Runtime Verification of Hierarchical Decentralized Specifications

Antoine El Hokayem

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Antoine EL HOKAYEM

Thèse dirigée par **Yliès Falcone**, Université Grenoble Alpes

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Runtime Verification of Hierarchical Decentralized Specifications

Vérification à l'Exécution de Spécifications Décentralisées Hiérarchiques

Thèse soutenue publiquement le **18 Décembre 2018**,
devant le jury composé de :

M. Saddek Bensalem
Professeur, Université Grenoble Alpes, Président du jury
M. Thierry Jéron
Directeur de Recherche, INRIA Rennes Bretagne-Atlantique, Rapporteur
M. Martin Leucker
Professeur, Université de Lübeck, Rapporteur
M. Sylvain Hallé
Professeur, Université du Québec à Chicoutimi, Examinateur
M. Klaus Havelund
Senior Research Scientist, NASA Jet Propulsion Laboratory, Examinateur
M. Leonardo Mariani
Professeur, Université de Milano Bicocca, Examinateur
M. Yliès Falcone
Maître de Conférences, Université Grenoble Alpes, Directeur de thèse

Runtime Verification of
Hierarchical Decentralized Specifications

Abstract

Runtime Verification (RV) [LS09a, FHR13, BFB+17a] is a lightweight formal method which consists in verifying that a run of a system is correct with respect to a specification. The specification formalizes the behavior of the system typically using logics (such as variants of Linear-Time Temporal Logic, LTL) or finite-state machines. The system is considered as a black box that feeds events to a monitor. An event is a set of atomic propositions that describe some abstract operations or states in the system. The sequence of events transmitted to the monitor is referred to as the trace. Based on the received events, the monitor emits verdicts in a truth domain that indicate whether the run complies or not with the specification. For this thesis, we consider the truth domain to be a set $\{\top, \bot, ?\}$ where verdicts $\top$ and $\bot$ indicate respectively that a program complies or violates the specification, and verdict ? indicates that no final verdict could be reached yet. Truth domains can also include additional verdicts such as currently true and currently false, to indicate a finer grained truth value [BLS10]. While RV comprehensively deals with monolithic systems, multiple challenges are presented when scaling existing approaches to decentralized systems, that is, systems with multiple components with no central observation point. These challenges are inherent to the nature of decentralization; the monitors have a partial view of the system and need to account for communication and consensus.

In this thesis, we focus particularly on three challenges: managing partial information, separating monitor deployment from the monitoring process itself, and reasoning about decentralization in a modular and hierarchical way. Several algorithms have been designed [BF12, FCF14, BFRT16, CF16a] and used [Bar13] to monitor decentralized systems. They typically consist of starting with an initial global formula of the system, then depending on the technique, the approaches utilize monitors to relay partial information, decide consensus, or monitor a partial part of the specification and rely their information to other monitors.

We focus on the notion of decentralized specification wherein multiple specifications are provided for separate parts of the system. The system specification is constructed bottom up, by defining specifications over other specifications. Taking into account dependencies and abstracting subspecifications provides various advantages such as allowing for realistic monitor synthesis of the specifications, and the ability to modularize specifications. We recall that synthesis algorithms are doubly exponential in the size of the formula, it becomes infeasible to synthesize a monitor for a large formula representing all the properties of a system. We also present a general monitoring algorithm for decentralized specifications, and a general datastructure to encode automata execution with partial observations. This allows us to compare existing approaches in a uniform manner predictably since we use automata. We develop the THEMIS tool, which provides a platform for designing decentralized monitoring algorithms, metrics for algorithms, and simulation to better understand the algorithms. THEMIS provides tools for designing reproducible experiments which can be extended to include new algorithms and metrics.

We illustrate the approach with multiple applications. First, we use decentralized specifications to analyze, adapt, compare, and simulate three existing decentralized monitoring algorithms [CF16a]. We perform a worst-case analysis and look at the usefulness of simulations to determine the advantages or disadvantages of certain algorithms in specific scenarios. We compare the algorithms under two scenarios. The first scenario explores synthetic benchmarks, that is, we consider random traces and specifications. This allows us to account for different types of behavior. The second scenario explores a specific example associated with a common pattern in programming. For that, we consider a publish-subscribe system, where multiple publishers subscribe to a channel (or topic), the channel publishes events to the subscribers. We use the Chiron user interface example [ACD+99, Tea99], along with the specifications formalized for it [DAC99a].

Second, we use decentralized specifications to check various properties in a smart apartment. The properties can be broken down into three types: behavioral correctness of the apartment sensors, detection of specific user activities (known as activities of daily living), and composition of properties of the previous types. The context of the smart apartment provides us with a complex system with a large number of components with two different hierarchies to group properties and sensors: geographically within the same room, floor, or globally in the apartment, and logically following the different types of properties. This allows us to re-use specifications, and combine them to: (1) scale beyond existing centralized RV techniques, and (2) greatly reduce computation and communication costs.

Furthermore, we elaborate on utilizing decentralized specifications for the decentralized online monitoring of multithreaded programs. We first expand on the limitations of existing tools and approaches when meeting the challenges introduced by concurrency, and ensure that concurrency needs to be taken into account by considering partial orders in traces, we provide perspectives on the monitoring of multithreaded programs using a form of decentralized specifications, where different specifications are provided for the various concurrency regions in a program. We detail the description of such concurrency areas in a single program execution, and provide a general approach which allows re-using existing RV techniques. In our setting, monitors are deployed within specific threads, and only exchange information upon reaching synchronization regions defined by the program itself. That is, they use the opportunity of a lock in the program, to evaluate information across threads. As such, we refer to this approach as *opportunistic* RV for multithreaded programs. The general approach provides additional semantics for delimiting and processing concurrent regions, so that the result can be checked soundly using existing techniques. By using the existing synchronization, our approach reduces additional overhead and interference to synchronize at the cost of adding a delay to determine the verdict. We utilize a textbook example of *readers-writers* as it contains concurrent regions, and show how opportunistic RV is capable of expressing specifications on concurrent regions, without incurring significant delay. We present a manual monitoring implementation for *readers-writers*, and show that the overhead of our approach scales particularly well with the number of concurrent events in a given region.

Keywords: *decentralized monitoring, decentralized specifications, runtime verification, LTL, LTL3 monitors, smart homes, multithreaded programs, eventual consistency, concurrency.*

Vérification à l'Exécution de
Spécifications Décentralisées Hiérarchiques

Résumé

La vérification à l'exécution [LS09a, FHR13, BFB+17a] (RV) est une méthode formelle légère qui consiste à vérifier qu'une exécution d'un système est correcte par rapport à une spécification. La spécification exprime de manière rigoureuse le comportement attendu du système, en utilisant généralement des formalismes basés sur la logique (comme les variantes de la Logique Temporelle Linéaire, LTL) ou les machines à états finies. Le système est considéré comme une boite noire qui fournit des évènements à un moniteur. Un évènement est un ensemble de propositions atomiques qui décrit des opérations ou des états abstraits dans le système. La suite d'évènements transmis au moniteur est appelée la trace. À partir des évènements reçus, le moniteur produit des verdicts dans un domaine de vérité qui indique si l'exécution est conforme ou non à la spécification. Pour cette thèse, nous considérons le domaine de vérité $\{\top, \bot, ?\}$, dans lequel les verdicts $\top$ et $\bot$ indiquent respectivement qu'un programme respecte ou viole la spécification, et le verdict ? indique qu'aucun verdict final n'a pu être atteint pour le moment. Les domaines de vérité peuvent aussi inclure des verdicts additionnels tels que « correct pour le moment » et « incorrect pour le moment », pour indiquer une valeur de vérité plus précise [BLS10]. Alors que la verification a l'éxecution traite les systèmes monolithiques de manière exhaustive, plusieurs difficultés se présentent lors de l'application des techniques existantes à des systèmes décentralisés, c-à-d. des systèmes avec plusieurs composants sans point d'observation central. Ces difficultés sont inhérentes à la nature de la décentralisation; les moniteurs ont une vue partielle du système et requièrent une prise en compte des problèmes de communication et de consensus.

Dans cette thèse, nous nous concentrons particulièrement sur trois problèmes : la gestion de l'information partielle, la séparation du déploiement des moniteurs du processus de vérification lui-même et le raisonnement sur la décentralisation de manière modulaire et hiérarchique. Plusieurs algorithmes ont été conçus [BF12, FCF14, BFRT16, CF16a] et utilisés [Bar13] pour vérifier des systèmes décentralisés. Ils consistent généralement à partir d'une formule initiale globale du système, puis, selon la technique, les approches utilisent des moniteurs pour relayer l'information partielle, décider les consensus, ou vérifier une partie de la spécification et relayer l'information aux autres moniteurs.

Nous nous concentrons sur la notion de spécification décentralisée dans laquelle plusieurs spécifications sont fournies pour des parties distinctes du système. La spécification du système est construite de bas en haut, en définissant les spécifications en s'appuyant sur d'autres spécifications. Prendre en compte les dépendances et abstraire les sous-spécifications a divers avantages tels que permettre une synthèse de moniteurs à partir des spécifications complexes et la possibilité de modulariser les spécifications. Nous rappelons que comme les algorithmes de synthèse sont doublement exponentiels sur la taille de la formule, il devient impossible de synthétiser un moniteur pour une grosse formule représentant toutes les propriétés du système. Nous présentons également un algorithme général de vérification pour les spécifications décentralisées et une structure de données pour représenter l'exécution d'un automate avec observations partielles. Cela nous permet de comparer les approches existantes de manière uniforme et prévisible puisque nous utilisons les automates. Nous développons l'outil THEMIS, qui fournit une plateforme pour concevoir des algorithmes de vérification décentralisée, des mesures pour les algorithmes et une simulation pour mieux comprendre les algorithmes. THEMIS fournit des outils pour concevoir des expérimentations reproductibles qui peuvent être étendues pour inclure de nouveaux algorithmes et de nouvelles mesures.

Nous illustrons notre approche avec diverses applications. Premièrement, nous utilisons des spécifications décentralisées pour analyser, adapter, comparer et simuler trois algorithmes de vérification décentralisée existants [CF16a]. Nous menons une analyse de pire cas et étudions l'utilité des simulations pour déterminer les avantages et inconvénients de certains algorithmes dans des scénarios spécifiques. Nous comparons des algorithmes dans deux scénarios. Le premier scénario explore des benchmarks synthétiques : nous considérons des traces et spécifications générées aléatoirement. Cela nous permet de prendre en compte différents types de comportement. Le deuxième scénario explore un exemple spécifique associé avec un schéma répandu en programmation. Pour cela, nous considérons un système de type publication-abonnement, dans lequel de multiples diffuseurs s'abonnent à un canal (ou sujet), le canal publie des évènements aux abonnés. Nous utilisons l'exemple de l'interface humain-machine Chiron [ACD+99, Tea99], avec la spécification formalisée pour cet exemple [DAC99a].

Deuxièmement, nous utilisons des spécifications décentralisées pour vérifier diverses propriétés dans un appartement intelligent. Les propriétés peuvent être classées dans trois catégories : correction du comportement des capteurs de l'appartement, détection d'activité spécifiques de l'utilisateur (activités de la vie quotidienne) et composition de propriétés des deux catégories précédentes. Le contexte de l'appartement intelligent nous fournit un système complexe avec un grand nombre de composants avec deux hiérarchies différentes pour groupe les propriétés et les capteurs : géographiquement dans la même pièce, le même étage ou globalement dans l'appartement, et suivant logiquement les différents types de propriétés. Cela nous permet de réutiliser les spécifications, et de les combiner pour: (1) aller au-delà des techniques RV centralisées existantes, et (2) réduire grandement les coûts en calcul et communication.

Enfin, nous élaborons sur l'utilisation de spécifications décentralisées pour la vérification décentralisée pendant l'exécution de programmes parallélisés. Nous commençons par discuter les limitations des approches et des outils existants lorsque les difficultés introduites par le parallélisme sont rencontrées et affirmer que le parallélisme doit être pris en compte en utilisant des ordres partiels dans les traces, nous ouvrons des perspectives sur la vérification de programmes parallèles en utilisant une forme de spécifications décentralisées, où les différentes spécifications sont fournies pour les diverses zones parallèles dans un programme. Nous détaillons la description de telles zones de parallélisme d'une unique exécution d'un programme et décrivons une approche générale qui permet de réutiliser des techniques RV existantes. Dans notre configuration, les moniteurs sont déployés dans des fils d'exécutions spécifiques et échangent de l'information uniquement lorsque des points de synchronisation définis par le programme lui-même sont atteints. Autrement dit, ils prennent partie d'un verrou dans le programme pour évaluer l'information à travers les fils d'exécutions. Ainsi, nous qualifions cette approche de *vérification opportuniste* pour les programmes parallèles. L'approche générale fournit une sémantique additionnelle pour délimiter et traiter les zones concurrentes, afin que le résultat puisse être vérifié correctement en utilisant les techniques existantes. En utilisant les points de synchronisation existants, notre approche réduit les interférences et surcoûts résultant de la synchronisation, au prix d'un retard pour déterminer le verdict. Nous utilisons un exemple typique des *lecteurs et rédacteurs* parce qu'il contient des zones concurrentes et montre la capacité de la vérification opportuniste d'exprimer des spécifications sur des zones concurrentes, sans entraîner de retard significatif. Nous présentons une implémentation de vérification manuelle pour le problème des *lecteurs et rédacteurs* et montrons que le surcoût de notre approche passe particulièrement bien à l'échelle par rapport au nombre d'évènements concurrents dans une zone donnée.

Mots-clés: *vérification décentralisée, monitoring décentralisé, spécifications décentralisées, vérification à l'exécution, runtime verification, LTL, moniteurs LTL3, habitats intelligents, programmes parallèles, concurrence.*

Acknowledgements

I would first like to thank Yliès Falcone for being always available to discuss, guide, and provide valuable discussions. The discussions and guidance were invaluable to bringing the work presented in this thesis to life, and also introducing me to the academic world in general. The energy, enthusiasm, and discipline of Yliès made some contributions happen that would have otherwise been missed, and helped improve the maturity of the work and the level of polish.

I would also like to thank Martin Leucker and Thiérry Jéron for reviewing the manuscript and providing feedback that overall improved the quality of the thesis. I am also grateful for Saddek Bensalem, Sylvain Hallé, Klaus Havelund, and Leonardo Mariani for being part of the jury.

I am grateful to all the team members of CORSE for the discussions we had during work, also for providing feedback on the various forms of the work featured in this thesis. In particular, I would like to thank Kevin for his early support and discussions which helped me settle in, Laercio, Fabian, Raphaël, and Manuel for discussions, reviews and feedback on the work featured in the thesis. A special thank you to Imma for minimizing the impact of having to deal with huge paperwork throughout the three years of my stay with CORSE, and helping with the organization of the defense in itself.

Ultimately, a PhD does not only involve the thesis and the academic work. I am very thankful to the various people who were a pleasure to be around during my stay in Grenoble: Emilie, Fabian (in particular, for actively organizing plans), Lotfi, Luis, Rana, Rani, Raphaël, Rim, and Sahar.

Finally, this thesis would not have been possible without the support of my family who have been very encouraging in the pursuit of this path.

Contents

CHAPTER 1

Introduction

"When we had no computers, we had no programming problem either.
When we had a few computers, we had a mild programming problem.
Confronted with machines a million times as powerful,
we are faced with a gigantic programming problem."

– Edsger W. Dijkstra, EWD963, 1986

Computer and information systems are becoming ubiquitous in everyday life. Human societies now rely heavily on automation. Computer systems are responsible for controlling a wide range of systems spanning space, aviation, nuclear reactors, military equipment, medical equipment, business processes, financial markets, supply chain management, entertainment (video games, movie production), and phone applications. Computer systems now form the infrastructure of a modern society and are becoming more complex. Modern information systems operate on a larger scale than ever, spanning multiple computers, or utilizing complex architectures and parallelism. As we grow more dependent on computer systems, and as the systems themselves become more complex, it is crucial to be able to know that they are indeed running as expected. Nobody wants to live in a building with faulty infrastructure after all.

Unfortunately, computer systems often include errors (informally referred to as *"bugs"*). Software errors are not all dangerous, as they can impact various parts of the system, ranging from minor inconvenience, to failure of the software to perform its functions. Some errors can even go unnoticed until combined with other errors, leading to yet more confusion. Ensuring that a system has no errors is a massive, and sometimes impossible task; as such, multiple techniques are introduced and combined to ensure more confidence that the system performs as expected – as Dijkstra so well put it: *"Testing shows the presence, not the absence of bugs"*. By employing more techniques, we raise our confidence level in the software, but also increase the cost of the software. Therefore, reliability is one of many other considerations (such as performance) when building software, and it is often not the priority. Improving reliability while maintaining low costs is a challenge in software development.

Reliability is a priority for *critical systems*. A critical system is a system whose failure results in loss of life, major environmental harm (or disaster), or significant economic loss. Consider a phone application: a possible crash is easily averted in the worst case by simply rebooting the specific phone in question. While inconvenient, and so long as it does not happen frequently, it has no significant impact. However, a phone update which modifies the firmware of the phone and causes hardware damage is to be considered a critical system, as phones will no longer be operable and thus a recall is necessary, introducing a financial burden on the company. Historically various errors have caused critical systems to fail or provide unreliable information. To illustrate the scale and potential consequences, we present *four* failures in Example 1.

Example 1 (Failure of critical systems) The Therac-25 was a computer-controlled radiation therapy machine produced in 1982. Between the years 1985 and 1987, the faulty software controlling the machine led to administering lethal doses of radiation to patients killing three and critically injuring three [Baa08]. The error was a

synchronization error, wherein the machine would be in different modes based on how fast the operator entered keystrokes, without updating properly the display to the operator, leaving them unaware of the issue.

In December 1994, Intel recalled its early Intel Pentium processors due to an error affecting the float-pointing unit [Nic11]. The error (referred to as the *FDIV* bug) caused some divisions to return an incorrect result. The total cost of the product recall was $475 million.

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded forty seconds after its lift-off during its first voyage [LLF+96]. The rocket was a result of a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. The report states the main cause to be a *"software exception [..] during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer"*.

Finally, we present the case of Oko, the soviet early-warning missile defense system during the cold war era. Oko utilizes various satellites to warn against incoming nuclear ballistic missiles by detection of their engines' exhaust plume in infrared light. On 26 September 1983, shortly after midnight, the computer reported that one intercontinental ballistic missile was heading toward the Soviet Union from the United States [Hof99]. Luckily, the officer on duty – lieutenant colonel Stanislav Petrov, correctly assumed that it was a false alarm. Petrov reasoned that a first strike by the United States would involve more than one missile, as it would trigger the Soviet Union to retaliate with its full nuclear arsenal. Multiple false alarms were issued (and ignored) on the same night. The false alarms were caused by rare alignment of sunlight on high-altitude clouds and the satellites' orbits. Had Petrov followed protocol, *the software error could have started a nuclear war*. *

Realizing that the cost of failure is high in some situations, multiple techniques have been developed to increase the confidence in the correctness of systems. The improvements encompass all aspects of the system's cycle, including techniques for managing the process of the system development, regulation that requires specific methods as well as processes be utilized to set minimums for industries, and approaches for verifying the code which constitutes the system. For the scope of this thesis, we will be focusing on verifying the correctness of the code itself.

1.1 Software Verification

Software verification is a discipline of software engineering concerned with ensuring that software meets the expected behavior by its designers. The expected behavior is referred to as a *specification*. Techniques for software verification are numerous. When discussing them we focus on six considerations.

1.1.1 Considerations for Verification

First, we focus on the *soundness* and *completeness* of the approach. Soundness states that the approach is capable of detecting correctly when the program conforms to or violates the specification. That is, whenever the verification logic detects that a program has conformed to (resp. violated) the specification, the program has indeed conformed to (resp. violated) the specification. Completeness states that all conformance and violations are detected by the approach. In this case, the approach does not miss cases where the program violates or complies with the specification. A sound and complete approach is capable of detecting all situations where a program violates or complies with the specification, and all the detected instances are detected correctly. Verification techniques seek to always be sound; when discussing them we will focus on completeness.

Second, we consider the *rigor of the method*: this involves typically the modeling of the system or specification. As such, we distinguish between informal approaches and formal approaches. Formal approaches rely on mathematical models of the system and expected behavior (specification). The additional modeling, formalization, and checking of the system can be expensive both in terms of budget and computational resources. Formal approaches may also be required in specific domains by standards. For example, they are required for avionics software compliant with the DSO-178C standard [GP12], and for railway software compliant with the highest safety integrity level (SIL4) in the CENELEC 50128 standard [EN501, FFM12b].

Third, we determine the *degree of internal knowledge of a system*: a system is seen either as a black box, a gray box, or a white box. A black-box system is a system whose internals are not known. That is, the verification procedure has no access to the internals of the system, but typically only to its interface. In contrast, a white-box system is a system whose entire code is known. A gray-box system is a system which some internals are known but others are not. For example, consider a process that invokes a system call or another library. While the code of the process may be known, a developer is only aware of the interface of the additional call, but not its code.

Fourth, we distinguish whether the analysis occurs *using only the program source, or during the program execution*. The former is referred to as *static analysis*, it relies on information extracted from the program source (high language source file, or binary format). Static analysis is generally performed once to verify the source, paying the computational cost up front and once for the verification. The latter is referred to as *dynamic analysis*, it relies on information extracted while the program is executing. Dynamic analysis is performed at every execution, and thus its overhead is directly added to the program execution. Since dynamic analysis is performed during the execution, it typically considers states reached by the program at runtime. Additionally dynamic analysis is useful when we are dealing with non white-box systems, as certain conditions cannot be modeled or controlled for.

Fifth, we distinguish the *level of automation*. An approach is either *manual* or *automatic*. Manual approaches require developers to write the verification code. Automatic approaches are capable of synthesizing all needed verification code, and also integrating it automatically with the target program. Automatic approaches are preferred as they reduce the possibility of human error in the verification code. Manual approaches are typically well adapted to the target system and specification, they are generally more fine tuned, and have better performance. We also classify interactive approaches as manual, since the developer input may be required during the verification.

Sixth, we focus on the *scalability* of the approach. The more scalable the approach, the more it is able to tackle more complex and larger systems. Scalability in our setting is based on the computational cost needed to perform the verification.

Using the different conditions to classify techniques, we now present some representative verification techniques.

1.1.2 Testing

Testing generally involves establishing a relationship between a given input and an expected output of the part, or entirety of the program being tested [Kas18, BMP18, Run06]. Testing can be done manually, or through tests ran or generated automatically.

Manual testing. Developers rely on a wide set of techniques when performing manual testing [IML09]. These techniques includes pre-scripted scenarios in which a developer performs manually a testing scenario set forth by a designer, top-down exploration of the software in which a developer starts from the highest level operations in the software and refines the testing to more details, or simply relying on experience instead of documentation to attempt to find bugs. These techniques are all informal, manual, and not complete. While time consuming, they do verify minimal functionality with respect to usage scenarios.

Automated test execution. To avoid the tedious effort required to perform manual testing, it is possible to utilize assertions and exception handling to write custom (informal) tests that can be executed automatically alongside the application at runtime [Run06]. Testing frameworks – like the successful JUnit [DF14, Bec04] framework for Java – are capable of managing and automatically executing a large collection of tests. The tests target typically an interface at various levels of an application (method, or all methods in a class), referred to as a *unit*. Tests are called *unit tests*, they verify parts of the software independently Unit testing adapts to all levels of the system knowledge as tests operate at the level of an interface. Unit tests account for the input generated by the tester, they are not complete. Executing unit tests is highly scalable, as it can be applied to a large system, as it targets units of it. However, designing tests for a full system is a tedious task.

Automated test generation. Designing tests is also challenging when accounting for edge cases, or unexpected input for the system. While the automatic test execution relies on a human tester to design the test, it is also possible to generate the tests automatically [GA14, RC17, McM04]. Different approaches combine static and dynamic

Table 1.1: Comparison of verification techniques.

Technique	Complete	Formal	Knowledge	Stage	Automated	Scalable
Testing	no	some	grey-box	dynamic	some	execution only
Model Checking	yes	yes	white-box	static	yes	no
Bounded Model Checking	no	yes	white-box	hybrid	yes	yes
Theorem Proving	no	yes	white-box	static	some	no
Assertion Checking	no	yes	white-box	dynamic	some	yes
Runtime Verification	no	yes	black-box	dynamic	yes	yes

analysis, and range from formal approaches that synthesize tests from proofs [EH07] or from a formal model of the system (known as *conformance testing* [AMF14, ACM+18]) to non-formal approaches that synthesize tests using genetic algorithms [RDCN18].

1.1.3 Techniques Relying on Formally Modeling Programs

Model checking. Model checking approaches [GV08, CE81, QS08] reason on a model of the program. As such, they formalize the program behavior as a model and the specification using (temporal) logics, and then verify that the model complies or violates the specification. This allows such approaches to be complete, i.e. to exhaustively account for all edge cases that a programmer might be unaware of. Model checking approaches need to model all possible states of a program, and as such often end up suffering from a combinatorial explosion, and are therefore not very scalable. To avoid explicitly enumerating all possible states and improve scalability, it is possible to use (sound) abstractions of the program semantics with techniques like abstract interpretation [CC77]. Furthermore, it is possible to define abstractions over groups of states by constraint solving. This is known as bounded model checking [CBRZ01].

Contracts. Another approach to model the behavior of programs is to use contracts. Contracts are formally based on Hoare triples [Hoa69], and are used to describe the expected behavior of a system. A triple $(\{P\}\ C\ \{Q\})$ consists of a given system operation (C) and two predicates (also called assertions): a pre-condition (P) and a post-condition (Q) governing the effect of the operation on the system. The triple states that if P holds before executing an operation C, then Q must hold after executing C, assuming C terminates without errors[1]. Hoare triples allow us to also express invariants on the program states (i.e., having the same predicate as pre-condition and post-condition). The principle is used for modeling Java programs with the *Java Modeling Language* (JML) [LBR06], and C programs with the *ANSI/ISO C Specification Langage* (ACSL) [BFM+18, Dor15]. Contracts can be checked statically using deductive reasoning with techniques such as weakest precondition [Dij75], and theorem proving [Duf91]. Checking contracts statically can be costly, and as such is hard to scale. They can also be checked dynamically by evaluating the predicates at runtime, as in assertion checking [SKV17, AGVY11, KHBZ15, RLL+13]. Finally, static and dynamic approaches can be combined to minimize the assertions needed to be checked online. Frameworks such as Frama-C [KKP+15] use a plugin system to combine multiple approaches. Furthermore, while assertions are typically written down by users, they can also be generated automatically. For example, Frama-C automatically generates assertions to account for valid memory accesses and pointer de-referencing.

