_1) \land (X_\beta f_1) \land \varphi, (X_\beta d_3 \lor X_\beta f_1) \land (X_\beta f_1) \land \varphi\})\) is created.

The update event \((\beta_1, f_1)\) updates nodes \(\eta_2, \eta_3\) and \(\eta_4\) such that \(\eta_1 = ((f_1, f_2, d_3), (1, 0), \{\varphi\}), \eta_3 = ((f_1, f_2, f_3), (1, 1), \{F, \varphi, \varphi\})\) and \(\eta_4 = ((f_1, \bot, \bot), (1, 2), \{F, \varphi, \varphi\})\).

Table 7.2 shows the step-by-step reconstructing and monitoring of the associated computation lattice. The highlighted nodes are the removed nodes using Algorithm 7.8 (p. 79), but for the sake of better understanding we show them.

### 7.2.6 Correctness of Formula Progression on the Lattice

In Section 7.2 (p. 69), we introduced how from an unobservable partial trace \(t\) of a distributed CBS one can construct a set of paths representing the set of compatible global-traces of \(t\) in form of a lattice. Furthermore, in Section 7.2.5 (p. 85) we adapted formula progression over the constructed lattice with respect to a given LTL formula \(\varphi\). What we obtained is a directed lattice \(L^\varphi\) starting from the initial node and ending up with frontier node \(\eta^f\). The set of formulas attached to the frontier node, that is \(\eta^f \cdot \Sigma\), represents the progression of the initial formula over the set of path of the lattice.

**Definition 7.45** (Progression on a partial trace). Function \(\text{PROG} : \text{LTL} \times Q \cdot (\text{GAct} \cdot Q)^* \rightarrow \text{LTL}\) is defined as:

- \(\text{PROG}(\varphi, \text{init}) = \varphi\),
- \(\text{PROG}(\varphi, \sigma) = \varphi'\),
- \(\text{PROG}(\varphi, \sigma \cdot (\alpha \cup \beta) \cdot q) = \text{prog}(\text{UPD}(\text{PROG}(\varphi, \sigma), Q), q)\) where
  - \(Q = \{q[i] \mid \beta_i \in \beta\}\) is the set of updated states,
  - function \(\text{UPD} : \text{LTL} \times Q^R \rightarrow \text{LTL}\) is defined as:
    - \(* \text{ UPD}(\varphi, \{\varepsilon\}) = \varphi,\)
    - \(* \text{ UPD}(\varphi, Q^c \cup \{q_i\}) = \text{upd}_\varphi(\text{UPD}(\varphi, Q^c), q_i)\).

with \(Q^R \subseteq \{q \in Q_i^c \mid i \in [1.. |B|]\}\) the set of subsets of ready states of the components.
7.2. CONSTRUCTION OF THE COMPUTATION LATTICE OF DISTRIBUTED CBSS

Function $\text{PROG}$ uses functions $\text{prog}$ (Definition 7.42, p. 86), $\text{upd}_\varphi$ (Definition 7.43, p. 87) and function $\text{UPD}$. Since after each global action in the partial trace we reach a partial state (see Definition 4.9, p. 39), function $\text{upd}_\varphi$ does not need to check among multiple partial states to find whose formula must be updated. That is why $\text{PROG}$ uses the simplified version of function $\text{upd}_\varphi$ by eliminating the scheduler index input. Moreover functions $\text{prog}$ modified in such way to take as input a partial state in $Q$ instead of a state in $Q^l$, because as we above mentioned, the index of schedulers does not play a role in the progression of an LTL formula on a partial trace.

Given a partial state $t$ as per Definition 4.9 (p. 39) and an LTL property $\varphi$, by $\eta_f \Sigma$ we denote the set of LTL formulas of the frontier node of the constructed computation lattice $\mathcal{L}^\varphi$, we have the two following proposition and theorems:

**Proposition 7.46.** Given an LTL formula $\varphi$ and a partial trace $t$, there exists a partial trace $t'$ such that $\text{PROG}(\varphi, t') = \text{progression}(\varphi, R_\beta(t'))$ with:

$$t' = \begin{cases} t & \text{if } \text{last}(t)[i] \in Q_i^l \text{ for all } i \in [1..|\mathcal{B}|], \\ t \cdot \beta \cdot q & \text{otherwise.} \end{cases}$$

Where $\beta \subseteq \bigcup_{i \in [1..|\mathcal{B}|]} \{\beta_i\}$, $\forall i \in [1..|\mathcal{B}|], q[i] \in Q_i^l$ and progression is the standard progression function on a global trace as described in [11].
Table 7.3: Overview of monitoring approaches w.r.t different settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Behavioral semantics</th>
<th>Observable trace</th>
<th>What is obtained after Instrumentation</th>
<th>Relation with the actual global trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Global-state semantics for components</td>
<td>Global trace</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Interactions are managed by one scheduler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-threaded</td>
<td>Partial-state semantics for components</td>
<td>Partial trace</td>
<td>Witness trace</td>
<td>Bisimilar global trace</td>
</tr>
<tr>
<td></td>
<td>Interactions are managed by one scheduler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed</td>
<td>Partial-state semantics for components</td>
<td>Local partial-traces</td>
<td>Computation lattice</td>
<td>A set of compatible global traces</td>
</tr>
<tr>
<td></td>
<td>Interactions are managed by several schedulers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Proposition 7.46 states that progression of an LTL formula on a partial trace of a distributed system (as per Definition 4.9, p. 39) using PROG results similar to the standard progression of the LTL formula on the corresponding refined global trace using progression if we allow the system to be stabilized by the execution of \( \beta \) actions of busy components. Intuitively, our progression method over on a trace of a distributed system follows the standard progression technique on a trace of a sequential system where the global state of the system is always defined.

**Theorem 7.47** (Soundness). For a partial trace \( t \) and LTL formula \( \varphi \), we have

\[
\forall \varphi' \in \eta^f.\Sigma, \exists t' \in \mathcal{P}(t). \text{PROG}(\varphi, t') = \varphi'.
\]

Theorem 7.47 states that each formula of the frontier node is derived from the progression of formula \( \varphi \) on a compatible partial-trace of \( t \).

**Theorem 7.48** (Completeness). For a partial trace \( t \) and an LTL formula \( \varphi \), we have:

\[
\eta^f.\Sigma = \{ \text{PROG}(\varphi, t') \mid t' \in \mathcal{P}(t) \}.
\]

Theorem 7.48 states that the set of formulas in the frontier node is equal to the set of progression of \( \varphi \) on all the compatible partial-traces of \( t \).

Figure 7.5 (p. 91) depicts our monitoring approach for a distributed CBS.

**Overview.** Table 7.3 combines together all our monitoring approaches on different settings. In the sequential setting, we deal with a component-based system with global-state semantics in which the observable trace is a global trace. Such global trace itself is suitable for runtime monitoring. For multi-threaded setting the observable trace is a partial trace. The proposed instrumentation allows the observer to reconstruct the witness trace of the observed partial trace. The unique witness trace is bisimilar to the global trace of the
corresponding sequential system. For distributed setting we deal with a set of observable local partial-traces associated to several schedulers. Each local partial-trace consists of the partial states of the components in the scope of the corresponding scheduler and the actions that the scheduler manages. The events associated to each local partial-trace are sent to the observer. The observer constructs and evaluate the computation lattice on-the-fly. The computation lattice contains a set of global traces compatible with the partial trace of the system.

**Summary:** In this chapter, we reconstructed on-the-fly the global trace using the events obtained from the instrumented model with abstract semantics at runtime. Moreover, we introduced online monitoring techniques for each execution setting. In the next chapters, we implement these techniques and evaluate their performance by applying them on several case studies.
Part II

Implementation and Evaluation
The BIP Framework

Chapter abstract

In this chapter, for the sake of completeness, we provide a succinct description of the BIP framework and refer to [5, 6] for the detailed and fully-formalized operational semantics. BIP is an implementation of the general framework presented in Chapter 4 (p. 33), and allows us to evaluate our monitoring approaches on actual implementations.

BIP (Behavior, Interaction, Priority) is a powerful and expressive framework for the formal construction of heterogeneous systems. BIP supports a construction methodology of components as the superposition of three layers: behavior, interaction, and priority. Layering favors a clear separation between behavior and structure. The behavior layer describes the operational semantics of atomic components. Atomic components are transition systems endowed with a set of local variables and a set of ports labeling individual transitions. Ports are used for synchronization and communication with other components. Transitions can be guarded by some constraints over local variables. Local variables of an atomic component can be sent or modified through the interacting ports. The interaction layer defines a set of connectors over the ports of atomic components describing the synchronizations (so-called interactions) between atomic components. An interaction is a synchronous action among (some of) the components which have one of their ports involved in the interaction. The priority layer describes scheduling policies for interactions. Composite components are built from connected atomic components along with a set of priority rules.
8.1 Multi-Threaded BIP Model

In this section we describe the behavior of a BIP model with partial-state semantics. We use such a model to illustrate our monitoring approach on a multi-threaded component-based systems.

Atomic Components. An atomic component is endowed with a finite set of local variables $X$ taking values in a set Data. Atomic components synchronize and exchange data with other components through ports.

Definition 8.1 (Port). A port $p[x_p]$, where $x_p \subseteq X$, is defined by a port identifier $p$ and some data variables in a set $x_p$.

Variables attached to ports are purposed to transfer values between interacting components (see also Definition 8.3, p. 99, for interactions). The variables attached to the port are also used to determine whether a communication through this port can take place (see below).

Definition 8.2 (Atomic component with partial state in BIP). An atomic component in BIP is defined as a tuple $B = (P \cup \{\beta\}, L \cup L^\perp, T, X)$, where $P$ is the set of ports, $\beta \notin P$ is a special port dedicated for internal action, $L$ is the set of ready (control) locations, $L^\perp$ is the set of busy locations such that $L^\perp \cap L = \emptyset$ and $T \subseteq ((L \times P \times G(X) \times \{\} \times L^\perp) \cup (L^\perp \times \{\beta\} \times \text{true} \times F^*(X) \times L))$ is the set of transitions, and $X$ is the set of variables.

$G(X)$ denotes the set of Boolean expressions over $X$ and $F(X)$ the set of assignments of expressions over $X$ to variables in $X$. For each transition $\tau = (l, p, g_{\tau}, f_{\tau}, l') \in T$, $g_{\tau}$ is a Boolean expression over $X$ (the guard of $\tau$), $f_{\tau} \in \{x := f^x(X) | x \in X \land f^x \in F(X)\}^*$: the computation step of $\tau$, a sequence of assignments to variables.

The behavior of the atomic component is an LTS $(Q, P \cup \{\beta\}, \rightarrow)$ where $Q = (L \cup L^\perp) \times (X \rightarrow \text{Data})$ is the set of states, and $\rightarrow = \{((l, v), p(v_p), (l', v')) \in Q \times P \cup \{\beta\} \times Q | \exists \tau = (l, p, g_{\tau}, f_{\tau}, l') \in T. g_{\tau}(v) \land v' = f_{\tau}(v \backslash v_p)\}$ is the transition relation.

A state is a pair $(l, v) \in Q$, where $l \in (L \cup L^\perp)$, $v \in X \rightarrow \text{Data}$ is a valuation of the variables in $X$. The evolution of states $(l, v) \xrightarrow{p[v_p]} (l', v')$, where $v_p$ is a valuation of the variables $x_p$ attached to port $p$, is possible if there exists a transition $(l, p[x_p], g_{\tau}, f_{\tau}, l')$, such that $g_{\tau}(v) = \text{true}$. As a result, the valuation $v$ of variables is modified to $v' = f_{\tau}(v \backslash v_p)$.

We use the dot notation to denote the elements of atomic components, e.g., for an atomic component $B$, $B.P$ denotes the set of ports of the atomic component $B$, $B.L$ denotes its set of locations, etc.
Interaction. For a given CBS build from a set of atomic component $B = \{B_1, \ldots, B_{|B|}\}$, we assume that their respective set of ports are pairwise disjoint, that is for two $i \neq j$ from $\{1 \ldots |B|\}$ we have $B_i.P \cap B_j.P = \emptyset$. Therefore $\bigcup_{i=1}^{\vert B \vert} \{B_i.P\}$ is defined as a set of all ports in the system. An interaction is a set of port defined as follows.

**Definition 8.3** (Interaction). An interaction $a$ is a tuple $(\mathcal{P}_a, F_a)$, where $\mathcal{P}_a = \{p_{i\lfloor x_i \rfloor} \mid p_i \in B_i.P\}_{i \in I}$ is the set of ports such that $\forall i \in I. \mathcal{P}_a \cap B_i.P = \{p_i\}$ and $F_a$ is a sequence of assignments to the variables in $\bigcup_{i \in I} x_i$.

When clear from context, an interaction $(\{p_{\lfloor x_p \rfloor}\}, F_a)$ consisting of only one port $p$ is denoted by $p$.

**Definition 8.4** (Composite component in BIP). A composite component $M^\perp = \Gamma(B_1, \ldots, B_{|B|})$ is defined from a set of atomic components $B = \{B_1, \ldots, B_{|B|}\}$ and $\Gamma = Int \cup \beta$ where $Int$ is the set of multi-party interactions and $\beta = \{\{\beta_i\}\}_{i=1}^{\vert B \vert}$ is the set of singleton busy interactions.

A state $q$ of a composite component $\Gamma(B_1, \ldots, B_{|B|})$ is an $|B|$-tuple $q = (q_1, \ldots, q_{|B|})$, where $q_i = (l_i, v_i)$ is a state of atomic component $B_i$. The semantics of the composite component $M^\perp$ is an LTS $(Q^\perp, \Gamma, \rightarrow)$, where $Q^\perp = B_1.Q \times \ldots \times B_{|B|}.Q$ is the set of states and $\rightarrow$ is the least set of transitions satisfying the global behavior defined in Definition 4.8 (p. 37).

**Priorities.** In composite components modeled in BIP, many interactions can be enabled at the same time, introducing a degree of non-determinism in the product behavior. Non-determinism can be restricted by means of priorities, specifying which of the interactions should be preferred among the enabled one.

Recall that, since we only deal with the execution traces of a component-based system, we assume that the obtained traces are correct with respect to the priorities. Although our monitoring technique can be applied on the prioritized BIP model, defining such a model is out of the scope of this work and only make the model complex.

**Remark 8.5.** Behavior of a multi-threaded composite component in BIP follows the semantic rules defined in Definition 4.17 (p. 41) which is interpreted by a centralized scheduler. Since in a multi-threaded BIP model the scheduling process is done in another level which is not accessible by the users, we adapt the instrumentation defined in Section 6.1.3 (p. 56) on the level of components and existing interactions. Moreover, we do not present the model of BIP scheduler for multi-threaded setting.

**Running example.** We use a task system, called Task, to illustrate our monitoring approach on a multi-threaded BIP model. The system consists of a task generator ($Generator$) along with 3 task executors
Example 8.6 (Multi-threaded composite component in BIP). Figure 8.2 depicts the corresponding composite component of system Task with partial-state semantics $\Gamma(Worker_1, Worker_2, Worker_3, Generator)$, where each $Worker_i$ for $i \in [1..3]$ is identical to the component $Worker$ and $Generator$ is the component depicted in Fig. 8.2.

- Figure 8.1a depicts a model of component $Generator$\(^1\) defined as follows:
  
  - $Generator.P = \{deliver[\emptyset], newtask[\emptyset], \beta\}$.
  - $Generator.L = \{hold, delivered, \bot\}$,
  - $Generator.T = \{(hold, deliver, true, [\ ], \bot), (\bot, \beta, true, [\ ], delivered),
  (delivered, newtask, true, [\ ], \bot), (\bot, \beta, true, [\ ], hold)\}$,
  - $Generator.X = \emptyset$.

- Figure 8.1b depicts a model of component $Worker$ defined as follows:

\(^1\)For the sake of simpler notation, the variables attached to the ports are not shown.
To simplify the depiction of these components, we represent each busy location \( \Gamma = \) component in BIP. The corresponding distributed atomic component is obtained based on the layers, i) the distributed atomic components, ii) the distributed schedulers, and iii) the conflict resolution interactions since they are executed within the same scheduler. In this section, we briefly represent a distributed BIP model which has been extensively studied in [5, 6, 18]. A distributed BIP model contains three layers, i) the distributed atomic components, ii) the distributed schedulers, and iii) the conflict resolution protocol components. The distributed version of the atomic components in BIP is obtained based on the original atomic component as per Definition 8.2 (p. 98) with the difference that a separate port \( \beta^j \) associated to each corresponding scheduler \( S_j \in S \) is added.

**Definition 8.7 (Distributed atomic component in BIP).** Let \( B = (P \cup \{\beta\}, L \cup L^\perp, T, X) \) be an atomic component in BIP. The corresponding distributed atomic component is \( B^d = (P^d, L^d, T^d, X^d) \), such that:

- \( P \cup \{\beta^j \mid j \in [1..|S|] \cap B \in \text{scope}(S_j) \} \subseteq P^d \),
- \( L \cup \{l^\perp \mid j \in [1..|S|] \cap B \in \text{scope}(S_j) \cap l \in L \} \subseteq L^d \),
- \( X^d = X \cup @ \), where @ is the set of variables used for conflict resolution purposes,
- For each set of transitions \( \{(l, p, g_r, [\cdot], l'), (l', \beta, \text{true}, f_r, l'')\} \in T \), we include a new set of transitions \( \{(l, p, g_r, [\cdot], l'^R[1]), (l'^R[1], \beta R[1], \text{true}, f_r, l'^R[2]), \ldots (l'^R[n], \beta R[n], \text{true}, [\cdot], l'')\} \in T^d \) with

### 8.2 Distributed BIP Model

In Section 8.1 (p. 98) we defined a multi-threaded BIP model relying on a single centralized scheduler for executing all multi-party interaction. Such models allow parallelism between computation in the components. However, concurrency between interactions is not possible (i.e., simultaneous execution of several interactions) since they are executed within the same scheduler. In this section, we briefly represent a distributed BIP model which has been extensively studied in [5, 6, 18]. A distributed BIP model contains three layers, i) the distributed atomic components, ii) the distributed schedulers, and iii) the conflict resolution protocol components. The distributed version of the atomic components in BIP is obtained based on the original atomic component as per Definition 8.2 (p. 98) with the difference that a separate port \( \beta^j \) associated to each corresponding scheduler \( S_j \in S \) is added.
Figure 8.3: Worker$^d$

Figure 8.4: Communication messages among Worker$^d_1$ and schedulers $S_1$ and $S_2$

$R$ is a $n$-tuple where $n = |\{j \in [1..|S|] \mid B \in \text{scope}(S_j)\}|$ such that $\forall m \in [1..n]. R[m] \in \{j \in [1..|S|] \mid B \in \text{scope}(S_j)\}$ and $\forall k \in [1..n-1]. R[k] < R[k+1]$.

These transitions send the update state of the component to the corresponding schedulers.

**Definition 8.8** (Distributed scheduler in BIP). Scheduler $S$ is defined as a tuple $(P, L, T, X)$, such that:

- $P$ is the set of ports such that $\{p^i_a \mid a \in Int \land \text{managed}(a) = S \land B_i \in \text{scope}(S) \land i \in [1..|B|]\} \cup \{\beta_i \mid B_i \in \text{scope}(S) \land i \in [1..|B|]\} \subset P$,
- $L$ is the set of locations,
- $X$ is the set of variables,
- $T \subseteq L \times P \times L$ is the set of transitions.

Port $p^i_a \in S.P$ is associated to the notification of component $B_i \in B$ involved in interaction $a \in Int$. When the scheduler triggers an interaction, a set of notifications is sent to the component involved in the interaction. Port $\beta_i$ is associated to the notification of the scheduler of the update state of component $B_i \in B$. 

Example 8.9 (Distributed BIP model). We present the distributed version of the model depicted in Figure 8.2 (p. 100) using two schedulers $S_1$ ans $S_2$ in charge of the execution of interactions. We partition the set of interactions $\Gamma$ into two classes $\Gamma_1 = \{ex_{12}, ex_{13}, ex_{23}\}$ and $\Gamma_2 = \{r_1, r_2, r_3, f_1, f_2, f_3, n_i\}$ where $\text{managed}(\Gamma_1) = S_1$ and $\text{managed}(\Gamma_2) = S_2$. Schedulers are connected with the distributed components in their scope. Figure 8.3 (p. 102) shows the distributed version of component $Worker$ presented in Figure 8.1b (p. 100). Each transition in $Worker^d$ corresponding to the execution of an action is followed by two busy transitions labeled by $\beta^1$ and $\beta^2$ sending the updated state to schedulers $S_1$ and $S_2$ respectively. Figure 8.4 (p. 102) shows the messages sent by the schedulers in order to trigger the associated actions in component $Worker_1^d$ and the messages send by component $Worker_1^d$ to notify the schedulers about its updated state. Components $Worker_2^d$, $Worker_3^d$ and $Generator^d$ are connected to the schedulers similarly (for the sake of simpler presentation they are not shown).

Remark 8.10. We do not present the model of the third level, i.e., the conflict resolution protocol layer. The main reason is that we deal only with the executions of the interactions and the busy actions of components. Resolving the (possible) existing conflicts i) takes place before these executions which is considered as the internal process of the schedulers and ii) does not influence the satisfaction of the property.
Monitoring Multi-Threaded BIP Models

Chapter abstract

In this chapter, we apply our monitoring approach on multi-threaded BIP models. We define explicitly how the proposed instrumentation can be adapted to a BIP composite component with concurrent behavior. We present the actual algorithms used in the instrumented system to reconstruct global states. We introduce RVMT-BIP, a prototype tool implementing the monitoring approach.