We next present *runtime verification*, a formal method which consists in verifying that a single run of a system is correct with respect to a specification.

[1] Hoare triples as defined in [Hoa69] do not express the termination property for the entire program, it must be checked separately.

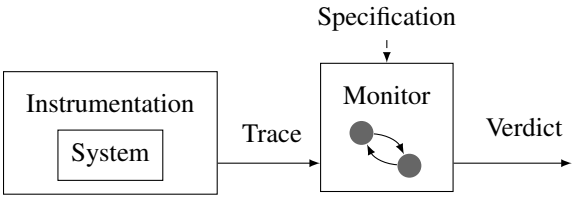


Figure 1.1: Overview of basic concepts of RV and their interactions.

1.2 Runtime Verification

1.2.1 Overview

Runtime Verification (RV) [LS09a, FHR13, BFB+17a] is a lightweight formal method which consists in verifying that a run of a system is correct with respect to a formal specification typically expressed in logics (such as variants of Linear-Time Temporal Logic, LTL) or finite-state machines. Figure 1.1 illustrates how RV is used to perform verification. First, a monitor is *synthesized* (usually automatically) from the *specification*. It contains code necessary to perform verification against the specification. Second, the system is *instrumented* to generate and feed *events* to a *monitor*. An event usually consists of a set of *observations*. An observation associates an atomic proposition that describes some abstract operation or state in the system to a truth value. The sequence of events transmitted to the monitor is referred to as the *trace*. The system can be typically seen as a black-box since, for performing verification, all that is required is the trace. Third, based on the received trace, the monitor emits *verdicts* in a truth domain that indicates whether or not the run complies with the specification. A typical *truth domain* is the set $\{\top, \bot, ?\}$ where verdicts $\top$ and $\bot$ indicate respectively that a program complies or violates the specification, and verdict ? indicates that no final verdict could be reached yet. Truth domains can also include additional verdicts such as currently true and currently false, to indicate a finer grained truth value. We illustrate a simple case of a light switch and a light bulb in Example 2.

Example 2 (Switch triggers bulb) Consider a system that contains a light switch and a light bulb. The possible states for each of the switch and bulb consist of being *on* or *off*. Let us associate the states with the atomic propositions s and ℓ, respectively. Using the atomic propositions, we are able to generate observations about the system. For example, the observation $\langle s, \bot\rangle$ indicates that the switch is in the state *off*. An event is simply a set of observations. The event $\{\langle s, \top\rangle, \langle \ell, \bot\rangle\}$ indicates that the switch is *on*, and the light is *off*. We next define (informally) a property of the system: *"The light bulb must be on one timestamp after the switch is on, until the switch is turned off"*. Now let us consider two traces: $tr_0 \stackrel{\text{def}}{=} \{\langle s, \bot\rangle, \langle \ell, \bot\rangle\} \cdot \{\langle s, \top\rangle, \langle \ell, \bot\rangle\} \cdot \{\langle s, \top\rangle, \langle \ell, \top\rangle\} \cdot \{\langle s, \bot\rangle, \langle \ell, \bot\rangle\}$ and $tr_1 \stackrel{\text{def}}{=} \{\langle s, \bot\rangle, \langle \ell, \bot\rangle\} \cdot \{\langle s, \top\rangle, \langle \ell, \bot\rangle\} \cdot \{\langle s, \top\rangle, \langle \ell, \bot\rangle\}$. We see that tr_0 complies with the specification, while tr_1 violates it as the light is not turned on after the switch is turned on. *

Since RV targets one execution of the program, it explores a smaller number of states than model-checking does, avoiding the state explosion problem. RV sacrifices completeness in order to be scalable and expressive when verifying only one run of the system. Table 1.1 shows how RV compares to some of the techniques mentioned in the earlier section (Section 1.1). We note that verifying that the currently running program is behaving according to the specification is useful, as some factors could impact the program beyond its own code. These are typically environmental (external) to the program, such as hardware failure and radiation induced faults [dOPSR16]. This allows the program to detect correctly the error, and also possibly react to it (see Section 1.2.2). A challenge for RV is to keep the overhead low, since the overhead is paid at *runtime* for every execution of the program. Multiple considerations are needed to monitor even a single run. We elaborate on the major concepts often dealt with in runtime verification in the next section.

1.2.2 A Field With Many Considerations ([FKRT18])

Monitoring even a single execution of a program introduces considerations on many levels. In [FKRT18], the authors suggest multiple concepts that RV approaches tend to deal with. We show a high-level mindmap in Figure 1.2.

Figure 1.2: High-level mindmap overviewing the taxonomy of Runtime Verification (from [FKRT18]).

Traces. First, an RV approach has to consider the trace, as it provides all the information about the system to the RV technique. Traces can take on two **roles**, they can be used to represent the information extracted from the program execution, they can also be used as a mathematical object to reason about properties of the RV technique (such as considering finite, or infinite traces). Furthermore, an RV technique must define the necessary **information** from the program, and encode it appropriately. For example, it must determine whether the information is events generated by function calls or triggers, sampling of the system state, or a continuous signal. To build up the trace, information is often associated with a notion of time, as such **evaluation** pertains to whether the trace captures points in time or intervals.

Specifications. The information needed in the trace depends on the chosen specification formalism. Specifications are defined under a certain view and assumptions on the system (referred to as the **system model**). Specifications are either **implicit** or **explicit**. An **implicit** specification denotes correctness properties that are important but do not necessarily need to be defined by the developer. These properties are universally understood and can be hard-coded. Examples of implicit specifications include deadlock freedom, data-race freedom, and ensuring safe memory accesses. An **explicit** specification denotes properties that are important to the programmer and typically defined using behavioral formalisms (like finite state machines and time logics).

Monitors. Specifications are used to create monitors. Monitors are the components responsible for performing the verification. As such RV, techniques have to deal with monitor **generation** and **execution**. The automatic creation of monitors from specifications is the problem of monitor synthesis (c.f. [BLS11, BL11]). Monitor execution can be either direct or indirect. A directly executable monitor is simply code in the target programming language that contains the verification logic. An indirectly executable monitor consists of a reification of more generic monitoring logic (e.g. part of some framework), that is initialized with specific parameters to monitor the specification. Finally, the **decision procedure** of the executing monitor can be analytical or operational. An analytical decision procedure is capable of randomly seeking records (e.g. in a database) to determine relationships

between the various records with respect to the execution. An operational decision procedure decides as it receives information, using an iterative process over the trace. Operational decisions procedures utilize formalisms like automata or are based on formula rewriting.

Deployment. Monitors are designed to verify a given execution of the system, and therefore decisions must be made to determine when, where, and how they are placed in the system. The **stage** of a monitor determines *when* the verification occurs. We distinguish between two stages: *online* and *offline*. Online monitoring consists in running the monitors *during* the execution, while offline monitoring consists in analyzing the trace of the execution postmortem. The **placement** of a monitor determines *where* the verification occurs. Monitors can be placed to execute in the same address space as the target system (i.e. same process or thread), in which case monitoring is said to be *inline*. In contrast, monitors can be run in a separate process or device (for example, on GPUs or FPGAs in parallel), in which case it monitoring is said to be *outline*. The **architecture** of monitors determines *how* monitors are placed. That is, it determines how many monitors will be run in the system, and whether or not monitors will communicate together. If the system contains only one monitor, it is said to be *centralized*, otherwise it is said to be *decentralized*.

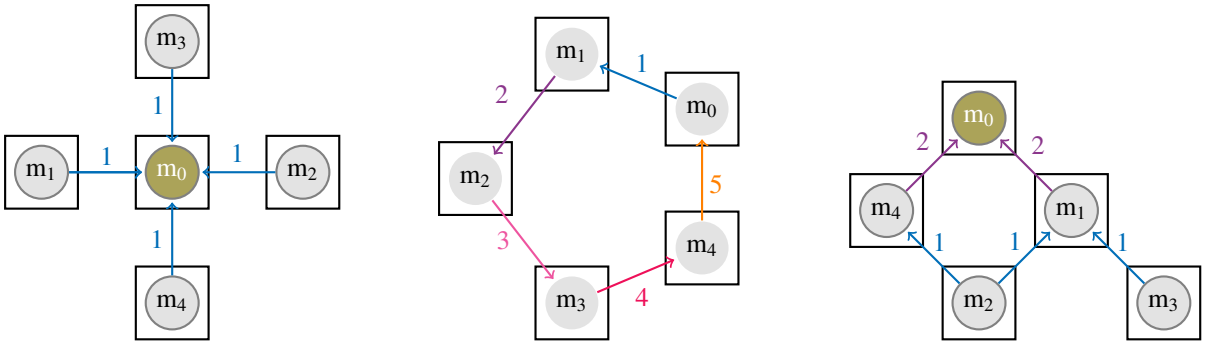
Interference. By introducing monitors in a system for verification, it is often the case that a technique interferes with the existing system in order to monitor it. Monitors or trace capturing techniques may affect the memory layout of the application, or the schedule of threads in a concurrent program. They can also incur additional delays in systems such as real-time systems. There is interest in minimizing the interference of runtime verification, which can rely on the platform itself providing the trace such as JVM snapshotting and hardware modules for non-intrusive trace capture.

Reaction. Runtime verification techniques are primarily used to observe the system and verify the run against a specification. In this sense, they are said to be **passive**, since they only observe, and then report or fail when a specification is violated. The monitoring reaction can also be **active**, in which case the monitoring logic interferes in the target system to ensure that it is compliant with the specification. This is the case of *runtime enforcement* [Fal10, FMFR11, FMRS18], in which monitors are capable of snapshotting, rolling back, and suppressing events in a program. It is also possible to allow the program to deviate from the specification for a fixed amount of time, after which the monitor logic is supposed to restore the compliance with the specification or fail.

In this thesis, we focus on tackling concepts pertaining to passive monitoring when monitors are deployed in a decentralized fashion. We next introduce the problem of *decentralized runtime verification*.

1.3 Decentralized Runtime Verification

Decentralized runtime verification [CF16a, FCF14, NCMG17, SVAR04] (DRV) is concerned with verifying systems of multiple components. A *component* can be seen as a separate computing unit such as a thread, a process, or an interface with another system. In DRV, checking the specification requires observations from multiple components and cannot be performed on each component in isolation. In this setting, there is no single observation point, monitors may be deployed on various components and need to communicate to perform the verification. We use the term *global state* to refer to all the observations across all components of the system, and the term *partial state* (and *partial observations*) to refer to a subset of all observations. To illustrate the setting consider the switch and the bulb from Example 2. In this setting, the observations on each of the switch and bulb are not in a monolithic architecture, but rather are separated. As such, one must consider deploying code on the switch or bulb to possibly send observations to the other in order to verify the specification. DRV approaches differ based on the assumptions on the system. For example: synchronous versus asynchronous communication, presence of a global clock, and reliable or unreliable communication links. While we elaborate on the approaches in detail in Chapter 3, we present three general strategies as examples of monitor setups from [CF16a] in Section 1.3.1, discuss their common challenges in Section 1.3.2, and present the thesis contributions in Section 1.3.3.



(a) Orchestration (b) Migration (c) Choreography

Figure 1.3: Different DRV strategies. Numbers illustrate hops (delay) needed to propagate information.

1.3.1 Strategies

Organizing monitors in a decentralized architecture requires consideration for their topology when integrated in the system. In [CF16a] the authors discuss three general yet different algorithms that organize monitors: Orchestration, Migration, and Choreography. They are illustrated in Figure 1.3. In this section, we introduce a high-level view of the strategies, and use them as examples in the remainder of the thesis. In particular, we analyze and compare them in detail in Chapter 8.

Orchestration. The orchestration algorithm (Orch) shown in Figure 1.3a emulates a centralized monitoring algorithm. To do so, it first sets up a main monitor (m_0) in charge of monitoring the entire specification. Then, since that monitor cannot access all observations on all components, Orch introduces one monitor per component to forward the observations to the main monitor. In Orch, only one component performs monitoring, while the components simply send all observations. While this is a simple approach to monitoring, it does not effectively distribute the computation across components, and requires that all observations be communicated at all times, which can be costly (Chapter 8).

Migration. A migration algorithm shown in Figure 1.3b performs monitoring by moving the monitor across components to complete the information needed for verification. In brief, a migration algorithm places the monitor on an initial component, captures observations, then decides whether or not to transfer the monitor to another component using a heuristic. As such, it attempts to minimize communication messages as a monitor is transferred whenever information from other components is needed. The migration algorithm relies on a heuristic to decide on transferring the monitor, in this thesis we use two heuristics: earliest obligation from [CF16a] and round-robin. The *earliest obligation* heuristic ensures that the atomic proposition with the least timestamp (soonest) determines which component to choose next. The *round-robin* heuristic always transfers the monitor in a cyclical manner starting from m_0. By using the heuristics we define two variants of the migration algorithms: Migr and Migrr, respectively.

Choreography. The choreography algorithm (Chor) shown in Figure 1.3c performs an analysis on the specification (written in LTL), splits the specification into subspecifications, assigns monitors to each subspecification, and organizes the monitors in a directed acyclic graph (DAG). As such, the information needed to analyze the specification propagates in the DAG, allowing each monitor to complete its verification, until it reaches the root monitor (m_0), which determines the verdict. In Chor, a monitor may perform verification multiple times to determine the verdict associated with the subspecificiation on the same trace but starting at different timestamps (the process is called *respawning*). Orch can be seen as a special case of Chor with a tree of height 1, where leaves consist of subspecifications formed only by a single atomic proposition.

The presented strategies encompass different approaches to decentralize monitors when performing DRV. However, multiple challenges and common considerations arise when performing DRV using similar strategies. We elaborate on the challenges in the next section (Section 1.3.2).

1.3.2 Challenges

Multiple common challenges arise when performing DRV. In this thesis, we focus particularly on three challenges: managing partial observations, separating deployment from monitoring, and reasoning about decentralization in a modular and hierarchical way.

Partial observations. Since algorithms performing DRV have no access to the global state of the system, they must maintain partial information on the execution. The partial information for a given monitor depends on the component assigned to it, and the information communicated by other monitors. Therefore, a common theme to all approaches pertains to managing partial observations, maintaining possible reachable verdicts, and then resolving the global state when sufficient information is obtained. While this can be implemented arbitrarily by buffering information in memory and always waiting for all information to be available, it is sometimes the case that monitoring can progress when sufficient partial information is gathered. It is then of interest to be able to model partial information in a uniform way across the different strategies in order to parametrize and assess them.

Separation of problems. DRV approaches are concerned with the monitor deployment given a specific global specification. They typically analyze the specification to determine the monitor architecture while also managing the monitoring itself. As such, the problem of generating the monitor network and assigning monitors to components is often tangled with the problem of the monitoring itself. While some DRV approaches (detailed in Section 3.2.2) enable specifications to assign specific atomic propositions to components which are used when synthesizing the monitor network, they do not explicitly model the deployment and monitoring as two separate phases of monitoring. Separating the problems of deployment from monitoring allows for more control and analysis of the monitoring approaches, and more importantly opens the possibility to enable the deployment to target the system architecture.

Modular hierarchical specifications. Using a single global specification to represent system properties hides the relationships between various parts of a decentralized system. A global specification requires that the monitoring technique perform additional (potentially costly) analysis to determine the relationships and dependencies between the various subspecifications. Having a large specification presents scalability issues both in the design and maintenance of specifications, and the automatic synthesis of monitors. Therefore, we seek to modularize specifications and determine relationships between subspecifications and the various components of a system.

1.3.3 Thesis contributions

We summarize the contributions of the thesis as follows:

- To account for the challenge of managing partial observations, we introduce in Chapter 5 a data structure (Execution History Encoding, `EHE`) which encodes an execution of an automaton. The `EHE` keeps track of potential reachable states given partial information, by modeling paths to possible states as boolean expressions. By doing so, the `EHE` allows us to make the most of partial information. By performing boolean simplifications we are able to determine reachable states without requiring all atomic propositions be provided. Furthermore, the `EHE` data structure replicates under strong eventual consistency regardless of the order of messages. That is, for any two monitors exchanging their `EHE`, they are able to determine the same information about the verification of the system.
- The `EHE` data structure also allows us to unify the analysis of different decentralization strategies, as the `EHE` relies on boolean formulae instead of LTL formulae which are easier to manipulate at runtime and have a known minimal form [BU08] (see Section 3.1.3 for details).

- We also present decentralized specifications in Chapter 6, which allow specifications to account for decentralization. A decentralized specification consists of a set automata-based specifications that are able to reference each other. Referencing other specifications explicitly models the relationships between specifications in the system, allowing us to separate the problem of generating subspecifications (i.e. synthesis of a decentralized specification) from the monitoring itself, and thus establish a generic decentralized monitoring algorithm consisting of two phases: *setup* which performs deployment, and *monitor* which accounts for the monitoring technique. We elaborate on their semantics (i.e. how to evaluate the references), and two of their properties. The first extends the property of what one can monitor (i.e. *monitorability*) to decentralized specs. The second defines the property of *compatibility* between a specification and a system architecture.
- We detail the design decisions behind developing `THEMIS` (Chapter 7), a framework designed to handle decentralized specifications. We show how `THEMIS` can be used to design, analyse and simulate decentralized monitoring algorithms .
- We present two applications of decentralized specifications and `EHE`. First, they are used to compare the three strategies (mentioned in Section 1.3.1) using analysis and simulation in Chapter 8. This extends previous work [CF16a] with analysis, and replicates the experiment with additional metrics. Second, we use decentralized specifications to monitor smart homes and user behavior in Section 9, on real traces of over 36,000 timestamps spanning 27 sensors in a smart apartment. We show how to go beyond system properties, to specify user behavior using RV, and more complex interdependent specifications defined on up to 27 atomic propositions. We illustrate the modularity of decentralized specifications to: (1) scale beyond existing centralized RV techniques, and (2) greatly reduce computation and communication costs.
- We instantiate decentralized specifications to account for RV of multithreaded programs. After detailing the limitations and challenges of RV for multithreaded programs, we utilize a two-level decentralized specification. The decentralized specifications determines behavior at two levels. At the first level, properties are defined on the thread itself. At the second level, properties are defined over concurrency regions (referred to as *scopes*).

1.4 Thesis Overview

The thesis is split into three parts. We present an overview of the parts and their chapters.

Chapter 2: We introduce the general concepts often used in runtime verification. Particularly, we introduce two formalisms: Linear-time temporal logic (LTL), and LTL_3 monitors. Furthermore, we present aspect-oriented programming on which a large amount of RV techniques rely for instrumentation.

Chapter 3: We present related work, focusing on techniques applicable for decentralized RV. We tackle techniques relying on formula rewriting, specific approaches for monitoring distributed and multithreaded programs, and finally we cover stream-based RV which models various input sources as named streams.

Part One: Hierarchical Decentralized Specifications

Part one tackles the challenges presented in Section 1.3.2 by introducing the theory used in the remainder of the thesis and the `THEMIS` tool designed to incorporate it. In particular it introduces the `EHE` data structure used to encode partial information, and decentralized specifications.

Chapter 4: We introduce the basic concepts and notation used in the remainder of the thesis.

Chapter 5: We introduce the *Execution History Encoding* (`EHE`) data structure. The `EHE` data structure encodes the execution of an automaton while preserving soundness and determinism.

Chapter 6: We define hierarchical decentralized specifications. We elaborate on the basic structure of a decentralized specification, elaborate on its semantics, and detail properties of monitorability and compatibility. *Monitorability* ensures that given a specification, monitors are able to eventually emit a verdict, for all possible traces. *Compatibility* ensures that a monitor topology can be deployed on a given system.

Chapter 7: In this chapter, we explain the design goals and architecture of `THEMIS`. `THEMIS` is a modular tool to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ. `THEMIS` is designed with the ability to interface with other tools while providing a uniform workflow for designing algorithms, metrics, and running reproducible experiments.

Part Two: Applications

Part two illustrates two applications of decentralized specifications. The first application compares various decentralized monitoring algorithms using a unified analysis and simulation approach. The second applies runtime verification to a smart home, which presents a hierarchical architecture and multiple inter-dependencies between specifications.

Chapter 8: In this chapter, we aim to compare different decentralized monitoring algorithms in terms of computation, communication, and memory overhead. We elaborate on the general phases of decentralized monitoring algorithms and illustrate the approach to analyze and simulate them by adapting the algorithms explained in Section 1.3.1.

Chapter 9: We use decentralized specifications to check various specifications in a smart apartment. The specifications can be broken down into three types: behavioral correctness of the apartment sensors, detection of specific user activities (known as activities of daily living), and composition of specifications of the previous types. We illustrate how decentralized specifications allow us to re-use specifications, and combine them to: (1) scale beyond existing centralized RV techniques, and (2) greatly reduce computation and communication costs.

Part Three: Instantiation for Multithreaded RV

Part three reuses the concept of decentralized specifications modifying the semantics to target multithreaded programs. We begin by introducing the challenges to multithreaded RV, and then present a two-level decentralized specification that targets multithreaded programs.

Chapter 10: We review some of the main RV approaches and tools that handle multithreaded Java programs to highlight the challenges RV faces when targeting multithreaded programs. We discuss their assumptions, limitations, expressiveness, and suitability when tackling parallel programs such as producer-consumer and readers-writers. By analyzing the interplay between specification formalisms and concurrent executions of programs, we identify *four* questions RV practitioners may ask themselves in order to classify and determine the situations in which it is sound to use the existing tools and approaches.

Chapter 11: We introduce the decentralized monitoring of multithreaded programs using the main idea behind decentralized specifications. Monitors are deployed to monitor specific threads, and only exchange information upon reaching synchronization regions defined by the program itself. That is, they use the opportunity of a lock in the program, to evaluate information across threads. We utilize a textbook example of *readers-writers* as it contains concurrent regions, and show how opportunistic monitoring is capable of expressing specifications on concurrent regions, without incurring significant delay.

1.5 Associated Publications

[EHF17a] presented at ISSTA'17 introduces the basic theory behind `EHE` and decentralized specifications only commenting on the soundness property. Furthermore it also features the preliminary analysis and simulation presented in Chapter 8.

[EHF17b] is a tool demonstration paper presented at ISSTA'17 for `THEMIS`, it discusses an older version of `THEMIS` presented in Chapter 7.

[EHF18a] is a case study paper in RV'18 for monitoring smart homes which we present in more detail in Chapter 9.

[EHF18b] presents a tutorial in RV'18 covering monitoring multithreaded programs and the limitations of existing RV tools. We cover the topic in Chapter 9.

CHAPTER 2

Expressing Properties and Instrumenting Programs

Contents

We introduce the general concepts often used in runtime verification. Particularly, we introduce two formalisms in Section 2.1: Linear-time temporal logic (LTL), and LTL_3 monitors. The two formalisms are used in the remainder of the thesis when defining decentralized specifications (Chapter 6), and when writing specifications (in Chapters 8, 9, and 11). Furthermore, we introduce in Section 2.2 aspect-oriented programming (AOP) on which a large amount of RV techniques rely for instrumentation. We use AOP in the `THEMIS` tool (Section 7.6) to automatically instrument metrics and we discuss limitations of using AOP for RV when monitoring multithreaded programs (Section 10.3).

2.1 Specifying Expected System Behavior

The specification formalizes the expected behavior for the system. The execution of a program is checked against the specification to ensure it is correct. Since RV is a formal technique, it is expected that specifications be expressed using a formalism. In this section, we introduce two basic formalisms on which extensions are typically used. First, we introduce linear-time temporal logic in Section 2.1.1 to specify the behavior of programs with respect to time. Then, we introduce specific Moore automata generated to monitor programs referred to as LTL_3 monitors.

2.1.1 Specifying Behavior with Linear-time Temporal Logic (LTL)

Linear-time Temporal Logic (LTL) intoduced by Pnueli [Pnu77] is a formalism used to express properties over a sequence of system states. The sequence order is of temporal nature, the sequence shows the evolution of the

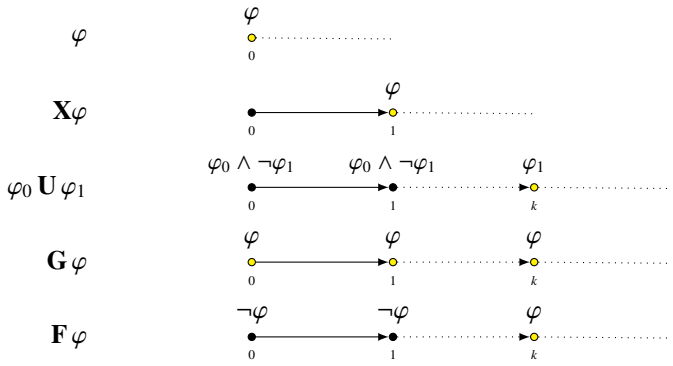


Figure 2.1: LTL operators and their effect on evaluating formulae on a sequence of states. Numbers indicate timestamps (index in the sequence).

system state across time. We denote by AP the set of possible atomic propositions, then $\Sigma = 2^{AP}$ is the alphabet of atomic propositions. An LTL formula defined over Σ, can be expressed using the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi\,\mathbf{U}\,\varphi, \text{where } p \in AP$$

LTL formulae are evaluated over (infinite) words defined over Σ and ranging over Σ^ω as per Definition 1. The notation w^i indicates the sequence w starting at i (suffix), while $w(i)$ indicates the element of the sequence at i.

Definition 1 (LTL semantics) Let $w \in \Sigma^\omega$ and $i \in \mathbb{N}$. Satisfaction of an LTL formula by w at time (index) i is defined inductively:

$$\begin{array}{ll}
w^i \models p & \Leftrightarrow p \in w(i), \text{for any } p \in AP \\
w^i \models \neg\varphi & \Leftrightarrow w^i \not\models \varphi \\
w^i \models \varphi_0 \vee \varphi_1 & \Leftrightarrow w^i \models \varphi_0 \text{ or } w^i \models \varphi_1 \\
w^i \models \mathbf{X}\varphi & \Leftrightarrow w^{i+1} \models \varphi \\
w^i \models \varphi_0\,\mathbf{U}\,\varphi_1 & \Leftrightarrow \exists k \in [i, \infty[\ w^k \models \varphi_1 \text{ and } \forall l \in [i, k[\ w^l \models \varphi_0
\end{array}$$

An LTL formula consisting of an atomic proposition p is satisfied by checking if p holds at the current time, i.e p appears in the set of $w(i)$. The operator $\mathbf{X}$ is referred to as the *next* operator, as for a given timestamp i and a formula φ, it checks if φ is satisfied at the next timestamp (i.e. w^{i+1}). The operator $\mathbf{U}$ is called the *until* operator. Given two formulae φ_0 and φ_1, the formula $\varphi_0\,\mathbf{U}\,\varphi_1$, states that φ_0 must hold until a time instant k where φ_1 holds, at which φ_0 no longer affects the satisfaction of the formula. Using these fundamental operators, we can extend the syntax of LTL for convenience as follows: $\top \stackrel{\text{def}}{=} p \vee \neg p$, $\bot \stackrel{\text{def}}{=} \neg\top$, $\varphi_0 \wedge \varphi_1 \stackrel{\text{def}}{=} \neg(\neg\varphi_0 \vee \neg\varphi_1)$, $\mathbf{F}\,\varphi \stackrel{\text{def}}{=} \top\,\mathbf{U}\,\varphi$, and $\mathbf{G}\,\varphi \stackrel{\text{def}}{=} \neg\,\mathbf{F}\,(\neg\varphi)$. Operator $\mathbf{F}$ indicates that *eventually* (or *finally*) φ holds, it is typically used to express *liveness* properties, that is, it expresses states that are eventually reached by the program execution. Operator $\mathbf{G}$ indicates that *globally* (or *always*) φ holds, it is typically used to express *safety* properties, that is, it expresses states that the program execution must not reach. Figure 2.1 illustrates the introduced time operators, we see on the left a formula written in LTL using the operator, and on the right a trace which satisfies the formula.

Example 3 (Specifying behavior with LTL) Let us recall the light switch and bulb from Example 2. Our set of atomic propositions is $AP = \{\mathrm{s}, \ell\}$. We are interested in specifying the following: *"The light bulb must always be on one timestamp after the switch is on, until the switch is turned off"*. The corresponding LTL formula is then: $\mathbf{G}(\mathrm{s} \implies \mathbf{X}(\ell\,\mathbf{U}\,\neg\mathrm{s}))$ That is, we want to check that at all times ($\mathbf{G}$), when a switch is on (s), we verify at the next timestamp ($\mathbf{X}$) that the light switch is on until the switch is off ($\ell\,\mathbf{U}\,\neg\mathrm{s}$). Using LTL, we are able to specify the behavior of the interaction of the light switch and bulb. We note that, in this case, we chose to allow the light to stay on after the switch is toggled off. We only require the light be on while the switch is on, while other factors can impact it staying on as well, or give the light a given fading duration before it turns off, i.e. a few timestamps before it is turned off. It is also possible to enforce a stricter relation between the light and the switch by using a specification such as $\mathbf{G}(\mathrm{s} \Leftrightarrow \mathbf{X}\ell)$. *

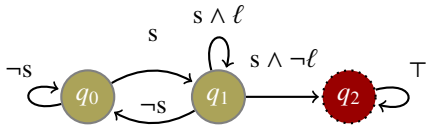


Figure 2.2: LTL_3 monitor for the light switch and bulb specification. The verdicts associated with the states are $\bot$: dotted red, and ?: single yellow.

2.1.2 Specifying Behavior with Moore Automata

An LTL_3 *monitor* is a typical automaton used in RV (c.f. [BLS11, FCF14]). An LTL_3 monitor is a complete, minimal, and deterministic Moore automaton where states are labeled with the verdicts in the set $\mathbb{B}_3 = \{\top, \bot, ?\}$, and transitions are labeled with expressions defined over an alphabet of atomic propositions. Atomic propositions are used to represent abstract states of the system. Verdicts $\top$ and $\bot$ respectively indicate that the current execution complies and does not comply with the specification, while verdict ? indicates that the verdict has not been determined yet. Verdicts $\top$ and $\bot$ are called "final", as once the monitor outputs $\top$ or $\bot$ for a given trace, it cannot output a different verdict for any extension of that trace. The automaton can be automatically generated from an LTL specification using various transformations detailed in [BLS11].

Example 4 shows the LTL_3 monitor for checking the interaction between the light switch and bulb.

Example 4 (LTL_3 Monitor) We introduced in Example 3 the property $\mathbf{G}(s \implies \mathbf{X}(\ell \,\mathbf{U}\, \neg s))$. The LTL_3 monitor that checks for the property is presented in Figure 2.2. The automaton consists of three states: q_0, q_1, and q_2 associated respectively with the verdicts ?, ?, and $\bot$. Upon reaching q_2, the verdict is final as it can no longer change. The final verdict indicates that, at some point in the execution, the light was off while the switch was on.∗

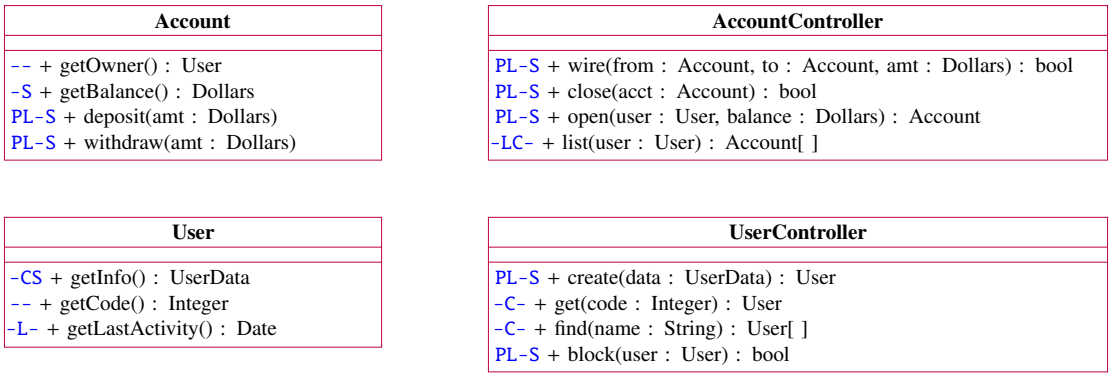
In the case of Example 4, it is possible to imagine one monitor running (with or alongside) the program, and having access to the global state of the program. We refer to such a scenario as *centralized monitoring of a centralized specification*. There is one global specification of the system, being checked by a given monitor that has access to all the information about the atomic propositions.

2.2 Instrumenting Programs with Aspect-Oriented Programming (AOP)

Specifications define the way a program execution should behave typically over events. Events represent abstract states of a program. Eventually, it is required that these events be extracted during an execution. This is typically done by instrumenting the program. While RV techniques are capable of defining their own instrumentation technique, there exist general purpose dedicated tools for instrumenting programs. In this section, we elaborate on the aspect-oriented programming paradigm as it provides a modular way to describe and intercept and inject code into an existing program. The AOP paradigm has an explicit compiler for an extension of Java called *AspectJ* [KHH+01a] which integrates its principles at the source-code level, and recently DiSL [MZA+15, MVZ+12] which integrates AOP concepts for dynamic analysis at the byte-code level.

2.2.1 Overview of AOP

A typical system consists of its main logic along with tangled code that implements multiple other functionalities. Such functionalities are often seen as secondary to the system. For example, logging is not particularly related to the main logic of most systems, yet it is often scattered throughout multiple locations in the code. Logging and the main code are separate domains and represent different *concerns*. A concern is defined in [CES97] as a "*domain used as a decomposition criterion for a system or another domain with that concern*". Domains include logging, persistence, and system policies such as security. Concerns are often found in different parts of a system, or in some cases multiple concerns overlap one region. These are called *crosscutting concerns*. Aspect-Oriented Programming (AOP) [KHH+01a] aims at *modularizing* crosscutting concerns by identifying a clear role for each of them in the system, implementing each concern in a separate module, and loosely coupling each module to only a limited number of other modules. Essentially, AOP defines mechanisms to determine the locations of the



P: Persistence L: Logging C: Caching S: Security Policy

Figure 2.3: Multiple concerns in a simple system

concerns in the system execution by introducing the concept of *joinpoints* and *pointcuts*. Then, it determines what to do at these locations by introducing the concept of *advices*. Finally, it provides a mechanism to coordinate all the advices happening at a location by introducing the concept of *weaving*. We elaborate on the various concepts in Section 2.2.3. Since RV can be seen as a crosscutting concern, AOP techniques are often used to instrument systems as utilizing pointcuts is a simple way to describe events.

2.2.2 Crosscutting Concerns

The implementation of crosscutting concerns mentioned in the introduction leads to two typical problems: *scattering* and *tangling* [LLO03].

- *Tangling* happens when concerns overlap in one region of the program. Consequently, enforcing one concern may affect others.
- *Scattering* is the dual situation of tangling. It happens when one concern is spread across different regions of the program. Scattering concerns go against encapsulation. Developers have to manually keep track of the location of a specific concern in multiple areas of the system.

In the following example, we illustrate the above two problems on an example.

Example 5 (Crosscutting concerns and their issues.) Figure 2.3 illustrates four different crosscutting concerns: logging, caching, persistence and security policy. We present a class diagram describing the main methods of the classes. We omitted describing their relationships for clarity. The class diagram methods are prefixed with the four concerns as flags. We identify each concern by a label, if the method is annotated by the label, then some code related to the logic of that concern is included in the method. For example, the method `Account.withdraw` has three *tangled* concerns: persistence, logging, and policy. Thus, method `withdraw` has to include code for persistence, logging, and logic. This code enforces the policy in addition to its own main logic. The policy concern is scattered across all four classes, hence maintaining it requires one to modify all four classes when a change is needed. ∗

2.2.3 AOP Concepts

The purpose of AOP is to localize crosscutting concerns in an aspect. An aspect is defined in [KHH+01a] as "*a well modularized implementation of a crosscutting concern*". These concerns are separated from the main program logic and contained in separate logical units. One example of separation of concerns is achieved by AspectJ [KHH+01a], which is an aspect-oriented extension to the Java programming language.

A *joinpoint* is a well-defined point in program execution where a concern needs to be handled. It acts as a reference point to coordinate the behavior of multiple concerns. A *pointcut* refers to a set of *joinpoints* and execution context information. Basic pointcuts can be composed and identified so as to increase re-usability. Pointcuts are

Listing 1 Logging aspect combining the pointcut with advice.

```
1  public aspect Logging {
2          private static Logger logger = Logger.getLogger(Logging.class.getName());
3          pointcut log() :
4          call(void Account.deposit(Dollars))
5                  || call(void Account.withdraw(Dollars))
6                  || call(* AccountController.*(*))
7                  || call(Date User.getLastActivity())
8                  || call(User UserController.create(UserData))
9                  || call(bool UserController.block(User)
10
11         after() returning(Object res) : log() {
12                 logger.info(thisJoinPoint.getSignature().toShortString()
13                         + Arrays.toString(thisJoinPoint.getArgs())
14                         + " -> " + res);
15         }
16 }
```

the syntactic elements used to select joinpoints. A pointcut specifies a function signature, a variable name, and a module that needs to be matched. Furthermore, pointcuts are able to specify dynamic execution constraints, such as a function being invoked while inside another function (e.g. `cflow` pointcut in AspectJ). A pointcut regulates scattering by describing the joinpoints needed to implement the concern. An *advice* defines the additional behavior to be executed at each specific joinpoint selected by a pointcut. An *aspect* serves as the modular unit that encapsulate advices, pointcuts, and additional behavior. Furthermore, aspects may introduce their own variables, methods, and fields. This is referred to as inter-type declarations. The term inter-type designates the fact that these extra objects and code are accessible in different types (based on the matching joinpoints). The main task of an AOP language implementation is to coordinate the execution of the non-aspect code with the aspect code. This coordination has to ensure a correct execution at the joinpoint of both primary and secondary concerns. This process is called *weaving* and can be done at compile-time, load-time, or run-time.

Example 6 (Implementation of a logging concern with AspectJ) Listing 1 implements parts of the logging concern shown in Example 5 using AspectJ. In the case of logging, the inter-type variable is a `Logger` object (Line 2). The pointcut expression (lines 4-9) specifies the various method invocations to be intercepted and names the pointcut `log`. The advice implements the logging code. It consists of code necessary to (i) capture the arguments of the method invoked using the `getArgs`() method on the `thisJoinPoint` object, (ii) capture the return value of the method invoked, and (iii) pass it to the logger. The advice is set to trigger after the pointcut (line 11), in which case, it means after a method returns. Effectively, for any of the methods defined in the pointcut, the logger will log the name of the method called, its arguments and return value. *

2.2.4 Fine-grained AOP Instrumentation

AOP is a paradigm introduced to tackle tangling and scattering (Section 2.2.2). The concepts introduced in Section 2.2.3 apply to programs in general. AspectJ [KHH+01a] enables AOP concepts at the source-code level and targets Java programs. However, in RV, one may also be interested in instrumenting programs at the byte-code level, as properties may account for a much more fine-grained granularity of events.

RV tools can opt to use bytecode instrumentation frameworks such as ASM [BLC02] and the *Byte Code Engineering Library* (BCEL) [The17]. However, bytecode instrumentation frameworks are low-level, require expertise for instrumentation, and are verbose and error-prone when describing the needed instrumentation. Therefore, finding higher-level abstractions for the instrumentation has been the target of works such as Soot [VGH+00] and Shrike [IBM15]. For example[1], Soot is used by the RVPredict tool [HMR14] for bytecode instrumentation (introduced in Section 3.3.1).

More recently, the *Domain Specific Language for Instrumentation* (DiSL) [MZA+15, MVZ+12] has been designed for dynamic program analysis. DiSL allows for AOP concepts to be expressed at the byte-code level while providing access to comprehensive static and dynamic context information. Furthermore, DiSL employs efficient weaving to reduce the overhead of the instrumentation.

[1] *JTrek* –a bytecode instrumentation tool– is used by the Java PathExplorer RV tool [HR04]. However, its webpage is no longer accessible.

CHAPTER 3

Approaches to Decentralized Monitoring and Related Work

Contents

Chapter abstract

This chapter presents related work, focusing on techniques applicable for decentralized RV. We tackle techniques relying on formula rewriting, and illustrate the limitations of using rewriting, a key motivation for using automata for decentralized specifications. We also discuss specific approaches for monitoring distributed and multithreaded programs. For the former, we discuss techniques that (1) evaluate predicates on a global state of the distributed system, (2) distribute explicitly the LTL specification on the system components, and (3) are able to monitor even when faults occur. For the latter, we discuss utilizing predictive trace analysis techniques to infer the order of events between the various threads, and elaborate on an instrumentation technique that interfers minimally with the schedule of the threads. Finally we cover stream-based RV which models various input sources as named streams, and discuss their relation to complex event processing approaches.

$$
\begin{array}{lll}
P(p \in \mathrm{AP}, \sigma) & = p \in \sigma & P(\top, \sigma) = \top \\
P(\varphi_1 \vee \varphi_2, \sigma) & = P(\varphi_1, \sigma) \vee P(\varphi_2, \sigma) & P(\bot, \sigma) = \bot \\
P(\varphi_1 \wedge \varphi_2, \sigma) & = P(\varphi_1, \sigma) \wedge P(\varphi_2, \sigma) & P(\neg\varphi, \sigma) = \neg P(\varphi, \sigma) \\
P(\varphi_1 \mathrm{U} \varphi_2, \sigma) & = P(\varphi_2, \sigma) \vee P(\varphi_1, \sigma) \wedge \varphi_1 \mathrm{U} \varphi_2 & P(\mathrm{X}\varphi, \sigma) = \varphi \\
P(\mathrm{G}\varphi, \sigma) & = P(\varphi, \sigma) \wedge \mathrm{G}\varphi & P(\mathrm{F}\varphi, \sigma) = P(\varphi, \sigma) \vee \mathrm{F}\varphi
\end{array}
$$

Figure 3.1: LTL rewriting rules (from [BLS11]).

We present an overview of approaches that are either directly used for decentralized runtime verification, or can be used for such purpose. In Section 3.1, we introduce the rewriting technique for RV, and show how it is used for decentralized monitoring. Then, we present in Section 3.2 and Section 3.3 approaches to monitor respectively distributed and multithreaded programs. Finally, we present stream-based RV in Section 3.4 as it is capable of decentralized monitoring. We also refer to [FPS18] for a recent overview.

3.1 Monitoring by Formula Rewriting

The first class of approaches consists in monitoring by formula rewriting. The property is written as a formula, and upon observing events in the system, the formula is rewritten and simplified until it is equivalent to $\top$ (true) or $\bot$ (false) at which point the algorithm terminates. Approaches to rewriting of LTL, and Metric Temporal Logic (MTL) – an extension to LTL with temporal operators – are presented in the centralized context in Section 3.1.1, and the decentralized one in Section 3.1.2. Limitations of rewriting, especially when used for partial observations are presented in Section 3.1.3. These approaches use a centralized specification to describe the system behavior. Furthermore, they rely on a global clock, as the rewrites are performed on a sequence of events.

3.1.1 Centralized Rewriting

LTL. The usage of rewriting for the efficient monitoring of LTL specifications is tackled in [RH05]. The approach focuses on monitoring LTL *online* using term rewriting, and is implemented in Maude [CDE+03], a framework for rewriting formulae. Generally, the rewriting is represented as a progression function noted P which takes as input an existing formula φ and an event σ, and returns a new formula φ'. An event, in the context of progression (in [BLS11, BF12, CF16a]) is a set of atomic propositions (in 2^{AP}) *observed* to be $\top$. The progression function P consists of performing multiple rewrite rules. We present those for LTL in Figure 3.1, based on the work in [BLS11]. The resulting formula $\varphi' = P(\varphi, \sigma)$ is rewritten to include the knowledge provided by reading σ for the given input formula φ. We illustrate rewriting in Example 7.

Example 7 (LTL progression.) Consider the formula $\varphi \stackrel{\text{def}}{=} \mathbf{G}(a \wedge b \vee c)$, defined over the atomic propositions in $AP \stackrel{\text{def}}{=} \{a, b, c\}$. Given an event $\sigma = \{c\}$, the resulting progressed formula using the rules in Figure 3.1 is $\varphi' = \mathrm{P}(\varphi, \sigma) = \mathrm{P}(a \wedge b \vee c, \sigma) \wedge \varphi = ((a \in \sigma) \wedge (b \in \sigma) \vee (c \in \sigma)) \wedge \varphi = \top \wedge \varphi = \varphi$. Notice that by applying the rule relative to operator **G** we doubled the size of the formula. After using basic boolean simplification rules, we reduce the size of φ' by simplifying it to be equivalent to φ. ∗

The size of the evolving formula is in the worst-case exponentially bounded by the size of the original LTL formula. Furthermore, an exponential space explosion cannot be avoided in certain unfortunate cases. The authors of [RH05] point out that using memoization to cache the results of rewriting (a feature provided by Maude), improves the outcome by an order of magnitude.

MTL. Focusing on real-time systems often involves large traces with the notion of time. In [TR05], the authors extend rewriting to use Metric Temporal Logic (MTL). MTL introduces discrete time operators to LTL formulae, which requires maintaining information about large portions of a given trace. Traces are typically available only incrementally and they are much larger than the formulae against which they are checked. When monitoring *online*,

it is impossible to store an entire execution trace and then perform the formal analysis by having random access to the trace. The authors present a general monitoring algorithm for monitoring MTL which is exponential in the size of the formula but independent from the size of the trace, while covering lower bound analysis on monitoring MTL formulae.

Improving rewriting. Future approaches that have focused on improving the rewriting of both LTL and MTL formulae include the EAGLE rewriting engine [BGHS04]. EAGLE improves the upper bounds for rewriting LTL, and is capable of rewriting MTL. An additional improvement on the EAGLE engine is introduced as RULER [BRH10], which allows rewriting rules to be named and carry parameters.

3.1.2 Decentralized Rewriting

LTL rewriting as presented in Section 3.1.1 has been extended by [BF12, CF16a] to perform decentralized monitoring. While still performing monitoring of one global formula of the decentralized system, the approach uses rewriting to monitor the three different strategies for decentralized monitoring introduced in Section 1.3.1: orchestration, migration, and choreography.

Orchestration is effectively an "emulation" of centralized monitoring. A monitor on each component sends the observations to a central monitor, the center monitor combines them to form an event, and regular centralized progression is performed on the formula by the main monitor, as explained in Section 3.1.1. For the remaining two approaches, the events in the formula account for partial observations. In the case of migration, when a monitor is unable to observe some atomic propositions, it adds an *obligation* in the formula. An obligation is defined using the past-time operators of LTL, to encode in the rewritten formula, that the atomic proposition must have held in the past. The monitor then decides to possibly move to another component, in order to observe the atomic propositions it is missing, and rewrite the obligations. To monitor choreography, an analysis is performed to split the formula into multiple subformulae. Each subformulae will be assigned a reference and a component The original formula is rewritten to add the references to the subformulae, and the progression function is extended to account for references.

The provided tool DECENTMON is used to compare the three strategies in terms of number of progressions, messages size, messages exchanged, and added delay. The comparison is provided on random traces and specifications, for multiple sizes of the decentralized alphabet, and different probability distributions.

3.1.3 Limitations of Rewriting

Overview. The rewriting techniques presented in Sections 3.1.1 and 3.1.2 are simple and elegant. However, rewriting varies significantly during runtime based on observations, thus analyzing the runtime behavior could prove difficult if not unpredictable. When excluding specific syntactic simplification rules, $\mathbf{G}(\top)$ could be rewritten $\top \wedge \mathbf{G}(\top)$ and will keep growing in function of the number of timestamps. As such, the size of the formula relies significantly on LTL simplification. In Example 7, we illustrated an example where the formula doubles in size in the presence of the operator $\mathbf{G}$. The event contained information about the global state. As such, we had information about all atomic propositions in AP. The growth in size is made worse in the presence of partial information, as some subformulae cannot be immediately evaluated. In the remainder of this section, we elaborate on the effect of simplification and using partial information on the size of the progressed LTL formula.

Limitations of LTL simplification. As we have seen in Example 7, a key factor in keeping the formula size small is simplification. In [CF16a], the authors elaborate on the limitations of minimizing LTL formulae. The size of the LTL formula representing the dynamic monitor state poses challenges as it may become too big [RH05, BRH10]. Therefore, it is necessary to simplify the formula at hand, that is, determining a smaller formula that remains semantically equivalent to the original formula. It is possible to simplify LTL by either (i) using translations of formulae into equivalent automata and using automata minimisation (cf. [BLS11, MC13]) or (ii) by generating optimal equivalent monitors by coinduction [SRA03]. The complexity of such procedures (at least PSPACE-hard) renders these techniques not applicable at runtime and not suitable for monitoring. Note also that existing axiomatisations of LTL could not be used either for their purposes since what is needed is a finite confluent rewriting

system. To the best of the authors' knowledge, the existence or non-existence of such rewriting system for LTL formula simplification is still an open question.

Effect of partial information on size. To illustrate progression with partial information, we utilize the decentralized progression function from [BF12] (we denote it by P'). In the decentralized setting, atomic propositions are associated with components, and a formula is progressed on a given component. To account for atomic propositions that are not found on the component on which the formula is being progressed (referred to in this paragraph as non-local), *obligations* are added to the formula. An obligation ensures that an atomic proposition was observed at a given point in the *past*. For that, we use the operator *previous* ($\overline{\mathbf{X}}$) which is the dual of operator $\mathbf{X}$. For a given index i, the operator $\overline{\mathbf{X}}\varphi$ evaluates φ on the event at index $i-1$. Information about obligation is progressed by adding $\overline{\mathbf{X}}$ to non-local atomic propositions, and obligations involving them. The progressed formula can then be transferred to another component to evaluate the obligations (following some heuristic). The history of events (the trace) typically needs to be kept to be able to evaluate the past. We illustrate decentralized progression with partial information in Example 8.

Example 8 (Progression with partial information.) Let us consider the same formula from Example 7, $\varphi = \mathbf{G}(a \wedge b \vee c)$, defined over the atomic propositions $AP \stackrel{\text{def}}{=} \{a, b, c\}$. However, we partition the atomic propositions over three components as follows $AP_0 \stackrel{\text{def}}{=} \{a\}$, $AP_1 \stackrel{\text{def}}{=} \{b\}$, and $AP_2 \stackrel{\text{def}}{=} \{c\}$. Let the sequence of events be $w \stackrel{\text{def}}{=} \{a, b\} \cdot \{a, c\} \cdot \{b, c\}$. We begin by progressing φ on component 0: $\varphi^0 = P'(\varphi, \{a, b\}) = P'(a \wedge b \vee c, \{a\}) \wedge \varphi = (P'(a, \{a\}) \wedge \overline{\mathbf{X}}b \vee \overline{\mathbf{X}}c) \wedge \varphi = (\overline{\mathbf{X}}b \vee \overline{\mathbf{X}}c) \wedge \varphi$. We can see that the partial observations have been added to the formula as obligations, thus increasing its size. Now we suppose that the monitoring logic chooses component 2 to progress the formula, we note that the event at index 0, had c be $\perp$. We now have: $\varphi^1 = P'(\varphi^0, \{c\}) = (P'(\overline{\mathbf{X}}b, \{c\}) \vee P'(\overline{\mathbf{X}}c, \{c\})) \wedge (P'(a, \{c\}) \wedge P'(b, \{c\}) \vee P'(c, \{c\})) \wedge \varphi = ((\overline{\mathbf{X}}\overline{\mathbf{X}}b) \vee \perp) \wedge (\overline{\mathbf{X}}a \wedge \overline{\mathbf{X}}b \vee \top) \wedge \varphi = (\overline{\mathbf{X}}\overline{\mathbf{X}}b) \wedge \varphi$. By performing repetitive simplifications, we are able to maintain the size of the formula manageable. However, we can see that the less information is available, the more obligations are added to the formula and propagated. Furthermore, we can see that the size varies significantly at runtime based on observed atomic propositions and the strategy used to pass around the formula. ∗

Using automata. To tackle the unpredictability of rewriting LTL formulae, another approach [FCF14] uses automata for monitoring regular languages, and therefore (i) can express richer specifications, and (ii) has more predictable runtime behavior. The system property is expressed as one big automaton shared among all monitors. The monitoring procedure keeps track of potential reachable states and updates the set of reachable states as soon as sufficient partial information is relayed between monitors. In particular, monitors communicate observations, and reached states. This approach avoids LTL rewriting. As such, it does not rely on minimizing LTL, and does not suffer from the exponential increase of the formula size at runtime. However, it requires that all monitors verify the same centralized specification automaton. Furthermore, monitors must be aware of the entire specification. In this thesis, we focus on avoiding LTL rewriting as well, and rely on automata when defining decentralized specifications in Chapter 6. In addition, we focus on *black-box* specifications, wherein monitors are capable to refer to other monitors without knowing explicitly their specification. Furthermore, the execution history encoding (EHE) data structure presented in Chapter 5, presents a uniform approach to encode reachable potential states with partial information and share it between monitors without having to send all information, while relying on rewriting and simplifying boolean expressions (instead of LTL). In Section 5.1.2, we illustrate how the EHE data structure can be used for decentralized monitoring of a centralized specification as in [FCF14].