In this chapter, we consider a multi-threaded BIP model $M^\perp = \Gamma(B_1, \ldots, B_{|B|})$ of behavior $(Q^\perp, \Gamma, \rightarrow)$, one can obtain the corresponding sequential BIP model $M^s = \text{Int}(B_1, \ldots, B_{|B|})$ of behavior $(Q, \text{Int}, \rightarrow_s)$ (see Section 4.3, p. 41). In the sequel, we consider a multi-threaded BIP model $M^\perp$ and its corresponding sequential model version $M^s$. Intuitively, from any trace of $M^\perp$, we want to reconstruct on-the-fly the corresponding trace in $M^s$ and evaluate a property which is defined over global states of $M^s$.

9.1 Model Transformation to Construct the Witness Trace

We propose a model transformation of a composite component $M^\perp = \Gamma(B_1, \ldots, B_{|B|})$ such that it can produce the witness trace on-the-fly. The transformed system can be plugged to a runtime monitor as described in [39]. Our model transformation consists of three steps: 1) instrumentation of atomic components (Section 9.1.1, p. 106), 2) adding of a new component, that is the reconstructor of global trace (RGT), which implements Definition 7.6, p. 65 (Section 9.1.2, p. 107), and 3) modification of interactions in $\Gamma$ such that (i) component RGT can interact with the other components in the system and (ii) new interactions connect RGT to a runtime monitor (Section 9.1.3, p. 111). Moreover, we prove the correctness of the model
transformation (Section 9.1.4, p. 113).

9.1.1 Instrumentation of Atomic Components

Given an atomic component as per Definition 8.2 (p. 98), we instrument this atomic component such that it is able to transfer its state through port \( \beta \). The state of an instrumented component is delivered each time the component moves out from a busy location. In the following instrumentation, the state of a component is represented by the values of variables and the current location.

**Definition 9.1** (Instrumenting an atomic component). Given an atomic component in partial-state semantics \( B = (P \cup \{\beta\}, L \cup L^\perp, T, X) \) with initial location \( l_0 \in L \), we define a new component \( B^r = (P^r, L \cup L^\perp, T^r, X^r) \) where:

- \( X^r = X \cup \{loc\} \), \( loc \) is initialized to \( l_0 \);
- \( P^r = P \cup \{\beta^r\} \), with \( \beta^r = \beta[X^r] \);
- \( T^r = \{(l, p, g_r, [ \_ ], l^\perp_r), (l^\perp_r, \beta, \text{true}, t_r; [loc := l'], l') \mid \{(l, p, g_r, [ \_ ], l^\perp_r), (l^\perp_r, \beta, \text{true}, t_r, l') \} \subseteq T\} \).

In \( X^r \), \( loc \) is a variable containing the current location. \( X^r \) is exported through port \( \beta \). An assignment is added to the computation step of each transition to record the location.

**Example 9.2** (Instrumenting an atomic component). Figure 9.2 (p. 107) shows the instrumented version of atomic components in system Task (depicted in Figure 8.1, p. 100).

- Figure 9.2a depicts component task generator, where
  
  \[
  Generator^r.P^r = \{\text{deliver}[,] \_ \}, \text{newtask}[,] \_ \}, \beta[\{loc\}], \}
  \]

  \[
  Generator^r.T^r = \{(\text{hold}, \text{deliver}, \text{true}, [ \_ ], \_ ), (\_ , \beta, \text{true}, [loc := delivered], \text{delivered}), (\text{delivered}, \text{newtask}, \text{true}, [ \_ ], \_ ), (\_ , \beta, \text{true}, [loc := hold], \text{hold})\}, \]

  \[
  Generator^r.X^r = \{loc\}. \]

---

**Diagram:**

9.1.1 Instrumentation of Atomic Components

- Initial Model With Partial States
- Building RGT Atom
- Atomic Components Instrumentation
- Modification of Interactions
- Model With Partial States That Produces the Witness Trace

Figure 9.1: Model transformation
9.1. Model Transformation to Construct the Witness Trace

9.1.2 Creating a New Atomic Component to Reconstruct Global States

Let us consider a composite component \( \mathbf{M} = \Gamma(B_1, \ldots, B_{|B|}) \) with partial-state semantics, such that:

- \( \text{init} = (q_0^1, \ldots, q_0^{|B|}) \) is the initial state,
- \( \text{Int} \) is the set of interactions in the corresponding composite component with global-state semantics such that \( \text{Int} = \Gamma \setminus \{ \{ \beta_i \} \}_{i=1}^{|B|} \), and
- the corresponding instrumented atomic components \( B_1^r, \ldots, B_{|B|}^r \) have been obtained through Definition 9.1 (p. 106) such that \( B_i^r \) is the instrumented version of \( B_i \).

We define a new atomic component, called RGT, which is in charge of accumulating the global states of the system \( \mathbf{M} \). Component RGT is an operational implementation, as a component of function RGT (Definition 7.6, p. 65). At runtime, we represent a global state as a tuple consisting of the valuation of variables and the location for each atomic component. After a new interaction gets fired, component RGT builds a new tuple using the current states of components. Component RGT builds a sequence with the generated tuples. The stored tuples are updated each time the state of a component is updated. Following Definition 9.1 (p. 106), atomic components transfer their states through port \( \beta \) each time they move from a busy location to a ready location. RGT reconstructs global states from these received partial states and
delivers them through the dedicated ports.

Definition 9.3 (RGT atom). Component RGT is defined as \((P, L, T, X)\) where:

- \(X = \bigcup_{i \in [1..|B|]} \{B_i^r.X^r\} \cup \{\{B_i^r.X^c\} \cup \{\{B_i^r.X^r\} \cup \{\{B_i^r.X^c\} \cup \{\{B_i^r.X^r\} \cup \{\{B_i^r.X^c\} \cup \{V, v, m\}, where \(B_i^r.X^c\) is a set containing a copy of the variables in \(B_i^r.X^r\).

- \(P = \bigcup_{i \in [1..|B|]} \{\beta_i[B_i^r.X^r]\} \cup \{p_a[\emptyset] | a \in \text{Int}\} \cup \{p'_a[\bigcup_{i \in [1..|B|]} \{B_i^r.X^c\}] | a \in \text{Int}\}.

- \(L = \{l\} is a set with one control location.

- \(T = T_{\text{new}} \cup T_{\text{upd}} \cup T_{\text{out}} is the set of transitions, where:
  - \(T_{\text{new}} = \{(l, p_a, \wedge_{a \in \text{Int}} (\neg gs_a), \text{new}(a), l) | a \in \text{Int}\},
  - T_{\text{upd}} = \{(l, \beta_i, \wedge_{a \in \text{Int}} (\neg gs_a), \text{upd}(i), l) | i \in [1..|B|]\},
  - T_{\text{out}} = \{(l, p'_a, gs_a, \text{get}, l) | a \in \text{Int}\}.

\(X\) is a set of variables that contains the following variables:

- the variables in \(B_i^r.X^r\) for each instrumented atomic component \(B_i^r\);
- a Boolean variable \(gs_a\) that holds \(\text{true}\) whenever a global state corresponding to interaction \(a\) is reconstructed;
- a tuple \((z_1, \ldots, z_{|B|})\) of Boolean variables initialized to \(\text{false}\);
- an \((|B|+1)\)-tuple \(v = (v_1, \ldots, v_{|B|}, v_{|B|+1}). \)

For each \(i \in [1..|B|]\), \(z_i\) is \(\text{true}\) when component \(i\) is in a busy location and \(\text{false}\) otherwise. For \(i \in [1..|B|]\), \(v_i\) is a state of \(B_i^r\) and \(v_{|B|+1} \in \text{Int}\). \(V\) is a sequence of \((|B|+1)\)-tuples initialized to \((q^0_1, \ldots, q^0_{|B|}, -)\). \(m\) is an integer variable initialized to 1.

\(P\) is a set of ports.

- For each atomic component \(B_i^r\) for \(i \in [1..|B|]\), RGT has a corresponding port \(\beta_i\). States of components are exported to RGT through this port.

- For each interaction \(a \in \text{Int}\), RGT has two corresponding ports \(p_a\) and \(p'_a\). Port \(p_a\) is added to interaction \(a\) (later in Definition 9.5, p. 111) in order to notify RGT when a new interaction is fired. A reconstructed global state which is related to the execution of interaction \(a\), is exported to a runtime monitor through port \(p'_a\).

RGT has three types of transitions:
9.1. MODEL TRANSFORMATION TO CONSTRUCT THE WITNESS TRACE

- The transitions labeled by port $p_a$, for $a \in \text{Int}$, are in $T_{\text{new}}$. When no reconstructed global state can be delivered (that is, the Boolean variables in $\{ gs_a \mid a \in \text{Int} \}$ are false), the transitions occur when the corresponding interaction $a$ is fired.

- The transitions labeled by port $\beta_i$, for $i \in [1..|B|]$, are in $T_{\text{upd}}$. When no reconstructed global state can be delivered, to obtain the state of component $B_i$, these transitions occur at the same time transition $\beta$ occurs in component $B_i$.

- The transition labeled by port $p'_a$ for $a \in \text{Int}$ are in $T_{\text{get}}$. If RGT has a reconstructed global state corresponding to the global state of the system after executing interaction $a \in \text{Int}$, these transitions deliver the reconstructed global state to a runtime monitor.

RGT uses three algorithms.

Algorithm $\text{new}$ (see Algorithm 9.1) implements the case of function $\text{acc}$ that corresponds to the occurrence of a new interaction $a \in \text{Int}$ (Definition 7.6, p. 65). It takes $a \in \text{Int}$ as input and then: 1) sets $z_i$ to true if component $i$ is involved in interaction $a$, for $i \in [1..|B|]$; 2) fills the elements of the $(|B|+1)$-tuple $v$ with the states of components after the execution of the new interaction $a$ in such a way that the $i$th element of $v$ corresponds to the state of component $B_i$. Moreover, the state of busy components is null. The $(|B|+1)$th element of $v$ is dedicated to interaction $a$, as a record specifying that tuple $v$ is related to the execution of $a$; 3) appends $v$ to $V$.

Algorithm 9.1 $\text{new}(a)$

1: \textbf{for} $i = 1 \rightarrow |B|$ \textbf{do} \hfill \triangleright \text{Check if component } B_i \text{ is involved in interaction } a.
2: \quad \textbf{if } B_i \cdot P \cap a \neq \emptyset \text{ then} \quad \triangleright \text{In case component } B_i \text{ is busy, } z_i \text{ is } \text{true.}
3: \quad \quad z_i := \text{true} \hfill \triangleright \text{The } i\text{th element of tuple } v \text{ is represented by } v_i.
4: \quad \quad v_i := \text{null} \hfill \triangleright v_i \text{ receives the state of } B_i^r.
5: \quad \textbf{else} \hfill \triangleright v_i \text{ receives the state of } B_i^r.
6: \quad \quad v_i := B_i^r \cdot X^r \hfill \triangleright v_i \text{ receives the state of } B_i^r.
7: \quad \textbf{end if} \hfill \triangleright \text{Last element of } v \text{ receives interaction } a.
8: \textbf{end for} \hfill \triangleright v \text{ is added to } V.
9: v_{|B|+1} := a
10: V := V \cdot v

Algorithm $\text{upd}$ (see Algorithm 9.2, p. 110) implements the case of function $\text{acc}$ which corresponds to the occurrence of transition $\beta$ of atomic component $B_i$ for $i \in [1..|B|]$. According to Definition 9.1 (p. 106), the current state of the instrumented atomic component $B_i^r$ for $i \in [1..|B|]$ is exported through port $\beta$ of $B_i^r$. Algorithm $\text{upd}$ takes the current state of $B_i^r$ and looks into each element of $V$ and replaces null values which correspond to $B_i^r$ with the current state of $B_i^r$. Finally, algorithm $\text{upd}$ invokes algorithm $\text{check}$ to check the elements of $V$. If any tuple of $V$, associated to $a \in \text{Int}$, becomes a global state and has no null
element, then the corresponding Boolean variable $gs_a$ is set to true.

**Algorithm 9.2** $\text{upd}(i)$

1: $z_i := \text{false}$  
2: for $j = 1 \rightarrow \text{length}(V)$ do  
3: if $V(j)_i == \text{null}$ then \hfill $\triangleright$ The $i^{th}$ element of the $j^{th}$ tuple in $V$ is represented by $V(j)_i$.  
4: $V(j)_i := B^r_i.X^r$ \hfill $\triangleright$ Update the null states.  
5: end if  
6: end for  
7: $\text{check()}$ \hfill $\triangleright$ Check the elements of sequence $V$ (cf. Algorithm 9.3)

**Algorithm 9.3** $\text{check()}$

1: for $i = m \rightarrow \text{length}(V)$ do \hfill $\triangleright$ Check those tuples of $V$ which have not been delivered to the monitor.  
2: if $\neg gs(V(i)|B|+1)$ then  
3: $b_{\text{tmp}} := \text{true}$ \hfill $\triangleright$ Make a temporary boolean variable initialized to true.  
4: for $j = 1 \rightarrow |B|$ do  
5: $b_{\text{tmp}} := b_{\text{tmp}} \land (V(i)_j \neq \text{null})$ \hfill $b_{\text{tmp}}$ remains true until a null is found in the $i^{th}$ tuple of $V$.  
6: end for  
7: $gs(V(i)|B|+1) := b_{\text{tmp}}$ \hfill $\triangleright$ Update the value of Boolean $gs$ associated to $V(i)|B|+1$.  
8: end if  
9: end for

Algorithm get (see Algorithm 9.4) is called whenever component RGT has a reconstructed global state to deliver. Algorithm get takes the $m^{th}$ tuple in $V$ and copies its values into $\{B^r_i.X^r\}_{i=1}^{|B|}$ and then increments $m$. Finally, algorithm get calls algorithm check in order to update the value of the Boolean variables $gs_a$ for $a \in \text{Int}$, because there are possibly several reconstructed global states associated to an interaction $a \in \text{Int}$. In this case, after delivering one of those reconstructed global states and resetting $gs_a$ to false, one must again set variable $gs_a$ to true for the rest of the reconstructed global states associated to interaction $a$. Note, to facilitate the presentation of proofs in Appendix A (p. 151), component RGT is defined in such a way that it does not discard the reconstructed global states of the system after delivering them to the monitor. In our actual implementation of RGT, these states are discarded because they are not useful after
being delivered to the monitor. At runtime, \( RGT.V \) contains the sequence of global states associated with
the witness trace (as stated later by Proposition 9.9, p. 115).

**Example 9.4 (Component RGT).** Figure 9.3 depicts the component RGT for system Task. For readability, only one instance of each type of transitions is shown. The execution of a new interaction \( a \in \{ ex_{12}, ex_{13}, ex_{23}, r_1, r_2, r_3, f_1, f_2, f_3, n_t \} \) in system Task is synchronized with the execution of transition \( p.a \) of the component RGT which applies the algorithm \( \text{new} \). Each busy interaction in the system Task is synchronized with the execution of transition \( \beta_i \) (\( i \in [1..4] \) are the indexes of the four components in system Task) which applies the algorithm \( \text{upd} \) to update the reconstructed states so far and check whether or not a new global state is reconstructed. Transition \( \beta_i, i \in [1..4] \), is guarded by \( \beta_i \) which ensures the delivery of the new reconstructed global state through the ports \( p.a \) as soon as they are reconstructed. At runtime, RGT produces the sequence of global states in the right-most column of Table 7.1 (p. 66).

### 9.1.3 Connections

After building component RGT (see Definition 9.3, p. 108) and instrumenting atomic components (see Definition 9.1, p. 106), we modify all interactions and define new interactions to build a new transformed composite component. To let RGT accumulate states of the system, first we transform all the existing interactions by adding a new port to communicate with component RGT, then we create new interactions that allow RGT to deliver the reconstructed global states of the system to a runtime monitor.

Given a composite component \( M^\perp = \Gamma(B_1, \ldots, B_{|B|}) \) with corresponding component RGT and instrumented components \( B^\tau = (P \cup \{ \beta^\tau \}, L \cup L^\perp, T^\tau, X^\tau) \) such that \( B^\tau = B_i^\tau \in \{ B_1^\tau, \ldots, B_{|B|}^\tau \} \), we define a new composite component.
Definition 9.5 (Composite component transformation). For a BIP model $M^r = \Gamma(B_1, \ldots, B_{|B|})$, we introduce a corresponding transformed model $M' = \Gamma'(B'_1, \ldots, B'_{|B|}, RGT)$ such that $\Gamma' = a^r_\gamma \cup a^r_\beta \cup a^m$ where:

- $a^r_\gamma$ and $a^r_\beta$ are the sets of transformed interactions such that:
  $$\forall a \in \Gamma . a'^r = \begin{cases} a \cup \{RGT.p_a\} & \text{if } a \in Int \\ a \cup \{RGT.\beta_i\} & \text{otherwise } (a \in \{\beta_i\}_{i \in [1..|B|]}) \end{cases}$$
  $$a^r_\gamma = \{a'^r \mid a \in Int\}, a^r_\beta = \{a'^r \mid a \in \{\beta_i\}_{i \in [1..|B|]}\}$$

- $a^m$ is a set of new interactions such that:
  $$a^m = \{a' \mid a \in Int\} \text{ where } \forall a \in Int . a' = \{RGT.p'_a\} \text{ is a corresponding unary interaction.}$$

For each interaction $a \in \Gamma$, we associate a transformed interaction $a'^r$ which is the modified version of interaction $a$ such that a corresponding port of component RGT is added to $a$. Instrumenting interaction $a \in Int$ does not modify sequence of assignment $F_a$, whereas instrumenting busy interactions $a \in \{\beta_i\}_{i=1}^{B_i}$ adds assignments to transfer attached variables of port $\beta_i$ to the component RGT. The transformed interactions belong to two subsets, $a^r_\gamma$ and $a^r_\beta$. The set $a^m$ is the set of all unary interactions $a'$ associated to each existing interaction $a \in Int$ in the system.

The set of the states of transformed composite component $B'$ is $Q' = B'_1.Q \times \ldots \times B'_{|B|}.Q \times RGT.Q$.

Example 9.6 (Transformed composite component). Figure 9.4 shows the transformed composite component of system Task. The goal of building $a'$ for each interaction $a$ is to enable RGT to connect to a runtime.
monitor. Upon the reconstruction of a global state corresponding to interaction \( a \in \text{Int} \), the corresponding interaction \( a' \) delivers the reconstructed global state to a runtime monitor.

**Example 9.7 (Monitoring system Task).** Figure 9.6 (p. 114) depicts the transformed system Task with a monitor for the homogeneous distribution of the tasks among the workers, where \( e_1 \), \( e_2 \), and \( e_3 \) are events related to the pairwise comparison of the number of executed tasks by Workers. For \( i \in [1..3] \), event \( e_i \) evaluates to true whenever \( |x(i \mod 3)+1-x_i| \) is lower than 3 (for this example). Component Monitor evaluates \((e_1 \land e_2 \land e_3)\) upon the reception of a new global state from RGT and emits the associated verdict till reaching bad state \( \bot \). The global trace \((\text{free}, \text{free}, \text{free}, \text{hold}) \cdot ex_1 \cdot (\text{done}, \text{done}, \text{free}, \text{delivered}) \cdot n_t \) (see Table 7.1, p. 66) is sent by component RGT to the monitor which in turn produces the sequence of verdicts \( \top_c \cdot \top_c \)(where \( \top_c \) is verdict “currently good”, see [9, 35]).

**9.1.4 Correctness of the Transformations**

Combined together, the transformations preserve the semantics of the initial model as stated in the rest of this section.

Intuitively, the component RGT defined in Definition 9.3 (p. 108) implements function RGT defined in Definition 7.6 (p. 65). Reconstructed global states can be transferred through the ports \( p'_{a} \) with \( a \in \text{Int} \). If interaction \( a \) happens before interaction \( b \), then in component RGT, port \( p'_{a} \) which contains the reconstructed global state after executing \( a \) will be enabled before port \( p'_{b} \). In other words, the total order between executed interactions is preserved.

In the transformed composite component \( \Gamma^r(B^r_1, \ldots, B^r_{|B|}; RGT) \), the notion of equivalence is used to relate the tuples constructed by component RGT to the states of the initial system in partial-state semantics. Below, we define the notion of equivalence between an \((|B|+1)\)-tuple \( v = (v_1, \ldots, v_{|B|}, v_{|B|+1}) \) and a state of the system \( q = (q_1, \ldots, q_{|B|}) \) such that, for \( i \in [1..|B|] \), \( v_i \) is a state of \( B^r_i \) and \( v_{|B|+1} \in \text{Int} \).

**Definition 9.8 (Equivalence of an \((|B|+1)\)-tuple and a state).** A \((|B|+1)\)-tuple \( v = (v_1, \ldots, v_{|B|}, v_{|B|+1}) \)
is equivalent to a state \( q = (q_1, \ldots, q_{\vert B \vert}) \) if:

\[
\forall i \in [1..\vert B \vert], v_i = \begin{cases} q_i & \text{if } q_i \in Q_i, \\ \text{null} & \text{otherwise} \end{cases}
\]

When an \((\vert B \vert + 1)\)-tuple \( v \) is equivalent to a state \( q \), we denote it by \( v \equiv q \).