3.2 Monitoring Distributed Systems

In this section, we introduce techniques which monitor in a decentralized fashion a centralized specification in distributed systems. These approaches do not assume a global clock, and rely on acquiring global snapshots of the distributed system.

3.2.1 Global Predicate Detection

Distributed slicing. Monitoring distributed systems typically consider the problem of detecting global predicates. Global predicate detection [NCMG17, OG07] consists in evaluating predicates on the global state of a distributed system. Enumerating all the global states of a distributed system with multiple processes is costly, as the state space could grow exponentially. As such, the authors rely on the technique of computational slicing [MSG07], which is an abstraction technique that enumerates only the global states satisfying a given predicate. The slice containing all the global states that satisfy the predicate can be exponentially smaller than that of the whole set of global states. In these approaches, the authors focus on performing *online distributed slicing* exploiting the information from all processes and the structure of the predicate. Approaches performing predicate detection are capable of distributing the evaluation across a distributed system, and evaluate regular predicates which include some temporal logic predicates (such as globally **G** and eventually **F**). The evaluation is done online, and as such can be seen as runtime verification.

Application to RV. In [MB15] the authors extend the approach from [NCMG17] beyond safety properties to monitor temporal properties in distributed systems. Starting from a global automaton representing the system specification, a monitor is generated per process with its own copy of the automaton (instantiated based on its local observations). In the presence of concurrent events, that is, events that can be ordered in multiple ways, the monitor explores the paths (in the specification automaton) resulting from the different possible interleavings of such events. The approach is tested on specifications defined for drones, notably relying on a leader drone and set of followers. The authors show that the overhead for number of messages is linear with respect to the number of generated events for three properties. While the authors claim the algorithm is sound and complete, no argument is made to justify soundness and completeness.

3.2.2 Distributing LTL Specifications

Another set of approaches consider a centralized specification defining the system behavior written in a variant of LTL. These approaches target asynchronous distributed systems, by allowing specifications to refer to specific snapshot states of the global system. In [SVAR04], the authors introduce distributed past-time LTL (PT-DTL) to specify *safety* properties across processes for a given distributed system. PT-DTL allows for a new operator that adds references to a given snapshot state for a given formula of a given process. The approach is extended beyond safety properties to the 3-valued LTL semantics in [SS14]. The authors of [SS14] improve the monitor generation process to allow for more monitorable properties. The process is improved by integrating PT-DTL monitor synthesis with the synthesis techniques of LTL_3 monitors.

3.2.3 Fault-tolerant Monitoring

Another class of research focuses on handling a different problem that arises in distributed systems – that of faults. In these approaches, monitors are subject to faults ranging from failure to receive messages of other monitors to receiving wrong state information.

Consensus. In [BFRT16, BFR+16], monitors are subject to many faults such as failing to receive correct observations or to communicate their state with other monitors. In this setting, all monitors monitor the same global specification. The problem handled is that of reaching consensus with fault-tolerance. That is, it is ensuring that all distributed monitors reach the same verdict. The approach tackles the problem by determining the necessary verdict domain needed to be able to reach a consensus. The added verdicts represent a degree of certainty of the formula evaluation. In addition to the added verdicts, it introduces a monitoring algorithm that uses the added verdicts to ensure a consistent verdict to be reached by all monitors.

Knowledge gaps. By only considering failure in receiving messages, the approach in [BKZ15] extends the MTL approach (explained in Section 3.1.1) to deal with incomplete knowledge and out-of-order messages. In this setting, the timed model of distributed computing [CF99] is used, which assigns a time for each observation based on local

clocks, and ensures observations follow a total order. Failures are modeled as knowledge gaps and the approach ensures that these gaps are resolved, the more monitors communicate. For this purpose, the approach extends the MTL verdict domain to include the three valued domain ($\mathbb{B}_3$), where verdict ? models the "unknown" verdict. Verdicts are partially ordered by their knowledge content wherein $? \prec \top$ and $? \prec \bot$. A given formula is decomposed into multiple subformulae to form a dynamic graph, where each node contains a subformula and a time interval, and edges represent guards. Each node is associated with a truth value, and a set of guards. Guards are of two types incoming to the node ("preconditions"), and outgoing from the node ("triggers"). Guards encode the operators and are used to propagate the truth values of the nodes. The graph is dynamic, nodes and guards are created and removed as the monitors exchange messages. The changes in the topology of the graph indicate the evolution of the verdicts for each subformula. For example, a node with a certain time interval, could be split into multiple nodes partitioning the interval, so as to illustrate the knowledge about part of an interval.

3.3 Monitoring Multithreaded Systems

3.3.1 Predictive Trace Analysis

Overview. Predictive Trace Analysis (PTA) approaches [CSR08, SWYS11, HMR14, HLR15] are dynamic analysis techniques that extend verification from the events in a given single run to a set of possible re-orderings of the events, given a model of the concurrency. PTA approaches model the program execution as a set of traces corresponding to the different orderings of a trace. As such, they encode the trace minimally, then restrict the set of valid permutations based on the model that is allowed. In this section, we illustrate how PTA approaches have been used to check for concurrency errors such as data-race freedom in multithreaded programs, and specifically extended to runtime verification [CSR08, HMR14, HLR15].

PTA and data-races. The approach in [SWYS11] verifies for the absence of data-race in multithreaded programs. Data-race detection is modeled as a constraint solving problem, wherein a formula is used to encode all valid re-orderings of traces to consider during the execution. The constraint checking at runtime relates the current run, to other symbolic permutations of the traces. A violation of the constraint amounts to detecting a concurrency error. While the approach proved to be reliable in detecting data-races, it suffered from weaknesses. It did not take into account data-flow, and thus made conservative estimations on certain concurrency areas to remain sound. This leads to some data-races to be missed. This approach is extended in [HMR14], where authors suggest encoding data-flow in the constraints, proving that their model is sound and maximal. That is, they are able to detect all data-races that occur, and that detected data-races are indeed true positives. The approach provides the tool RVPredict which detects efficiently data-races in multithreaded programs. While [HMR14] can, in theory, model behavioral properties, RVPredict monitors only data races, but does so very efficiently.

Behavioral properties. Using the same sound and maximal model for predictive trace analysis [HMR14], the approach in [HLR15] extends the specification past data-races to behavior. Specifications are able to include behavioral, user-specified events, and are extended with thread identifiers, atomic regions, and concurrency. Events are defined similarly to Java-MOP using AspectJ for instrumentation. Atomic regions are special events that denote either the start or end of an atomic region. Each atomic region is given an ID. The specification formalism used is regular expressions extended with the concurrency operator "||" which allows events to happen in parallel. For example the specification "read($t1$)||write($t2$)" allows a read event from thread 1 to happen in parallel with a write event from thread 2. We elaborate in detail on GPredict in Section 10.4.1.

3.3.2 Runtime Assertion Checking with Minimal Interference

An important problem that arises when monitoring multithreaded programs pertains to interference with the original program (discussed in Section 1.2.2). The monitoring logic may require additional synchronization to maintain a consistent global state. As such, it is able to modify the schedule of various threads while executing. We show how some of the existing RV tools cause such changes in the schedule later in the thesis in Section 10.3. In [KHBZ15], the authors present extensions to a runtime assertion checker for JML specifications. They are able

to verify assertions on shared data in multithreaded programs, with minimal interference with the program. This is accomplished by using JVM snapshots, then performing the checks asynchronously with some delay.

3.4 Stream-based Monitoring

Stream-based techniques include LOLA [DSS+05], BeepBeep [HKG17], and more recently, the Temporal Stream-Based Specification Language [DGH+17, CHL+18, DDG+18], and streams that process time intervals [KHJF18]. Stream-based specifications rely on named streams to provide events. These streams are then aggregated using various functions that modify the timing, filter events, and output new events as a new stream. The output domain extends beyond the Boolean domain and encompasses types. The stream approach to monitoring has the advantage of aggregating types, as such operations such as summing, averaging or pulling statistics across multiple streams is also possible.

Stream aggregation is provided by general-purpose functions which are more expressive than automata, but more complex to analyze. While streams are general enough to express monitoring, *they do not address decentralized monitoring explicitly*. As such, there is no explicit assignment of monitors to components and parts of the system, nor consideration of architecture. Furthermore, there is no algorithmic consideration addressing monitoring in a decentralized fashion.

3.4.1 Monitoring Synchronous Streams

LOLA [DSS+05] is a stream-based specification language designed to monitor synchronous streams. When using LOLA, a programmer is able to write stream expressions to define the relation between the input and output events on a stream. These expressions are typed and support parameters. Expressions can be built bottom-up starting with constants and stream atomic propositions. Expressions are combined using boolean or arithmetic operators with the two additional constructs: an *if-then-else* construct for branching, and an offset construct to retrieve the value of an atomic proposition at relative timestamps. LOLA defines dependency graphs between various stream information. Furthermore, LOLA defines properties for specifications on streams. The concepts of *well formed*, and *efficiently monitorable* LOLA specifications is introduced. The former ensures that dependencies in the trace can be resolved before they are needed, and the latter ensures that the memory requirement is no more than constant with respect to the length of the trace.

3.4.2 Monitoring Real-time Systems with Streams

Focusing more on real-time systems, the Temporal Stream-Based Specification Language (TeSSLa) [DGH+17, CHL+18] provides syntax wherein events are not passed to streams at discrete time intervals, but are treated as changes to a signal. TeSSLa allows aggregation operators on signals to be defined recursively and provides memory and delay guarantees, based on a similar analysis of dependency graphs and operators used in the aggregation. Furthermore, TeSSLa provides multiple "fragments" – a restriction over the TeSSLa semantics to enable certain guarantees. For example the boolean fragment restricts streams to booleans instead of arbitrary data types. By defining various fragments, TeSSLa manages bounds on the computation, delay and memory as a trade-off for expressiveness. In [DDG+18], TeSSLa monitors have been synthesized on FPGAs to monitor CPU cores, and memory accesses. Monitors are able to process trace data in real-time, enabling continuous observation of the state of a core.

3.4.3 Monitoring Streams of Time Intervals

Another stream-based verification approach is centered around reasoning about intervals of time. The nfer formalism [KHJF18] adopts a rule based approach to perform stream-based processing, where events consist of a labeled time interval (start, end) and a map of values. The formalism is inspired by Allen's Temporal Logic, and its semantics operates on matching, creating, merging, or splitting intervals, while comparing and modifying the map containing arbitrary values. The approach aims to handle global constraints on the pool of events and feasibly

process a large stream of events. An example of a constraint is *minimality*. Minimality states that there must exist at most one interval with the same time period and name while processing the stream. A DSL is introduced to write specifications for NFER. The semantics are implemented as rewriting rules written in both Scala and C. The resulting implementation is used to process three data-sets of large traces spanning at least 50,000 events.

3.4.4 Relation to Complex Event Processing

Overview of CEP. Complex Event Processing (CEP) [Luc05] encompasses a series of approaches that deal with processing events formed by aggregating other events. That is CEP consists of processing complex information in a system and outputting events that summarize, or provide a higher level view of the system. The primary objective of CEP is not verification, but rather understanding the state of a given system by recursively aggregating data. For example, by grouping a prepare and commit event in a database system, we can form a transaction event, then we can form more complex events that count the number of transactions performed or computing the average time needed to complete a transaction.

CEP and RV. Stream-based RV can be seen as a special case of CEP that aggregates the data in a way that performs verification [Hal16]. Tools used for CEP can be specialized to perform stream-based RV by defining aggregate functions that perform verification, as is the case of BEEPBEEP [HKG17]. It is possible to define a stream aggregation function that takes a stream of events as an input, and runs the sequence of events against an FSM, or a rewrite-based engine.

Part I

Hierarchical Decentralized Specifications

CHAPTER 4

Monitoring with Boolean Expressions

Contents

This chapter introduces the basic concepts and notation used in the remainder of the thesis. In particular, we introduce the concept of replicated data types (CRDTs) (Section 4.1), and their property of strong eventual consistency. CRDTs allow objects to have several replicas which eventually converge towards the same state. Then, we introduce the dict data structure (Section 4.2) used to build more complex data structures, define the basic concepts and notation for centralized and decentralized monitoring (Section 4.3), and we use them to define centralized monitoring of a centralized specification in the context of this thesis (Section 4.4).

4.1 Replicated Data Types

When monitoring decentralized systems, we see that algorithms require managing partial information. We presented three various strategies in Section 1.3.1. These strategies have utilized a single specification representing the behavior of the system, and multiple monitors. Monitors are deployed on different components, and need to communicate to infer information about the global state of the system. For monitors, we are interested in ensuring that they observe the same global state of the system once they exchange their information. To that end, we introduce the notion of a conflict-free replicated data type (CRDT) from [SPBZ11].

CRDTs and decentralized runtime verification. In [SPBZ11], the authors consider an object with multiple replicas at different parts of a distributed system. A replicated object with m replicas is a tuple $\langle S, s^{\text{init}}, \langle s_0, \ldots, s_{m-1}\rangle$, `query`, `update`, `merge`$\rangle$, where S denotes the possible states the objects can be in, s^{init} the initial state, $s_i \in S$ (for $i \in [0, m-1]$) denotes the state for each replica, and three operations that are performed on replicas: `query` which retrieves the object state for a given replica (i.e., s_i for a replica i), `update` which changes the state of a particular replica (i.e., modifies the object only affecting a given replica s_i), and `merge` which combines two replicas of the same object. When reasoning about the global state of a property for a decentralized system, the global state can be seen as an object, where each replica is the local copy for a given monitor deployed on a component. In particular,

we introduce later two data structures which represent replicated objects: memory (Section 4.3) and execution history encoding (Section 5.1). The memory represents observations received by monitors, and the execution history encoding represents the execution of the automaton associated with the global state of specification on a trace of observations. For these two objects, replicas constitute the local view associated with each monitor.

Since operations can occur on replicas, we are interested in ensuring that all replicas converge towards the same state of the object. In this context, *state equivalence* between any two replicas means that the result of operation `query` is the same across the replicas. CRDTs can be either state-based or operation-based. For state-based CRDTs (CvRDTs), a replica sends and receives another state for another replica, upon reception, the state of the remote replica is merged with the local state to create a new updated state. While for operation-based CRDTs (CmRDTs) the history of operations performed on a replica is communicated, and other replicas replay the operations to infer the state. For the purposes of this thesis, we focus on state-based replication, as we will mostly send the state of the object itself. However, both state-based and operation-based CRDTs are shown to be equivalent in [SPBZ11].

Properties of CvRDTs [SPBZ11]. The state of a specific replica is subject to change locally. The change should eventually be propagated for all replicas. Eventual consistency (Definition 2) states that replicas eventually converge to equivalent states.

Definition 2 (Eventual Consistency) Eventual consistency applied to replicated objects consists of the following conditions:

Eventual delivery: an update delivered at some replica is eventually delivered to all replicas.

Convergence: replicas that have been delivered the same updates eventually reach equivalent states.

Termination: operations `query`, `update`, and `merge` terminate.

Systems that are eventually consistent are capable of reaching the same state eventually. This may cause problems when updates can be conflicting, and a strategy for conflict-resolution (rollback) may be required. As such, a stronger condition is needed to ensure that upon receiving the *same* updates, replicas are in equivalent states. The added condition is referred to as strong eventual consistency (SEC).

Definition 3 (Strong Eventual Consistency (SEC)) Replicas that have been delivered the same updates have equivalent states.

To achieve SEC, the authors of [SPBZ11] distinguish objects with specific properties, called monotonic semilattice objects.

Definition 4 (Monotonic semilattice object) A state-based object equipped with a partial order $\leq$ with the following properties is a semilattice object: (1) the set of its states S forms a semilattice ordered by $\leq$, (2) merging a replica state s with a remote state s' computes the least upper bound (LUB) of the two states (i.e., $\texttt{merge}(s, s') = s \sqcup s'$), and (3) the state is monotonically non-decreasing across updates.

By converging to the LUB of the semilattice, and ensuring the `update` operation is monotonically non-decreasing, we guarantee that replicas performing the same merges converge to the same state.

Proposition 1 (CvRDT) By assuming eventual delivery and termination, a monotonic semilattice object is guaranteed to be SEC (see [SPBZ11] for details).

We illustrate a simple CvRDT called the *add-only set* in Example 9.

Example 9 (Add-only set) An add-only set [SPBZ11] is a CvRDT defined as $\langle 2^{\{v_0,v_1,v_2\}}, \emptyset, \langle s_0, s_1, s_2\rangle, \mathtt{q}, \mathtt{a}, \cup\rangle$, where S is the superset of all possible values that can be added, the initial state is the empty set, we have three replicas with the emptyset as the initial state, and the three operations are as follows: `q` which queries the CRDT returns the value for the local replica, `a` adds an element to the set (at the replica), and $\cup$ is the merge operations which perform set union. The states can be ordered using set inclusion ($\subseteq$) which forms a semilattice. Furthermore,

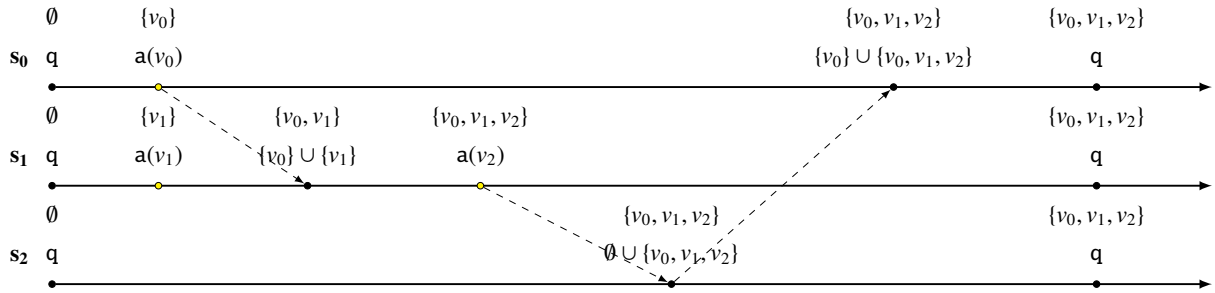


Figure 4.1: Add-only set replication using 3 replicas. For each replica we show the local state, and the operations performed. Update operations are shown in yellow, dashed arrows illustrate sending a state to another replica.

the merge operation which performs set union ($\cup$) only adds elements to the set, and thus is monotonically non-decreasing.

Figure 4.1 shows three replicas (s_0, s_1 and s_2) of an add-only set CvRDT object. The initial state is the empty set. The first two replicas perform concurrently an add operation, to add respectively the elements v_0 and v_1 to the set. At this point in the execution, the replicas all have different local states $\{v_0\}$, $\{v_1\}$ and $\emptyset$ for s_0, s_1 and s_2, respectively. Upon sending the state of s_0 to s_1, s_1 performs a merge and updates its state to be $\{v_0, v_1\}$. However since s_1 did not send its state to s_0, s_0 still has not seen update $\mathtt{a}(v_1)$. We notice that in the end, once all replicas have received information that precedes all updates, the query operation returns the same set for all three $\{v_0, v_1, v_2\}$. *

State-based replicated data types are capable of converging to the same state across multiple replicas. As such, when designing our data structures, we also would like to have the SEC property. This allows multiple monitors checking the same specification (i.e. decentralized monitoring of a centralized specification) to be easily spread in the system and eventually converge to the same view of the global system (whether for the observations, or the specification evaluation). In the next section (Section 4.2), we introduce the common data structure template that we rely on for the remainder of the thesis, designed to account for SEC.

4.2 The `dict` Data Structure

As introduced in Section 4.1 when monitoring decentralized systems, monitors typically have a state, and attempt to merge other monitor states with theirs to maintain a consistent view of the running system, that is, at no point in the execution, should two monitors receive updates that conflict with one another. We would like in addition, that any two monitors receiving the same information be in equivalent states. Therefore, we are interested in designing data structures that can replicate their state under strong eventual consistency (SEC).

We use a dictionary data structure (noted `dict`) as our basic building block that assigns a value to a given key. Data structure `dict` will be used to define the memory of a monitor (Section 4.3), and data structure `EHE` which encodes the execution of an automaton (Section 5.1).

We model `dict` as a partial function f. The domain of f (denoted by dom(f)) is the set of keys, while the codomain of f (denoted by codom(f)) is the set of values. `dict` supports two operations: `query` and `merge`. The `query` operation checks if a key $k \in$ dom(f) and returns f(k). If $k \notin$ dom(f), then it is undefined. The `merge` operation of a `dict` f with another `dict` g, is modeled as function composition. Two partial functions f and g are composed using operator $\dagger_{op}$ where $op : (\text{dom(f)} \times \text{dom(g)}) \to (\text{codom(f)} \cup \text{codom(g)})$ is a binary function.

$$\text{f} \dagger_{op} \text{g} : \text{dom(f)} \cup \text{dom(g)} \to \text{codom(f)} \cup \text{codom(g)}$$

$$\text{f} \dagger_{op} \text{g}(x) = \begin{cases} op(\text{f}(x), \text{g}(x)) & \text{if } x \in \text{dom(f)} \cap \text{dom(g)} \\ \text{g}(x) & \text{if } x \in \text{dom(g)} \setminus \text{dom(f)} \\ \text{f}(x) & \text{if } x \in \text{dom(f)} \setminus \text{dom(g)} \\ \text{undef} & \text{otherwise} \end{cases}$$

On sets of functions, $\dagger_{op}$ applies pairwise: $\biguplus^{op}\{f_1, \dots f_n\} = ((f_1 \dagger_{op} f_2) \dots f_n)$. The following two operators are used in the rest of the paper: $\dagger_2$ and $\dagger_\vee$. We define both of these operators to be commutative, idempotent, and associative to ensure SEC.

$$\dagger_2(x, x') = \begin{cases} x' & \text{if } x \prec x' \\ x & \text{otherwise} \end{cases} \qquad \dagger_\vee (x, x') = x \vee x'$$

Operator $\dagger_2$ acts as a replace function based on a total order ($\prec$) between the elements, so that it always chooses the highest element to guarantee idempotence and the join on the semilattice, while $\dagger_\vee$ uses the logical *or* operator to combine elements. Respectively, we denote the associated pairwise set operators by $\biguplus^2$ and $\biguplus^\vee$. Data structure `dict` can be composed by only using operation `merge`. When using `dict` to define subsequent data structures, we must ensure that the state itself can be ordered as a semilattice, and updates ensure the state is monotonically non-decreasing in that order (see Definition 4). Furthermore, we ensure that the `merge` is idempotent, commutative, and associative (with the two operators $\dagger_2$ and $\dagger_\vee$) so as to be insensitive to the order of message delivery and the same message being received multiple times.

4.3 Basic Monitoring Concepts

We recall the basic building blocks of monitoring. We consider the set of verdicts $\mathbb{B}_3 = \{\top, \bot, ?\}$ to denote the verdicts true, false, not reached (or inconclusive) respectively. A verdict in $\mathbb{B}_2 = \{\top, \bot\}$ is a *final* verdict. It indicates that the monitor has concluded its monitoring, and any further input will not affect it. Abstract states of a system are represented as a set of *atomic propositions* (*AP*). A monitoring algorithm typically includes additional information such as a timestamp associated with the atomic propositions. We capture this information as an encoding of the atomic propositions (*Atoms*), this encoding is left to the monitoring algorithm to specify.

Definition 5 (Event) An observation is a pair in $AP \times \mathbb{B}_2$ indicating whether or not a proposition is observed. An event is a set of observations in $2^{AP \times \mathbb{B}_2}$.

Example 10 (Event) We recall the example of a light switch and bulb from Example 2. We have $AP = \{s, \ell\}$. The event $\{\langle s, \top\rangle, \langle \ell, \bot\rangle\}$ indicates that the switch is observed to be on (i.e., the atomic proposition s is observed to be true), while the light bulb is observed to be off (i.e., the atomic proposition ℓ is observed to be false). ∗

A decentralized monitoring algorithm requires retaining, retrieving and communicating observations. Monitoring algorithms are versatile, and may require additional information associated with atomic propositions. This information can include timestamps indicating when the atomic proposition was observed, or a component ID, to determine where the atomic proposition was observed. As such, when stored, atomic propositions are typically encoded to add this additional information by the monitoring algorithm. To abstract the additional information, and remain general, the monitors store the encoded atomic proposition (instead of the atomic proposition itself), the encoded atomic proposition is referred to as *atom*. $Expr_{Atoms}$ (resp. $Expr_{AP}$) denotes the set of Boolean expressions over *Atoms* (resp. *AP*). When omitted, *Expr* refers to $Expr_{Atoms}$. An encoder is a function $enc : Expr_{AP} \to Expr_{Atoms}$ that encodes the atomic propositions into atoms. In this paper, we use two encoders: idt which is the identity function (it does not modify the atomic proposition), and ts_t which adds a timestamp t to each atomic proposition.

Definition 6 (Memory) A memory is a `dict` with the merge operator $\dagger_2$, and is modeled as a partial function $\mathcal{M} : Atoms \to \mathbb{B}_3$ that associates an atom to a verdict. The set of all memories is defined as *Mem*.

An event can be converted to a memory by encoding the atomic propositions to atoms, and associating their truth value: $memc : 2^{AP \times \mathbb{B}_2} \times (Expr_{AP} \to Expr_{Atoms}) \to Mem$.

Example 11 (Memory) We recall from Example 10 the event: $evt = \{\langle s, \top\rangle, \langle \ell, \bot\rangle\}$. At $t = 1$, the resulting memories using encoders idt and ts_1 are: $memc(evt, idt) = [s \mapsto \top, \ell \mapsto \bot]$, $memc(evt, ts_1) = [\langle 1, s\rangle \mapsto \top, \langle 1, \ell\rangle \mapsto \bot]$, respectively. ∗

In order to ensure SEC, we require that the state of the memory be a semilattice, with an update function that ensures the state is monotonically non-decreasing (see Section 4.1). For that, we require an order on the set of atoms (*Atoms*). This order can be assumed, as users can define an arbitrary order over AP, and extend it to the encoding. For encoder idt, we have $Atoms = AP$, no additional ordering is needed. For encoder ts, the order can be extended such that they are additionally ordered by timestamps. A memory $\mathcal{M}$ can be seen as a set of pairs $\{\langle x, y\rangle \mid \mathcal{M}(x) = y\}$. This is the same as treating it as an *add-only* set (Example 9). Therefore, a memory is a semilattice ordered by set-inclusion. An observation that is present in one but not the other memory is simply added (i.e. set union). Memories with conflicting observations can be replaced using an order between atoms. Since our assumption ensures that no two memories provide conflicting observations for the same atom[1], and are constructed only by merging existing memories, and if we impose a certain order on *Atoms*, then two memories $\mathcal{M}_1$ and $\mathcal{M}_2$ can be merged by applying operator $\dagger_2$, which computes the LUB. This ensures that the operation is idempotent, associative and commutative. Then, it follows from Proposition 1 that the memory data structure is a CvRDT. Monitors that exchange their memories and merge them have a consistent snapshot of the memory, regardless of the message ordering.