A tuple \((v_1, \ldots, v_{\vert B \vert}, v_{\vert B \vert + 1})\) and a state \((q_1, \ldots, q_{\vert B \vert})\) are equivalent if \( v_i = q_i \) for each position \( i \) where the state \( q_i \) of component \( B_r^i \) is also a state of the initial model, and \( v_i = \text{null} \) otherwise. The notion of equivalence is extended to traces and sequences of \((\vert B \vert + 1)\)-tuples. A trace \( t = q_0.a_1.q_1' \ldots a_k.q_k' \) and a sequence of \((\vert B \vert + 1)\)-tuples \( V = v(0) \cdot v(1) \ldots v(k) \) are equivalent, denoted \( t \equiv V \), if \( q_j' \) is equivalent to
v(j) for all j ∈ [0 .. k] and v(j)|_{B+1} = a_j for all j ∈ [1 .. k].

**Proposition 9.9** (Correctness of component RGT). ∀t ∈ Tr(M\). RGT.V ≡ acc(event(t)).

Proposition 9.9 states that, for any trace t, at any time, variable RGT.V encodes the witness trace acc(event(t)) of the current trace: RGT.V is a sequence of tuples where each tuple consists of the state and the interaction that led to this state, in the same order as they appear on the witness trace.

**Proof.** The proof is done by induction on the length of t ∈ Tr(M\), i.e., the trace of the system in partial-state semantics. The proof is given in Appendix A.1.5 (p. 157).

For each trace resulting from an execution with partial-state semantics, component RGT produces a trace of global states which is the witness of this trace in the initial model.

**Definition 9.10** (State stability). State (l, v) ∈ RGT.Q is said to be stable when ∀x ∈ {RGT.gs_a | a ∈ Int}.v(x) = false.

A state q in the the semantics of atomic component RGT is said to be stable when all Boolean variables in set {RGT.gs_a | a ∈ Int} evaluate to false with the valuation of variables in state q. In other words, the current state of component RGT is stable when it has no reconstructed global states to deliver. We say that the composite component B′ is stable when the state of its associated component RGT is stable.

**Example 9.11** (Stable state). We illustrate Definition 9.10 based on the execution trace in Table 7.1 (p. 66). By the evolution of system Task from step 4 to step 5, component RGT reconstructs the global state associated to the execution of ex12 and respectively sets boolean variable gs_{ex12} to true. Once gs_{ex12} becomes true, we say that the state of the component RGT is not stable. In component RGT, the execution of transition labeled by port p′_{ex12} delivers the reconstructed global state (i.e., (done, done, free, delivered)) to the monitor and sets boolean variable gs_{ex12} to false. Consequently, component RGT becomes stable. We say that component RGT is not stable whenever there exists at least one reconstructed global state which has not been delivered to the monitor. Whenever component RGT is not stable, we say that the system is not stable as well.

The following lemma states a property of the algorithms in Section 9.1.2 (p. 107) ensuring that whenever component RGT has reconstructed some global states, it transmits them to the monitor before the system can execute any new partial state can be created.

**Lemma 9.12.** In any state of the transformed system, if there is a non-empty set GS ⊆ {RGT.gs_a | a ∈ Int} in which all variables are true, the variables in {RGT.gs_a | a ∈ Int} \ GS cannot be set to true until all variables in GS are reset to false first.
The following lemma states that any state of the composite component $B^r$ can be stabilized by executing interactions in $a^m$.

**Lemma 9.13.** We shall prove that for any state $q \in Q^r$, there exists a state $q' \in Q^r$ reached after interactions in $a^m$ (i.e., $q \xrightarrow{(a^m)^*} q'$), such that $q'$ is a stable state (i.e., $\text{stable}(q')$).

We define a notion of equivalence between states of the transformed model and states of the initial system.

**Definition 9.14 (Equivalent states).** Let $q^r = (q^r_1, \ldots, q^r_{|B|}, q^r_{|B|+1}) \in Q^r$ be a state in the transformed model where $q^r_{|B|+1}$ is the state of component RGT, function $\text{equ} : Q^r \rightarrow Q^\bot$ is defined as follows: $\text{equ}(q^r) = q$, where $q = (q_1, \ldots, q_{|B|})$. ($\forall i \in [1..|B|]. q^r_i = q_i$) $\land \text{stable}(q^r_{|B|+1})$.

A state in the initial model is said to be equivalent to a state in the transformed model if the state of each component in the initial model is equal to the state of the corresponding component in transformed model and the state of component RGT is stable.

The following lemma is a direct consequence of Definition 9.14. The lemma states that, if an interaction is enabled in the transformed model, then the corresponding interaction is enabled in the initial model when the states of two models are equivalent.

**Lemma 9.15.** For any two equivalent states $q \in Q^\bot$ and $q^r \in Q^r$ (i.e., $\text{equ}(q^r) = q$), if interaction $a \in \Gamma$ is enabled in state $q$, then $a^r \in \Gamma^r$ is enabled at state $q^r$.

Based on the above lemmas, we can now state the correctness of our transformations.

**Theorem 9.16 (Transformation Correctness ).** $\Gamma(B_1, \ldots, B_{|B|}) \sim \Gamma'(B^r_1, \ldots, B^r_{|B|}, \text{RGT})$.

**Proof.** The proof relies on exhibiting a bi-simulation relation between the set of states of transformed model $M' = \Gamma'(B^r_1, \ldots, B^r_{|B|}, \text{RGT})$, that is $Q^r$, and the set of states of $M^\bot = \Gamma(B_1, \ldots, B_{|B|})$, that is $Q^\bot$. The proof is given in Appendix A.1.7 (p. 160).

Combined together, Theorem 9.16 and Lemma 9.15 imply that, for each state in the initial system, there exists an equivalent state in the transformed system in which all enabled interactions in the initial system are also enabled in the transformed system. Hence, we can conclude that the transformed system is as concurrent as the initial system.

Consequently, we can substantiate our claims stated in the introduction about the transformations: instrumenting atomic components and adding component RGT (i) preserves the semantics and concurrency of the initial model, and (ii) verdicts are sound and complete.
Remark 9.17 (Alternative RGT atoms). In the definition of atom RGT (Definition 9.3, p. 108), one can observe that whenever component RGT has reconstructed global states to deliver, the system cannot proceed and must wait until all the reconstructed global states are sent (because of the guards of transitions $T_{upd}$ and $T_{new}$). This gives precedence to monitoring rather than to the evolution of the system.

Three alternative definitions of RGT can be considered by changing the guards of the transitions in $T_{new}$ and $T_{upd}$. For both transitions, by suppressing the guards, one gives less precedence to the transmission of reconstructed global states. By suppressing the guards in transitions in $T_{new}$, we let the system starting a new interaction while there may be still some reconstructed global states for RGT to deliver. By suppressing the guards in transitions in $T_{upd}$, we let the system execute $\beta$-transitions while there may be still some reconstructed global states for RGT to deliver.

Suppressing these guards favors the performance of the system but may delay the transmission of global states to the monitor and thus it may also delay the emission of verdicts. There is thus a tradeoff between the performance of the system and the emission of verdicts.

9.2 Implementation of Witness Trace Construction

We implemented our monitoring approach in a tool called RVMT-BIP. RVMT-BIP is a prototype tool implementing the algorithms presented in Section 9.1 (p. 105).

Architecture of RVMT-BIP. RVMT-BIP (Runtime Verification of Multi-Threaded BIP) is a Java implementation of ca. 2,200 LOC. RVMT-BIP is integrated in the BIP tool suite [6]. The BIP (Behavior, Interaction, Priority) framework is a powerful and expressive framework for the formal construction of het-
erogeneous systems. RVMT-BIP takes as input a BIP CBS and a monitor description for a property, and outputs a new BIP system whose behavior is monitored against the property while running concurrently. RVMT-BIP uses the following modules:

- Module *Atomic Transformation* takes as input the initial BIP system and a monitor description. From the input abstract monitor description, it extracts the list of components, and the set of their states and variables that influence the truth-value of the property and are used by the monitor. Then, this module instruments the atomic components in the extracted list so as to observe their states and the values of the variables. Finally, the transformed components and the original version of the components that do not influence the property are returned as output.

- Module *Building RGT* takes as input the initial BIP system and a monitor description and produces component RGT (Reconstructor of Global Trace) which reconstructs and accumulates global states at runtime to produce “on-the-fly” the global trace.

- Module *Building Monitor* takes as input the initial BIP system and a monitor description and then outputs the atomic component implementing the monitor (following [39]). Component *Monitor* receives and consumes the reconstructed global trace generated by component RGT at runtime and emits verdicts.

- Module *Connections* constructs the new composite and monitored component. The module takes as input the output of the *Atomic Transformation*, *Building RGT* and *Building Monitor* modules and then outputs a new composite component with new connections. The new connections are purposed to synchronize instrumented components and component RGT in order to transfer updated states of the components to RGT. Instrumented components interact with RGT independently and concurrently.
Monitoring Distributed BIP Models

Chapter abstract

In this chapter, we apply the instrumentation method presented in Chapter 6 (p. 49) on a distributed BIP model. The instrumented model generates events at runtime which are used for computation lattice construction. The application of our monitoring approach, presented in Section 7.2 (p. 69), is introduced as a tool RVDIST, which receives events and evaluates the behavior of a distributed system on-the-fly in the observer level.

10.1 Model Transformation of Distributed BIP Models

We apply the instrumentation presented in Chapter 6 (p. 49) on a distributed BIP model defined in Section 8.2 (p. 101). The vector clock exchange between a shared component and the associated schedulers can be applied on the existing communication channels of a distributed BIP model. Indeed, we merge the controllers (defined in Section 6.1, p. 50) with the distributed components and the required data variables are attached to their existing ports.

Definition 10.1 (Transformation of a scheduler of a distributed BIP model). For a scheduler $S = (P, L, T, X)$ as per Definition 8.8 (p. 102), we define an instrumented scheduler $S^r = (P^r, L^r, T^r, X^r)$ such that:

- $X^r = X \cup \{vc\} \cup \{b_i \mid i \in [1..|B|] \wedge B_i \in B_s\}$ where $vc$ is a $|S|$-tuples and $b_i$ is a Boolean variable,
- $P^r = P \cup \{p^e_a \mid a \in Int \wedge managed(a) = S\} \cup \{p^s_{\beta_i} \mid i \in scope(S)\}$.
- For each set of transitions $\{(l, p^l_a, g, f, l') \mid B_i \in involved(a) \wedge B_i \in B_s\} \in T$ associated to
the notification of shared components involved in interaction $a$, we include a corresponding set
$\{ (l, p'_d, g, f; [b_i := \text{true}], l') | B_i \in \text{involved}(a) \land B_i \in B_s \} \in T^r$ such that port $p'_d$ carries the value of variable $vc$.

- For each set of transitions $\{ (l, p'_d, g, f, l') | B_i \in \text{involved}(a) \} \in T$ associated to the notification of the components involved in interaction $a$, we include a new transition $(l''', p'_d, \text{true}, [vc[j] := vc[j] + 1], l''') \in T^d$ to send the action event to the observer, where $j \in [1..|S|]$ is the index of scheduler $S$.

- For each transition $(l, \beta, g, f, l') \in T$ associated to the reception of the update state of shared component $B_i$, we include a transition $(l, \beta, g, f; [vc = \max(vc, vc')], l') \in T^r$ such that port $\beta$ carries vector clock $vc'$ stored in the shared component.

- For each transition $(l, \beta, g, f, l') \in T$, associated to the reception of the update state of shared component $B_i$, we include new transition $(l'', p'_{\beta}, (g \land b_i), f; [b_i := \text{false}], l'') \in T^r$ to send the update event to the observer.

- For each interaction $(l, \beta, g, f, l') \in T$, associated to the reception of the update state of non-shared component $B_i$, we include new transition $(l'', p'_{\beta}, g, f, l'') \in T^r$ send the update event to the observer.

We define transformation of distributed shared atomic component.

**Definition 10.2** (Transformation of distributed shared atomic component). For a distributed shared atomic component $B^d = (P^d, L^d, T^d, X^d)$ as per Definition 8.7 (p. 101), we define an instrumented component $B^{dr} = (P^{dr}, L^d, T^{dr}, X^{dr})$ such that $X^{dr} = X^d \cup \{vc\}$ where $vc$ is a $|S|$-tuples. The set of the ports $P^{dr} = P^d$ with the difference that ports in $P^{dr}$ carry the value of variable $vc$. The set of transitions $T^{dr}$ is defined as follows:

- For each transition $(l, p, g, f, l') \in T^d$ associated to a notification from a scheduler $S_j$ to execute an action (i.e., $p \in P$) we include transition $(l, p, g, f; [vc = \max(vc, vc')], l') \in T^{dr}$ where $vc'$ is the value of the vector clock of scheduler $S_j$ carried with port $p$ and sent to the shared component.

- For each transition $(l, \beta, g, f, l') \in T^d$ associated to notifying scheduler $S_j$ about the updated state of the shared component, we include transition $(l, \beta, g, f, l') \in T^{dr}$ where port $\beta$ carries the value of $vc$ and sends it to scheduler $S_j$.

We define a new atomic component in charge of accumulating the events sent from schedulers.

**Definition 10.3** (Observer). Component observer is defined as a tuple $O = (P, L, T, X)$

$P = \{p_a | a \in \text{Int} \} \cup \{\beta_i | i \in [1..|B|] \land j \in [1..|S|] \land B_i \in \text{scope}(S_j)\}$ is the set of ports,
10.1. MODEL TRANSFORMATION OF DISTRIBUTED BIP MODELS

- $L = \{l\}$ is the set consists of one location,

- $X$ is the set of variables.

- $T = \{(l, p, \text{true}, \text{MAKE}, l) \mid p \in P\}$ is the set of transitions.

Ports are dedicated to events. The observer receives action events associated to the execution of interaction $a \in \text{Int}$ through port $p_a$ so that the corresponding vector clock is carried with the port. The observer receives update events associated to the $\beta$ actions of the components. Port $\beta^i_j$ receives an update state of component $B_i$ which is sent by scheduler $S_j$ to the observer. The reception of each event triggers a transition which calls Algorithm $\text{MAKE}$ (see Algorithm 7.1, p. 75) augmented with formula progression defined in Definition 7.41 (p. 86).

Example 10.4 (Transformation of distributed BIP model). Figure 10.1 depicts the transformed version of distributed BIP model depicted in Figure 8.9 (p. 102). In this example we only show the communications among the transformed shared component $\text{Worker}^{dr}_1$, schedulers and observer. In component $\text{Worker}^{dr}_1$, ports carry the value of vector clock, either to receive from a scheduler (i.e., $vc'$) or to send the local vector clock (i.e., $vc$). For the sake of simpler presentation, the new assignments of the transitions labeled by the ports are shown next to the ports name. Generating the associated events is done using the existing send/receive channels and the extra messages are due to sending the generated events to the observer.

Figure 10.1: Model transformation of Example 8.9 (p. 102)
10. MONITORING DISTRIBUTED BIP MODELS

10.2 Implementation of Computation Lattice Construction

We present an implementation of our monitoring approach in a tool called RVDIST. RVDIST is a prototype tool implementing the algorithms presented in Section 7.2 (p. 69).

RVDIST Work-Flow. RVDIST is a prototype tool implementing algorithm MAKE enhanced with formula progression technique presented in Section 7.2.5 (p. 85), written in the C++ programming language. It consists of roughly 900 lines of code. RVDIST takes as input a configuration file describing the architecture of the distributed system (i.e., number of schedulers, number of components, initial state, LTL formula to be monitored, mapping of atomic propositions to components) and a list of events. Whenever an event is given to RVDIST, it invokes the monitoring algorithm MAKE (Algorithm 7.1, p. 75) to construct the computation lattice and simultaneously carries out the monitoring by formula progression over the constructed lattice. RVDIST outputs the evaluation of the constructed lattice by reporting the number of observed events, the number of existing nodes of the constructed lattice, the number of nodes which have been removed from the lattice due to optimizing the size of the lattice, the vector clock of the frontier node, the number of paths from the initial node to the frontier node which have been monitored (the set of all compatible global-traces), the set of formulas associated to the frontier node. Figure 10.2 depicts the work-flow of RVDIST.

RVDIST works with a sequence of events, and events can be produced from any distributed system with the semantics that can be modeled with LTSs. Therefore, RVDIST can be easily adapted to runtime monitor many distributed systems, either component-based or monolithic. One need to instrument the system to generate the decent events format.
Chapter abstract

In this chapter, we present the results of a set of experiments to evaluate our monitoring algorithms presented in Chapter 7 (p. 61). We present the systems and properties used in our case studies. We make experiments on four multi-threaded systems with RVMT-BIP and on two distributed systems with RVDIST, where each system is monitored against dedicated properties. We present the experimental results and discuss about the obtained results in terms of the performance of the monitoring techniques.

11.1 Evaluation of Monitoring Multi-threaded CBS

11.1.1 Case Studies

We present some case studies on executable BIP systems conducted with RVMT-BIP.

Process Completion of System Demosaicing

Demosaicing is an algorithm for digital image processing used to reconstruct a full color image from the incomplete color samples output from an image sensor. Figure 11.1 (p. 124) shows a simplified version of the processing network of Demosaicing. Demosaicing contains a Splitter and a Joiner process, a pre-demosaicing (Demopre) and a post-demosaicing (Demopost) process and three internal demosaicing Demo processes that run in parallel. The real model contains ca. 1,000 lines of code, consists of 26 atomic components interacting through 35 interactions. We consider two specifications related to process completion:
Figure 11.1: Processing network of system Demosaicing

\( \varphi_1 \): It is necessary that all the internal demosaicing units finish their process before the post-demosaicing unit starts processing. The post-demosaicing unit receives the output results of internal demosaicing units through port \textit{getimg}. We add variable \textit{port} to record the last executed port. Each demosaicing unit has a boolean variable \textit{done} which is set to \texttt{true} whenever the demosaicing process completes. This requirement is formalized as property \( \varphi_1 \) defined by the automaton depicted in Figure [11.2a](p. 125) where the events are \( e_1 : \text{Demopost.port} == \text{getimg} \) and \( e_2 : (\text{Demo}_1.\text{done} \land \text{Demo}_2.\text{done} \land \text{Demo}_3.\text{done}) \). From the initial state \( s_1 \), the automaton moves to state \( s_2 \) when all the internal demosaicing units finish their process. Receiving the processed images by post-demosaicing causes a move from state \( s_2 \) to \( s_1 \).

\( \varphi_2 \): Moreover, internal demosaicing units (\( \text{Demo}_1, \text{Demo}_2, \text{Demo}_3 \)) should not start the demosaicing process until the pre-demosaicing unit finishes its process. The pre-demosaicing unit sends its output to the internal demosaicing units through port \textit{transmit} and each internal demosaicing unit starts the demosaicing process by executing a transition labeled by port \textit{start}. This requirement is formalized as property \( \varphi_2 \) which is defined by the automaton depicted in Figure [11.2b](p. 125) where \( e_1 : \text{Demopre.port} == \text{transmit} \), \( e_2 : \text{Demo}_1.\text{port} == \text{start} \), \( e_3 : \text{Demo}_2.\text{port} == \text{start} \) and \( e_4 : \text{Demo}_3.\text{port} == \text{start} \). From the initial state \( s_1 \), whenever the pre-demosaicing unit transmits its processed output to the internal demosaicing units, the automaton moves to state \( s_2 \). Internal demosaicing units can start in different order. Moreover, all demosaicing units must eventually start their internal process and the automaton reaches state \( s_{12} \). From state \( s_{12} \), the automaton moves back to state \( s_2 \) whenever the pre-demosaicing unit sends the next processed data to the internal demosaicing units.

**Data-freshness of System Reader-WriterV1**

System Reader-WriterV1 (ca. 130 LOC) consists of a set of independent composite components. Each composite component consists of four components: a Reader, a Writer, a Clock, and a Poster (in total, 12 components and 9 interactions). Reader and Writer communicate with each other through Poster. The
data generated by *Writer* is written in a *Poster* that can be accessed by *Reader*. The Reader-Writer model is presented in Figure 11.3 (p. 126). We consider a specification related to data freshness:

\[ \varphi_3: \text{It is necessary that the data is up-to-date: component } \text{Reader} \text{ must read recent data produced by component } \text{Writer}. \text{ If } t_2 \text{ and } t_1 \text{ are the moments of reading and writing actions respectively, then the difference between } t_2 \text{ and } t_1 \text{ must be less than a specific duration } \delta, \text{ i.e., } (t_2 - t_1) \leq \delta. \text{ In the model, the time counter is implemented by a component } \text{Clock}, \text{ and the } \text{tick} \text{ transition occurs every second. This requirement is formalized as property } \varphi_3 \text{ which is defined by the automaton depicted in Figure 11.4a (p. 127), where } \delta = 2, e_1 \text{: Writer.port == write, } e_2 \text{: Clock.port == tick and } e_3 \text{: Reader.port == read. Whenever Writer writes into Poster, the automaton moves from the initial state } s_1 \text{ to } s_2. \text{ When Reader reads Poster, the automaton moves from } s_2 \text{ to } s_1. \text{ Reader is allowed to read Poster after one tick transition. In this case, the automaton moves from } s_2 \text{ to } s_3 \text{ after the tick, and then moves from } s_3 \text{ to } s_1 \text{ after reading Poster. } \varphi_3 \text{ also allows to read Poster after two tick transitions. In this case, the automaton moves from } s_2 \text{ to } s_4 \text{ after the first tick, then moves from } s_4 \text{ to } s_3 \text{ on the second tick, and finally moves from } s_3 \text{ to } s_1 \text{ after reading Poster.}

**Execution Order of System Reader-WriterV2**

System Reader-WriterV2 (ca. 150 LOC) is a more complex version of Reader-WriterV1 and involves several writers. This system has six components: *Reader, Writer_1, Writer_2, Writer_3, Clock and Poster*. The Writers are synchronized. *Reader* and Writers communicate with each other through *Poster*. The data generated by each writer is written to *Poster* and can then be accessed by *Reader*. Having several writers, a more complex specification on the execution order can be defined. We consider a specification related to the execution order:
\( \varphi_4 \): The writers should periodically write data to a poster in a specific order. The specification concerns 3 writers: Writer_1, Writer_2 and Writer_3. During each period, the writing order must be as follows: Writer_1 writes to the poster first, then Writer_2 can write only when Writer_1 has finished writing to the poster, Writer_3 can write only when Writer_2 finishes writing to the poster, and the same goes on for the next periods. To do so, each writer is assigned a unique id that is passed to the poster when it starts using the poster. This id is then used to determine the last writer that used the poster. For example, when Writer_2 wants to access the poster, it has to check whether the id stored in the poster corresponds to Writer_1 or not.