Corollary 1: A memory with operation $\dagger_2$ is a CvRDT. ◊

In this paper, we perform monitoring by manipulating expressions in *Expr*. The first operation we provide is rw, which rewrites the expression to attempt to eliminate *Atoms*.

Definition 7 (Rewriting an expression) An expression e is rewritten with a memory $\mathcal{M}$ using function rw : $Expr \times Mem \to Expr$ defined as follows:

$$
\begin{aligned}
\mathrm{rw}(e, \mathcal{M}) = \ & \texttt{match } e \texttt{ with} \\
& \mid a \in Atoms \quad \to \begin{cases} \mathcal{M}(a) & \text{if } a \in dom(\mathcal{M}) \\ a & \text{otherwise} \end{cases} \\
& \mid \neg e' \quad \to \neg \mathrm{rw}(e', \mathcal{M}) \\
& \mid e_1 \wedge e_2 \quad \to \mathrm{rw}(e_1, \mathcal{M}) \wedge \mathrm{rw}(e_2, \mathcal{M}) \\
& \mid e_1 \vee e_2 \quad \to \mathrm{rw}(e_1, \mathcal{M}) \vee \mathrm{rw}(e_2, \mathcal{M})
\end{aligned}
$$

Using information from a memory $\mathcal{M}$, the expression is rewritten by replacing atoms with a final verdict (a truth value in $\mathbb{B}_2$) in $\mathcal{M}$ when possible. Atoms that are not associated with a final verdict are kept in the expression. Operation rw yields a smaller formula to work with.

Example 12 (Rewriting) We extend the set of atomic propositions from Example 10 to include a motion sensor. We associate the motion sensor state with the atomic proposition pres, where if pres is observed to be $\top$, then the sensor is detecting motion. We have $AP = \{\mathrm{s}, \ell, \mathrm{pres}\}$. We consider the memory from Example 11: $\mathcal{M} = [\mathrm{s} \mapsto \top, \ell \mapsto \bot]$; and an expression $e = (\mathrm{s} \vee \ell) \wedge \mathrm{pres}$. In this case, we want to check if the light or switch are on only when the motion detects presence. We have $\mathcal{M}(\mathrm{s}) = \top$, $\mathcal{M}(\ell) = \bot$, $\mathcal{M}(\mathrm{pres}) = ?$. Since pres is associated with $? \notin \mathbb{B}_2$ then it will not be replaced when the expression is evaluated. The resulting expression is $\mathrm{rw}(e, \mathcal{M}) = (\top \vee \bot) \wedge \mathrm{pres}$. ∗

We eliminate additional atoms using Boolean logic. We denote by simplify(`expr`) the simplification of expression `expr` [2].

Example 13 (Simplification) Using the same setting as Example 12, we consider memory $\mathcal{M} = [\mathrm{s} \mapsto \top]$ and expression $e = (\mathrm{s} \wedge \ell) \vee (\mathrm{s} \wedge \neg\ell)$. We have $e' = \mathrm{rw}(e, \mathcal{M}) = (\ell \vee \neg\ell)$. We notice that rewriting e' does not yield a final verdict. However, atoms can be eliminated with simplify(e'). We finally get $\top$. ∗

We combine both rewriting and simplification in the eval function which determines a verdict from an expression e.

[1] We are able to handle conflicts by imposing an order on $\mathbb{B}_3$.
[2] This is also known as The Minimum Equivalent Expression problem [BU08].

Definition 8 (Evaluating an expression) The evaluation of a Boolean expression $e \in Expr$ using a memory $\mathcal{M}$ yields a verdict. Function eval : $Expr \times Mem \to \mathbb{B}_3$ is defined as:

$$\text{eval}(e, \mathcal{M}) = \begin{cases} \top & \text{if simplify}(\text{rw}(e, \mathcal{M}) \Leftrightarrow \top, \\ \bot & \text{if simplify}(\text{rw}(e, \mathcal{M}) \Leftrightarrow \bot, \\ ? & \text{otherwise.} \end{cases}$$

Function eval returns the verdict $\top$ (resp. $\bot$) if the simplification after rewriting is (Boolean) equivalent to $\top$ (resp. $\bot$), otherwise it returns verdict ?.

Example 14 (Evaluating expressions) We recall from Example 12 memory $\mathcal{M} = [\text{s} \mapsto \top, \ell \mapsto \bot]$; and expression $e = (\text{s} \vee \ell) \wedge \text{pres}$. We have simplify(rw($e, \mathcal{M}$)) = simplify($(\top \vee \bot) \wedge \text{pres}$) = pres, and eval($e, \mathcal{M}$) = ? which depends on pres. We cannot emit a final verdict before observing pres. ∗

A decentralized system is a set of components C. We assign a sequence of events to each component using a decentralized trace function.

Definition 9 (Decentralized trace) A decentralized trace of length n is a total function tr : $[1, n] \times C \to 2^{AP \times \mathbb{B}_2}$ (where $[1, n]$ denotes the interval of the n first non-zero natural numbers).

Function tr assigns an event to a component for a given timestamp. We denote by $\mathcal{T}$ the set of all possible decentralized traces. We additionally define function lu : $AP \to C$ to assigns an atomic proposition to a component. We assume that (1) no two components can observe the same atomic propositions[3], and (2) at least one atomic proposition is associated with a component (a component with no atomic propositions to monitor, can be simply considered excluded from the system under monitoring). Function lu is defined as lu(ap) = c s.t. $\exists t \in \mathbb{N}, \exists v \in \mathbb{B}_2 : \langle ap, v \rangle \in \text{tr}(t, c)$.

We consider timestamp 0 to be associated with the initial state, therefore our traces start at 1. The length of a trace tr is denoted by |tr|. An empty trace has length 0 and is denoted by ϵ. Monitoring using LTL or finite-state automata relies on sequencing the trace. Events must be totally ordered. A timestamp indicates simply the order of the sequence of events. As such, a timestamp represents a logical time, it can be seen as a *round number*. Every round consists in a transition taken on the automaton after reading a part of the word. While tr gives us a view of what components can locally see, we reconstruct the global trace to reason about all observations. A global trace of the system is therefore a sequence of events. A global trace encompasses all observations observed locally by components. While a global trace will never be used in practice, we use it for the purpose of reasoning about the global state (Section 4.4), and ensuring the correctness of our approach (Proposition 3).

Definition 10 (Reconstructing a global trace) Given a decentralized trace tr of length n, we reconstruct the global trace using function $\rho : \left([1, n] \times C \to 2^{AP \times \mathbb{B}_2}\right) \to \left([1, n] \to 2^{AP \times \mathbb{B}_2}\right)$ defined as $\rho(\text{tr}) = evt_1 \cdot \ldots \cdot evt_n$ s.t. $\forall i \in [1, n] : evt_i = \bigcup_{c \in C} \text{tr}(i, c)$.

For each timestamp $i \in [1, n]$, we take all observations of all components and union them to get a global event. Consequently, an empty trace yields an empty global trace, $\rho(\epsilon) = \epsilon$.

Example 15 (Traces) Using the switch and light bulb from Example 2, we define multiple components. We consider a system of two components lswitch and bulb, that are associated with atomic propositions s and ℓ respectively. An example decentralized trace of the system is given by tr = [1 ↦ lswitch ↦ $\{\langle \text{s}, \top \rangle\}$, 1 ↦ bulb ↦ $\{\langle \ell, \top \rangle\}$, 2 ↦ lswitch ↦ $\{\langle \text{s}, \top \rangle\}$, 2 ↦ bulb ↦ $\{\langle \ell, \bot \rangle\}$]. That is, component lswitch observes proposition s to be $\top$ at both timestamps 1 and 2, while bulb observes ℓ to be $\top$ at timestamp 1 and $\bot$ at timestamp 2. The associated global trace is: $\rho(\text{tr}) = \{\langle \text{s}, \top \rangle, \langle \ell, \top \rangle\} \cdot \{\langle \text{s}, \top \rangle, \langle \ell, \bot \rangle\}$. ∗

[3] This is not necessary, it makes the presentation clearer. For components sharing observations, we can encode their own ID in the atom to make it unique.

4.4 Monitoring a Centralized Specification with Expressions

We now focus on a decentralized system specified by one global automaton. We consider automata that emit 3-valued verdicts in the domain $\mathbb{B}_3$, similar to those in [CF16a, BLS11] for centralized systems. Using automata with 3-valued verdicts has been the topic of a lot of the Runtime Verification literature [BLS11, BF12, CF16a, FCF14, BKZ15], we focus on extending the approach for decentralized systems in [CF16a] to use a new data structure called Execution History Encoding (EHE). Typically, monitoring is done by labeling an automaton with events, then playing the trace on the automaton and determining the verdict based on the reached state. Specifications are similar to the Moore automata generated by [BLS11]. We modify labels to be Boolean expressions over atomic propositions (in $Expr_{AP}$). We choose to label the transitions with Boolean expressions as opposed to events, to keep a homogeneous representation (with EHE)[4].

Definition 11 (Specification) The specification is a deterministic and complete Moore automaton $\langle Q, q_0, \delta, \text{ver}\rangle$ where $q_0 \in Q$ is the initial state, $\delta : Q \times Expr_{AP} \to Q$ is the transition function and $\text{ver} : Q \to \mathbb{B}_3$ is the labeling function.

The labeling function associates a verdict with each state. When using multiple automata we use labels to separate them, $\mathcal{A}_\ell = \langle Q_\ell, q_{\ell_0}, \delta_\ell, \text{ver}_\ell\rangle$. We fix $\mathcal{A}$ to be a specification automaton for the remainder of this section. For monitoring, we are interested in events (Definition 5), we extend δ to events, and denote it by Δ[5].

Definition 12 (Transition over events) Given an event evt, we build the memory $\mathcal{M} = \text{memc}(evt, \text{idt})$. Then, function $\Delta : Q \times 2^{AP \times \mathbb{B}_2} \to Q$ is defined as follows:

$$\Delta(q, evt) = \begin{cases} q' & \text{if } evt \neq \emptyset \wedge \exists q' \in Q, \exists e \in Expr_{AP} : \delta(q, e) = q' \wedge \text{eval}(e, \mathcal{M}) = \top, \\ q & \text{otherwise.} \end{cases}$$

A transition is taken only when an event contains observations (i.e., $evt \neq \emptyset$). This allows the automaton to wait on observations before evaluating, as such it remains in the same state (i.e., $\Delta(q, \emptyset) = q$). Upon receiving observations, we use $\mathcal{M}$ to evaluate each label of an outgoing transition, and determine if a transition can be taken (i.e., $\exists q' \in Q, \exists e \in Expr_{AP} : \delta(q, e) = q' \wedge \text{eval}(e, \mathcal{M}) = \top$).

To handle a trace, we extend Δ to its reflexive and transitive closure in the usual way, and note it Δ^*. For the empty trace, the automaton makes no moves, i.e., $\Delta^*(q_0, \epsilon) = q_0$.

Example 16 (Monitoring using expressions) Recall the monitor from Example 4 monitoring the light switch and bulb interaction. Let us consider the global trace from Example 15: $evt_0 \cdot evt_1$, with $evt_0 = \{\langle \text{s}, \top\rangle, \langle \ell, \top\rangle\}$ and $evt_1 = \{\langle \text{s}, \top\rangle, \langle \ell, \bot\rangle\}$. The resulting memory at $t = 1$ is $\mathcal{M} = \text{memc}(evt_0, \text{idt}) = [\text{s} \mapsto \top, \ell \mapsto \top]$ (see Example 11). The transition from q_0 to q_1 is taken since $\text{eval}(\text{s}, \mathcal{M}) = \top$. Thus we have $\Delta(q_0, evt_0) = q_1$ with verdict $\text{ver}(q_1) = ?$. We continue by repeating the process for $t = 2$. The memory is $\mathcal{M}' = \text{memc}(evt_1, \text{idt}) = [\text{s} \mapsto \top, \ell \mapsto \bot]$. The transition from q_1 to q_2 is taken since $\text{eval}(\text{s} \wedge \neg\ell, \mathcal{M}) = \top$. Thus we have $\Delta(q_1, evt_1) = q_2$ with verdict $\text{ver}(q_2) = \bot$. We can see that for this trace, the property is violated. *

Remark 1 (Properties and normalization) We recall that the specification is a deterministic and complete automaton. Hence, there are properties on the expressions that label the transition function. For any $q \in Q$, we have:

1) $\forall \mathcal{M} \in Mem : (\exists \langle q, e\rangle \in \text{dom}(\delta) : \text{eval}(e, \mathcal{M}) = \top) \implies (\nexists \langle q, e'\rangle \in \text{dom}(\delta) \setminus \{\langle q, e\rangle\} : \text{eval}(e', \mathcal{M}) = \top)$; and
2) the disjunction of the labels of all outgoing transitions results in an expression that is a tautology.

[4] Indeed, an event can be converted to an expression by the conjunction of all observations, negating the terms that are associated with the verdict $\bot$.

[5] We note that in this case, we are not using any encoding ($Atoms = AP$).

The first property states that for all possible memories encoded with idt no two (or more) labels can evaluate to $\top$ at once. It results from determinism: no two (or more) transitions can be taken at once. The second property results from completeness: given any input, the automaton must be able to take a move. Furthermore, we note that for each pair of states $\langle q, q' \rangle \in Q \times Q$, we can rewrite δ such that there exists at most one expression $e \in Expr_{AP}$, such that $\delta(q, e) = q'$, without loss of generality. This is because for a pair of states, we can always disjoin the expressions to form only one expression, as it suffices that only one expression needs to evaluate to $\top$ to reach q'. By having at most one transition between any pair of states, we simplify the topology of the automaton. $*$

After introducing the fundamentals of using boolean expressions for monitoring, and examining centralized monitoring of centralized specifications, we shift the focus in the next chapter (Chapter 5) to account for partial observations by encoding the automaton execution using the execution history encoding data structure.

CHAPTER 5

Managing Partial Observations with Execution History Encoding

Contents

Chapter abstract

This chapter introduces the *Execution History Encoding* (EHE) data structure. The EHE data structure encodes the execution of an automaton using boolean expressions while preserving soundness and determinism. Furthermore, it is constructed to replicate under strong eventual consistency, ensuring that any two monitors exchanging their EHE are able to determine the same information about the global state. By analyzing the size of the EHE, we are able to determine the computation and communication costs incurred by decentralized monitoring algorithms utilizing it. Computation cost is affected by the rewriting and simplifying boolean of expressions, while communication impacts how large the EHE grows, and thus, the cost of exchanging it.

Introduction

This chapter introduces the *Execution History Encoding* (EHE) data structure to account for the challenge of managing partial observations. The EHE is used in the remainder of the thesis when encoding partial information.

Motivation. The EHE data structure encodes an execution of an automaton. The EHE keeps track of potential reachable states given partial information, by modeling paths to possible states as boolean expressions. Using boolean expressions allows us to utilize boolean simplification [BU08] to simplify information that is no longer necessary to determine the state. By doing so, the EHE allows us to make the most of partial information without requiring the full information to conclude. Furthermore, the EHE data structure provides additional properties useful in a decentralized context. Namely, we focus on the ability of the EHE to replicate under strong eventual consistency. That is, for any two monitors exchanging their EHE encoding the same automaton, they are able to determine the same information about the verification of the system. The EHE data structure also allows to unify the analysis of different decentralization strategies, as the EHE relies on rewriting boolean expressions instead of LTL. Therefore, EHE has a predictable size, and more predictable bounds, as opposed to standard rewriting based approaches for LTL for which the formula size evolves unpredictably at runtime, and we are unable to determine the minimal expression (see Section 3.1.3).

Chapter organization. In Section 5.1, we introduce the EHE data structure, we elaborate on its operations, we also illustrate properties, and utilize it for decentralized monitoring. In Section 5.2, we elaborate on costs for utilizing EHE, and identify the information delay parameter. Information delay is used to describe the delay it takes before partial knowledge is able to determine a global state. We study how information delay impacts the size of the EHE, which affects computation and communication costs.

Key contributions. The key contributions of this chapter can be summarized as follows:

1. Introducing the EHE data structure for encoding partial information;
2. Elaborating on two properties for EHE: determinism (Proposition 2) and soundness when encoding automata (Proposition 3);
3. Showing how EHE can be used for decentralized monitoring due to its strong eventual consistency property (Section 5.1.2); and
4. Presenting the cost model for EHE, as a unified model for analyzing algorithms with partial information (Section 5.2).

5.1 Monitoring using Execution History Encoding

The execution of the specification automaton, is in fact, the process of monitoring, upon running the trace, the reached state determines the verdict. An execution of the specification automaton can be seen as a sequence of states $q_0 \cdot q_1 \cdot \ldots q_t \ldots$. It indicates that, for each timestamp $t \in \mathbb{N}^*$, the automaton is in the state q_t[1]. In a decentralized system, a component receives only local observations and does not necessarily have enough information to determine the state at a given timestamp. Typically, when sufficient information is shared between various components, it is possible to know the state q_t that is reached in the automaton at t (we say that the state q_t has been found, in such a case). The aim of the EHE is to construct a data structure which follows the current state of an automaton, and in case of partial information, tracks the possible states the automaton can be in. For that purpose, we need to ensure strong eventual consistency in determining the state q_t of the execution of an automaton. That is, after two different monitors share their EHE, they should both be able to find q_t for t (if there exists enough information to infer the global state), or if not enough information is available, they both find no state at all.

[1] We note that in the case of RV, traces are typically finite.

5.1.1 The Execution History Encoding (EHE) data structure

Execution History Encoding (EHE) is a data structure designed to encode an execution of an automaton using boolean expressions while accounting for partial observations.

Definition 13 (Execution History Encoding - EHE) An Execution History Encoding (EHE) of the execution of an automaton $\mathcal{A}$ is a partial function $\mathcal{I} : \mathbb{N} \times Q \to Expr$.

Intuitively, for a given execution, an EHE encodes the conditions to be in a state at a given timestamp as an expression in $Expr$. $\mathcal{I}(t, q)$ is an expression used to track whether the automaton is in state q at t, i.e., $\mathcal{I}(t, q)$ holds iff the automaton is in state q at timestamp t. We begin by defining the EHE at timestamp $t = 0$ which indicates the initial state of the execution. For a given automaton with an initial state q_0, we know that we are indeed in the initial state at $t = 0$. As such, the initial EHE for the beginning of the execution is the function $[0 \mapsto q_0 \mapsto \top]$. For future timestamps, the EHE is extended inductively based on reachable states.

Definition 14 (Constructing an EHE) An EHE encoding the execution till timestamp t, noted $\mathcal{I}^t$ is constructed inductively using function $\text{mov} : (\mathbb{N} \times Q \to Expr) \times \mathbb{N} \times \mathbb{N} \to (\mathbb{N} \times Q \to Expr)$

$$\begin{aligned} \mathcal{I}^t &\stackrel{\text{def}}{=} \text{mov}([0 \mapsto q_0 \mapsto \top], 0, t) \\ \text{mov}(\mathcal{I}, t_s, t_e) &\stackrel{\text{def}}{=} \begin{cases} \text{mov}(\mathcal{I}', t_s + 1, t_e) & \text{if } t_s < t_e, \\ \mathcal{I} & \text{otherwise,} \end{cases} \end{aligned}$$

with $\mathcal{I}' = \mathcal{I} \dagger_\vee \bigvee\!\!\biguplus_{q' \in \text{next}(\mathcal{I}, t_s)} \{t_s + 1 \mapsto q' \mapsto \text{to}(\mathcal{I}, t_s, q', \text{ts}_{t_s+1})\}$, and

$$\begin{aligned} \text{to}(\mathcal{I}, t, q', \text{enc}) &\stackrel{\text{def}}{=} \bigvee_{\{\langle q, e' \rangle \mid \delta(q, e') = q'\}} (\mathcal{I}(t, q) \wedge \text{enc}(e')) \\ \text{next}(\mathcal{I}, t) &\stackrel{\text{def}}{=} \{q' \in Q \mid \exists \langle t, q \rangle \in \text{dom}(\mathcal{I}), \exists e \in Expr : \delta(q, e) = q'\}. \end{aligned}$$

The automaton is in the initial state at $t = 0$. We start building up $\mathcal{I}$ with the initial state and associating it with expression $\top$: $[0 \mapsto q_0 \mapsto \top]$. Then, for a given timestamp t, we use function next to check the next set of reachable states in the automaton (at $t + 1$) by looking at the outgoing transitions for all states in $\mathcal{I}$ at t (i.e., we find a state q' such that $\exists \langle t, q \rangle \in \text{dom}(\mathcal{I}), \exists e \in Expr : \delta(q, e) = q'$).

We now build the necessary expression to reach a state q' from multiple states by disjoining the transition labels using $\text{to}(\mathcal{I}, t, q', \text{enc})$, as it suffices to take only one such path to reach q'. Since the label consists of expressions in $Expr_{AP}$ we use an encoder (enc) to get an expression in $Expr_{Atoms}$. If an expression $\mathcal{I}(t, q)$ encodes the condition to reach q at t, and q' is reachable from q at $t + 1$ using the condition e', then it suffices to compute the conjunction.

Finally, $\mathcal{I}'$ is obtained by considering the next states and merging all their expressions with $\mathcal{I}$: $\mathcal{I}' = \mathcal{I} \dagger_\vee \bigvee\!\!\biguplus_{q' \in \text{next}(\mathcal{I}, t_s)} \{t_s + 1 \mapsto q' \mapsto \text{to}(\mathcal{I}, t_s, q', \text{ts}_{t_s+1})\}$. We recall from Section 4.2 that operator $\dagger_\vee$ performs the disjunction between entries, while operator $\bigvee\!\!\biguplus$ on EHE adds expressions for given timestamps and states that are not present, and merges multiple EHEs row by row using disjunction when the entry exists. As such, an EHE is assembled for $t + 1$ by combining all expressions for reachable states at $t + 1$ using $\bigvee\!\!\biguplus$. The assembled EHE for $t + 1$ is then combined with the EHE for t ($\mathcal{I}$) using $\dagger_\vee$, to form the EHE that contains both ($\mathcal{I}'$).

We use the notation $\text{rounds}(\mathcal{I})$, to denote all the timestamps that the EHE encodes, i.e., $\text{rounds}(\mathcal{I}) = \{t \in \mathbb{N} \mid \langle t, q \rangle \in \text{dom}(\mathcal{I})\}$. Similarly to automata notation, if multiple EHEs are present, we use a label in the subscript to identify them and their respective operations ($\mathcal{I}_\ell$ denotes the EHE of $\mathcal{A}_\ell$).

Example 17 (Constructing an EHE) We encode the execution of the automaton presented in Example 16. For this example, we use the encoder ts_n which appends timestamp n to an atomic proposition. We have $\mathcal{I}^0 = [0 \mapsto q_0 \mapsto \top]$. From q_0, it is possible to go to q_0 or q_1, therefore $\text{next}(\mathcal{I}^0, 0) = \{q_0, q_1\}$. To stay at q_0 at $t = 1$, we must be at q_0

Table 5.1: A tabular representation of $\mathcal{I}^2$

t	q	e
0	q_0	$\top$
1	q_0	$\neg\langle 1, \mathrm{s}\rangle$
1	q_1	$\langle 1, \mathrm{s}\rangle$
2	q_0	$(\neg\langle 1, \mathrm{s}\rangle \wedge \neg\langle 2, \mathrm{s}\rangle) \vee (\langle 1, \mathrm{s}\rangle \wedge \neg\langle 2, \mathrm{s}\rangle)$
2	q_1	$(\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle \wedge \langle 2, \ell\rangle) \vee (\neg\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle))$
2	q_2	$\langle 1, \mathrm{s}\rangle \wedge \langle 2, \mathrm{s}\rangle \wedge \neg\langle 2, \ell\rangle$

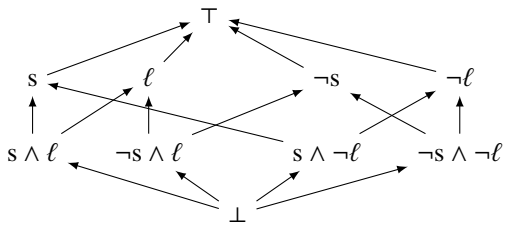


Figure 5.1: Ordering boolean expressions with $\Longrightarrow$.

at $t = 0$, and have : $\mathrm{to}(\mathcal{I}^0, 0, q_0, \mathrm{ts}_1) = \mathcal{I}^0(0, q_0) \wedge \neg\langle 1, \mathrm{s}\rangle$. To move to q_1 at $t = 1$, we must be at q_0 at $t = 0$. The following condition must hold: $\mathrm{to}(\mathcal{I}^0, 0, q_1, \mathrm{ts}_1) = \mathcal{I}^0(0, q_0) \wedge \langle 1, \mathrm{s}\rangle = \langle 1, \mathrm{s}\rangle$. The encoding up to timestamp $t = 2$ is obtained with $\mathcal{I}^2 = \mathrm{mov}(\mathcal{I}^0, 0, 2)$ and is shown in Table 5.1. We notice that when a state can be reached from multiple states, their expressions are disjoined. For instance, to reach q_0 at $t = 2$, we can either have stayed at q_0 at $t = 1$ and taken the loop transition or have moved to q_1, then taken the transition back to q_0 ($\neg\langle 2, \mathrm{s}\rangle$). ∗

Constructing an EHE with function mov is done only through merges using operator $\dagger_\vee$. We can impose a partial order on the set of boolean expressions using $\Longrightarrow$, as it is reflexive, transitive, and antisymmetric. As such, $\Longrightarrow$ guarantees a semilattice construction with $\top$ as the maximal element. We illustrate the order between two atomic propositions s and ℓ in Figure 5.1. We see that using logical OR ($\vee$) computes the LUB of the lattice. By creating an arbitrary order on the states (noted $\prec^{\mathrm{Q}}$) in the specification automaton, we can then compare any two entries of the EHE. One can see that similarly to memory in Definition 6, we can enumerate the state of the EHE $\mathcal{I}$ as a set of tuples: $\{\langle t, q, e\rangle \mid \mathcal{I}(t, q) = e\}$. We are then able to compare any such two tuples $\langle t, q, e\rangle \sqsubseteq \langle t', q', e'\rangle$ iff $t \leq t' \wedge q \preceq^{\mathrm{Q}} q' \wedge e \implies e'$. By merging with $\dagger_\vee$ (Section 4.2) the entries not found in both are added using set union, and conflicting entries are disjoined ($\vee$), effectively computing the LUB. As such, the EHE is a CvRDT.

Corollary 2: An EHE constructed with mov and merged with $\dagger_\vee$ is a CvRDT. ◊

By constructing the EHE, we have for each timestamp t and each state q in the EHE an expression. Using information from the execution stored in a memory $\mathcal{M}$, if $\mathrm{eval}(\mathcal{I}(t, q), \mathcal{M})$ is $\top$, then we know that the automaton is indeed in state q at timestamp t. Given a memory $\mathcal{M}$ which stores atoms, function sel determines if a state is reached at a timestamp t. If the memory does not contain enough information to evaluate the expressions, then the state is undef. The state q at timestamp t with a memory $\mathcal{M}$ is determined by:

$$\mathrm{sel}(\mathcal{I}, \mathcal{M}, t) = \begin{cases} q & \text{if } \exists q \in Q : \mathrm{eval}(\mathcal{I}(t, q), \mathcal{M}) = \top, \\ \text{undef} & \text{otherwise.} \end{cases}$$

We note that q such that $\mathrm{eval}(\mathcal{I}(t, q), \mathcal{M}) = \top$ is unique. Since we are encoding deterministic automata, we recall from Remark 1 that when a state q is reached at a timestamp t resulting from an execution, no other state can be reached at t for the same execution. Moreover, the EHE construction using operation mov and encoder ts preserves determinism.

Proposition 2 (Deterministic EHE) Given an EHE $\mathcal{I}$ constructed with operation mov using encoder ts, we

have:

$$\forall t \in \text{rounds}(\mathcal{I}), \forall \mathcal{M} \in \mathit{Mem}, \exists q \in Q : \text{eval}(\mathcal{I}(t,q), \mathcal{M}) = \top \implies \forall q' \in Q \setminus \{q\} : \text{eval}(\mathcal{I}(t,q'), \mathcal{M}) \neq \top.$$

Determinism is preserved since, by using encoder ts, we only change an expression to add the timestamp. By construction, when there exists a state q s.t. $\text{eval}(\mathcal{I}(t,q), \mathcal{M}) = \top$, such a state is unique, since the EHE is built using a deterministic automaton. The full proof is in Appendix A.

Function verAt is a short-hand to retrieve the verdict at a given timestamp t:

$$\text{verAt}(\mathcal{I}, \mathcal{M}, t) = \begin{cases} \text{ver}(q) & \text{if } \exists q \in Q : q = \text{sel}(\mathcal{I}, \mathcal{M}, t), \\ ? & \text{otherwise.} \end{cases}$$

Example 18 (Monitoring with EHE) We recall from Example 17 the constructed EHE shown in Table 5.1. Let us consider the global trace from Example 15: $evt_0 \cdot evt_1$, with $evt_0 = \{\langle s, \top\rangle, \langle \ell, \top\rangle\}$ and $evt_1 = \{\langle s, \top\rangle, \langle \ell, \bot\rangle\}$. We create a memory with the events of the two timestamps. Let $\mathcal{M} = \text{memc}(evt_0, \text{ts}_1) \dagger_2 \text{memc}(evt_1, \text{ts}_2) = [\langle 1, s\rangle \mapsto \top, \langle 1, \ell\rangle \mapsto \top, \langle 2, s\rangle \mapsto \top, \langle 2, \ell\rangle \mapsto \bot,]$. It is possible to infer the state of the automaton at $t = 2$ using $\mathcal{I}^2 = \text{mov}([0 \mapsto q_0 \mapsto \top], 0, 2)$ by using $\text{sel}(\mathcal{I}^2, \mathcal{M}, 2)$, we evaluate:

$$\begin{array}{lll} \text{eval}(\mathcal{I}^2(2,q_0), \mathcal{M}) & = (\neg\langle 1, s\rangle \wedge \neg\langle 2, s\rangle) \vee (\langle 1, s\rangle \wedge \neg\langle 2, s\rangle) & = \bot \\ \text{eval}(\mathcal{I}^2(2,q_1), \mathcal{M}) & = (\langle 1, s\rangle \wedge \langle 2, s\rangle \wedge \langle 2, \ell\rangle) \vee (\neg\langle 1, s\rangle \wedge \langle 2, s\rangle)) & = \bot \\ \text{eval}(\mathcal{I}^2(2,q_2), \mathcal{M}) & = \langle 1, s\rangle \wedge \langle 2, s\rangle \wedge \neg\langle 2, \ell\rangle & = \top \end{array}$$

We find that q_2 is the selected state, with verdict $\text{ver}(q_2) = \bot$. *

While the construction of an EHE preserves the determinism found in the automaton, an important property is in ensuring that the EHE encodes correctly the execution of the automaton.

Proposition 3 (Soundness) Given a decentralized trace tr of length n, we reconstruct the global trace $\rho(\text{tr}) = evt_1 \cdot \ldots \cdot evt_n$, we have: $\Delta^*(q_0, \rho(\text{tr})) = \text{sel}(\mathcal{I}^n, \mathcal{M}^n, n)$, with:
$\mathcal{I}^n = \text{mov}([0 \mapsto q_0 \mapsto \top], 0, n)$, and
$\mathcal{M}^n = \biguplus^2_{t\in[1,n]}\{\text{memc}(evt_t, \text{ts}_t)\}$.

EHE is sound wrt the specification automaton; both the automaton and EHE will indicate the same state reached with a given trace. Thus, the verdict is the same as it would be in the automaton. The proof is by induction on the reconstructed global trace ($|\rho(\text{tr})|$).

Proof sketch We first establish that both the EHE and the automaton memories evaluate two similar expressions modulo encoding to the same result. That is, for the given length i, the generated memories at $i+1$ with encodings idt and ts_{i+1} yield similar evaluations for the same expression e. Then, starting from the same state q_i reached at length i, we assume $\Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_i) = \text{sel}(\mathcal{I}^i, \mathcal{M}^i, i) = q_i$ holds. We prove that it holds at $i+1$, by building the expression (for each encoding) to reach state q_{i+1} at $i+1$, and showing that the generated expression is the only expression that evaluates to $\top$. As such, we determine that both evaluations point to q_{i+1} being the next state. The full proof is in Appendix A.

5.1.2 Decentralized Monitoring with EHE

EHE provides interesting properties for decentralized monitoring. Two (or more) components sharing EHEs and merging them will be able to infer the same execution history of the automaton. That is, components will be able to aggregate the information of various EHEs, and are able to determine the reached state, if possible, or that no state was reached. Merging two EHEs of the same automaton with $\dagger_\vee$ allows us to aggregate information from two partial histories.

However, two EHEs for the same automaton contain the same expression if constructed with mov. To incorporate the memory in an EHE, we generate a new EHE that contains the rewritten and simplified expressions for each entry. To do so we define function inc to apply to a whole EHE and a memory to generate a new EHE: $\text{inc}(\mathcal{I}, \mathcal{M}) = \biguplus^{2}_{\langle t,q\rangle \in \text{dom}(\mathcal{I})}\{[\langle t, q\rangle \mapsto \text{simplify}(\text{rw}(\mathcal{I}(t, q), \mathcal{M}))]\}$. We note, that for a given $\mathcal{I}$ and $\mathcal{M}$, $\text{inc}(\mathcal{I}, \mathcal{M})$ maintains the invariant of Proposition 2. We are simplifying expressions or rewriting atoms with their values in the memory which is what eval already does for each entry in the EHE. That is, $\text{inc}(\mathcal{I}, \mathcal{M})$ is a valid representation of the same deterministic and complete automaton as $\mathcal{I}$. However, $\text{inc}(\mathcal{I}, \mathcal{M})$ incorporates information from memory $\mathcal{M}$ in addition.

Proposition 4 (Memory obsolescence)

$$\forall \langle t, q\rangle \in \text{dom}(\text{inc}(\mathcal{I}, \mathcal{M})) : \text{eval}(\mathcal{I}(t, q), \mathcal{M}) \Leftrightarrow \text{eval}(\text{inc}(\mathcal{I}, \mathcal{M})(t, q), []).$$

Proof : Follows directly by construction of inc and the definition of eval (which uses functions simplify and rw).■

Proposition 4 ensures that it is possible to directly incorporate a memory in an EHE, making the memory no longer necessary. This is useful for algorithms that communicate the EHE, as they do not need to also communicate the memory.

By rewriting the expressions, the EHEs of two different monitors receiving different observations contain different expressions. However, since they still encode the same automaton, and observations do not conflict, merging with $\dagger_\vee$ shares useful information.

Corollary 3: Given an EHE $\mathcal{I}$ constructed using function mov, and two memories $\mathcal{M}_1$ and $\mathcal{M}_2$ that do not observe conflicting observations[2], the two EHEs $\mathcal{I}_1 = \text{inc}(\mathcal{I}, \mathcal{M}_1)$ and $\mathcal{I}_2 = \text{inc}(\mathcal{I}, \mathcal{M}_2)$ have the following properties $\forall \langle t, q\rangle \in \text{dom}(\mathcal{I}')$:

1) $\mathcal{I}' = \mathcal{I}_1 \dagger_\vee \mathcal{I}_2$ is deterministic (Proposition 2);
2) $\text{eval}(\mathcal{I}'(t, q), []) \implies \text{eval}(\mathcal{I}(t, q), \mathcal{M}_1 \dagger_2 \mathcal{M}_2)$;
3) $\text{eval}(\mathcal{I}'(t, q), []) = \top \implies \text{eval}(\mathcal{I}'(t, q), \mathcal{M}_1) = \top \wedge \text{eval}(\mathcal{I}'(t, q), \mathcal{M}_2) = \top$;
4) $\text{eval}(\mathcal{I}'(t, q), []) = \top \implies \text{eval}(\mathcal{I}_1(t, q), \mathcal{M}_1) \neq \bot \wedge \text{eval}(\mathcal{I}_2(t, q), \mathcal{M}_2) \neq \bot$. ◊

The first property ensures that the merge of two EHEs that incorporate memories are still indeed representing a deterministic and complete automaton, this follows from Proposition 2 and Proposition 4. Since operation $\dagger_\vee$ disjoins the two expressions, and since the two expressions come from EHEs that each maintain the property, the additional disjunction will not affect the outcome of eval. The second property extends Proposition 4 to the merging of EHE with incorporated memories. It follows directly from Proposition 4, and the assumptions that the memories have no conflicts. The third property adds a stronger condition. It states that merging two EHEs with incorporated memories results in an EHE that evaluates to true, cannot evaluate to anything else with the two different memories. This follows from the second property and the fact that the memories do not have conflicting observations. Finally, the fourth property ensures that merging an EHE with an entry that evaluates to $\bot$ does not result in an entry that evaluates to $\top$. That is, if an EHE has already determined that a state is not reachable, merging it with another EHE does not result in the state being reachable. This ensures the consistency when sharing information. This property follows from the merging operator $\dagger_\vee$ which uses $\vee$ to merge entries in two EHEs. We recall that an entry in $\langle t, q\rangle \in \text{dom}(\mathcal{I}')$ is constructed as: $\text{eval}(\mathcal{I}_1(t, q), \mathcal{M}_1) \vee \text{eval}(\mathcal{I}_2(t, q), \mathcal{M}_2)$. For $\text{eval}(\mathcal{I}'(t, q), [])$ to be $\top$, either $\text{eval}(\mathcal{I}_1(t, q), \mathcal{M}_1)$ or $\text{eval}(\mathcal{I}_2(t, q), \mathcal{M}_2)$ has to be $\top$, if one is already $\bot$, then the other has to be $\top$. This leads to a contradiction, since both $\mathcal{I}_1$ and $\mathcal{I}_2$ encode the same deterministic automaton, as such, the automaton cannot be in two states at once.

Example 19 (Reconciling information) We consider the specification presented in Example 4, and the decentralized trace and two components: lswitch and bulb presented in Example 15. We recall the trace $\text{tr} = [1 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top\rangle\}, 1 \mapsto \text{bulb} \mapsto \{\langle \ell, \top\rangle\}, 2 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top\rangle\}, 2 \mapsto \text{bulb} \mapsto \{\langle \ell, \bot\rangle\}]$. Furthermore, we associate respectively two monitors m_0 and m_1 with components lswitch and bulb. We focus on the timestamp at $t = 2$. The monitors can observe the propositions s and ℓ respectively and use one EHE each: $\mathcal{I}^2_0$ and $\mathcal{I}^2_1$ respectively. Their memories are respectively $\mathcal{M}^2_0 = [\langle 1, \text{s}\rangle \mapsto \top, \langle 1, \text{s}\rangle \mapsto \top]$ and $\mathcal{M}^2_1 = [\langle 1, \ell\rangle \mapsto \top, \langle 2, \ell\rangle \mapsto \bot]$. Table 5.2 shows the EHEs (where $\mathcal{I}^2$ denotes the non-rewritten EHE). The columns $[\mathcal{M}^2_0]$ and $[\mathcal{M}^2_1]$ show the result of performing eval

Table 5.2: Reconciling information by combining EHEs.$\mathcal{I}^2$ indicates the non-rewritten EHE. The columns $[\mathcal{M}_0^2]$ and $[\mathcal{M}_1^2]$, the result of performing eval on the EHE $\dagger_\vee^2$ using memories $\mathcal{M}_0^2$ and $\mathcal{M}_1^2$ respectively. A dash (-) indicates the expression is the same as $\mathcal{I}^2$.

t	q	$\mathcal{I}^2$	$\mathcal{I}_0^2$	$\mathcal{I}_1^2$	$\dagger_\vee^2$	$[\mathcal{M}_0^2]$	$[\mathcal{M}_1^2]$
0	q_0	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
1	q_0	$\neg\langle 1, s\rangle$	$\bot$	-	-	$\bot$	?
1	q_1	$\langle 1, s\rangle$	$\top$	-	$\top$	$\top$	$\top$
2	q_0	$(\neg\langle 1, s\rangle \wedge \neg\langle 2, s\rangle) \vee (\langle 1, s\rangle \wedge \neg\langle 2, s\rangle)$	$\bot$	-	-	$\bot$	?
2	q_1	$(\langle 1, s\rangle \wedge \langle 2, s\rangle \wedge \langle 2, \ell\rangle) \vee (\neg\langle 1, s\rangle \wedge \langle 2, s\rangle))$	$\langle 2, \ell\rangle$	$\neg\langle 1, s\rangle \wedge \langle 2, s\rangle)$	$\langle 2, \ell\rangle \vee (\neg\langle 1, s\rangle \wedge \langle 2, s\rangle)$	?	?
2	q_2	$\langle 1, s\rangle \wedge \langle 2, s\rangle \wedge \neg\langle 2, \ell\rangle$	$\neg\langle 2, \ell\rangle$	$\langle 1, s\rangle \wedge \langle 2, s\rangle$	$\neg\langle 2, \ell\rangle \vee (\langle 1, s\rangle \wedge \langle 2, s\rangle)$	$\top$	$\top$

on the EHE $\dagger_\vee^2$ using memories $\mathcal{M}_0^2$ and $\mathcal{M}_1^2$ respectively.

Constructing the EHE $\mathcal{I}^2$ follows similarly from Example 17. We show the rewriting for both $\mathcal{I}_0^2$ and $\mathcal{I}_1^2$ respectively in the next two columns. Then, we show the result of combining the rewrites using $\dagger_\vee$. We notice initially that since s is $\bot$, m_0 could evaluate $\langle 1, s\rangle = \top$ and know that the automaton is in state q_1. However, for m_1, this is not possible until the expressions are combined. By evaluating the combination, m_1 determines that the automaton is in state q_0 at $t = 1$. We see at $t = 2$ for both q_1 and q_2 the expression resulting from combining the EHE is much weaker than the one present in each of the individual EHEs. After evaluating with the local memory, both monitors determine that the automaton is in state q_2.

In this case, we are only looking for expressions that evaluate to $\top$. We notice that monitor m_0 can determine that q_0 is not reachable (since $\neg\langle 1, s\rangle = \bot$) while m_1 cannot, as the expression $\neg\langle 1, s\rangle$ cannot yet be evaluated to a final verdict, and thus the combination evaluates to ?. This does not affect the outcome, as we are only looking for one expression that evaluates to $\top$, since both $\mathcal{I}_0^2$ and $\mathcal{I}_1^2$ are encoding the same execution. In the future, we would like to also propagate the information about the non-reachable states by tweaking the combination of EHEs. ∗

5.2 Data Structure Costs

We first consider the parameters and the cost for the basic functions of the EHE and *memory* data structures. We use s_E to denote the size necessary to encode an element of the set E. For example, s_{AP} is the size needed to encode an element of set AP. To do so, we first address the cost to store partial functions and merge them. Then, we parametrize the size of the EHE by accounting for delay introduced to relay partial observations. Finally, we present the size of both EHE and *memory* data structures.

5.2.1 Storing and Merging Partial Functions

Since memory and EHE are partial functions, to assess their required memory storage and iterations, we consider only the elements defined in the function.

Storing. The size of a partial function f, denoted $|f|$, is the size to encode all $x = f(x)$ mappings. We recall that $|\operatorname{dom}(f)|$ the number of entries in f. The size of each mapping $x = f(x)$ is the sum of the sizes $|x| + |f(x)|$. Therefore $|f| = \sum_{x \in \operatorname{dom}(f)} |x| + |f(x)|$.

Merging Merging two memories or two EHEs is linear in the size of both structures in both time and space. In fact, to construct $f \dagger_{\text{op}} g$, we first iterate over each $x \in \operatorname{dom}(f)$, check whether $x \in \operatorname{dom}(g)$, and if so assign

$$
\begin{array}{cccc}
 & t & \mapsto q & \mapsto \top \\
\delta_t & \left\{\begin{array}{lll}
 & q_0 & \mapsto e_{10} \\
 & q_1 & \mapsto e_{11} \\
t+1 & \mapsto & \vdots \\
 & q_{|Q|-1} & \mapsto e_{1(|Q|-1)}
\end{array}\right\} & |Q| \\
 & \left\{\begin{array}{lll}
 & q_0 & \mapsto e_{20} \\
t+2 & \mapsto & \vdots \\
 & q_{|Q|-1} & \mapsto e_{2(|Q|-1)}
\end{array}\right\} & |Q| \\
 & \vdots & \\
 & \left\{\begin{array}{lll}
 & q_0 & \mapsto e_{\delta_t 0} \\
 & q_1 & \mapsto e_{\delta_t 1} \\
t+\delta_t & \mapsto & \vdots \\
 & q_{|Q|-1} & \mapsto e_{\delta_t(|Q|-1)}
\end{array}\right\} & |Q|
\end{array}
$$

Figure 5.2: Size of the EHE (worst-case) with respect to information delay.

$\mathrm{op}(f(x), g(x))$, otherwise assign $f(x)$. Finally we assign $g(x)$ to any $x \in \mathrm{dom}(g) \cap \overline{\mathrm{dom}(f)}$. This results in $|\mathrm{dom}(f \dagger_{\mathrm{op}} g)| = |\mathrm{dom}(f) \cup \mathrm{dom}(g)|$ which is at most $|\mathrm{dom}(f)| + |\mathrm{dom}(g)|$.

5.2.2 Information delay

The main goal of the EHE data structure is to keep track of partial states of an automaton execution. Keeping track of partial states becomes unnecessary once enough information is gathered to determine which state was reached during an execution. An EHE associates an expression with a state for any given timestamp. When an expression e associated with a state q_{kn} for some timestamp t_{kn} is evaluated to $\top$, we know that the automaton is in q_{kn} at t_{kn}. We call q_{kn} a '*known*' (or stable) state. The information delay δ_t is the number of timestamps needed to reach a new known state from an existing known state. That is, it is the number of timestamps in the EHE storing partial information without determining a stable state. Information delay is a runtime measure, as it depends on the updates done to the EHE as it evolves through time. Given an EHE at timestamp t_{kn} such that q_{kn} is a known state for a given memory $\mathcal{M}^{t_{\mathrm{kn}}}$ i.e. $\mathrm{sel}(\mathcal{I}^{t_{\mathrm{kn}}}, \mathcal{M}^{t_{\mathrm{kn}}}, t_{\mathrm{kn}}) = q_{\mathrm{kn}}$. The next known timestamp is the least timestamp $t_{\mathrm{newkn}} > t_{\mathrm{kn}}$, such that $\mathrm{sel}(\mathcal{I}^{t_{\mathrm{newkn}}}, \mathcal{M}^{t_{\mathrm{newkn}}}, t_{\mathrm{newkn}}) \neq \mathrm{undef}$, where $\mathcal{I}^t$ and $\mathcal{M}^t$ at some timestamp t is used to denote respectively the changes to the EHE and memory through time in the execution[3]. The information delay for this evaluation of a state is $\delta_t = t_{\mathrm{newkn}} - t_{\mathrm{kn}}$. While information delay needs to be computed each time a stable state is reached, it is often the case that it is measured for a whole execution of an algorithm, in which case we can consider an average information delay and a maximum information delay, where we aggregate the various information delays (for reaching each known state) by computing their average and maximum. Since we know the automaton is in q_{newkn}, prior information is no longer necessary, therefore it is possible to discard all entries in $\mathcal{I}$ with $t < t_{\mathrm{newkn}}$. Thus, reducing the number of expressions in the EHE. This can be seen as a garbage collection strategy [WB86, SPBZ11] for the memory and EHE. We next show how the information delay parameter affects the size of the EHE.

5.2.3 EHE Encoding

For the EHE data structure, we consider the three functions: mov, eval, and sel[4] (see Section 5.1). Function mov depends on the topology of the automaton. We quantify it using the maximum size of the expression that labels a transition in a normalized automaton (see Remark 1) L, and the number of states in the automaton $|Q|$). From a known state each application of mov considers all possible transitions and states that can be respectively taken and reached, for each outbound transition, the label itself is added. Therefore, the rule is expanded by L per outbound state for each move beyond t_{kn}. We illustrate the expansion in Figure 5.2, where for each timestamp, we associated with each state an expression.

[3] We note that t_{newkn} is not necessarily equal to $t_{\mathrm{kn}} + 1$, as the EHE can determine a known state by simplification, and therefore skip intermediate states. We allow skipping states as it is reasonable for LTL_3 semantics, since final verdicts do not change for all suffixes.

[4] verAt is simply a sel followed by a $O(1)$ lookup

We use $S(n)$ to denote the size of an expression at a given timestamp after a known state. As such to reach a given state, we require a previous expression (i.e., $S(n-1)$), and add the label of a given transition (i.e., L). In the worst-case, the automaton is a fully connected graph, a state can be reached by all other states (including itself). Hence, we require the disjunction of $|Q|$ such expressions. The recurrence relation is given by: $S(n) = |Q|(S(n-1) + L)$. The size of the expression at the known timestamp is 1, as the expression $\top$. By summing all timestamps we have, $\sum_{i=0}^{n} |Q|^i L = L \sum_{i=0}^{n} |Q|^i$ where $\sum_{i=0}^{n} |Q|^i$ is a geometric series of ratio equal to $a > 1$. We can then deduce that the size of the expression is exponential in the number of timestamps, i.e. $S(n) = \Theta(|Q|^n L)$.

An EHE contains $\delta_t |Q|$ expressions, its' size is then (in the worst-case):

$$|\mathcal{I}^{\delta_t}| = \Theta(\delta_t \times |Q| \times |Q|^{\delta_t} L) = \Theta(|Q|^{\delta_t+1} \delta_t L).$$

For a given expression e, we use $|\mathsf{e}|$ to denote the size of e, i.e., the number of atoms in e. Given a memory $\mathcal{M}$, the complexity of function eval($\mathsf{e}, \mathcal{M}$) is the cost of simplify(rw($\mathsf{e}, \mathcal{M}$)). Function rw($\mathsf{e}, \mathcal{M}$) looks up each atom in e in $\mathcal{M}$ and attempts to replace it by its truth-value. The cost of a memory lookup is $\Theta(1)$, and the replacement is linear in the number of atoms in e. It effectively takes one pass to syntactically replace all atoms by their values, therefore the cost of rw is $\Theta(|\mathsf{e}|)$. However, applying function simplify() requires solving the Minimum Equivalent Expression problem which is Σ_2^p-complete [BU08], it is exponential in the size of the expression, making it the most costly function. $|\mathsf{e}|$ is bounded by $\delta_t L$. Function sel() requires evaluating every expression in the EHE. For each timestamp we need at most $|Q|$ expressions, and the number of timestamps is bounded by δ_t.

5.2.4 Memory

The memory required to store $\mathcal{M}$ depends on the trace, namely the amount of observations per component. Recall that once a state is known, observations can be removed, the number of timestamps is bounded by δ_t. The size of the memory is then:

$$\sum_{t=i}^{i+\delta_t} |\mathrm{tr}(c,t)| \times (\mathrm{s}_{\mathbb{N}} + \mathrm{s}_{AP} + \mathrm{s}_{\mathbb{B}_2}).$$

The size of the memory depends for each timestamp on the number of observations associated with the component ($|\mathrm{tr}(c,t)|$), and the size of each observation. The size of an observation is the size needed to encode the timestamp ($\mathrm{s}_{\mathbb{N}}$), the atomic proposition (s_{AP}) and the verdict ($\mathrm{s}_{\mathbb{B}_2}$).

Conclusion

We have introduced the execution history encoding (EHE) data structure. The EHE data structure is used to encode the possible states reached when executing an automaton given partial information. We elaborated on the properties of EHE, particularly on determinism, soundness, its ability to integrate the memory, and strong eventual consistency when used for decentralized monitoring of centralized specifications. We have introduced a cost model for the EHE that relies on information delay. The information delay represents the time needed to acquire sufficient partial information to infer the next state in the execution of the specification automaton. We show that the size of the EHE scales exponentially with the information delay, and linearly with the maximum length of the expressions labeling the specification automaton.

In the next chapter (Chapter 6), we shift the focus from centralized specifications to decentralized specifications. While the EHE property of SEC is useful when performing decentralized monitoring of centralized specifications, the EHE can be used as a general cost model when dealing with partial information, as using boolean expression simplification we make the most out of the partial information.