This requirement is formalized as property \( \varphi_4 \) which is defined by the automaton depicted in Figure 11.4b (p. 127) where:

\[
\begin{align*}
\text{– } e_1 &: (\text{Writer}_1.\text{port} == \text{write} \land \text{Poster}.\text{port} == \text{write} \land \text{Clock}.\text{port} == \text{getTime}), \\
\text{– } e_2 &: (\text{Writer}_2.\text{port} == \text{write} \land \text{Poster}.\text{port} == \text{write} \land \text{Clock}.\text{port} == \text{getTime}), \\
\text{– } e_3 &: (\text{Writer}_3.\text{port} == \text{write} \land \text{Poster}.\text{port} == \text{write} \land \text{Clock}.\text{port} == \text{getTime}).
\end{align*}
\]

When Writer_1 writes to the poster, the automaton moves from initial state \( s_1 \) to state \( s_2 \). From state \( s_2 \), the automaton moves to state \( s_3 \) when Writer_2 writes to the poster. From state \( s_3 \), the automaton moves to the initial state \( s_1 \) when Writer_3 writes to the poster. This writing order must always be followed.

Distribution of Tasks in System Task

We consider our running example system Task and a specification of the homogeneous distribution of the tasks among the workers:
11.1. EVALUATION OF MONITORING MULTI-THREADED CBS

11.1.2 Evaluation Principles

For each system and all its properties, we synthesized a BIP monitor following [38, 39] and combined it with the CBS output from RVMT-BIP. We obtain a new CBS with corresponding RGT and monitor components. We run each system by using various numbers of threads and observe the execution time. Executing these systems with a multi-threaded controller results in a faster run because the systems benefit from the parallel threads. Additional steps are introduced in the concurrent transitions of the system. Note, these are asynchronous with the existing interactions and can be executed in parallel. These systems can also execute with a single-threaded controller which forces them to run sequentially. Varying the number of threads allows us to assess the performance of the (monitored) system under different degrees of parallelism. In particular, we expected the induced overhead to be insensitive to the degree of parallelism. For instance, an undesirable behavior would have been to observe a performance degradation (and an overhead increase) which would mean either that the monitor sequentializes the execution or that the monitoring infrastructure

\( \varphi_5 \): The satisfaction of this specification depends on the execution time of each worker. Different tasks may have different execution times for different workers. Obviously, the faster a worker completes each task, the higher is the number of its accomplished tasks. After executing a task, the value of the variable \( x \) of a worker is increased by one. Moreover, the absolute difference between the values of variable \( x \) of any two workers must always be less than a specific integer value (which is 3 for this case study). This requirement is formalized as property \( \varphi_5 \) which is defined by the automaton depicted in Figure 11.5 where\( e_1 : |worker_1.x - worker_2.x| < 3 \), \( e_2 : |worker_2.x - worker_3.x| < 3 \) and \( e_3 : |worker_1.x - worker_3.x| < 3 \). The property holds as long as \( e_1, e_2 \) and \( e_3 \) hold.
Table 11.1: Results of monitoring Demosaicing, Reader-WriterV1, Reader-WriterV2, and Task with RVMT-BIP

<table>
<thead>
<tr>
<th>system</th>
<th># executed interactions</th>
<th>execution time and overhead according to the number of threads</th>
<th># events</th>
<th># extra executed interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Demosaicing (26,35)</td>
<td>1,300</td>
<td>18.98</td>
<td>10.24</td>
<td>7.75</td>
</tr>
<tr>
<td>Demosaicing (27,69)</td>
<td>3,051</td>
<td>19.02</td>
<td>11.53</td>
<td>8.17</td>
</tr>
<tr>
<td>Demosaicing (27,46)</td>
<td>1,850</td>
<td>&lt;0.1%</td>
<td>7.9%</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>Reader-WriterV1 (12,9)</td>
<td>120,000</td>
<td>61.48</td>
<td>29.67</td>
<td>20.03</td>
</tr>
<tr>
<td>ReaderWriterV1 (13,12)</td>
<td>200,000</td>
<td>1.6%</td>
<td>63.29</td>
<td>21.96</td>
</tr>
<tr>
<td>Reader-WriterV2 (6,7)</td>
<td>20,000</td>
<td>5.8%</td>
<td>32.06</td>
<td>21.45</td>
</tr>
<tr>
<td>ReaderWriterV2 (7,12)</td>
<td>85,000</td>
<td>13.90</td>
<td>22.72</td>
<td>13.77</td>
</tr>
<tr>
<td>Task (4,10)</td>
<td>399,999</td>
<td>32.1%</td>
<td>59.88</td>
<td>60.01</td>
</tr>
<tr>
<td>Task (5,16)</td>
<td>600,197</td>
<td>5.7%</td>
<td>123.98</td>
<td>71.73</td>
</tr>
</tbody>
</table>

is not suitable for multi-threaded systems. We also extensively tested the functional correctness of RVMT-BIP, that is whether the verdicts of the monitors are sound and complete.

11.1.3 Results and Conclusions

**Performance evaluation.** Table [11.1] and Table [11.2] (p. 129) report the timings obtained when checking the following specifications: complete process property on Demosaicing, data freshness and execution ordering property on Reader-Writer systems, and task distribution property on Task, with RVMT-BIP and RV-BIP respectively. Each measurement is an average value obtained over 100 executions of these systems. In these tables, the columns have the following meanings:

- Column *system* indicates the systems. System in italic format represents the monitored version of the initial system. Moreover, (x, y) in front of the system name means that x (resp. y) is the number of components (resp. interactions) of the system. The monitored property is written below each monitored system name with a value (z) which indicates that z components have variables influencing the truth-value of the property (and were thus instrumented by RVMT-BIP or RV-BIP).

- Column *# executed interactions* indicates the number of interactions executed by the engine which also represents the number of functional steps of the system.
Table 11.2: Results of monitoring Demosaicing, Reader-WriterV1, Reader-WriterV2, and Task with RV-BIP

<table>
<thead>
<tr>
<th>System</th>
<th># Executed Interactions</th>
<th>Execution Time and Overhead w.r.t. Different Number of Threads</th>
<th># Events</th>
<th># Extra Executed Interactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demosaicing (26,35)</td>
<td>1,300</td>
<td>18.98 10.24 7.75 6.85 6.58 6.09 6.33 6.45 6.29 6.27</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Demosaicing (27,37)</td>
<td>2,450</td>
<td>3.5% 167% 316% 376% 402% 429% 392% 384% 369% 407%</td>
<td>1,300</td>
<td>1,300</td>
</tr>
<tr>
<td>Reader-WriterV1 (12,9)</td>
<td>120,000</td>
<td>61.48 29.67 20.03 20.00 20.05 20.21 20.60 21.54 21.92 22.13</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Reader-WriterV2 (6,7)</td>
<td>20,000</td>
<td>32.06 21.45 12.04 11.37 11.33 11.44 11.49 11.53 11.58</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Task (4,10)</td>
<td>399,999</td>
<td>117.28 70.18 60.91 60.06 58.98 60.01 60.93 61.77 63.13 65.45</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Task (5,12)</td>
<td>500,197</td>
<td>121.61 70.12 72.25 75.11 75.66 80.54 81.62 84.58 89.65 90.21</td>
<td>100,198</td>
<td>100,198</td>
</tr>
</tbody>
</table>

- Columns **Execution time and overhead according to the number of threads** report (i) the execution time of the systems when varying the number of threads and (ii) the overhead induced by monitoring (for monitored systems).

- Column **Events** indicates the number of reconstructed global states (events sent to the associated monitor).

- Column **Extra executed interactions** reports the number of additional interactions (i.e., execution of interactions which are added into the initial system for monitoring purposes).

As shown in Table 11.1 (p. 128), using more threads reduces significantly the execution time in both the initial and transformed systems. Comparing the overheads according to the number of threads shows that the proposed monitoring technique (i) does not restrict the performance of parallel execution and (ii) scales up well with the number of threads.

**Performance comparison of RV-BIP and RVMT-BIP.** To illustrate the advantages of monitoring multi-threaded systems with RVMT-BIP, we compared the performance of RVMT-BIP and RV-BIP ([39]); see Table 11.1 (p. 128) and Table 11.2 for the results. Monitoring with RV-BIP amounts to use a standard runtime verification technique, i.e., not tailored to multi-threaded systems. At runtime, the RV-BIP monitor consumes the global trace (i.e., sequence of global states) of the system (where global snapshots are obtained...
by synchronization among the components) and yields verdicts regarding property satisfaction. It has been shown in [39] that RV-BIP efficiently handles CBSs with sequential executions.

In the following, we highlight some of the main observations and draw conclusions:

1. Fixing a system and a property, the number of events received by the monitors of RV-BIP and RVMT-BIP are similar, because both techniques produce monitored systems that are observationally equivalent to the initial ones [39, 72]. Moreover, increasing the number of threads does not change the global behavior of the system, therefore the number of events is not affected by the number of threads.

2. Fixing a system and a property, the number of extra interactions imposed by RVMT-BIP is greater than the one imposed by RV-BIP. In the monitored system obtained with RVMT-BIP, after the execution of an interaction, the components that are involved in the interaction and influencing the truth-value of the property independently send their updated state to component RGT (whenever their internal computation is finished). In the monitored system obtained with RV-BIP, after the execution of an interaction influencing the truth value of the property, all the updated states will be sent at once (synchronously) to the component monitor. Hence, the evaluation of an event in RV-BIP is done in one step and the number of extra interactions imposed by RV-BIP is the same as the number of monitored events (see Table 11.2, p. 129).

3. In spite of the higher number of extra interactions imposed by RVMT-BIP, during a multi-threaded execution, the fewer synchronous interactions of monitored components imposed by RV-BIP induces a significant overhead. This phenomenon is especially visible for the two most concurrent systems: Demosaicing and Task.

4. *On the independence of components:* Consider systems Demosaicing and Task, which consist of independent components with low-level synchronization and high degree of parallelism, and for which the monitored property requires the states of these independent components. On the one hand, at runtime, RV-BIP imposes synchronization among the components whose execution influences the truth value of the property and the component monitor. It results in a loss of the performance when executing with multiple threads. On the other hand, RVMT-BIP collects updated states of the components independently right after their state update. Consequently, with RVMT-BIP, the system performance in the multi-threaded setting is preserved (systems Demosaicing and Task) as a negligible overhead is observed. This is a usual and complex problem which depends on many factors such as platform, model, external codes, compiler, etc. This renders the computation of the number of threads leading to peak performance complex.
5. *Synchronization of independent components*: In RV-BIP, the thread synchronizations and the synchronization of components with the monitor induce a huge overhead especially when concurrent component are concerned with the desired property (system Demosaicing and property $\varphi_1$).

6. *Synchronized components*: We observe that, for system ReaderWriterV2, the overhead obtained with RVMT-BIP monitor is slightly higher than the one obtained with RV-BIP monitors. Indeed, system ReaderWriterV2 consists of 3 writers synchronized by a clock component. Moreover, property $\varphi_4$ is defined over the states of all the writers. As a matter of fact, if one of the writers needs to communicate with component RGT, then all the other writers need to wait until the communication ends. That is, when the concurrency of the monitored system is limited by internal synchronizations, the global-state reconstruction performed by RVMT-BIP is less effective than the technique used by RV-BIP from a performance point of view.

7. *Synchronized components in independent composite components*: If the initial system (i) consists of independent composite components working concurrently, (ii) the components in each composite are highly synchronized (low degree of parallelism in each composition) and (iii) the desired property is defined over the states of the components of a specific composite component, then RVMT-BIP performs similarly to RV-BIP. Indeed, in the monitored system, the independent entities (i.e., composite component) are able to run as concurrently as in the initial system and the overhead is caused by the synchronized components. However, by increasing the number of threads, RVMT-BIP monitors offer better performance (system Reader-WriterV1).

11.2 Evaluation of Monitoring Distributed CBS

We present the evaluation of our monitoring approach on two case studies carried out with RVDIST.

11.2.1 Case Studies

We present a realistic example of a robot navigation and a model of two phase commit protocol (TPC). The two case studies are executable BIP systems.

Robotic Application

We consider a navigating robot system consisting of a set of modules ROBLOCO, ROBLASER, ROBMAP and ROBMOTION. ROBLOCO is in charge of the robot low-level controller to track the speed and the position of the robot using three main units *controller*, *signal* and *manager*. The motor *controller* receives
data from the motor signal. In parallel, the manager, which is associated to the odo task activities to measure the distance traveled by the robot, reads the signals from the encoders on the wheels and produces a current position. ROBLASER is in charge of the laser to produce the free space in the laser’s range tagged with the position where the scan has been made. ROBMAP aggregates the successive scanned data. ROBMOTION has one task plan which, given a goal position, computes the appropriate speed to reach it using the current position, and avoiding obstacles. We deal with the most complex module, i.e., ROBLOCO, involving three schedulers in charge of the execution of the dedicated actions. ROBLOCO (ca. 1200 LOC) has 34 components (with a total number of 265 ports and 193 control locations) and 117 multi-party interactions synchronizing the actions of components. Three schedulers are in charge of the execution of the dedicated actions.

To prevent deadlocks in the system, it is required that whenever the controller is free, at some point in future, the signal must start sending data to the controller before the manager starts managing a new odo task. The deadlock freedom requirement can be defined as LTL formula \( \varphi_1 \).

\[
\varphi_1: G(\text{ControlFree} \implies (X\neg\text{ManagerStartodo} U\text{SignalStart})).
\]

**Two Phase Commit (TPC)**

We consider the distributed transaction commit [52] problem where a set of nodes called resource managers \( \{rm_1, \ldots, rm_n\} \) have to reach agreement on whether to commit or abort a distributed transaction. Resource managers are able to locally commit or abort a transaction based on a local decision. In a fault-free system, it is required the global system to commit as a whole if each resource manager has locally committed, and to abort as a whole if any of the resource managers has locally aborted. In case of global abort, locally-committed resource managers may perform roll-back steps to undo the effect of the last transaction [89].

Two phase commit protocol is a solution proposed by [51] to solve the transaction commit problem. It uses a transaction manager that coordinates between resource managers to ensure they all reach one global decision regarding a particular transaction. The global decision is made by the transaction manager based on the feedback from resource managers after making their local decision (LocalCommit/LocalAbort).

The protocol, running on a transaction, uses a client, a transaction manager and a non-empty set of resource managers which are the active participants of the transaction. The protocol starts when client sends remote procedure to all the participating resource managers. Then each participating resource manager \( rm_i \) makes its local decision based on its local criteria and reports its local decision to transaction manager. LocalCommit\(_i\) is true if resource manager \( rm_i \) can locally-commit the transaction, and LocalAbort\(_i\) is true if resource manager \( rm_i \) cannot locally-commit the transaction. Each participant resource managers
11.2. EVALUATION OF MONITORING DISTRIBUTED CBS

Table 11.3: Results of monitoring ROBLOCO and TPC with RVDIST

| System | property | $|\varphi|$ | # observed events | # lattice size | frontier node VC |
|--------|----------|------------|------------------|---------------|-----------------|
|        |          |            |                  | optimized | not-optimized |                |
| ROBLOCO| $\varphi_1$ | 3          | 3463             | 17        | 10602          | (730,352,485)  |
|        | $\varphi_2$ | 10         | 4709             | 11        | 2731           | (402,402,402,601) |
| TPC    | $\varphi_3$ | 26         |                  |           |                |                |

stays in wait location until it hears back from transaction manager whether to perform a global commit or abort for the current transaction. After all local decisions have been made and reported to the transaction manager, the latter makes a global decision (GlobalCommit/GlobalAbort) that all the system will agree upon. When GlobalCommit is true, the system will globally-commit as a whole, and it will abort as a whole when GlobalAbort is true. We consider two specifications related to TPC protocol correctness:

$$
\varphi_2: G (\bigwedge_{i=1}^n \neg \text{LocalAbort}_i \Rightarrow X (\neg \text{LocalAbort}_i \land \neg \text{LocalCommit}_i) \land \text{GlobalAbort})
$$

$$
\varphi_3: G (\bigwedge_{i=1}^n \text{LocalCommit}_i \Rightarrow X (\bigwedge_{i=1}^n (\neg \text{LocalAbort}_i \land \neg \text{LocalCommit}_i)) \land \text{GlobalCommit})
$$

Property $\varphi_2$ states that, sending locally abort in any resource managers for a current transaction implies the global abort (GlobalAbort) on that transaction before the resource manager locally aborts or commits the next transaction, that is, none of the resource managers commit. Property $\varphi_3$ states that, if all the resource managers send locally commit for a current transaction, then all the resource managers commit the transaction (GlobalCommit) before the resource managers locally aborts or commits the next transaction.

For each system we applied the model transformation defined in Section 6.1 and run them in a distributed setting. Each instrumented system produces a sequence of event which is generated and sent from the controllers of its schedulers. The events are sent to the RVDIST where the associated configuration file is already given. Upon the reception of each event, RVDIST applies the online monitoring algorithm introduced in Section 7.2.5 and outputs the result consists of the information stored in the constructed computation lattice and evaluation of the desired LTL property so far.

11.2.2 Results and Conclusions

Although we can not compare the performance of RVDIST with RVMT-BIP or RV-BIP in terms of execution time, we comment about the results obtained by monitoring the two case studies.

Table 11.3 and Figure 11.6 (p. 135) present the results checking specifications deadlock freedom on ROBLOCO and protocol correctness on TPC. The columns of the table have the following meanings:
– Column $|\varphi|$ shows the size of the monitored LTL formula. Note, the size of formulas are measured in terms of the operators entailment inside it, e.g., $G(a \land b) \lor Xc$ is of size 2.

– Column observed event indicates the number of action/update events sent by the controllers of the schedulers.

– Column lattice size reports the size of constructed lattice using optimization algorithm is used vs. the size of constructed lattice when non-optimized algorithm is used.

– Column frontier node VC indicated the vector clock associated to the frontier node of the constructed lattice.

Figures 11.6a, 11.6b (p. 135) show how the size of constructed lattice varies in two systems as they evolve. Having shared components in system is not the only reason to have a small lattice size, what is more important is how often the shared components are used as a part of executed interactions. The more execution with shared components results the more dependencies in the generated events and thus the smaller lattice size.

In system ROBLOCO, after receiving 3463 events, the size of the obtained computation lattice is 17, whereas the size of non-optimized lattice is 10602 which is quite large in terms of storage space and iteration process. It shows how efficient our optimization algorithm minimize and optimize the monitoring process. Figure 11.6a (p. 135) shows how the size of constructed lattice varies over time with the evolution of ROBLOCO system. Although what we need in the constructed computation lattice as the verdicts of the monitor output is only stored in the frontier node, but the rest of the nodes are necessary to be kept at runtime in order to extend the lattice in case of reception of new events.

In system TPC we also obtained a very small size of the lattice after the reception of 4709 events. As it is shown in Table 11.3 (p. 133) the size and complexity of the LTL property does not change the structure of the constructed lattice, it only effects on the progression process. The frontier-node vector clock shows how many interactions have been executed by each scheduler at the end of the system run.

Our monitoring algorithm implemented in RVDIST provide a lightweight tool to runtime monitor the behavior of a distributed CBS. RVDIST keeps the size of the lattice small even for a long run. The model instrumentation defined in Chapter 6 (p. 49) can be adapted on distributed BIP models, so that the communication between the controllers are done through the existing send/receive message channels. The extra messages are induced by to the communication between the controllers of schedulers and the observer. For each event the observer receives a message corresponding to the event.
11.2. Evaluation of Monitoring Distributed CBS

Discussion: Shared Component vs. Lattice Size

In the following we investigate how the number of shared components affects the size of the computation lattice over a very simple example of a distributed CBS with multiparty interactions.

Example 11.1 (Shared component, lattice size). Let us consider a component-based system with four independent components $Comp_1, \ldots, Comp_4$. Each component has two actions $Action_1, Action_2$ which are designed to only be executed with the following order: $Action_1, Action_2, Action_1$ and then the component terminates. We distribute the execution of actions using four schedulers $Sched_1, \ldots, Sched_4$. For the sake of simplicity, we consider each action of the components as a singleton interaction of the system such that $Act = \{Comp_i, Action_1, Comp_i, Action_2 \mid i \in \{1 \ldots 4\}\}$. Each scheduler manages a subset of $Act$. We define
various partitioning of the interactions to obtain the following settings:

1. Each scheduler manages the actions of only one component, such that the set of actions $\text{Comp}_i.Action_1$ and $\text{Comp}_i.Action_2$ are managed by Scheduler $Sched_i$, for $i \in [1..4]$. In this setting, no component is in the scope of more than one scheduler.