CHAPTER 6

Hierarchical Decentralized Specifications

Contents

Chapter abstract

In this chapter, we define hierarchical decentralized specifications. We define the basic structure of a decentralized specification as Moore automata similar to Section 2.1.2, but with added constraints on the atomic propositions, and references to other monitors. After a brief informal overview of decentralized specifications, we elaborate on the semantics to evaluate the references, and detail properties of monitorability and compatibility. *Monitorability* ensures that given a specification, monitors are able to eventually emit a verdict, for all possible traces. *Compatibility* ensures that a monitor topology can be deployed on a given system.

Introduction

Motivation. We noticed that so far, *decentralized monitoring* allows for several monitors that monitor the *same* specification. Typically, a decentralized monitoring algorithm will consider one (global) specification, and either generate necessary monitors that are tasked with monitoring a part of the specification, or allow for consensus among multiple monitors to find the global verdict (Chapter 3). We focus on multiple monitors each having their own independent specification, of which others are normally unaware. We thus consider *decentralized monitoring of decentralized specifications*. We note that centralized monitoring of decentralized specifications makes little sense as there does not exist more than one monitor. We noted that existing techniques often start from a global specification and then synthesize local monitors with either a copy of the global specification [BFRT16] or a completely different specification to monitor (typically a subformula of the original formula) [BF12, FCF14]. In this case, we would like to split the problem of generating equivalent decentralized specifications from a centralized one (synthesis) from the problem of monitoring. In addition, works on characterizing what one can monitor (i.e., monitorability [KVB+99, PZ06, FFM12a]) for centralized specifications exist [BLS11, FFM12a, DL14], but do not extend to decentralized specifications. For example, by splitting an LTL formula ad hoc, it is possible to obtain a non-monitorable subformula as we illustrate in Example 20. Obtaining a non-monitorable subformula interferes with the completeness of a monitoring algorithm.

Example 20 (Non-monitorable subformulae) We use the example from [CF16a] of the formula $\varphi \stackrel{\text{def}}{=} \mathbf{G}\,\mathbf{F}(a) \wedge \neg(\mathbf{G}\,\mathbf{F}(a))$, where $\mathbf{G}\,\mathbf{F}(a)$ means that a should hold infinitely often. One can see that φ is monitorable, as it can be simplified to $\bot$, since it has the form $\varphi' \wedge \neg\varphi'$. We notice that if we split φ into two subformulae $\mathbf{G}\,\mathbf{F}(a)$ and $\neg(\mathbf{G}\,\mathbf{F}(a))$, the obtained subformulae are non-monitorable. ∗

In addition to the separation of deploying monitors and the monitoring itself, using a large global specification to specify the whole behavior of the system can prove problematic when synthesizing monitors. We recall that the monitor synthesis algorithm presented in [BLS11] is doubly exponential in the length of the formula, and alphabet (atomic propositions) in a formula. We point out that since we are dealing with partial observations, we have to use atomic propositions and not events, as such the size of the alphabet grows sufficiently large, that for small systems, we often find it difficult to synthesize a monitor.

Example 21 (Limitation of synthesis) Let us consider a simple property that verifies a sensor-actuator system such as a switch and a light (as used in Example 2). One can verify that whenever a switch is pressed, the light must stay on until the switch is no longer pressed. We consider n to be the number of pairs of such switches and lights. The global property of the system is a large conjunction: $\bigwedge_{i=1}^{n} \mathbf{G}(\mathrm{s}_i \implies \mathbf{X}(\ell_i\,\mathbf{U}\,\neg\mathrm{s}_i))$. Within a reasonable limit of n (generally $n \geq 3$), existing synthesis tools will timeout when attempting to produce the synthesized monitor within a reasonable time frame (2 hours). ∗

Building specifications bottom up by composing existing specifications via references provides a modular way to build and hide subformulae (and atomic propositions) as they are composed. By referencing other subspecifications, we model explicitly the dependency relationships between various specifications. The generated hierarchy provides advantages to scalability and performance, on which we elaborate in Chapter 9.

Chapter organization. The chapter is split into three sections. In Section 6.1, we introduce an overview of decentralized specifications, their structure, and their semantics. In Section 6.2, we elaborate two properties (compatibility and monitorability) of decentralized specifications, and present algorithms to decide them. In Section 6.3, we discuss future perspectives, and problems introduced by decentralized specifications.

Key contributions. The key contributions of this chapter can be summarized as follows:

1. Introducing of decentralized specifications wherein each monitor has their own specification limited to what is observable for the monitor, and references to other monitors;
2. Elaborating on the semantics used to evaluate the references in specifications;

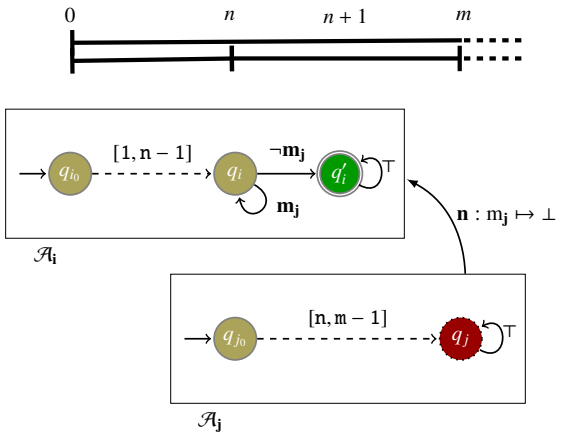


Figure 6.1: Evaluating Monitor References

3. Presenting a two step view of decentralized monitoring, which consists in the separation between topology of monitors and the monitoring;
4. Characterizing and computing two properties of decentralized specifications: *compatibility* of a specification to a target system, and *monitorability* of a given decentralized specification (extended from monitorability of centralized specifications); and
5. Discussing other properties of interest for future exploration tackling synthesis and equivalence.

6.1 Decentralized Specifications

In this section, we shift the focus to a specification that is decentralized. A set of automata represent various requirements (and dependencies) for different components of a system. In this section, we define the notion of a decentralized specification and its semantics, and in Section 6.2, we define various properties on such specifications.

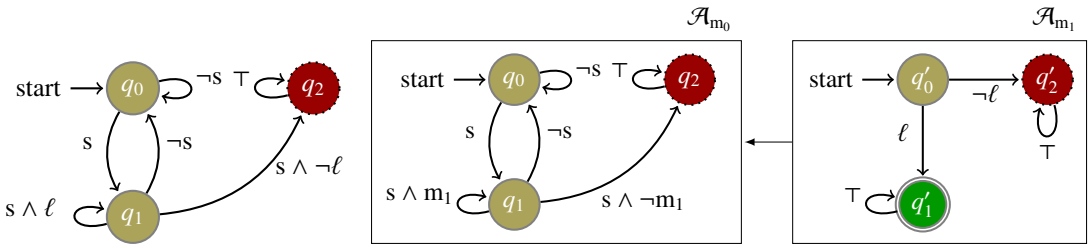
6.1.1 Informal Overview

Informally, a decentralized specification considers the system as a set of components, defines a set of LTL_3 monitors (see Section 2.1.2), additional atomic propositions that represent references to monitors, and attaches each monitor to a component. Attaching monitors to components allows a monitor specification to explicitly reference atomic propositions that are associated with the component. However, the transition labels in a monitor are restricted to only atomic propositions related to the component on which the monitor is attached, and references to other monitors.

A monitor reference is evaluated as if it were an oracle as shown in Figure 6.1. That is, to evaluate a monitor reference m_j, in a monitor $\mathcal{A}_i$, at a timestamp n, the monitor referenced ($\mathcal{A}_j$) is executed starting from the initial state by looking at observations in the trace starting at n. The atomic proposition m_j at n takes the value of the final verdict reached by the monitor $\mathcal{A}_j$ starting its evaluation from n. Details of the semantics are provided in Section 6.1. Furthermore, to evaluate reference we need the resulting oracle execution to be able to reach a final verdict, which is not always guaranteed. As such, it is important to define some of the properties of decentralized specifications such as *monitorability*, which indicates that a final verdict is co-reachable from any state in a given monitor (Section 6.2.1). We elaborate on characterizing and computing properties of decentralized specifications in Section 6.2.

6.1.2 Decentralizing a Specification

We recall that a decentralized system consists of a set of components $\mathcal{C}$. To decentralize the specification, instead of having one automaton, we have a set of specification automata (Definition 11) $\text{Mons} = \{\mathcal{A}_\ell \mid \ell \in AP_{\text{mons}}\}$,



(a) Centralized (b) Decentralized

Figure 6.2: Monitor(s) for the centralized and decentralized light switch and bulb specification presented in Example 4. The verdicts associated with the states are ⊥: dotted red , ⊤: double green, and ?: single yellow.

where AP_{mons} is a set of monitor labels. We refer to these automata as *monitors*. To each monitor, we associate a component using a function $\mathcal{L} : \text{Mons} \to C$. However, the transition labels of a monitor $\text{mon} \in \text{Mons}$ are expressions restricted to either observations local to the component the monitor is attached to (i.e., $\mathcal{L}(\text{mon})$), or references to other monitors. Transitions are labeled over $AP_{\text{mons}} \setminus \{\text{mon}\} \cup \{ap \in AP \mid \text{lu}(ap) = \mathcal{L}(\text{mon})\}$. This ensures that the monitor is labeled with observations it can locally observe or depend on other monitors. To evaluate a trace as one would on a centralized specification, we require one of the monitors to be a starting point, we refer to that monitor as the *root monitor* (rt ∈ Mons).

Definition 15 (Decentralized specification) A decentralized specification is a tuple $\langle AP_{\text{mons}}, \text{Mons}, C, \mathcal{L}, \text{rt}\rangle$, where AP_{mons} is the set of monitor references, Mons is the set of monitors, C is the set of components, $\mathcal{L}$ is a function assigning monitors to components, and rt is the root monitor.

We note that a centralized specification is a special case of a decentralized specification, with one component (global system, sys), and one monitor (g) attached to the sole component, i.e. $\langle\{\text{g}\}, \{\mathcal{A}_\text{g}\}, \{\text{sys}\}, [\mathcal{A}_\text{g} \mapsto \text{sys}], \mathcal{A}_\text{g}\rangle$.

As automata expressions now include references to monitors, we first define function dep : $Expr \to 2^{\text{Mons}}$, which determines monitor dependencies in a given expression. Then, we define the semantics of evaluating (decentralized) specifications with references.

Definition 16 (Monitor dependency) The set of monitor dependencies in an expression `e` is obtained by function dep : $Expr \to 2^{\text{Mons}}$, defined as[1]: dep(e) = `match` e `with`:

$\mid id \in AP_{\text{mons}}$	$\to \{\mathcal{A}_{id}\}$	$\mid e_1 \wedge e_2$	$\to \text{dep}(e_1) \cup \text{dep}(e_2)$
$\mid \neg e$	$\to \text{dep}(e)$	$\mid e_1 \vee e_2$	$\to \text{dep}(e_1) \cup \text{dep}(e_2)$

Function dep finds all monitors referenced by expression `e`, by syntactically traversing it.

Example 22 (Decentralized specification) Figure 6.2b shows a decentralized light switch and bulb specification corresponding to the centralized specification in Example 4 (shown in Figure 6.2a for side-by-side comparison). We recall from Example 15 that the system consists of two components the light switch and bulb, labeled lswitch and bulb, respectively. We associated the components lswitch and bulb with the monitors $\mathcal{A}_{\text{m}_0}$ and $\mathcal{A}_{\text{m}_1}$, respectively. We use $\mathcal{A}_{\text{m}_0}$ as the root monitor for the decentralized specification. We consider the two atomic propositions s and ℓ can only be observed by component lswitch and bulb respectively. $\mathcal{A}_{\text{m}_0}$ depends on the verdict from m_1 and only observations local to lswitch, while $\mathcal{A}_{\text{m}_1}$ is only labeled with observations local to bulb. Given the expression $\text{s} \wedge \text{m}_1$, we have $\text{dep}(\text{s} \wedge \text{m}_1) = \{\mathcal{A}_{\text{m}_1}\}$. ∗

6.1.3 Semantics of a Decentralized Specification

The transition function of the decentralized specification is similar to the centralized automaton with the exception of monitor ids.

Definition 17 (Semantics of a decentralized specification) Consider the root monitor $\mathcal{A}_{\text{rt}}$ and a decentralized trace tr with index $i \in [1, |\text{tr}|]$ representing the timestamps. Monitoring tr starting from $\mathcal{A}_{\text{rt}}$ emits the verdict $\text{ver}_{\text{rt}}(\Delta'^{*}_{\text{rt}}(q_{\text{rt}_0}, \text{tr}, 1))$ where for a given monitor label ℓ:

$$\Delta'^{*}_{\ell}(q, \text{tr}, i) = \begin{cases} \Delta'^{*}_{\ell}(\Delta'_{\ell}(q, \text{tr}, i), \text{tr}, i+1) & \text{if } i < |\text{tr}| \\ \Delta'_{\ell}(q, \text{tr}, i) & \text{otherwise} \end{cases}$$

$$\Delta'_{\ell}(q, \text{tr}, i) = \begin{cases} q' & \text{if } \text{tr}(i, \mathcal{L}(\mathcal{A}_\ell)) \neq \emptyset \wedge \exists e \in Expr_{AP} : \\ & \quad \delta_\ell(q, e) = q' \wedge \text{eval}(e, \mathcal{M}) = \top \\ q & \text{otherwise} \end{cases}$$

$$\text{where} \quad \mathcal{M} = \text{memc}(\text{tr}(i, \mathcal{L}(\mathcal{A}_\ell)), \text{idt}) \dagger_2 \biguplus^{2}_{\mathcal{A}_{\ell'} \in \text{dep}(e)} \{[\ell' \mapsto \text{ver}_{\ell'}(q_{\ell'_f})]\}$$

$$\text{and} \quad q_{\ell'_f} = \Delta'^{*}_{\ell'}(q_{\ell'_0}, \text{tr}, i)$$

For a monitor $\mathcal{A}_\ell$, we determine the new state of the automaton starting at $q \in Q_\ell$, and running the trace tr from timestamp i to timestamp t by applying $\Delta'^{*}_{\ell}(q, \text{tr}, i)$. To do so, we evaluate one transition at a time using Δ'_ℓ as would Δ^{*}_{ℓ} with Δ_ℓ (see Definition 12). To evaluate Δ'_ℓ at any state $q \in Q_\ell$, we need to evaluate the expressions so as to determine the next state q'. The expressions contain atomic propositions and monitor ids. For atomic propositions, the memory is constructed using $\text{memc}(\text{tr}(i, \mathcal{L}(\mathcal{A}_\ell)), \text{idt})$ which is based on the event with observations local to the component the monitor is attached to (i.e., $\mathcal{L}(\mathcal{A}_\ell)$). However, for monitor ids, the memory represents the verdicts of the monitors. To evaluate each reference ℓ' in the expression, the remainder of the trace starting from the current event timestamp i is evaluated recursively on the automaton $\mathcal{A}_{\ell'}$ from the initial state $q_{\ell'_0} \in \mathcal{A}_{\ell'}$. Then, the verdict of the monitor is associated with ℓ' in the memory.

Example 23 (Monitoring of a decentralized specification) We consider the decentralized specification from Example 22. We have the monitors $\mathcal{A}_{\text{m}_0}$ (root) and $\mathcal{A}_{\text{m}_1}$ associated to components lswitch and bulb respectively. Furthermore, we consider the decentralized trace from Example 15: $\text{tr} = [1 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top \rangle\}, 1 \mapsto \text{bulb} \mapsto \{\langle \ell, \top \rangle\}, 2 \mapsto \text{lswitch} \mapsto \{\langle \text{s}, \top \rangle\}, 2 \mapsto \text{bulb} \mapsto \{\langle \ell, \bot \rangle\}]$.

To evaluate tr on $\mathcal{A}_{\text{m}_0}$ (from Figure 6.2b), we use $\Delta'^{*}_{\text{m}_0}(q_0, \text{tr}, 1)$. To do so, we first evaluate $\Delta'_{\text{m}_0}(q_0, \text{tr}, 1)$. In this case, the expressions only depend on the atomic proposition s, which does not depend on any other monitor. We have $\mathcal{M}^1_{\text{m}_0} = \text{memc}(\langle \text{s}, \top \rangle, \text{idt}) = [\text{s} \mapsto \top]$, and $\text{eval}(\text{s}, \mathcal{M}^1_{\text{m}_0}) = \top$. Thus, we obtain $\Delta'_{\text{m}_0}(q_0, \text{tr}, 1) = q_1$.

In q_1 at $t = 2$, we now evaluate $\Delta'_{\text{m}_0}(q_1, \text{tr}, 2)$. Transitions from q_1 are labeled with expressions that depend on m_1. Therefore, we evaluate the decentralized trace on $\mathcal{A}_{\text{m}_1}$ starting at $t = 2$ by evaluating $\Delta'^{*}_{\text{m}_1}(q'_0, \text{tr}, 2)$. We start by evaluating $\Delta'_{\text{m}_1}(q'_0, \text{tr}, 2)$. We have $\mathcal{M}^2_{\text{m}_1} = \text{memc}(\langle \ell, \bot \rangle, \text{idt}) = [\ell \mapsto \bot]$, and $\text{eval}(\neg\ell, \mathcal{M}^2_{\text{m}_1}) = \top$. Thus, we obtain $\Delta'_{\text{m}_1}(q'_0, \text{tr}, 2) = q'_2$ labeled by the verdict $\bot$. Having reached a final verdict for m_1, we can construct the memory for m_0. We have $\mathcal{M}^2_{\text{m}_0} = \text{memc}(\langle \text{s}, \top \rangle, \text{idt}) \dagger_2 [\text{m}_1 \mapsto \bot] = [\text{s} \mapsto \top, \text{m}_1 \mapsto \bot]$. Knowing that $\text{eval}(\text{s} \wedge \neg \text{m}_1, \mathcal{M}^2_{\text{m}_0}) = \top$, we conclude that the next state is $\Delta'_{\text{m}_0}(q_1, \text{tr}, 2) = q_2$. Since q_2 is labeled by verdict $\bot$, the monitoring concludes and we detect a violation of the specification. *

6.2 Properties for Decentralized Specifications

A key advantage of using decentralized specifications is to make the association of monitors with components explicit. Since monitors have been explicitly modeled as a set of automata with dependencies between each other, we can now determine properties on decentralized specifications. In this section, we revisit the concept of *monitorability*, characterize it for automata, define it for decentralized specifications, and describe an algorithm for deciding monitorability. Furthermore, we explore *compatibility*, that is the ability of a decentralized specification to be deployed on a given architecture.

Figure 6.3: A trivial nonmonitorable specification

6.2.1 Decentralized Monitorability

An important notion to consider when dealing with runtime verification is that of monitorability [PZ06, Fal10]. In brief, monitorability of a specification determines whether or not an RV technique is applicable to a specification. That is, a monitor synthesized for a non-monitorable specification is unable to check if the execution complies or violates the specification for all possible traces. Consider the automaton shown in Figure 6.3, one could see that there is no state labeled with a final verdict. In this case, we can trivially see that no trace allows us to reach a final verdict. We also notice similar behavior when monitoring LTL expressions with the pattern $\mathbf{GF}(p)$ with p is an atomic proposition. The LTL expression requires that at all times $\mathbf{F}(p)$ holds $\top$, while $\mathbf{F}(p)$ requires that p eventually holds $\top$. As such, at any given point of time, we are unable to determine a verdict, since if p is $\bot$ at the current timestamp, it can still be $\top$ at a future timestamp, and thus $\mathbf{F}(p)$ will be $\top$ for the current timestamp. And if $\mathbf{F}(p)$ is $\top$ at the current timestamp, the $\mathbf{G}$ requires that it be $\top$ for all timestamps, so in the future there could exist a timestamp which falsifies it. Consequently, when monitoring such an expression, a monitor will always output ?, as it cannot determine a verdict for any given timestamp. In this section, we first characterize monitorability in terms of automata and EHE for both centralized and decentralized specifications. Then, we provide an effective algorithm to determine monitorability.

Characterizing Monitorability

Centralized monitorability of properties Monitorability in the sense of [PZ06] is defined on traces. A property is monitorable if for all finite traces t (a sequence of events) in the set of all (possibly infinite) traces, there exists a continuation t' such that monitoring $t \cdot t'$ results in a true or false verdict. Informally, it can be seen as whether or not continuing to monitor the property after reading t can still yield a final verdict. We note that this definition deals with all possible traces, it establishes monitorability to be oblivious of the input trace.

Centralized monitorability in automata We express monitorability to reach "true" or "false" verdict to the notion of reaching a *final verdict*, and associate it with automata. For automata, monitorability can be analyzed in terms of reachability and states.

> **Definition 18 (Monitorability of an automaton.)** Given a automaton $\mathcal{A} = \langle Q, q_0 \in Q, \delta, \text{ver}\rangle$, a state $q \in Q$ is monitorable (noted monitorable(q)) iff $\text{ver}(q') \in \mathbb{B}_2$ or $\exists q' \in Q$ such that $\text{ver}(q') \in \mathbb{B}_2$ and q' is reachable from q. Automaton $\mathcal{A}$ is said to be monitorable (noted monitorable($\mathcal{A}$)) iff $\forall q \in Q :$ monitorable(q).

Defining monitorability using reachability is consistent with [PZ06]. After reading a finite trace t and reaching q ($q = \Delta^*(q_0, t)$), there exists a continuation t' that leads the automaton to a state q' ($q' = \Delta^*(q, t')$), such that $\text{ver}(q') \in \mathbb{B}_2$. We note that an automaton is monitorable according to this definition iff, in the automaton, all paths from the initial state q_0 lead to a state with a final verdict. As such, it is sufficient to analyze the automaton to determine monitorability irrespective of possible traces (see Section 6.2.1)[2]. We illustrate monitorability of automata in Example 24.

Example 24 (Centralized monitorability of automata.) Figure 6.2a illustrates the automaton that expresses the light switch and bulb specification. It is monitorable, as the states q_0, q_1, and q_2 are monitorable. For both q_0 and q_1, it is possible to reach q_2 labeled with the final verdict $\bot$. We note that monitorability is a necessary but not sufficient condition for termination (with a final verdict). An infinite trace consisting of repetitions of the event $\{\neg\langle s, \bot\rangle, \langle \ell, \bot\rangle\}$ never lets the automaton reach q_2. However, monitorability guarantees the possibility of reaching a final verdict. If a state q is not monitorable, we know that it is impossible to reach a final verdict from q, and can abandon monitoring. ∗

[2]The expressions leading to q' must all be also satisfiable. However, satisfiability is guaranteed as our automaton is normalized, see Remark 1.

Centralized monitorability with EHE Reachability in automata can be expressed as well using the EHE data structure. A path from a state q to a state q' is expressed as an expression over atoms. We define paths(q, q') to return all possible paths from q to q'.

$$\text{paths}(q, q') = \{e \in Expr \mid \exists t \in \mathbb{N} : \mathcal{I}^t(t, q') = e \wedge \ \mathcal{I}^t = \text{mov}([0 \mapsto q \mapsto \top], 0, t)\}$$

Each expression is derived similarly as would an execution in the EHE (Definition 13). We start from state q and use a logical timestamp starting at 0 incrementing it by 1 for the next reachable state. A state q is monitorable iff $\exists e_f \in \text{paths}(q, q_f)$, such that (1) e_f is satisfiable; (2) ver(simplify$(e_f)) \in \mathbb{B}_2$. The first condition ensures that the path is able to lead to the state q_f, as an unsatisfiable path will never evaluate to true. The second condition ensures that the state is labeled by a final verdict. An automaton is thus monitorable iff all its states are monitorable. We note that paths(q, q') can be infinite if the automaton contains cycles, however path expressions could be "compacted" using the pumping lemma. Using EHE we can frame monitorability as a satisfiability problem which can benefit from additional knowledge on the truth values of atomic propositions. For the scope of this paper, we focus on computing monitorability on automata in Section 6.2.1.

Decentralized monitorability In the decentralized setting, we have a set of monitors Mons. The labels of automata include monitor ids (AP_{mons}). We recall that the evaluation of a reference $\ell \in AP_{\text{mons}}$ consists in running the remainder of the trace on $\mathcal{A}_\ell$ starting from the initial state q_{ℓ_0}. As such, for any dependency on a monitor $\mathcal{A}_\ell$, we know that ℓ evaluates to a final verdict iff monitorable$(\mathcal{A}_\ell)$. We notice that monitorability of decentralized specification is recursive, and relies on the inter-dependencies between the various decentralized specifications. This is straightforward for EHE, since a path is an expression. For a path e_f, the dependent monitors are captured in the set dep(e_f). The additional condition on the path is thus: $\forall \mathcal{A}_\ell \in \text{dep}(e_f) :$ monitorable$(\mathcal{A}_\ell)$.

Computing Monitorability

Centralized specification We compute the monitorability of a centralized specification $\mathcal{A}$, with respect to a set of final verdicts $\mathbb{B}_2$[3]. We denote monitorability by monitorable$(\mathcal{A}, \mathbb{B}_2)$. In the remainder of the thesis we always use $\mathbb{B}_2$, thus, we write monitorable$(\mathcal{A})$. Computing monitorability consists in checking that all states of the automaton are co-reachable from states with final verdicts. As such, it relies on a traversal of the graph starting from the states that are labeled with final verdicts. To do so, we use a variation of the work-list algorithm. We begin by adding all states labeled by a final verdict to the work list. These states are trivially monitorable. Conversely, any state that leads to a monitorable state is monitorable. As such, for each element in the work list, we add its predecessors to the work list. We maintain a set of marked states (Mark), that is, states that have already been processed, so as to avoid adding them again. This ensures that cycles are properly handled. The algorithm stabilizes when no further states can be processed (i.e., the work list is empty). All marked states (Mark) are therefore monitorable. To check if an automaton is monitorable, we need all of its states to be monitorable. As such we verify that |Mark| = $|Q|$. The number of edges between any pair of states can be rewritten to be at most 1 (as explained in Section 4.4). As such, one has to traverse the graph once, the complexity being linear in the states and edges (i.e., $O(|Q| + |\delta|)$). Hence in the worst case, an automaton forms a complete graph, and we have $\binom{|Q|}{2}$ edges. The worst case complexity is quadratic in the number of states (i.e., $O(|Q| + \frac{1}{2}|Q|(|Q| - 1))$).