2. We consider setting 1 with the only difference that action $\text{Comp}_1.Action_2$ is managed by scheduler $Sched_2$. In this setting, $\text{Comp}_1$ is a shared component.

3. We consider setting 2 with the only difference that action $\text{Comp}_2.Action_2$ is managed by scheduler $Sched_3$. In this setting, $\text{Comp}_1$ and $\text{Comp}_2$ are shared components.

4. We consider setting 3 with the only difference that action $\text{Comp}_3.Action_2$ is managed by scheduler $Sched_4$. In this setting, $\text{Comp}_1$, $\text{Comp}_2$ and $\text{Comp}_3$ are shared components.

5. We consider setting 4 with the only difference that action $\text{Comp}_4.Action_2$ is managed by scheduler $Sched_1$. In this setting, all the components are shared components.

We design the components to be involved only in three interactions. Therefore, the execution of such a system in any aforementioned setting generates the same amount of action/update events, that is 12 action events, 12 update events (24 events in total), no matter which scheduler manages which interaction. The only difference of the events obtained in those different settings is the vector clock of the action events (their logical dependencies) and the sender of action/update events. Table 11.4 and Figure 11.7 (p. 136) represent the results with respect to the above-mentioned settings. Columns in Table 11.4 have the following meaning:
Table 11.4: Results of lattice construction w.r.t different settings of Example 11.1 (p. 135)

<table>
<thead>
<tr>
<th># shared component</th>
<th># lattice nodes</th>
<th># removed nodes</th>
<th># paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>175</td>
<td>81</td>
<td>10,681,263</td>
</tr>
<tr>
<td>1</td>
<td>88</td>
<td>72</td>
<td>1,616,719</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
<td>60</td>
<td>572,847</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
<td>52</td>
<td>316,035</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>47</td>
<td>251,177</td>
</tr>
</tbody>
</table>

- Column *shared component* indicates the number of shared components in each setting.
- Column *lattice nodes* shows the number of the nodes of the constructed lattice in each setting.
- Column *removed nodes* indicates the number of removed nodes in the lattice using the optimization algorithm.
- Column *path* indicates the number of paths of the constructed computation lattice.

Considering the first setting (no shared component), the execution of the system generates a set of concurrent (independent) action events, in the sense that any two action events from two different schedulers are not causally related. Therefore, RVDIST constructs a complete (maximal) computation lattice in order to cover all the compatible global traces. The size of the constructed computation lattice as well as the number of paths of the lattice decreases when considering more shared components (see Figure 11.7 p. 136).
Part III

Discussion
Chapter abstract
In this chapter, we compare our monitoring approach with related work in runtime verification of multi-threaded and distributed systems.

12.1 Runtime Verification of Multi-Threaded Systems

Several approaches are related to the runtime verification of multi-threaded CBS, as they either target CBSs or address the problem of concurrently runtime verifying systems.

Runtime Verification of Single-threaded CBSs  Dormoy et al. proposed an approach to runtime check the correct reconfiguration of components at runtime \[27\]. They propose to check configurations over a variant of RV-LTL where the usual notion of state is replaced by the notion of component configuration. RV-LTL is a 4-valued variant of LTL dedicated to runtime verification introduced in \[9\] and used in \[34\]. Our approach offers several advantages compared to the approach in \[27\]. First, our approach is not bound to temporal logic since it only requires a monitor written as a finite-state machine. This state-machine can be then generated by several already existing tools (e.g., Java-MOP) since it uses a generic format to express monitors. Thus, existing monitor synthesis algorithms from various specification formalisms can be reused, up to a syntactic adaptation layer. Second, the instrumentation of the initial system and the addition of the monitor is formally defined, contrarily to \[27\] where the process is only overviewed. Moreover, our proposed approach for monitoring multi-threaded CBSs leverages the formal semantics of BIP allowing us
to provide a formal proof of the correctness of the approach. All these features confers to our approach a higher-level of confidence.

In [38, 39], the authors proposed a first approach for the runtime verification of CBSs. The approach in [38, 39] takes a CBS and a regular property as input and generates a monitor implemented as a component. Then, the monitor component is integrated within an existing CBS. At runtime, the monitor consumes the global trace (i.e., sequence of global states) of the system and yields verdicts regarding property satisfaction. The technique in [38, 39] only efficiently handles CBSs with sequential executions: if applied to a multi-threaded CBS, the monitor would sequentialize completely the execution. Hence, the approach proposed in this thesis can be used in conjunction with the approach in [38, 39] when dealing with multi-threaded CBSs: (only) the monitor-synthesis algorithm in [38, 39] can be used to obtained a monitor that can be plugged to the RGT component (defined in this thesis) reconstructing the global states of the system.

Synthesizing Correct Concurrent Runtime Monitors In [44], the authors investigate the synthesis of correct monitors in a concurrent setting, whereby (i) the system being verified executes concurrently with the synthesized monitor (ii) the system and the monitor themselves consist of concurrent sub-components. Authors have constructed a formally-specified tool that automatically synthesizes monitors from sHML (adaptation of SafeHML (SHML) a sub-logic of the Hennessy-Milner Logic) formulas so as to asynchronously detect property violation by Erlang programs at runtime. SHML syntactically limits specifications to safety properties which can be monitored at runtime. Our approach is not bounded to any particular logic. Moreover, properties in our approach are not restricted to safety properties but supports co-safety properties, and properties that are neither safety nor co-safety properties. Moreover, the monitored properties can express the desired behavior not only on the internal states of components but also on the states of external interactions.

Decentralized Runtime Verification The approaches in [11, 32, 7] decentralize monitors for linear-time specifications on a system made of synchronous black-box components. Moreover, monitors only observe the outside visible behavior of components to evaluate the formulas at hand. The decentralized monitor evaluates the global trace by considering the locally-observed traces obtained by local monitors. To locally detect global violations and satisfactions, local monitors need to communicate, because their traces are only partial with respect to the global behavior of the system. Given an LTL property \( \varphi \), the objective is to create sound formula derived from \( \varphi \) that can be monitored on each local trace, while minimizing inter-component communication. However, they assume that the projection of the global trace upon each component is well-defined and known in advance. Moreover, all monitors consume events from the trace synchronously.
Inspired by the decentralized monitoring approach to LTL properties in [11], Kouchnarenko and Weber [59] define a progressive FTPL semantics allowing a decentralized evaluation of FTPL formula over component-based systems. Complementarily, Kouchnarenko and Weber [58] propose the use of temporal logics to integrate temporal requirements to adaptation policies in the context of Fractal components [19]. The policies are used for specifying reflection or enforcement mechanisms, which refers respectively to corrective reconfiguration triggered by unwanted behaviors, and avoidance of reconfiguration leading to unwanted states. However, the approaches in [58, 59] fundamentally differ from ours because (i) they target architectural invariants and (ii) our approach is specific to CBSs that can be executed in a multi-threaded fashion. The components in [58, 59] are seen as black boxes and the interaction model considers only unidirectional connections. On the contrary, our approach leverages the internal behavior of components and their interactions for the instrumentation and global-state reconstruction.

**Monitoring Safety Properties in Concurrent Systems** The approach in [84] addresses the monitoring of asynchronous multi-threaded systems against temporal logic formulas expressed in MTTL. MTTL augments LTL with modalities related to the distributed/multi-threaded nature of the system. The monitoring procedure in [84] takes as input a safety formula and a partially-ordered execution of a parallel asynchronous system, and then predicts a potential property violation on one of the causally-consistent interleavings of the observed execution. Our approach mainly differs from [84] in that we target CBSs. Moreover, we assume a central scheduler and we only need to monitor the unique causally-consistent global trace with the observed partial trace. Also, we do not place any expressiveness restriction on the formalism used to express properties.

**Parallel Runtime Verification of Sequential Programs** Berkovich et al. [13, 14] introduce parallel algorithms to speed up the runtime verification of sequential programs against complex LTL formulas using a graphics processing unit (GPU). Berkovich et al. consider two levels of parallelism: the monitor (i) works along with the program in parallel, and (ii) evaluates a set of properties in a parallel fashion. Monitoring threads are added to the program and directly execute on the GPU. The approach in [13, 14] is not tailored to CBSs and is a complementary technique that adds significant computing power to the system to handle the monitoring overhead. Note that, as shown by our experiments, our approach preserves the performance of the monitored system. Finally, our approach is not bound to any particular logic, and allows for Turing-complete monitors.
12. RELATED WORK

12.2 Runtime Verification of Distributed Systems

In Chapter 2, p. 13 we present work on monitoring distributed systems. In this section we compare our approach with these once.

Decentralized LTL Monitoring  A close work to the approach presented in this thesis has been exposed in [11]. The authors presented an algorithm for decentralized monitoring LTL formulas for synchronous distributed systems. Compared to our setting, we target asynchronous distributed component systems. Moreover, we target partial-state semantics CBSs, where the global states of the systems is not available at runtime. Hence, instead of having a global trace at runtime, we are dealing with a set of possible partial traces which possibly could happen during the run of the system.

Model-based runtime analysis of distributed reactive systems.  In [8], the authors presented a framework for detecting and analyzing synchronous distributed systems faults in a centralized manner using local LTL properties that require only the local traces. Compared to our approach, we considered the assumption that the global properties can not be projected and checked on individual components or individual schedulers. Therefore, local traces can not be directly used for verifying the properties.

Reachability analysis on distributed executions.  In [26], the authors presented an algorithm for trace checking of distributed programs by building the lattice of all reachable global states of the distributed system, based on the on-the-fly observation of the partial order of message causality. No monitor has been proposed in their work for the purpose of verification whereas in our algorithm we synthesize a runtime monitor which evaluates on-the-fly the behavior of the system based on the reconstructed computation lattice. Moreover, in our distributed setting, schedulers do not communicate directly by sending-receiving messages, and we deal with partial traces.

Efficient decentralized monitoring of safety in distributed systems.  In [83], the authors designed a method for monitoring safety properties in distributed systems relying on the existing communication among processes. The main noteworthy difference between their work and our work is that our monitoring algorithm is sound, in the sense that we evaluate the behavior of the distributed system based on all of the possible partial traces of the distributed system. In our work, each evaluated trace could have happened as the actual trace of the system, and could have generated the same events.
Efficient online monitoring of LTL properties for asynchronous distributed systems. In [64], the authors define an online monitoring to verify any LTL property on a finite sequence of a distributed system. Our approach mainly differs in that we target distributed CBSs where the traces are defined over the set of the partial states of the system. Nevertheless, their technique in filtering out the irrelevant event (those that do not take part in a monitor move) could be used in our work to decrease the size of the constructed lattice.

Detecting temporal logic predicates on the happened-before model. Authors in [82] used CTL for specifying properties of distributed computation and interpret them on a finite lattice of global states and check that a predicate is satisfied for an observed single execution trace of the program. We deal with a set of events at runtime generated by the schedulers which results in an infinite computation lattice. Although the computation lattice in our method is made of the observed partial states, we could check the satisfaction of temporal predicates defined over the global states of the system, which mean that we could monitor the system even if the global state of the is not defined.
Chapter abstract

In this chapter, we draw the conclusions of our monitoring approaches for multi-threaded and distributed CBSs. We outline avenues for future work, such as decentralization of the proposed monitor for distributed CBSs, using static analysis to optimized the instrumentation and monitoring process, going beyond monitoring by runtime enforcement, and finally extending our on real-time CBSs.

13.1 Summary

In this thesis, we tackled the problem of runtime verification of concurrent CBSs in two different settings: for multi-threaded and distributed CBSs. The goal is to verify the satisfaction or violation of properties referring to the global states of the system on-the-fly. To this end, we introduced an abstract semantic model of CBSs consisting of a set of components, each of which is endowed with a set of actions, and a set of schedulers. Each scheduler is in charge of executing the dedicated subset of multi-party interactions (joint actions). The execution of each interaction triggers the set of actions of corresponding components involved in the interaction. In the concurrent setting, schedulers execute interactions by knowing the partial states of the system. Therefore, the observable trace of the system is a sequence of partial states which is not suitable for verifying global-state properties. Moreover, in the distributed setting the total order of the executions of the interactions is not observable.

Our technique consists of two steps, (i) model instrumentation of the CBSs to extract the events of the system, and (ii) synthesizes a centralized monitor which collects the events, reconstructs the corresponding
global trace(s), and verifies the desired properties on-the-fly.

Monitoring Multi-Threaded CBSs. In the multi-threaded setting, where one scheduler manages the execution of the interactions, the instrumented system outputs a sequence of totally-ordered events. These events encode the partial trace of the system. We proved that for each partial trace of a multi-threaded CBS there exists a unique corresponding global trace, that is the witness trace. The witness trace consists of a sequence of global states that could be observed as if the system was executed in the sequential setting (single-threaded). In this setting, our approach (i) integrates a global trace reconstructor, i.e., a component that receives the events and produces the witness trace at runtime, and (ii) verifies on-the-fly any linear-time property referring to the global state of the system while preserving the performance and benefits from concurrency.

We implemented our approach in a prototype tool RVMT-BIP. We evaluated the performance and functional correctness of RVMT-BIP against four case studies. The experimental results show the effectiveness of our approach and that monitoring with RVMT-BIP induces a cheap overhead at runtime.

Monitoring Distributed CBSs. In the distributed setting, where several schedulers manage the execution of the interactions, the instrumented system outputs a sequence of partially-ordered events. Since the total ordering of the events is not observable, we deal with a set of compatible partial traces of the system. We showed that each compatible partial trace could have occurred as the actual run of the system. Moreover, for each compatible partial trace, there exists a corresponding-compatible global trace (inspiring from the witness trace of a multi-threaded CBS). The set of compatible global traces is represented in the form of a lattice. In this setting, our proposed approach (i) integrates a centralized observer which collects the local events of all schedulers (ii) constructs the computation lattice, and (iii) verifies on-the-fly any LTL property over the constructed lattice. We introduced a novel online LTL monitoring technique on the computation lattice, so that each nodes carries a set of formulas evaluating the set of paths start from the initial node and end up with the node. The set of formulas attached to the frontier node of the constructed lattice represents the evaluation of all the compatible global traces with respect to the given LTL formula.

We implemented our monitoring approach in a prototype tool called RVDIST. RVDIST executes in parallel with the distributed system and takes as input the events generated from each scheduler and outputs the evaluated computation lattice. The experimental results show that, thanks to the optimization applied in the online monitoring algorithm, the size of the constructed computation lattice is insensitive to the the number of received events, and the lattice size is kept reasonable.
13.2 Future Work

Many opportunities for extending the scope of this thesis remain to be explored. This section presents some of these research perspectives.

**Decentralized monitoring.** A first direction for monitoring distributed CBSs is to decentralizing the runtime monitor, such that the satisfaction or violation of specifications can be detected by local monitors alone. For distributed CBSs with large number of schedulers, a centralized algorithm requires a single process to perform high number of computations, and to store very large data. By distributing the monitors we indeed decrease the load of monitoring process on a single process. Indeed, the centralized monitoring execution steps must be decomposed into multiple steps, so that these steps are executed by each local monitor independently.

**Static analysis.** Another direction is to use static analysis to detect a set of global states that can never occur at runtime, so that some nodes of the lattice can be ignored and the paths consisting of these nodes are never explored. Moreover, using static analysis leads to design an adaptive model instrumentation, in the sense that the runtime monitor only receives events that potentially affect the truth value of the property. Thus, the computation lattice size and monitoring load is decreased.

**Runtime enforcement.** Runtime verification might provide a sufficient assurance to check whether or not the desired property is satisfied. However, for some classes of systems e.g., safety-critical systems, a misbehavior might be not acceptable. To prevent this, a possible solution is to enforce the desired property so that the monitor does not only observe the current program execution, but it also controls it in order to ensure that the expected property is fulfilled. Runtime enforcement was initiated by the work of Schneider [79] on security automata. Runtime enforcement consists in using a monitor to watch the current execution sequence and after it whenever it deviates from the property by for instance halting the system. This line of work has been also studied for program monitoring in [62, 63, 41, 74, 40] and for monitoring sequential component-based systems in [37].

**Timed model.** Another possible direction is to extend the proposed framework to runtime verify and enforce timed specifications on timed components. Recently, the real-time multi-threaded and distributed CBSs [92, 93] have been proposed, where each interaction has a timing constraint. In this setting, schedulers take into account the timing constraints before execute any interactions, e.g., before triggering an interaction,
the scheduler has to ensure that no interaction with an earlier deadline is enabled. In [25], the authors presented a technique for augmenting vector clocks with real time to enable better ordering of events.
A.1 Proofs Related to the Approach for Monitoring Multi-Threaded CBS

Before tackling the proof of correctness of our approach, we provide an intuitive description of the proof content. The correctness of our approach relies on three results.

The first result concerns the witness trace. Given a multi-threaded CBS $M^\perp$ with partial-state semantics as per Definition 4.2 (p. 40) and the global-state semantics version $M^\ast$ as per Definition 4.3 (p. 41) such that $M^\perp$ can execute concurrently and which is bi-similar to $M^\ast$. Any trace of an execution of $M^\perp$ can be related to the trace of a unique execution of $M^\ast$, i.e., its witness. Property 7.4 (p. 64) states that any witness trace corresponds to the execution in global-state semantics that has the same sequence of interaction executions, i.e., that the witness relation captures the abovementioned relation between a system in global-state semantics and the corresponding system in partial-state semantics. Property 7.5 (p. 64) states that from any execution in partial-state semantics, the witness exists and is unique.

The second result states that function $RGT$ builds the witness trace from a trace in partial-state semantics in an online fashion. Theorem 7.14 (p. 68) states the correctness of this function.

The third result states that the transformed components, the synthesized components, and their connection are correct. That is, the obtained system (i) computes the witness and implements function $RGT$ (Proposition 9.9, p. 115), and (ii) is bisimilar to the initial system (Theorem 9.16, p. 116).

Proof outline. The following proofs are organized as follows. The proof of Property 7.4 (p. 64) is in Appendix A.1.1 (p. 152). The proof of Property 7.5 (p. 64) is in Appendix A.1.2 (p. 152). Some intermediate
lemmas with their proofs are introduced in Appendix A.1.3 (p. 153) in order to prove Theorem 7.14 (p. 68) in Appendix A.1.4 (p. 156). The proofs of Propositions 7.10 (p. 67) and 9.9 (p. 115) are respectively given in Appendices A.1.4 (p. 156) and A.1.5 (p. 157). Some intermediate definitions and lemmas with their proofs are given in Appendix A.1.6 (p. 158) in order to prove Theorem 9.16 (p. 116) in Appendix A.1.7 (p. 160).

A.1.1 Proof of Property 7.4 (p. 64)

We shall prove that:
\[ \forall (t_1, t_2) \in W. \text{interactions}(t_1) = \text{interactions}(t_2), \]
where \( W \) is the witness relation defined in Definition 7.1 (p. 63) (using a bi-simulation relation \( R \)), and \( \text{interactions}(t) \) is the sequence of interactions of trace \( t \).

Proof. The proof is done by structural induction on \( W \).

- Base case. By definition of \( W \), \((\text{init}, \text{init}) \in W \) and \( \text{interactions}(\text{init}) = \epsilon \).

- Induction case. Let us consider \((t_1, t_2) \in W \) and suppose that \( \text{interactions}(t_1) = \text{interactions}(t_2) \).

  According to the definition of \( W \), there are two rules for constructing a new element in \( W \).

  - Consider \((t_1 \cdot a \cdot q_1, t_2 \cdot a \cdot q_2) \in W \) such that \( a \in \text{Int} \) and \((q_1, q_2) \in R \). We have \( \text{interactions}(t_1 \cdot a \cdot q_1) = \text{interactions}(t_1) \cdot a \) and \( \text{interactions}(t_2 \cdot a \cdot q_2) = \text{interactions}(t_2) \cdot a \), and thus the expected result using the induction hypothesis.

  - Consider \((t_1, t_2 \cdot \beta \cdot q_2) \in W \) such that \((\text{last}(t_1), q_2) \in R \). We have \( \text{interactions}(t_2 \cdot \beta \cdot q_2) = \text{interactions}(t_2) \) and thus the expected result using the induction hypothesis.

A.1.2 Proof of Property 7.5 (p. 64)

We shall prove that:
\[ \forall t_2 \in \text{Tr}(M^\perp), \exists! t_1 \in \text{Tr}(M^*) . (t_1, t_2) \in W, \]
where \( M^* \) is a component-based system (with set of traces \( \text{Tr}(M^*) \)) and \( M^\perp \) is the corresponding component-based system with partial-state semantics (with set of traces \( \text{Tr}(M^\perp) \)).

Proof. First, let us note that from the weak bi-simulation of a global-state semantics model with its corresponding partial-state semantics model [6], we can conclude that, for any trace in the partial-state semantics
model, there exists a corresponding trace in the global-state semantics model. We prove that the witness
trace is unique by contradiction.

Let us assume that for a trace in partial-state semantics \( t_2 \in \text{Tr}(M^\perp) \), there exist two witness traces \( t_1', t_1 \in \text{Tr}(M^s) \) s.t. \((t_1, t_2), (t_1', t_2) \in W \) and \( t_1 \neq t_1' \). From Property 7.4 (p. 64), \( \text{interactions}(t_1) = \text{interactions}(t_2) \) and \( \text{interactions}(t_1') = \text{interactions}(t_2) \), therefore \( \text{interactions}(t_1) = \text{interactions}(t_1') \). Moreover, \( t_1 \) and \( t_1' \) have the same initial state because of the definition of \( W \) and \((t_1, t_2), (t_1', t_2) \in W \). From the semantics of composite components, a sequence of interactions is associated to a unique trace (from a unique initial state). This is thus a contradiction.

\[ \square \]

A.1.3 Intermediate Lemmas

We prove the intermediate lemmas that are needed to prove Theorem 7.14 (p. 68).

**Proof of Lemma 7.8 (p. 66).** We shall prove \( \forall (t_1, t_2) \in W. |\text{acc(event}(t_2))| = |t_1| = 2 \times |s| + 1 \), where \( s = |\text{interactions}(t_1)| \), where \( \text{acc} \) is the accumulator used in the definition of function \( \text{RGT} \) (Definition 7.6, p. 65), and function \( \text{interactions} \) (defined in Section 4.3, p. 41) returns the sequence of interactions in a trace (removing \( \beta \)).

**Proof.** The proof is done by structural induction on \( W \).

- **Base case.** By definition of \( W \), \((\text{init}, \text{init}) \in W \) and we have \( \text{acc(event}(\text{init})) = \text{init} \), \( |\text{init}| = 1 \) and \( |\text{interactions}(\text{init})| = |e| = 0 \).

- **Induction case.** Let us consider \((t_1, t_2) \in W \) such that \( |\text{interactions}(t_2)| = s \) and suppose that Lemma 7.8 (p. 66) holds for \((t_1, t_2) \). According to the definition of \( W \), there are two rules for constructing a new element in \( W \).

  - Consider \((t_1 \cdot a \cdot q_1, t_2 \cdot a \cdot q_2) \in W \) such that \( a \in \text{Int} \) and \((q_1, q_2) \in R \). According to Definition 7.6 (p. 65), \( \text{acc(event}(t_2 \cdot a \cdot q_2)) = \text{acc(event}(t_2)) \cdot a \cdot q_2 \). Using the induction hypothesis, \(|\text{acc(event}(t_2))| = |t_1| \). Hence \(|\text{acc(event}(t_2 \cdot a \cdot q_2))| = |\text{acc(event}(t_2))| + 2 = |t_1| + 2 = |t_1 \cdot a \cdot q_1| \).

  - Consider \((t_1, t_2 \cdot \beta \cdot q_2) \in W \) such that \((\text{last}(t_1), q_2) \in R \). According to Definition 7.6 (p. 65) and using the definition of operator \( \text{map} \), we have \(|\text{acc(event}(t_2 \cdot \beta \cdot q_2))| = |\text{map} [x \mapsto \text{upd}(q, x)] (\text{acc(event}(t_2)))| = |\text{acc(event}(t_2))| \), and thus we obtain the expected result using the induction hypothesis.

\[ \square \]
Proof of Lemma 7.9 (p. 67). We shall prove that: \( \forall t \in \text{Tr}(M^\perp), \exists k \in [1..s], q_k \in Q \implies \forall z \in [1..k], q_z \in Q, q_{z-1} \xrightarrow{a_z} q_z \) where \((q_0 \cdot a_1 \cdot q_1 \cdot \cdots \cdot a_s \cdot q_s) = \text{acc}(\text{event}(t))\).

Proof. According to Lemma 7.8 (p. 66) and the definition of function acc (see Definition 7.6, p. 65), a state is generated and added to sequence acc(event(t)) just after the execution of an interaction \(a \in \text{Int}\). This state is obtained from the last state in acc(event(t)), say \(q\), such that the new state has state information about less components than \(q\) because the states of all components involved in \(a\) are undetermined and the states of all other components are identical. Since after any busy transition, function upd (see Definition 7.6, p. 65) updates all the generated partial states that do not have the state information regarding the components that performed a busy transition, the completion of each partial state guarantees the completion of previously generated states. Therefore, if there exists a global state (possibly completed through function upd) in trace acc(event(t)), then all the previously generated states are global states.

Moreover, the sequence of reconstructed global states follow the global-state semantics. This results stems from two facts. First, according to the definition of function upd, whenever function upd completes a partial state in the trace by adding the state of a component for which the last state in the trace is undetermined, it uses the next state reached by this component according to partial-state semantics. Second, according to Definition 8.2 (p. 98), an intermediate busy state, say \(\perp\), is added between the starting state \(q\) and arriving state \(q'\) of any transition \((q, p, q')\). Moreover, the transitions \((q, p, \perp)\) and \((\perp, \beta, q')\) in the partial-state semantics replace the previous transition \((q, p, q')\) in the global-state semantics. Hence, whenever a component in partial-state semantics is in a busy state \(\perp\), the next state that it reaches is necessarily the same state as the one it would have reached in the global-state semantics.

Proof of Proposition 7.10 (p. 67). We shall prove that \(\forall t \in \text{Tr}(M^\perp)\).

\[
|\text{discriminant}(\text{acc}(\text{event}(t)))| \leq |\text{acc}(\text{event}(t))| \quad \wedge \quad \text{discriminant}(\text{acc}(\text{event}(t))) = q_0 \cdot a_1 \cdot q_1 \cdot \cdots \cdot a_d \cdot q_d \quad \Rightarrow \quad \forall i \in [1..d], q_{i-1} \xrightarrow{a_i} q_i,
\]

where acc is the accumulator function and discriminant is the discriminant function used in the definition of function RGT (Definition 7.6, p. 65) such that \(\text{RGT}(t) = \text{discriminant}(\text{acc}(\text{event}(t)))\).

Proof. The proof directly follows from the definitions of functions acc and discriminant, and Lemma 7.9. Let us consider \(t \in \text{Tr}(M^\perp)\).

Regarding the first conjunct, according to the definition of function discriminant (Definition 7.6, p. 65), \(\text{discriminant}(\text{acc}(\text{event}(t)))\) is the longest prefix of \(\text{acc}(\text{event}(t))\) such that the last state of function
discriminant(acc(event(t))) is a global state. Thus, the length of discriminant(acc(event(t))) is always lesser than or equal to the length of sequence acc(event(t)).

Regarding the second conjunct, all the states of discriminant(acc(event(t))) are global ready-states and follow the global-state semantics (according to Lemma 7.9, p. 67). Moreover, one can note that function discriminant removes the longest suffix made of partial states output by function acc and function acc only updates partial states.

Proof of Lemma 7.12 (p. 67). We shall prove that: \(\forall t \in \text{Tr}(\mathcal{M}^\perp)\). \(\text{last(acc(event(t)))} = \text{last}(t)\).

Proof. The proof is done by induction on the length of the trace in partial-state semantics, i.e., \(t \in \text{Tr}(\mathcal{M}^\perp)\).

- Base case: The property holds for the initial state. Indeed, in this case \(t = \text{init}\) and according to the definition of function acc (see Definition 7.6, p. 65) last(acc(event(init))) = init.

- Induction case: Let us assume that \(t = q_0 \cdot a_1 \cdot q_1 \cdot \cdots \cdot a_m \cdot q_m\) is a trace in partial-state semantics and acc(event(t)) = \(q'_0 \cdot a'_1 \cdot q'_1 \cdot \cdots \cdot a'_s \cdot q'_s\) such that \(q_m = q'_s\). We have two cases according to whether the next move of the partial-state semantics model is an interaction or a busy transition:
  
  - If \(a_{m+1} \in \text{Int}\), then according to the definition of the function acc, we have: \(\text{last(acc(event(t \cdot a_{m+1} \cdot q_{m+1})) = q_{m+1}}\).
  
  - If \(a_{m+1} \in \{\beta_i\}_{i=1}^{\mathcal{B}}\), then according to the definition of function acc, we have: \(\text{last(acc(t \cdot a_{m+1} \cdot q_{m+1})) = \text{upd}(q_{m+1}[i], q'_s)\). From the induction hypothesis: \(\text{upd}(q_{m+1}[i], q'_s) = \text{upd}(q_{m+1}[i], q_m)\) and from the fact that the only difference between state \(q_m\) and state \(q_{m+1}\) is that in state \(q_m\) the state of the component that executed \(a_{m+1}\) (i.e., \(B_i\)) is a busy state, while in state \(q_{m+1}\) it is not a busy state. From the definition of function upd (Definition 7.6, p. 65), we can conclude that \(\text{upd}(q_{m+1}[i], q_m) = q_{m+1}\).

In both cases, \(\text{last(acc(event(t)))} = \text{last}(t)\).
A.1.4 Proof of Theorem 7.14 (p. 68)

We shall prove that, for a given BIP model $M^\perp = \Gamma(B_1, \ldots, B_{|B|})$ with behavior $(Q^\perp, \Gamma, \rightarrow)$ as per Definition 8.4 (p. 99) with the set of traces $\text{Tr}(M^\perp)$ and its corresponding sequential model with global-state semantics $M^\sharp$ with behavior $(Q, \text{Int}, \rightarrow_s)$ and the set of traces $\text{Tr}(M^\sharp)$, the following holds:

$$\forall t \in \text{Tr}(M^\perp).$$

$$\text{last}(t) \in Q \implies \text{RGT}(t) = W(t)$$

$$\land \text{last}(t) \notin Q \implies \text{RGT}(t) = W(t') \cdot a,$$

$$t' = \min_{t_p \in \text{Tr}(M^\perp)} \{ t_p \in \text{Tr}(M^\perp) \mid \exists a \in \text{Int}, \exists t'' \in \text{Tr}(M^\perp). t = t_p \cdot a \cdot t''$$

$$\land \exists i \in [1 .. |B|]. (B_i, P \cap a \neq \emptyset) \land (\forall j \in [1 .. \text{length}(t'')]. \beta_i \neq t''(j))\}$$

where function $\text{RGT}$ is defined in Definition 7.6 (p. 65) and $W$ is the witness relation defined in Definition 7.1 (p. 63).

Proof. For any trace in partial-state semantics $t$, we consider two cases depending on whether the last element of $t$ belongs to $Q$ of not:

- If $\text{last}(t) \in Q$, according to Lemma 7.12 (p. 67), we have $\text{last}(\text{acc(event}(t))) \in Q$ and therefore $\text{RGT}(t) = \text{discriminant(\text{acc(event}(t))}) = \text{acc(event}(t))$. Let us assume that $\text{acc(event}(t)) = q_0 \cdot a_1 \cdot q_1 \cdots a_s \cdot q_s$, with $q_0 = \text{init}$. According to Lemma 7.9 (p. 67) we have $\forall k \in [1 .. s]. q_{k-1} \rightarrow_s a_k \Rightarrow \text{acc(event}(t)) \in \text{Tr}(M^\sharp)$. Moreover, $\text{interactions(\text{acc(event}(t))) = \text{interactions}(t)}$ (according to Lemma 7.13, p. 67). Furthermore, according to definition of the witness relation (Definition 7.1, p. 63), from the unique initial state, since $\text{acc(event}(t))$ and $t$ have the same sequence of interactions, $\text{acc(event}(t)), t \in W$. Therefore, $\text{acc(event}(t)) = \text{RGT}(t) = W(t)$.

- If $\text{last}(t) \notin Q$, we treat this case by induction on the length of $t$. Let us assume that the proposition holds for some $t \in \text{Tr}(M^\perp)$ (induction hypothesis). Let us consider $t = t' \cdot a'_1 \cdot q'_1 \cdot a'_2 \cdot q'_2 \cdots a'_k \cdot q'_k$, with $k > 0$. Let us assume that the splitting of $t$ is $t' \cdot a'_1 \cdot t''$, where $t'$ is the minimal sequence such that there exists at least one component that is involved in interaction $a'_1 \in \text{Int}$ and that is still busy. (We note that in this case $t'$ do exist because last($t$) $\notin Q$ implies that the system has made at least one move.) Let $i$ be the identifier of this component and $a'_1$ be $s^{th}$ interaction in trace $t$ such that $a'_1 = \text{interactions}(t)(s)$. Let us consider $t \cdot a'_{k+1} \cdot q'_{k+1}$, the trace extending $t$ by one interaction $a'_{k+1}$. We distinguish again two subcases depending on whether $a'_{k+1} \in \text{Int}$ or not.

- Case $a'_{k+1} \in \text{Int}$. We have last($t$) $\notin Q$ and then last($t \cdot a'_{k+1} \cdot q'_{k+1}$) $\notin Q$ (because $a'_{k+1} \in \text{Int}$,
i.e., the system performs an interaction, and the state following an interaction is necessarily a partial state). Moreover, $\text{RGT}(t) = \text{RGT}(t \cdot a_k^1 \cdot q_k^1)$, i.e., the reconstructed global state does not change. Hence, the components which are busy after $a_1^i$ are still busy. Consequently, the splitting of $t$ and $t \cdot a_k^1 \cdot q_k^1$ are the same. Following the induction hypothesis, $t \cdot a_k^1 \cdot q_k^1$ has the expected property.

- Case $a_k^1 = \beta_j$, for some $j \in [1..|B|]$. We distinguish again two subcases.

  * If $i = j$, that is the busy interaction $\beta_j$ concerns the component(s) for which information was missing in $t''$ (component $i$). If component $i$ is the only component involved in interaction $a_1^i$ for which information is missing in $q_1^1 \cdots q_k^1$, the reconstruction of the global state corresponding to the execution of $a_1^i$ can be done just after receiving the state information of component $i$. After receiving $q_{k+1}$, which contains the state information of component $i$, the partial states of $\text{acc}(t)$ are updated with function $\text{upd}$. That is, $\text{RGT}(t \cdot a_k^1 \cdot q_{k+1}) = \text{RGT}(t) \cdot q_0^0 \cdot a_k^1 \cdot q_1^1 \cdots q_{m-1}^1 \cdot a_m^m$, where $m > 0$, $q_0^0$ is the reconstructed global state associated with interaction $a_1^i$, and $a_m^m = \text{interactions}(t)(s + m)$ is the first interaction executed after $t$ for which there exists at least one involved component which is still busy. Indeed, some interactions after $a_1^i$ in trace $t$ (i.e., $a_m^m = \text{interactions}(t)(s + p)$ for $m > p > 0$) may exist and be such that component $i$ is the only component involved in them for which information is missing to reconstruct the associated global states. In this case, updating the partial states of $\text{acc}(t)$ with the state information of component $i$ yields several global states i.e., $q_1^1 \cdots q_{m-1}^1$. Then, the splitting of $t$ changes as follows: $t = t'' \cdot a_m^m \cdots a_k^1 \cdot q_{k+1}^1$, where $t'' = t' \cdot a_1^i \cdot q_1^1 \cdots q_{k-1}^1$ and $q_k^1$ is the system state before interaction $a_m^m$. Therefore, $\text{RGT}(t \cdot a_k^1 \cdot q_{k+1}^1) = W(t'') \cdot a_m^m$ and the property holds again.

  * If $i \neq j$, we have $\text{RGT}(t) = \text{RGT}(t \cdot a_k^1 \cdot q_{k+1}^1)$. Hence, the splitting of $t$ and $t \cdot a_k^1 \cdot q_{k+1}^1$ are the same. Following the induction hypothesis, $t \cdot a_k^1 \cdot q_{k+1}^1$ has the expected property.

\[ \square \]

### A.1.5 Proof of Proposition 9.9 (p. 115)

We shall prove that, for a given BIP model $M^\perp = \Gamma(B_1, \ldots, B_{|B|})$ and the transformed composite component $M^r = \Gamma^r(B_1^r, \ldots, B_{|B|}^r)$ obtained as per Definition 9.5 (p. 111), we shall prove that for any execution of the system with partial-state semantics with trace $t \in \text{Tr}(M^\perp)$, component RGT (Definition 9.3 p. 108) implements function RGT (Definition 7.6 p. 65), that is $\forall t \in \text{Tr}(M^\perp). \text{RGT} \cdot V \cong \text{acc}($event$(t))$. 

\[ \square \]
Proof. The proof is done by induction on the length of $t \in \text{Tr}(M^\perp)$, i.e., the trace of the system in partial-state semantics.

- **Base case.** By definition of function $\text{RGT}$, at the initial state $\text{acc}(\text{event}(\text{init})) = \text{init}$. By definition of component $\text{RGT}$, $V$ is initialized as a tuple representing the initial state of the system. Therefore, $\text{RGT}.V \cong \text{acc}(\text{event}(\text{init}))$.

- **Induction case.** Let us suppose that the proposition holds for a trace $t \in \text{Tr}(M^\perp)$, that is $\text{RGT}.V \cong \text{acc}(\text{event}(t))$. According to the definition of function $\text{RGT}$ (see Definition 9.3, p. 108), we have $\text{RGT}(t) = \text{discriminant}(\text{acc}(\text{event}(t)))$. Consequently, there exists $t' \in \text{Tr}(M^\perp)$ of the form $t' = q_0' \cdot a_1' \cdot q_1' \cdots q_k'$, with $k \geq 0$, such that $\text{acc}(\text{event}(t)) = \text{RGT}(t) \cdot t'$. We distinguish two cases depending on the action of the system executed after $t$:

  - The first case occurs when the action is the execution of an interaction $a_{k+1}'$, followed by a partial state $q_{k+1}'$. On the one hand, we have $\text{acc}(\text{event}(t \cdot a_{k+1}' \cdot q_{k+1}')) = \text{acc}(\text{event}(t)) \cdot a_{k+1}' \cdot q_{k+1}'$. On the other hand, in component $\text{RGT}$, according to Algorithm 9.1 (p. 109) (line 6), the corresponding transition $\tau \in T_{\text{new}}$ extends the sequence of tuples $V$ by a new ($|B|+1$)-tuple $v$ which consists of the current partial state of the system such that $V = V \cdot v$ and $v \cong q_{k+1}'$. Therefore, we have $\text{RGT}.V \cong \text{acc}(\text{event}(t))$ as expected.

  - The second case occurs when the next action is the execution of a busy transition. On the one hand, function $\text{RGT}$ updates all the partial states $q_0', \ldots, q_k'$. On the other hand, according to Algorithm 9.2 (p. 110) (lines 2-6), in component $\text{RGT}$, the corresponding transition $\tau \in T_{\text{upd}}$ updates the sequence of tuples $V$ such that $\text{RGT}.V \cong \text{acc}(\text{event}(t))$ hold.

Moreover, function $\text{RGT}$ and component $\text{RGT}$ similarly create new global states from the partial states whenever new global states are computed. On the one hand, after any update of partial states, through function $\text{discriminant}$, function $\text{RGT}$ outputs the longest prefix of the generated trace which corresponds to the witness trace. On the other hand, after any update of the sequence of tuples $V$, component $\text{RGT}$ checks for the existence of fully completed tuples in $V$ to deliver them to through the dedicated ports to the runtime monitor.

\[\square\]

### A.1.6 Proofs of Intermediate Lemmas

In the following proofs, we will consider several mathematical objects in order to prove the correctness of our framework:
– a composition with partial-state semantics $M^\perp = \Gamma(B_1, \ldots, B_{|B|})$ of behavior $(Q^\perp, \Gamma, \to)$;
– the transformed composite component $M^r = \Gamma^r(B_1^r, \ldots, B_{|B|}^r, RGT)$ of behavior $(Q^r, \Gamma^r, \to_r)$.

$M^r$ is obtained from $M^\perp$ by following the transformations described in Definition 9.5 (p. 111).

**Proof of Lemma 9.12** (p. 115). We shall prove that in any state of the transformed system, if there is a non-empty set $GS \subseteq \{\text{RGT}.gs_a \mid a \in \text{Int}\}$ in which all variables are true, the variables in $\{\text{RGT}.gs_a \mid a \in \text{Int}\} \setminus GS$ cannot be set to true until all variables in $GS$ are reset to false first.

Proof. According to the definition of component RGT (Definition 9.3, p. 108), on the one hand only the transitions in set $T_{upd}$ are able to set the value of the variables in $\{\text{RGT}.gs_a \mid a \in \text{Int}\}$ to true; on the other hand the transitions in set $T_{upd}$ are guarded by $\bigwedge_{a \in \text{Int}}(\neg gs_a)$ which means that all of the Boolean variables in $\{\text{RGT}.gs_a \mid a \in \text{Int}\}$ must be false for one of these transitions to execute. Therefore, in any state $q \in Q^r$ such that such a set $GS$ exists, the transitions in $T_{upd}$ are not possible. Moreover, the only possible transitions in state $q$ are the transitions in set $T_{out}$ which effect is to reset the value of the variables in $\{\text{RGT}.gs_a \mid a \in \text{Int}\}$ to false using algorithm $\text{get}$. \qed

**Proof of Lemma 9.13** (p. 116). We shall prove that for any state $q \in Q^r$, there exists a state $q' \in Q^r$ reached after interactions in $a^m$ (i.e., $\xrightarrow{(a^m), r} q'$), such that $q'$ is a stable state (i.e., stable($q'$)).

Proof. Let us consider a non-stable state $q \in \text{RGT}.Q$. The interactions in $a^m$ involve to execute ports in $\{p_a' \mid a \in \text{Int}\}$ and transitions in $T_{out}$. Since $q$ is a non-stable state, at least one of the variables in $\{\text{RGT}.gs_a \mid a \in \text{Int}\}$ evaluates to true in $q$ (see Definition 9.10, p. 115). Such transitions entail to execute algorithm $\text{get}$ (Algorithm 9.4, p. 110) which resets the Boolean variable to false by delivering the associated reconstructed global state(s) to the monitor. After executing algorithm $\text{get}$, if there exists another Boolean variable in $\{\text{RGT}.gs_a \mid a \in \text{Int}\}$ that evaluates to true, according to Lemma 9.12 component RGT returns to a situation where only again algorithm $\text{get}$ can execute (through the interactions in set $a^m$). The above process executes until the system eventually reaches a state $q'$ where no interaction in $a^m$ is enabled. Therefore, in $q'$ all Boolean variables in $\{\text{RGT}.gs_a \mid a \in \text{Int}\}$ evaluate to false, because interactions in $a^m$ are unary interactions, each involving port RGT $p_a'$ (Definition 9.5, p. 111) guarded by $\bigwedge_{a \in \text{Int}}(\neg gs_a)$ (Definition 9.3, p. 108). According to Definition 9.10 a state is stable when all Boolean variables in $\{\text{RGT}.gs_a \mid a \in \text{Int}\}$ evaluate to false. Thus, $q'$ is stable. \qed

**Proof of Lemma 9.15** (p. 116). Let us consider two states: $q$ of the initial model and $q'$ its corresponding state in the transformed model such that $\text{equ}(q') = q$. There exists an enabled interaction in the initial model
(a \in \Gamma) in state q \in Q^\perp$, if and only if the corresponding interaction in the transformed model $(a^r \in \Gamma^r)$ is enabled at state $q^r$.

**Proof.** According to the definitions of interaction transformation and atom RGT (Definition 9.3, p. 108), ports RGT.\textit{pa}, for $a \in Int$, are always enabled. Since for a given interaction $a$, $a^r$ and $a$ differ only by port RGT.\textit{pa}, we can conclude that $a^r \in a^\gamma$ is enabled if and only if $a \in \Gamma$ is enabled. $\blacksquare$

### A.1.7 Proof of Theorem 9.16 (p. 116)

Before tackling the proof of the theorem, we convey a remark preparing the definition of the weak bisimulation relation defined in the proof.

Following Definition 9.5 (p. 111), the set of interactions $\Gamma^r$ of the instrumented system is partitioned as $\Gamma^r = a^\gamma_r \cup a^\beta_r \cup a^m$, where $a^\gamma_r$ is the set of interactions of the initial system augmented by RGT port, $a^\beta_r$ is a set containing the busy interactions of the initial system (one for each component) augmented by RGT port, and $a^m$ is a new set of interactions used for monitoring purposes. First, we note that the set of interactions in the instrumented system $a^\gamma_r$ and $a^\beta_r$ are isomorphic to the sets of interactions Int and $\{\{\beta_i\}\}_{i=1}^{|B|}$ of the initial system because they contain only an additional port to notify component RGT. We can thus identify these sets of interactions. Moreover, as usual in monitoring, the actions used for monitoring purposes (i.e., interactions in $a^m$) are considered to be unobservable. These interactions do not influence the state of the system and execute independently of the interactions in $a^\gamma_r \cup a^\beta_r$; these are interactions occurring between RGT and the monitor which are components introduced in the instrumentation. See also [39], for more arguments along these lines related to the instrumentation of single-threaded CBSs.

**Proof.** We exhibit a relation $R \subseteq Q^\perp \times Q^r$ between the set of states of the initial model with partial-state semantics and the set of states of the transformed model. We define the relation such that $R = \{(q, q^r) \mid \exists z^r \in Q^r \cdot q^r \xrightarrow{(a^m)^\ast \cdot a^r} z^r \land \text{equ}(z^r) = q\}$, and we shall prove that relation $R$ satisfies the following properties to establish that $R$ is a weak bi-simulation:

1. $(q, q^r) \in R \land \exists z^r \in Q^r \cdot q^r \xrightarrow{a^m \cdot a^r} z^r) \implies (q, z^r) \in R$;
2. $(q, q^r) \in R \land \exists z^r \in Q^r \cdot q^r \xrightarrow{a^\gamma_r + a^\beta_r} z^r) \implies \exists z \in Q^\perp \cdot (q \xrightarrow{a} z \land (z, z^r) \in R)$;
3. $(q, q^r) \in R \land q \xrightarrow{\Gamma} z) \implies \exists z^r \in Q^r \cdot (q^r \xrightarrow{(a^m)^\ast \cdot a^r} z^r \land (z, z^r) \in R)$.

Let us consider $q = (q_1, \cdots, q_{|B|})$ and $q^r = (q^r_1, \cdots, q^r_{|B|}, q^r_{|B|+1})$ such that $(q, q^r) \in R$.

**Proof of (i):**

Since $(q, q^r) \in R$, there exists a stable state $q'' \in Q^r$ which is reached after unobservable interactions in $a^m$. 


After the execution of some unary interaction \( \alpha \in a^m \), the corresponding Boolean variable \( \text{RGT} . gs_\alpha \) is set to \text{false} (Algorithm 9.4, p. 110). Let us consider that the next state after the execution of some interaction \( \alpha \in a^m \) is \( z' = (z'_1, \ldots, z'_{|B|}, z'_{|B|+1}) \). If \( z'_{|B|+1} \) is a stable state then \( \text{equ}(z') = q \) thus \( (q, z') \in R \), and if \( z'_{|B|+1} \) is not a stable state according to Lemma 9.13 (p. 116), after interaction \( \alpha \in a^m \), the state of \( \text{RGT} \) (that is \( z'_{|B|+1} \)) will be stable, therefore we conclude that \( (q, z') \in R \).

Proof of (ii):

Let us consider \( z'' = (z''_1, \ldots, z''_{|B|}, z''_{|B|+1}) \) and \( z = (z_1, \ldots, z_{|B|}) \). When some \( a'' \in (a_r^\ast \cup a_g^\ast) \) is enabled, from the definition of the semantics of transformed composite component and Lemma 9.15 (p. 116), we can deduce that the corresponding interaction \( \alpha \in \Gamma \) is enabled (recall, that for each interaction \( a \in \Gamma \) in the initial model with partial-state semantics there exists a corresponding interaction \( a'' \) in the transformed model, as per Definition 9.5, p. 111). Executing the corresponding interactions \( a \) and \( a'' \) changes the local states \( q_i'' \) and \( q_i \), for \( i \in [1 \ldots |B|] \), to \( z'_i \) and \( z_i \) for \( i \in [1 \ldots |B|] \) respectively, in such a way that \( z''_i = z_i \), for \( i \in [1 \ldots |B|] \), because the transformations do not modify the transitions of the components of the initial model. After \( a'' \), we have two cases depending on whether \( z''_{|B|+1} \) is stable or not.

- If \( z''_{|B|+1} \) is stable, from the definition of relation \( R \), we have \( (z, z'') \in R \).

- If \( z''_{|B|+1} \) is not stable, then according to Lemma 9.13 (p. 116), \( z''_{|B|+1} \) will be stable after some interactions \( \alpha \in a^m \) (that is \( z''_{|B|+1} \mapsto \text{stable}(z''_{|B|+1}) \)). Therefore, \( (z, z'') \in R \).

Proof of (iii):

Let us consider \( z'' = (z''_1, \ldots, z''_{|B|}, z''_{|B|+1}) \). When \( a \in \Gamma \) is enabled in the initial model, we can consider two cases depending on whether the corresponding interaction \( a'' \) in the transformed model is enabled or not.

- If \( a'' \) is enabled, we have two cases for the next state of component RGT:

  - if \( a'' \in a_r'' \), according to the definition of atom RGT, \( z''_{|B|+1} \) is stable and \( (z, z'') \in R \).

  - if \( a'' \in a_g'' \), we have two cases:

    * If RGT has some global states to deliver (that is \( z''_{|B|+1} \) is not stable), then, according to Lemma 9.13 (p. 116), RGT will be stable after some interactions in \( a^m \). Hence, \( (z, z'') \in R \).

    * If RGT has no global state, then atom RGT is stable and \( (z, z'') \in R \).

- If \( a'' \) is not enabled, according to the definition of atom RGT, we can conclude that RGT has some global states to deliver, thus \( q'' \) is not stable. According to Lemma 9.13 (p. 116), a not stable system
becomes stable after executing some interactions in $a^m$. Therefore, according to Lemma 9.15 (p. 116), $a^r$ is necessarily enabled when the system is stable. Consequently, the same reasoning followed for the previous case can be conducted in which $a^r$ is initially enabled. Henceforth, $(z, z^r) \in R$.

\[\square\]

A.2 Proofs Related to the Approach for Monitoring distributed CBS

In the following, we consider a distributed system $M$ consisting a set of schedulers and components $\{S_1, \ldots, S_{|S|}, B_1, \ldots, B_{|B|}\}$ with the global behavior $(Q, GAct, \rightarrow)$ as per Definition 4.8 (p. 37) and the transformed version of $M$ due to monitoring purposes (see Section 6, p. 49), that is $M^c$ consisting instrumented schedulers and shared components $\{S_1 \otimes C^s_1, \ldots, S_{|S|} \otimes C^s_{|S|}, B'_1, \ldots, B'_{|B|}\}$ with behavior $(Q^c, GAct_{c, \rightarrow_c})$ as per Definition 6.13 (p. 59) where $C^s_j$ for $j \in [1..|S|]$ is the controller of scheduler $S_j$ as per Definition 6.1 (p. 51) such that $\forall i \in [1..|B|], B'_i = B_i \otimes C^b_i$ such that $C^b_i$ is the controller of the share component $B_i$ if $B_i \in B_s$ as per Definition 6.4 (p. 54) and $B'_i = B_i$ otherwise.

We consider the global state of system $M$ as $q = (q_{s_1}, \ldots, q_{s_{|S|}}, q_{b_1}, \ldots, q_{b_{|B|}}) \in Q$, where $q_{s_j}$ is the state of scheduler $S_j$ for $j \in [1..|S|]$, and $q_{b_i}$ is the state of component $B_i$ for $i \in [1..|B|]$. Moreover, the global state of system $M^c$ is $q^c = (q'_{s_1}, q'_{sc_1}, \ldots, q'_{s_{|S|}}, q'_{sc_{|S|}}, q'_b, q'_{bc_1}, \ldots, q'_{b_{|B|}}, q'_{bc_{|B|}}) \in Q^c$, where $q_{sc_j}$ is the state of the controller of scheduler $S_j$ for $j \in [1..|S|]$, $q_{bc_i}$ is the state of the controller of shared component $B_i$ for $i \in [1..|B|]$ if $B_i \in B_s$ and empty otherwise. The first result concerns the correctness of the transformed model $M^c$ through the instrumentation defined in Section 6.1 (p. 50).

The instrumented model $M^c$ generates events and sends them the observer in order to reconstruct the set of compatible global traces, that is, the computation lattice $L$. The second results concerns the correctness (soundness and completeness) of computation lattice construction presented in Section 7.2 (p. 69) with respect to the obtained events from the instrumented model.

The third result states the correctness (soundness and completeness) of the monitoring algorithm applied on the constructed lattice presented in Section 7.2.5 (p. 85), that is, our algorithm verifies the set of compatible global traces of the system.

Proof outline. The following proofs are organized as follows. Some intermediate definitions and lemmas are introduced in Appendix A.2.1 (p. 163) in order to prove Proposition Proposition 6.14 (p. 59) in Appendix A.2.2 (p. 163). The proof of Property 7.18 (p. 71) is in Appendix A.2.3 (p. 164). The proof of Property 7.21 (p. 72) is in Appendix A.2.4 (p. 164). The proof of Proposition 7.30 (p. 81) is in Appendix A.2.5 (p. 164). The proof of Proposition 7.38 (p. 84) is in Appendix A.2.6 (p. 165). The proof
of Proposition 7.39 (p. 84) is in Appendix A.2.7 (p. 167). The proof of Proposition 7.46 (p. 91) is in Appendix A.2.8 (p. 168). The proof of Theorem 7.47 (p. 92) is in Appendix A.2.9 (p. 169). The proof of Theorem 7.48 (p. 92) is in Appendix A.2.10 (p. 170).

A.2.1 Intermediate Definition and Lemma

We give some intermediate definition and lemma that are needed to prove Proposition 6.14 (p. 59) to prove the bi-simulation of $M$ and $M^c$. First we define a relation between the states of two systems.

**Definition A.1.** Relation $\text{equ} \subseteq Q \times Q_c$ is the smallest set that satisfies the following rule.

$$(q, q') \in \text{equ} \implies q_{s_j} = q'_{s_j} \land q_b_i = q'_{b_i}, \forall j \in [1..|S|], \forall i \in [1..|B|]$$

Two states $q \in Q$ and $q' \in Q_c$ are in relation $\text{equ}$ where the states of scheduler $S_j$ for $j \in [1..|S|]$ and the state of component $B_i$ for $i \in [1..|B|]$ in global state $q$ are the same as they are in global state $q'$.

The following lemma is a direct consequence of Definition A.1.

**Lemma A.2.** A global action $\alpha \in GAct$ is enabled in the initial system at global state $q$, and in the transformed system at global state $q'$ if $(q, q') \in \text{equ}$.

Since controllers do not induce any restriction in the system in the sense that they do not hold the execution of the system, any enabled action in the initial system at state $q$ is also enabled in the augmented system at state $q'$ if $(q, q') \in \text{equ}$.

A.2.2 Proof of Proposition 6.14 (p. 59)

We shall prove the existence of a bi-simulation between initial and transformed model, that is relation $R = \{(q, q') \in Q \times Q_c \mid (q, q') \in \text{equ}\}$ satisfies the following property:

$$(q, q') \in R \land \exists \alpha \in GAct, \exists z \in Q, q \xrightarrow{\alpha} z \implies \exists \alpha \in GAct, \exists z' \in Q_c, (q' \xrightarrow{\alpha} c z' \land (z, z') \in R)$$

**Proof.** After executing global action $\alpha$, states of the schedulers managed the interactions of the global action and components involved in the interactions are changed following the semantic rules defined in Definition 4.8 (p. 37). Moreover, because of the equality of $q$ and $q'$, global action $\alpha$ is enabled at state $q'$ in the transformed system. Execution of global action $\alpha$ in the transformed system following the semantic rules defined in Definition 6.2 (p. 53) results the new states of the schedulers managed the action $\alpha$ which are the same as the states of the schedulers in the initial model after the execution of global action $\alpha$. 
A.2.3 Proof of Property 7.18 (p. 71)

We shall prove that a computation lattice can be extended by an action event \( e \in E_a \) from just one node.

**Proof.** The proof is done by contradiction. Let us assume that there exists two nodes \( \eta, \eta' \in \mathcal{L}.\text{nodes} \) in such that \( \text{extend}(\eta, e) \) and \( \text{extend}(\eta', e) \) both are defined. According to Definition 7.20 (p. 72), the vector clocks of the nodes \( \eta \) and \( \eta' \) are in relation \( \mathcal{J}_L \), that is \( (\eta.\text{clock}, \eta'.\text{clock}) \in \mathcal{J}_L \). Moreover, its concluded that nodes \( \eta \) and \( \eta' \) are associated to two concurrent action events. Based on the definition 7.22 (p. 72), joint node of \( \eta \) and \( \eta' \) has the same vector clock as \( e.\text{clock} \), which means that we received an action event whose vector clock is already dedicated to another node in the lattice. Reception of an action event with a vector clock similar to the vector clock of the joint node of \( \eta \) and \( \eta' \) defeats the concurrency between \( \eta \) and \( \eta' \) and contradict the assumption. Therefore there exist at most one node in the lattice for with function \( \text{extend} \) is defined. 

A.2.4 Proof of Property 7.21 (p. 72)

We shall prove that the meet of two nodes \( \eta, \eta' \in \mathcal{L}.\text{nodes} \) in relation \( \mathcal{J}_L \) exists in \( \mathcal{L}.\text{nodes} \).

**Proof.** According to Definition 7.17 (p. 71), for a node \( \eta \) in the lattice where \( \eta.\text{clock} = (c_1, \ldots, c_{|S|}) \) we have \( \exists \eta' \in \mathcal{L}.\text{nodes} . \eta'.\text{clock} = (c'_1, \ldots, c'_{|S|}) \) such that \( \exists j \in [1..|S|], c'_j = c_j - 1 \). Node \( \eta' \) is the node that lattice \( \mathcal{L} \) has been extended from, to make node \( \eta \). If two nodes of the lattice satisfy relation \( \mathcal{J}_L \). According to Definition 7.20 (p. 72), they have extended the lattice from a unique node which exists in the lattice according to the above argument.

A.2.5 Proof of Proposition 7.30 (p. 81)

We shall prove that reception of certain events results computing a unique lattice and unique queue of stored events.

\[
\zeta, \zeta' \in E^* . \left( \forall S_j \in S . \zeta \downarrow S_j = \zeta' \downarrow S_j \right) \implies \text{MAKE}(\zeta) = \text{MAKE}(\zeta')
\]

**Proof.** The proof is done by induction over the length of the sequence of received events.

- Base case. for the sequences of events with the length of one the proposition holds.
– Let us suppose that the proposition holds for two sequences of events $\zeta$ and $\zeta'$ such that $\text{MAKE}(\zeta) = \text{MAKE}(\zeta')$. By extending $\zeta$ and $\zeta'$ via two events $e$ and $e'$ in different order such that $\zeta \cdot e_1 \cdot e_2$ and $\zeta' \cdot e_2 \cdot e_1$ are the new sequences, we have following possible cases:

– If either $e_1, e_2 \in E_a$ such that $e_1 \not\rightarrow e_2 \lor e_2 \not\rightarrow e_1$ or $e_1, e_2 \in E_\beta$, these events are said to be independent events in the sense that using one of them in order to extend/update the lattice or storing it in the queue does not depend to the reception of the other event.

– If $e_1, e_2 \in E_a$ and there exist happened-before relation between them such that for instance $e_1 \rightarrow e_2$, and $e_2$ is received before $e_1$. After the reception of event $e_2$, one can’t find node $\eta$ in the lattice in which $\text{extend}(\eta, e_2)$ is defined, thus event $e_2$ is added to the queue. After the reception of event $e_1$ and extending the lattice with the associated node, the algorithm recalls event $e_2$ in the queue, as if event $e_1$ has been received earlier than $e_2$. In other words, the algorithm reorder the received events by using the queue $\kappa$.

– If $e_1 \in E_a, e_2 \in E_\beta$, where $e_2$ is an update event contains the state of component $B_i$ for $i \in [1..|B|]$. Updating the lattice with event $e_2$ or storing event $e_2$ in the queue depends on the other events in the queue such that if there exists an action event associated to execution of an action concerning the component $B_i$, then $e_2$ must be stored in the queue. However, based on our assumption, it never be the case that from a specific scheduler, the observer receives an update event associated to component component $B_i$ earlier than receiving the action event associated to the execution of an action concerning component $B_i$. Therefore, updating the lattice with event $e_2$ or storing event $e_2$ in the queue does not depend on event $e_1$ which is going to be received later.

Moreover, extending the lattice with the action event $e_2$ or storing the action event $e_2$ in the queue does not depend on any update event.

\[ \square \]

A.2.6 Proof of Proposition 7.38 (p. 84)

We shall prove that for any possible set of events $\zeta$ of a given global trace $t$, the projection of the paths in the constructed lattice on scheduler $S_j$ with $j \in [1..|S|]$ results the refined local trace of the scheduler.

$\forall \zeta \in \Theta(t), \forall \pi \in \Pi(\text{MAKE}(\zeta) . \text{lattice}), \forall j \in [1..|S|], \pi \downarrow S_j = \mathcal{R}_\beta(s_j(t))$.

Proof. The proof is done by induction over the length of the global trace $t$.

– Base case. The proposition holds initially where $t = \text{init}$. 
- Let us suppose that the proposition holds for a global trace \( t \).

We have two cases for the next action of the system, which leads to the extension of the global trace:

- Any extension of the global trace by execution of an action \( a \in Int \) (i.e., \( t \cdot a \cdot q \)) generates the associated action event \( e = (a, we) \), and if there exist a node in the lattice \( \eta \in \text{lattice} \) such that \( \text{extend}(\eta, e) \) is defined, then event \( e \) extends the lattice from \( \eta \) toward the direction of the scheduler manages interaction \( a \). We consider two cases whether or not the lattice is extended from the frontier node:

  - if \( \eta \) is the frontier node (i.e., \( \eta = \eta^f \))

    \[
    \Pi(\text{MAKE}(\zeta, e).\text{lattice}) = \{ \pi \cdot a \cdot \eta' \mid \pi \in \Pi(\text{MAKE}(\zeta).\text{lattice}) \wedge \eta' = \text{extend}(\eta, e) \}.
    \]

    We have two cases \( \forall j \in [1..|S|] \):

    * if \( S_j \neq \text{managed}(a) \) we have \( s_j(t \cdot a \cdot q) = s_j(t) \), and \( \Pi(\text{MAKE}(\zeta, e).\text{lattice}) \downarrow_{S_j} = \Pi(\text{MAKE}(\zeta).\text{lattice}) \downarrow_{S_j} \).

      * if \( S_j = \text{managed}(a) \) we have \( s_j(t \cdot a \cdot q) = s_j(t) \cdot a \cdot q \), and \( \Pi(\text{MAKE}(\zeta, e).\text{lattice}) \downarrow_{S_j} = \Pi(\text{MAKE}(\zeta).\text{lattice}) \downarrow_{S_j} \cdot a \cdot (\eta'.\text{state}) \) where \( \eta' \) is the new node.

  - if \( \eta \) is not the frontier node

    \[
    \{ \pi(0..k) \cdot a \cdot \eta' \mid \pi \in \Pi(\text{MAKE}(\zeta).\text{lattice}) \wedge k \in [0..\text{length}(\pi)] \wedge \pi(k) = \eta \wedge \eta' = \text{extend}(\eta, e) \}.
    \]

    A set of new paths starting from the initial node and ending with node \( \eta' \) is added to the set of paths.

First, Algorithm MAKE extends the lattice by generating node \( \eta' = \text{extend}(\eta, e) \).

Since \( \eta \) is not the frontier node, there exists node \( \eta'' \in \mathcal{L}.\text{nodes} \) such that \( (\eta'.\text{clock}, \eta''.\text{clock}) \in \mathcal{J}_L \), meaning that execution of interaction \( a \) is concurrent with the execution associated to node \( \eta'' \). Extending the lattice with the possible joints leads to an extended lattice up to a new frontier \( \eta_{\text{new}}^f = \text{extend}(\eta', (a, \max(\text{vc}, \eta'.\text{clock}))) \), where \( \eta_{\text{new}}.\text{state} = q \).

For each path \( \pi \) in the initial lattice where \( \pi = \pi_1 \cdot a' \cdot \pi_2 \) such that \( a' \) is concurrent to action \( a \), and the vector clock of the first node of \( \pi_2 \) is in relation \( \mathcal{J}_L \) with the vector clock of node \( \eta' \), we have a set of new paths \( \pi' = \{ (\pi_1 \cdot a \cdot q_1 \cdot a' \cdot \pi_2'), (\pi_1 \cdot a' \cdot q_2 \cdot a \cdot \pi_2'), (\pi_1 \cdot (a \cup a') \cdot \pi_2') \} \) where \( \pi_2' \) has the same sequence of actions with \( \pi_2 \) with the difference that \( \pi_2' \) begins from the joint of \( \eta' \) and the first node of \( \pi_2 \) and \( \pi_2' \) ends up with node \( \eta_{\text{new}}^f \). Projection of each new path on schedulers results the following \( \forall j \in [1..|S|] \):

  * if \( S_j \neq \text{managed}(a) \) then \( \pi' \downarrow_{S_j} = \pi \downarrow_{S_j} \)
A.2. PROOFS RELATED TO THE APPROACH FOR MONITORING DISTRIBUTED CBS

* if \( S_j = \text{managed}(a) \) then \( \pi' \downarrow S_j = \pi \downarrow S_j \cdot a \cdot \pi'_2 \downarrow S_j \)

- Any extension of the global trace by execution of a busy action (i.e., \( t \cdot \beta_i \cdot q \)) generates an update event

\[
e = (\beta_i, q_i).
\]

According to Algorithm \textsc{Make}, procedure \textsc{UpdateEvent} uses the state information of event \( e \) and update the busy state of the nodes. Similarly, refine function updates the busy states associated to the component \( B_i \) using \texttt{upd} function. Therefore, for each updated path of the lattice \( \pi \) we have \( \forall j \in [1..|S|] . \pi \downarrow S_j = \mathcal{R}_\beta(s_j(t \cdot \beta_i \cdot q)) \).

In either case, the proposition holds.

A.2.7 Proof of Proposition 7.39 (p. 84)

We shall prove that for any possible set of events \( \zeta \) of a given global trace \( t \), there exists a unique path in the constructed lattice associated to each compatible trace, such that \( \forall \zeta \in \Theta(t), \forall t' \in \mathcal{P}(t), \exists \pi \in \Pi(\textsc{Make}(\zeta).\mathcal{lattice}) . \pi = \mathcal{R}_\beta(t') \)

Proof. The proof is done by induction over the length of the global trace \( t \).

- Base case. The proposition holds initially where \( t = \text{init} \).
- Let us assume that the proposition holds for a global trace \( t \).
- We have two cases for the next action of the system, which leads to the extension of the global trace:

  - Any extension of the global trace by execution of a global action \( a \in 2^{Int} \) (i.e., \( t \cdot \alpha \cdot q \)) results the new set of compatible traces \( \mathcal{P}(t \cdot \alpha \cdot q) \) in which for each trace \( t' \in \mathcal{P}(t) \) we have a set of extended traces considering all the possible ordering among the actions in \( \alpha \) that is \( \mathcal{P}(t \cdot \alpha \cdot q) = \{ t'' \cdot t'''' | t'' \in \mathcal{P}(t), t'''' \in \mathcal{P}(\text{last}(t'') \cdot \alpha \cdot q) \} \). According to Proposition 7.38 p. 84 execution of an action causes the lattice extension considering all the possible orderings with the rest of the execution of the system. Therefore for each trace \( t' \) in set \( \mathcal{P}(t \cdot \alpha \cdot q) \) there exists a corresponding path \( \pi \) in the lattice such that \( \pi = \mathcal{R}_\beta(t') \).

  - Any extension of the global trace by execution of a global action \( \beta \subseteq \bigcup_{i \in [1..|B|]} \{ \beta_i \} \) (i.e., \( t \cdot \beta \cdot q \)) does not extend the lattice but the state of the nodes of the lattice. Moreover, according to Definition 7.24 (p. 73), updating the nodes of the path corresponding to a compatible trace \( t \) is done in such a way that the refine function updates the partial states of \( t \) as per Definition 7.35 (p. 83). Therefore, based on our assumption updated path \( \pi \) is still equal to the corresponding refined compatible trace trace \( t' \) after update by the internal action \( \beta \).

For both cases, the proposition holds.

\qed
A.2.8 Proof of Proposition 7.46 (p. 91)

We shall prove that for a given an LTL formula $\varphi$ and a global trace $t$, there exists a global trace $t'$ such that $PROG(\varphi, t') = progression(\varphi, R_\beta(t'))$ with:

$$t' = \begin{cases} 
  t & \text{if last}(t)[i] \in Q_i^f \text{ for all } i \in [1..B], \\
  t \cdot \beta \cdot q & \text{otherwise.}
\end{cases}$$

Where $\beta \subseteq \bigcup_{i \in [1..B]} \{ \beta_i \}$, $\forall i \in [1..B], q[i] \in Q_i^f$ and $progression$ is the standard progression function.

Proof. The proof is done by induction over the length of the global trace $t$.

- Base case. The proposition holds initially where $PROG(\varphi, init) = progression(\varphi, R_\beta(init)) = progression(\varphi, init)$.

- Let us assume that the proposition holds for a global trace $t$.

- We shall prove that the proposition holds for any possible extension of $t$. We consider two cases based on the last element of $t$:

  - if last$(t)[i] \in Q_i^f$ for all $i \in [1..B]$, the next action only could be in set $\alpha \subseteq 2^{Int}$ because all the components are ready and no $\beta$ action is possible. Then, the new global trace is $t \cdot \alpha \cdot q$ and $q$ is a global state in which the state of components involved in $\alpha$ are busy state. According to Definition 7.45 (p. 90), function $PROG(\varphi, t \cdot \alpha \cdot q)$ uses sub-function prog which postpones the formula evaluation by using $X_\beta$ until the busy components execute their busy actions. As soon as the busy components are done with their computations and the corresponding schedulers generate the update events, function $PROG$ evaluates all postponed propositions. Therefore, there exists a stabilized global trace $t' = t \cdot \alpha \cdot q \cdot \beta \cdot q'$ with ready states for all components in $q'$. Moreover, according to Definition 7.35 (p. 83), the refined trace of $t'$ consists of sequence of global ready state so that the construction of ready states is postponed until the occurrence of busy actions of busy components. Hence, $PROG(\varphi, t \cdot \alpha \cdot q \cdot \beta \cdot q') = progression(\varphi, R_\beta(t \cdot \alpha \cdot q \cdot \beta \cdot q'))$.

  - if last$(t)[i] \notin Q_i^f$ for all $i \in [1..B]$, we consider two cases based on the next action:

    * If the next action is a busy action $\beta \subseteq \bigcup_{i \in [1..B]} \{ \beta_i \}$, it follows our base assumption, therefore the proposition holds.
    * If the next action contains actions in set $\alpha \subseteq 2^{Int}$, the components involved in $\alpha$ are added to the set of busy components and their states evaluations are postponed until their next
A.2. PROOFS RELATED TO THE APPROACH FOR MONITORING DISTRIBUTED CBS

ready state (following the first case argument).

In both cases, the proposition holds.

A.2.9 Proof of Theorem 7.47 (p. 92)

We shall prove that for a global trace \( t \) and LTL formula \( \varphi \), each formula attached to the frontier node of the reconstructed computation lattice \( L \) corresponds to the evaluation of a compatible trace, that is \( \forall \varphi' \in \eta^f \Sigma, \exists t' \in \mathcal{P}(t) \cdot PROG(\varphi, t') = \varphi' \).

Proof. The proof is done by induction over the length of the global trace \( t \).

- According to Definition 7.41 (p. 86), initially lattice has only one node \( \text{init}^L = (\text{init}, (0, \ldots, 0), \{\varphi\}) \).
  \( \mathcal{P}(t) = \{\text{init}\} \) and \( PROG(\varphi, \text{init}) = \varphi \) therefore the theorem holds for the initial state.

- We assume that for a global trace \( t \) progression of all the compatible traces of \( t \) exists in the set of formula of the frontier node.

- Any extension creates the corresponding nodes. if the extension take place from the frontier node \( \eta^f \), then the set of formulas of the new frontier is \( \Sigma = \{\text{prog}(LTL', q) \mid LTL' \in \eta^f \Sigma\} \). and for all compatible trace \( t' \in \mathcal{P}(t) \) the corresponding extended compatible trace is \( (t \cdot a \cdot q) \in \mathcal{P}(t \cdot a \cdot q) \).
  According to Definition 7.45 (p. 90), evaluation of each formula \( PROG(\varphi, t \cdot a \cdot q) = \text{prog}(\varphi, q) = \text{prog}(\text{prog}(\varphi, \eta^f \text{state}), q) \). Base on our assumption \( \text{prog}(\varphi, \eta^f \text{state}) \) is the associated progressed formula of the compatible trace \( t' \) which exists in \( \eta^f \Sigma \).

- Extension of the lattice from a non-frontier node also leads to a new frontier node such that the number of compatible traces are increased as well as the number of path of the lattice (see Proposition 7.39 p. 84). For each newly generated path in the lattice there exists a corresponding compatible trace. The set of formulas associated to the new frontier node \( \eta^f \Sigma = \{\text{prog}(LTL, \eta^f \text{state}) \mid LTL \in \eta \Sigma \land (\eta \triangleright \eta^f \lor \exists N \subseteq L^f \text{nodes}. \eta = \text{meet}(N, L) \land \eta^f = \text{joint}(N, L))\} \). Each formula associated to the new frontier is evaluation of one path of the lattice. Similarly, the progression of global trace by function \( PROG \) is done using function \( \text{prog} \), so that the evaluation of each compatible global trace results a formula which exists in the set of formula of new frontier node associated to the compatible global trace.

- Any extension of the global trace by busy actions \( \beta \) on one hand updates the formulas of lattice nodes using function \( \text{upd}_\varphi \) in order to ascertain the truth of falsity of associated \( X_\beta \) modalities, and on
the other hand function $\text{PROG}$ updates the evaluation of the compatible traces, i.e., $\text{PROG}(\varphi, t \cdot \beta \cdot q) = \text{UPD}(\varphi, Q^r)$ where $Q^r$ is the set of update states after busy actions. According to Definition 7.45 (p. 90), function $\text{UPD}$ uses function $\text{upd}_\varphi$ so the progressed formula of each compatible global trace is updated in the similar way as the formulas in the frontier node are updated.

\[ \square \]

A.2.10 **Proof of Theorem 7.48 (p. 92)**

We shall prove that the set of formula in the frontier node of the constructed lattice consists in the evaluation of the compatible global traces of the system, that is $\eta^f.\Sigma = \{ \text{PROG}(\varphi, t') \mid t' \in P(t) \}$.

**Proof.** The proof is straightforward since the lattice construction is complete (Proposition 7.39, p. 84), in the sense that for each compatible global trace there exists a path in the constructed lattice and according to Theorem 7.47 (p. 92) each formula in the frontier node corresponds to the evaluation of a compatible global trace.
B.1 RVMT-BIP

RVMT-BIP is a tool integrated in the BIP tool suite for the runtime verification of multi-threaded BIP models. In the following we present the practical details of RVMT-BIP. Note that, Linux, Java 1.6 and C++11 are prerequisite for the installation of the tool.

How to install and run RVMT-BIP.

1. Download the RVMT-BIP tool-set\(^1\).

2. Extract the package.

   The main directory contains:

   - **lib/** directory that contains libraries used by the tool,
   - **bin/** directory that contains the tool binaries,
   - **setup.sh** script to set up the environment, and
   - **README** file that describes the tool in detail.
   - **examples/** directory that contains some examples in separate folders, each folder contains:

     - A BIP file,
     - External C++ codes (if applicable),

\(^1\)RVMT-BIP is available for download at \([70]\).
3. To use the tool on command line, you have to source the `setup.sh` in order to add the binary files to your path,

```bash
$ source setup.sh
```

4. Go to the RVMT main directory and make sure that `bin/RVMT` is executable (using `chmod u+x bin/RVMT`), then choose a specific example directory in `examples/` and run the RVMT-BIP tool to build monitored version of your selected example model, using this command:

```bash
$ RVMT [name-of-input-package] [name-of-root-component] [path-to-output-file]
```

- RVMT takes an input BIP model, which is specified by **package name** and **root-component name**. The package name is matched the file name that contains it (ie. package `Sample` is stored in a file named `Sample.bip`) and root-component name is the declaration of compound type.
- The output results, which are the monitored version of input BIP model along with two other files (`RGT.cpp` and `RGT.hpp`) as further execution requirements, will be stored in the user-defined path `[path-to-output-file]`.

5. Code generation, compile and execute the monitored BIP file. Note that, for this step it is prerequisite that the BIP engine is already installed on your machine.

- code generation:
  ```bash
  $ cmake ..
  ```

- compile and execute:
  ```bash
  $ make
  $ ./system -s -threads [threads-number]
  ```
Example B.1. We provide the BIP model of system Task which is defined in Example 8.6 (p. 100) and depicted in Figure 8.2 (p. 100) to illustrate the functionality of RVMT-BIP. Let us consider task.bip file with root-component name top along with the associated files Monitor.xml, Map_Event_Guard and Guide located in source folder. One can follow the steps below to monitor task.bip at runtime.

- The following commands should be executed on the folder where task.bip is located. For the sake of simplicity the associated files of RVMT-BIP are already copied here.
- $ source setup.sh
- $ RVMT task top output

In the RVMT main directory, output directory consist of the monitored version of input.bip is created.

- RVMT$ cd output
- RVMT/output$ mkdir out
- RVMT/output$ bipc.sh -I . -p task -d "top()" -gencpp-output-dir out -genCPP-follow-used-packages -genCPP-ld-l pthread -genCPP-cc-I $PWD
- RVMT/output$ mkdir out/build
- RVMT/output$ cd out/build
- RVMT/output/out/build$ cmake ..
- RVMT/output/out/build$ make
- And finally, execute using 3 threads:
  $ ./system -s --threads 3

BIP code of system Task task.bip

@ cpp(include="unistd.h")
package task
extern function printf(string, int)
extern function usleep(int)
port type Port()
connector type Sync3 (Port port1, Port port2, Port port3)
define port1 port2 port3
connector type Unary (Port port1)
define port1
end
atom type worker()
data int x
export port Port exec()
export port Port finish()
place done, free
initial to free do {x = 0;}
on exec from free to done do {x = (x + 1);}
on finish from done to free
end
atomic type task()
export port Port deliver()
export port Port newtask()
place ready, delivered
initial to ready
on deliver from ready to delivered
on newtask from delivered to ready
end
compound type top()
component task generator()
component worker worker1()
component worker worker2()
component worker worker3()
connector Sync3 ex12 (worker1.exec, worker2.exec, generator.deliver)
connector Sync3 ex23 (worker2.exec, worker3.exec, generator.deliver)
connector Sync3 ex13 (worker1.exec, worker3.exec, generator.deliver)
connector Unary f1 (worker1.finish)
connector Unary f2 (worker2.finish)
connector Unary f3 (worker3.finish)
connector Unary nt (generator.newtask)
end
end

monitor description Monitor.xml

<VerificationMonitor kind="MEALY">
<States>
  <State initial="true" id="s1">
    <Transition output="currently_good" nextState="s1" event="event1"/>
    <Transition output="currently_bad" nextState="s2" event="event2"/>
  </State>
  <State id="s2">
    <Transition output="currently_good" nextState="s1" event="event1"/>
    <Transition output="currently_bad" nextState="s2" event="event2"/>
  </State>
</States>
<Alphabet>
  <string>event1</string>
  <string>event2</string>
</Alphabet>
</VerificationMonitor>

Map_Event_Guard

event1 : 
  !(((worker1_x-worker2_x>100)||(worker2_x-worker1_x>100))||
   ((worker1_x-worker3_x>100)||(worker3_x-worker1_x>100))||
   ((worker2_x-worker3_x>100)||(worker3_x-worker2_x>100)))
event2 : 
  ((worker1_x-worker2_x>100)||(worker2_x-worker1_x>100))||
   ((worker1_x-worker3_x>100)||(worker3_x-worker1_x>100))||
   ((worker2_x-worker3_x>100)||(worker3_x-worker2_x>100))
B.2 RVDIST

RVDIST is a prototype tool for LTL runtime verification of distributed (component-based) systems, written in the C++ programming language. RVDIST uses Spot [28] platform which is a C++11 library for LTL manipulation. RVDIST takes as input a configuration file config.ini describing the architecture of the distributed system (i.e., number of schedulers, number of components, initial state, LTL formula to be monitored, mapping of atomic propositions to components) and a list of events events.data with the following details.

Events and configuration format. Let us assume that we have $m$ scheduler and $n$ components.

- **events.data** consists of a list of events.
  - Each event is placed in one line.
  - Each event is either an action event or an update event.
  - **Action events format:** $1, j, vc[1], \ldots, vc[m], a[1], \ldots, a[n]$
    - $j$ is the index of scheduler $S_j$, with $j \in [1..m]$, which has executed the corresponding interaction and sent the action event,
    - $vc[1], \ldots, vc[m]$ are associated to the vector clock of the executed interaction,
    - $a[i]$ for $i \in [1..n]$ is 1 if component $B_i$ is involved in the interaction, and 0 otherwise.
  - **Update events format:** $2, j, i, q$
    - $j$ is the index of scheduler $S_j$ which generated and sent the update event,
    - $i$ is the index of the associated component $B_i$ with the update event,
    - $q$ is the updates state which is a mapping of the component state to an integer.

- **config.ini** has the parameters of system.
B.2. RVDIST

- schedulers: \( m \),
- components: \( n \),
- initial_component\(_{[i]}\) for \( i \in [1..n] \): the initial state of component \( B_i \),
- phi: LTL formula (desired property),
- component\(_{[p]}\): the associated component of atomic proposition \( p \),
- value\(_{[p]}\): the truth value of atomic proposition \( p \),
- states_component\(_{[i]}\) for \( i \in [1..n] \): number of the states of component \( B_i \),
- value_states_component\(_{[i]}\)\(_{[k]}\) for \( i \in [1..n] \) and \( k \): mapping of state \( k \) of component \( B_i \) to a string.

Example B.2. To illustrate how RVDIST is applied on a real model, we use the robotic model presented in Section 11.2.1 (p. 131), and show how the configuration file and the list of events are structured. Note that, we used system ROBLOCO which is a large BIP model in terms of lines of code (ca. 1200 LOC). Thus the actual code is not presented in this section.

```
Configuration

schedulers=3
components=3
initial_component_1=idle
initial_component_2=start
initial_component_3=idle
phi= G( p1 -> (X(!p3) U p2) )
component_p1=1
component_p2=2
component_p3=3
value_p1=free
value_p2=start
value_p3=startodo1
states_component_1=20
states_component_2=3
states_component_3=22
value_states_component_1_1=idle
```
value_states_component_1_2=ready
dvalue_states_component_1_3=managets
dvalue_states_component_3_20=startmon2
dvalue_states_component_3_21=endmon2
dvalue_states_component_3_22=finish

At runtime, 3463 events are sent to the observer. Some of them are listed below.

List of events

1,1,1,0,0,0,1,1 action event by scheduler_1, vc=(1,0,0), including component_1
2,1,1,19 update event by scheduler_1, reporting the updated state of component_1 (19)
1,1,2,0,0,0,1,1
2,1,1,20
1,1,3,0,0,0,1,1
2,1,1,19
1,1,4,0,0,0,1,1
.
.
.
1,3,726,351,484,1,1,0
2,3,3,20
1,1,730,350,479,0,1,1
2,1,1,20
1,3,726,351,485,1,1,0
2,3,3,22
1,2,726,352,485,1,0,0 action event by scheduler_2, vc=(726,352,485), including component_2 and component_3
2,2,2,2

Finally, we have the following results.
Results
- Number of Observed Events : 3463
- Number of Lattice Nodes : 17
- Number of Removed Nodes : 10602
- Vector Clock of the Frontier Node : 730 352 485
- Number of Paths to the Frontier : 2.491E+544
- Formula Associated to the Frontier node:
  # 1.36E+541: Phi
  # 2.490E+544: !p3 & !xb2p3 & (X!p3 U p2) & G(p1 -> (X!p3 U p2))

The results show that among 2.491E+544 paths, 1.36E+541 of them are currently satisfy the property, and 2.490E+544 of them are kept on hold, for scheduler_3 to send the updated state of component_3, to be evaluated. All the paths evaluations are stored in the frontier node with the vector clock of (730, 352, 485). Any further execution of the system would extend the lattice from one of the 17 remained nodes. Note that, in this example the number of the paths are approximate for the sake of simpler presentation. The tool outputs are accurate.
Bibliography


Distributed Systems Laboratory, The University of Texas at Austin, 1994. available via ftp or WWW at maple.ece.utexas.edu.