Decentralized specifications In the case of decentralized specifications, the evaluation of paths (using eval) in an automaton depends on other monitors (and thus other automata). To compute monitorability, we first build the monitor dependency set for a given monitor $\mathcal{A}_\ell$ (noted MDS$(\mathcal{A}_\ell)$) associated with a monitor label ℓ.

$$\text{MDS}(\mathcal{A}_\ell) = \biguplus_{\{e \in Expr \mid \exists q, q' \in Q_\ell :\, \delta_\ell(q,e) = q'\}} \text{dep}(e)$$

The monitor dependency list for a monitor contains all the references to other monitors across all paths in the given automaton ($\mathcal{A}_\ell$), by examining all the transitions. It can be obtained by a simple traversal of the automaton.

Second, we construct the monitor dependency graph (MDG), which describes the dependencies between monitors. The monitor dependency graph for a set of monitors Mons is noted MDG(Mons) = $\langle$Mons, DE$\rangle$ where DE is the set

[3] While we use $\mathbb{B}_2$, this can be extended without loss of generality to an arbitrary set $\mathbb{B}_f$.

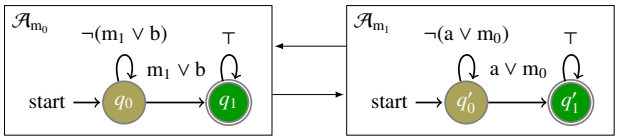


Figure 6.4: A monitorable decentralized specification with cyclically dependent monitors. When observing $\langle a, \top \rangle$, and $\langle b, \top \rangle$, the disjunction cancels out the dependency.

of edges which denotes the dependency edges between the monitors, defined as: $DE = \{\langle \mathcal{A}_\ell, \mathcal{A}_{\ell'} \rangle \in Mons \times Mons \mid \mathcal{A}_{\ell'} \in MDS(\mathcal{A}_\ell)\}$. A monitor $\mathcal{A}_{m_i}$ depends on another monitor $\mathcal{A}_{m_j}$ iff m_j appears in the expressions on the transitions of $\mathcal{A}_{m_i}$.

Proposition 5 (Sufficient conditions for monitorability of decentralized specifications.) A decentralized specification is monitorable if the two following conditions are met: (i) MDG(Mons) has no cycles; and (ii) $\forall \ell \in Mons :$ monitorable$(\mathcal{A}_\ell)$.

The first condition ensures that no cyclical dependency exists between monitors. The second condition ensures that all monitors are individually monitorable. We note, that both conditions are decidable. Furthermore, detecting cycles in a graph can be done in linear time with respect to the sum of nodes and edges, by doing a depth-first traversal with back-edge detection, or by finding strongly connected components [Tar72]. Thus, in worst case, it is quadratic in |Mons|. Monitorability is therefore quadratic in the number of monitors and states in the largest automaton.

We illustrate the monitorability of the decentralized light switch and bulb specification in Example 25.

Example 25 (Decentralized monitorability of decentralized specifications.) We consider the decentralized counterpart of the light switch and bulb presented in Example 24. The decentralized specification is shown in Figure 6.2b, it introduces two monitors $\mathcal{A}_{m_0}$ and $\mathcal{A}_{m_1}$. The set of monitors is $Mons \stackrel{\text{def}}{=} \{\mathcal{A}_{m_0}, \mathcal{A}_{m_1}\}$.

We compute the monitor dependency sets for each monitor. We have $MDS(\mathcal{A}_{m_0}) = dep(\top) \cup dep(s) \cup dep(\neg s) \cup dep(s \wedge m_1) \cup dep(s \wedge \neg m_1) = \{\mathcal{A}_{m_1}\}$, and $MDS(\mathcal{A}_{m_1}) = dep(\top) \cup dep(\ell) \cup dep(\neg \ell) = \emptyset$. Using the monitor dependency sets, we construct the monitor dependency graph: $MDG(Mons) = \langle Mons, \{\langle \mathcal{A}_{m_0}, \mathcal{A}_{m_1} \rangle\} \rangle$. The monitor dependency graph has no cycles, as it contains only one edge indicating the dependency of $\mathcal{A}_{m_0}$ on $\mathcal{A}_{m_1}$.

We now verify the monitorability of each monitor separately using centralized monitorability. Both $\mathcal{A}_{m_0}$ and $\mathcal{A}_{m_1}$ are monitorable as the states q_2 and q'_1 or q'_2 are reachable from all states. *

The requirement for no cycles is sufficient but not necessary, it is possible for certain cycles to exist while the decentralized specification is still able to reach a final verdict. This is because boolean expressions may cancel out the dependency, or dependencies can be on different timestamps (i.e., future transitions in the automaton). We illustrate a monitorable decentralized specification in *Figure* 6.4 with two monitors that depend on each other. Regardless of the choice of the root monitor, it is possible to still avoid the dependency if one operands of the disjunction holds true. That is, if we observe $\langle a, \top \rangle$ then it is no longer necessary to evaluate m_0, and therefore no real dependency exists.

6.2.2 Compatibility

A key advantage of decentralized specifications is the ability to associate monitors to components. This allows us to associate the monitor network with the actual system architecture constraints.

The monitor network is a graph $N = \langle Mons, E \rangle$, where Mons is the set of monitors, and E representing the communication edges between monitors. The monitor network is typically generated by a monitoring algorithm during its *setup* phase (See Section 8.1). For example, N could be obtained using the construction MDG(Mons) presented in Section 6.2.1. The system is represented as another graph $S = \langle C, E' \rangle$, where C is the set of components, and E' is the set of communication channels between components.

Defining compatibility We now consider checking for *compatibility*. Compatibility denotes whether a monitoring network can be actually deployed on the system. That is, it ensures that communication between monitors is possible when those are deployed on the components. We first consider the reachability in both the system and monitor graphs as the relations $\text{reach}_S : C \to 2^C$, and $\text{reach}_M : \text{Mons} \to 2^{\text{Mons}}$, respectively. Second, we recall that a monitor may depend on other monitors and also on observations local to a component. If a monitor depends on local observations, then it provides us with constraints on where it should be placed. We identify those constraints using the partial function $\text{cdep} : \text{Mons} \to C$. We can now formally define compatibility. Compatibility is the problem of deciding whether or not there exists a *compatible assignment*.

Definition 19 (Compatible assignment) A compatible assignment is a function $\text{compat} : \text{Mons} \to C$ that assigns monitors to components while preserving the following properties:

1) $\forall m_1, m_2 \in \text{Mons} : m_2 \in \text{reach}_M(m_1) \implies \text{compat}(m_2) \in \text{reach}_S(\text{compat}(m_1))$;
2) $\forall m \in \text{dom}(\text{cdep}) : \text{cdep}(m) = \text{compat}(m)$.

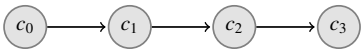
The first condition ensures that reachability is preserved. That is, it ensures that if a monitor m_1 communicates with another monitor m_2 (i.e. $m_2 \in \text{reach}_M(m_1)$), then m_2 must be placed on a component reachable from where m_1 is placed (i.e. $\text{compat}(m_2) \in \text{reach}_S(\text{compat}(m_1))$). The second condition ensures that dependencies on local observations are preserved. That is, if a monitor m depends on local observations from a component $c \in C$ (i.e. $\text{cdep}(m) = c$), then m must be placed on c (i.e. $\text{cdep}(m) = \text{compat}(m)$).

Computing compatibility We next consider the problem of finding a *compatible assignment* of monitors to components. Algorithm 1 finds a compatible assignment for a given monitor network ($\langle \text{Mons}, E \rangle$), system ($\langle C, E' \rangle$), and an initial assignment of monitors to components (cdep). The algorithm can be broken into three procedures: procedure verifyCompatible verifies that a (partial) assignment of monitors to components is compatible, procedure compatibleProc takes as input a set of monitors that need to be assigned and explores the search space (by iterating over components), and finally, procedure compatible performs necessary pre-computation of reachability, verifies that the constraint is first compatible, and starts the search.

We verify that an assignment of monitors to components ($s : \text{Mons} \to C$) is compatible using algorithm verifyCompatible (Lines 1-8). We consider each assigned monitor ($m \in \text{dom}(s)$). Then, we constrain the set of reachable monitors from m to those which have been assigned a component ($M' = \text{reach}_M(m) \cap \text{dom}(s)$). Using M', we construct a new set of components using s (i.e., $C' = \{s(m') \in C \mid m' \in M'\}$). Set C' represents the components on which reachable monitors have been placed. Finally, we verify that the components in the set C' are reachable from where we placed m (i.e., $C' \subseteq \text{reach}_S(s(m))$). If that is not the case, then the assignment is not compatible (Line 4). To iterate over all the search space, that is, all possible assignments of monitors to components, procedure compatibleProc (Lines 9-24) considers a set of monitors to assign (M), selects a monitor $m \in M$ (Line 13), and iterates over all possible components, verifying that the assignment is compatible (Lines 14-22). If the assignment is compatible, it iterates over the remainder of the monitors (i.e., $M \setminus \{m\}$), until it is empty (Line 16). If the assignment is not compatible, it discards it and proceeds with another component. For each monitor we seek to find at least one compatible assignment. One can see that the procedure eventually halts (as we exhaust all the monitors to assign), and is affected exponentially based on the number of monitors to assign $|\text{Mons} \setminus \text{dom}(\text{cdep})|$ (Line 31) with a branching factor determined by the possible values to assign ($|C|$, Line 14). It is important to note that the number of monitors to assign is in practice particularly small. The number of monitors to assign includes monitors that depend only on other monitors and not local observations from components, as the dependency on local observations requires that a monitor be placed on a given component (that is, it will be in dom(cdep)).

Example 26 (Compatibility) Figure 6.5 presents a simple monitor network of 3 monitors, and a system graph of 4 components. We consider the following constraint: $\text{cdep} = [m_0 \mapsto c_0, m_2 \mapsto c_2]$. For compatibility, we must first verify that cdep is indeed a compatible (partial) assignment, then consider placing m_1 on any of the components (i.e., both properties of Definition 19). Procedure compatible computes the set of reachable nodes for both the monitor network and the system. They are presented in Figure 6.5c and Figure 6.5d, respectively. We then proceed with line 28 to verify the constraint (cdep) using procedure verifyCompatible. We consider both m_0 and m_2. For m_0 (resp. m_2) we generate the set (Line 3) $\{c_0\}$ (resp. $\{c_2\}$), and verify that it is indeed a subset of $\text{reach}_S(c_0)$ (resp. $\text{reach}_S(c_2)$). This ensures that the constraint is compatible.

(a) Monitor Network



(b) System

m_0	$\{m_0, m_1\}$
m_1	$\{m_1\}$
m_2	$\{m_2, m_1\}$

(c) reach_{M}

c_0	$\{c_0, c_1, c_2, c_3\}$
c_1	$\{c_1, c_2, c_3\}$
c_2	$\{c_2, c_3\}$
c_3	$\{c_3\}$

(d) reach_{S}

Figure 6.5: Example Compatibility

We then proceed to place m_1 by calling compatibleProc(cdep, $\{m_1\}$, $\{c_0, c_1, c_2, c_3\}$, reach_{M}, reach_{S}). While procedure compatibleProc attempts all components, we consider for the example placing m_1 on c_1. On line 15, partial function s' is cdep $\dagger_2$ $[m_1 \mapsto c_1]$. We now call verifyCompatible to verify s'. We consider the assigned monitors m_0, m_1, and m_2. For m_0 (resp. m_1, m_2) we generate the set $\{c_0, c_1\}$ (resp. $\{c_1\}$, $\{c_2, c_1\}$). We notice that for m_0, $\{c_0, c_1\}$ is indeed a subset of $\text{reach}_{\text{S}}(c_0)$. This means that m_0 is able to communicate with m_1. However, it is not the case for m_2, the set $\{c_2, c_1\}$ is not a subset of $\text{reach}_{\text{S}}(c_2) = \{c_2, c_3\}$. The monitor m_2 will not be able to communicate with m_1 if m_1 is placed on c_1. Therefore, assigning m_1 to c_1 is incompatible. Example of compatible assignments for m_1 are c_2 and c_3 as both of those components are reachable from c_2.

Procedure compatibleProc continues by checking other components, and upon reaching c_2 or c_3 stops and returns that there is at least one compatible assignment. Therefore, the monitor network (Figure 6.5a) is compatible with the system (Figure 6.5b). ∗

6.3 Future Extensions

By introducing decentralized specifications, we separate the monitor topology from the monitoring algorithm. As such, we address future directions that result from analyzing the topology of monitors, and thus, define properties on such topologies, studying the monitoring by improving metrics, and applying decentralized specifications to the problem of runtime enforcement [Fal10].

Optimized compatibility The first direction is to extend the notion of *compatibility* (Section 6.2.2) to not only decide whether or not a specification is applicable to the architecture of the system, but also use the architecture to optimize the placement. That is, one can generate a decentralized specification that balances computation to suit the system architecture, or optimize specific algorithms for specific layouts of decentralized systems.

Verdict equivalence We can also compare decentralized specifications to ensure that two specifications emit the same verdict for all possible traces, we elaborate on this property as *verdict equivalence*. We consider two decentralized specifications $\mathcal{D}$ and $\mathcal{D}'$, constructed with two sets of monitors Mons and Mons$'$ (as per Section 6.1). Let the root monitors be rt, and rt$'$, respectively. We recall the notation from Section 4.4, for a given monitor label ℓ, q_{ℓ_0}, Δ_ℓ and ver_ℓ indicate the initial state, transition relation and the verdict function for a given monitor (automaton). One way to assess equivalence is to verify, that for all traces, both specifications yield similar verdicts. It suffices to evaluate the trace on the transition function starting from the root monitor, and check the verdict of the reached state. That is, two decentralized specifications $\mathcal{D}$ and $\mathcal{D}'$ are *verdict equivalent* iff $\forall t \in \mathcal{T} : \text{ver}_{\text{rt}}(\Delta'^{*}_{\text{rt}}(q_{\text{rt}_0}, t, 1)) = \text{ver}_{\text{rt}'}(\Delta'^{*}_{\text{rt}'}(q_{\text{rt}'_0}, t, 1))$. The verdict equivalence property establishes the basis for comparing two specifications that eventually output the same verdicts for the same traces. For all possible traces ($\forall t \in \mathcal{T}$), we first evaluate the trace on the root monitor of $\mathcal{D}$ (i.e., $q_{\text{f}} = \Delta'^{*}_{\text{rt}}(q_{\text{rt}_0}, t)$), and similarly we evaluate the same trace on the root monitor of $\mathcal{D}'$ (i.e., $q'_{\text{f}} = \Delta'^{*}_{\text{rt}'}(q_{\text{rt}'_0}, t)$). The states q_{f} and q'_{f} reached respectively for each decentralized specification $\mathcal{D}$ and $\mathcal{D}'$ need to be labeled by the same verdict. While both specifications yield the same verdict for a given trace, one could also extend this formulation to add bounds on delay.

Specification synthesis Another interesting problem to explore is that of *specification synthesis*. Specification synthesis considers the problem of generating a decentralized specification, using various inputs. Typically, we

Algorithm 1 Computing Compatibility

```
1:  procedure verifyCompatible(s, reach_M, reach_S)                          ▷ Verify assignment s
2:      for each m ∈ dom(s) do                                               ▷ Consider only assigned monitors
3:          if {s(m') | m' ∈ (reach_M(m) ∩ dom(s))} ⊈ reach_S(s(m)) then     ▷ Check reachability
4:              return false
5:          end if
6:      end for
7:      return true
8:  end procedure
9:  procedure compatibleProc(s, M, C , reach_M, reach_S)                     ▷ Explore assignments
10:     if M = ∅ then                                                        ▷ No monitors left to assign
11:         return ⟨true, s⟩                                                 ▷ Successfully assigned all monitors
12:     end if
13:     m ← pick(M)                                                          ▷ Pick a monitor from those left to assign
14:     for each c ∈ C do                                                    ▷ Explore assigning monitor to all possible components
15:         s' ← s †_2 [m ↦ c]                                               ▷ Add assignment to the existing solution
16:         if verifyCompatible(s', reach_M, reach_S) then                   ▷ Is it compatible?
17:             ⟨res, sol⟩ ← compatibleProc(s', M\{m}, C, reach_M, reach_S)  ▷ Recurse on the rest
18:             if res then                                                  ▷ Found a compatible assignment for all the rest of M
19:                 return ⟨res, sol⟩
20:             end if
21:         end if
22:     end for
23:     return ⟨false, []⟩                                                   ▷ No compatible assignment found
24: end procedure
25: procedure compatible(⟨Mons, E⟩, ⟨C, E'⟩, cdep)
26:     reach_M ← computeReach(⟨Mons, E⟩)                                    ▷ Precompute reachability
27:     reach_S ← computeReach(⟨C, E'⟩)
28:     if ¬verifyCompatible(cdep, reach_M, reach_S) then                    ▷ Check constraint first
29:         return ⟨false, []⟩                                               ▷ Constraint not satisfied
30:     end if
31:     return compatibleProc(cdep, Mons \ dom(cdep), C, reach_M, reach_S)   ▷ Begin exploring
32: end procedure
```

would expect another specification as reference and possibly the system architecture. For example, given a centralized specification, we generate a decentralized specification, by splitting the specification into subspecifications and assigning the subspecifications to monitors. Generating a decentralized specification using a centralized one as reference is used in some algorithms such as choreography [CF16a] (See Section 8.1). Starting from an LTL formula, the formula is split into subformulas hosted on the various components of the system (this is detailed further in Section 8.1). Given a decentralized specification $\mathcal{D}$, and a system graph $\langle C, E' \rangle$, the problem consists in generating a specification $\mathcal{D}'$. The variants of the synthesis problem depend on the properties that $\mathcal{D}'$ must have, we list (non-exhaustively) example properties:

1. $\mathcal{D}'$ is monitorable (Section 6.2.1);
2. $\mathcal{D}'$ is compatible with $\langle C, E' \rangle$ (Section 6.2.2);
3. $\mathcal{D}'$ and $\mathcal{D}$ are verdict equivalent.

Synthesis problems could also be expanded to handle optimization techniques, with regards to specifications. The specification determines the computation and communication needed by the monitors. As such, it is possible to optimize, the size of automata, and references so as to fine-tune load and overhead for a given system architecture.

Conclusion

We have introduced decentralized specifications wherein a specification is provided for each monitor, the specification is limited to what the monitor can observe and references other specifications. We presented their semantics, and assessed two of their properties: *monitorability* and *compatibility*. *Monitorability* ensures that given a specification, monitors are able to eventually emit a verdict, for all possible traces. *Compatibility* ensures that a monitor topology can be deployed on a given system. Furthermore, we elaborated on the potential properties to explore in the future. In the next chapter (Chapter 7), we present our tool that enables us to generate and monitor decentralized specifications, and thus enable programers to implement decentralized specifications.

CHAPTER 7

THEMIS: A Framework for Decentralized Monitoring of Decentralized Specifications

Contents

Chapter abstract

In this chapter, we explain the design goals and architecture of THEMIS. THEMIS is a modular tool to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ. It consists of a library and command-line tools. THEMIS provides an API, data structures and measures for decentralized monitoring. These building blocks can be reused or extended to modify existing algorithms, design new more intricate algorithms, and elaborate new approaches to assess existing algorithms. THEMIS is designed with the ability to interface with other tools while providing a uniform workflow for designing algorithms, metrics, and running reproducible experiments.

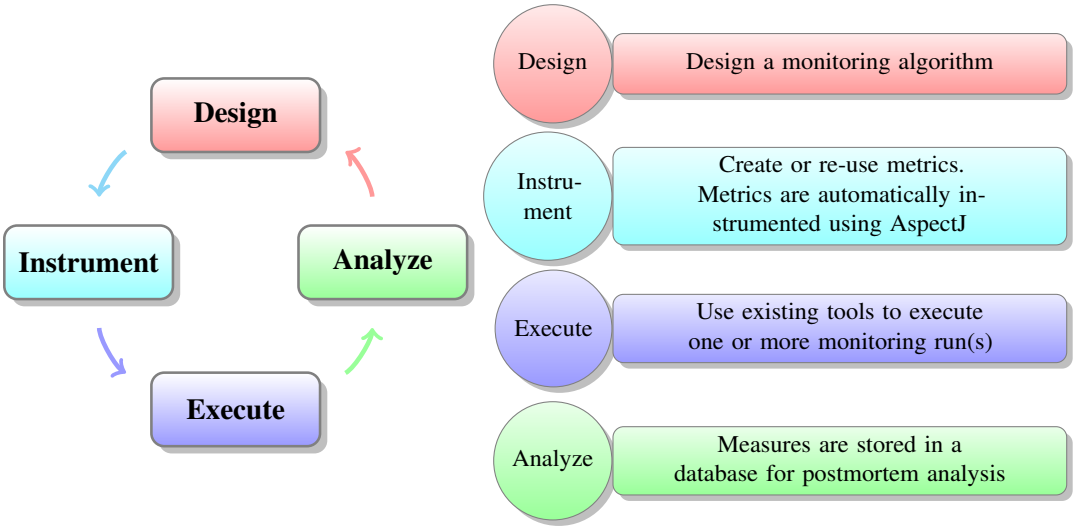


Figure 7.1: Using the THEMIS Framework

Introduction

Methodology THEMIS [EHF17c] is written in Java, uses AspectJ [KHH+01a] and is provided as a library with a set of command-line tools. The primary goal of THEMIS is to design and analyze decentralized monitoring algorithms. It is addressed mostly to researchers to experiment, tune, and compare decentralized monitoring algorithms. To assess the behavior of an algorithm, we identify four phases (Figure 7.1): design, instrument, execute, and analyze. The design phase consists in elaborating a monitoring algorithm. THEMIS implements the generalized decentralized monitoring steps (*setup* and *monitor*, and provides an API to describe the operations. Furthemore, THEMIS provides the data structures used throughout the thesis: Memory and EHE (Chapter 5). These operations are used as building-blocks to assemble an algorithm. The instrument phase consists in the definition of measures. Measures are instrumented into both THEMIS and the algorithms at load-time (bytecode loading in the JVM). Measures are defined at a high-level as they operate on the API and data structures. The execute phase consists in using the THEMIS set of tools to run simulations of the monitoring algorithms and record the measures. The analyze phase consists in using the recorded measures to study, compare, and refine the algorithm. An instrumented program can directly interact with THEMIS by providing a stream of events. In a future chapter (Chapter 8), we illustrate the usage of THEMIS to assess three different decentralized monitoring algorithms on synthetic and real traces.

Design Goals The main design goal of THEMIS is to provide a general API for decentralized monitoring. That is, to provide an environment that accounts for changes at all levels: traces, specification, monitoring logic. By doing so, we allow for new approaches implementing the API to benefit from all existing metrics and analysis. Additionally, this allows metrics to be assessed at the abstract level, for example the metric `messages sent` could be simply reused if new algorithms exchange messages. Furthermore, to accomplish this goal, we also aimed that our measures be stored per run in a database. This allows for analysis and benchmarking to be reduced to querying and analysis of the database. This effectively separates the analysis from the monitoring. Third-party tools can be used to explore and analyze the data. Another important design goal is reproducibility. We wanted to minimize the effort of re-running older simulations or comparing new approaches with older ones. This is reflected with the `Experiment` command-line tool which, in short, allows users to bundle all traces, specifications and algorithms. Since metrics are designed to work at the API level and data structures, any algorithm using the same building blocks can be measured similarly without added effort. This allows for new algorithms or variants of older algorithms to be easily compared against older ones with the same data and measures. By accomplishing these two primary goals, we minimize the overhead needed to design new algorithms and study them, and let researchers focus on the algorithm and the monitoring itself. Finally, THEMIS is designed to introduce decentralized specifications (Chapter 6). While some approaches [BKZ15, CF16a] do in effect introduce a decentralized specification, they primarily focus on presenting one global formula of the system from which they derive multiple specifications. THEMIS encompasses [CF16a] and in addition supports any decentralized specification.

Chapter organization. We begin by introducing the general client-server architecture of THEMIS in Section 7.1. Then, elaborate on the modules needed to write specifications (Section 7.2) and components that read traces (Section 7.3). We elaborate on the usage of data structures (Section 7.4) to write decentralized monitoring algorithms (Section 7.5). To assess decentralized monitoring algorithms, we illustrate the design and integration of measures (Section 7.6). In Section 7.7, we discuss THEMIS nodes which consitute the servers on which algorithms are executed. And finally, in Section 7.8, we elaborate on clients that interact with nodes to deploy components and monitors.

Key contributions. The key contributions of this chapter can be summarized as follows:

1. We present the THEMIS framework with an API and data structure that implement the ideas in this thesis;
2. The modular design allows for THEMIS to interact with other programs whether for monitor synthesis, LTL and Boolean simplification, visualization, and trace and specification generation;
3. THEMIS presents a uniform workflow for designing, assessing, recording simulations, or even directly running decentralized monitoring algorithms;
4. THEMIS allows for reproducible experiments by allowing all algorithms, metrics, traces and specifications to be packaged and re-run with different algorithms or metrics; and
5. Metrics are instrumented using AspectJ, and stored in a database for postmortem analysis by any third party software.

7.1 Architecture Overview

We begin by introducing an overview of the main modules that constitute the THEMIS framework. Then, we elaborate on each module and its design decisions and considerations. As mentioned at the start of the chapter, the primary goal of THEMIS is to facilitate the design and analysis of decentralized monitoring algorithms, while introducing decentralized specifications. To accomplish this, the monitoring is abstracted using a general API and data structures. The general design of THEMIS is a client-server design.

Server. The server is referred to as a THEMIS *node*. One or more nodes can be deployed on a given platform. A node receives information (via commands) to deploy components and monitors on the current platform. Each component contains one or multiple peripheries, it represents a logical component to monitor. A periphery is an input stream to the component, that generates observations. Peripheries follow a stream interface, and wait on a call to generate the next observations. Monitors are attached to components, and receive the observations that components receive. Thus, a node follows a *publish subscribe* model. Components can be seen as topics, where a monitor registers to a topic. Peripheries produce a stream of observations for components. Peripheries can include reading traces from files, over network sockets, or can be generated in a stream. Nodes can communicate with other nodes in a distributed manner, through sockets. The implementation of a node defines the high level assumptions of monitoring, we elaborate on node implementations in Section 7.7.

Client. A THEMIS client is charged with processing a specification, and based on the algorithm, generate and deploy monitors on one or more nodes. As such, one of the client responsibilities is to invoke the *setup* phase of a decentralized monitoring algorithm. The client controls deployed nodes, and provides them with necessary information to perform monitoring, and a callback to allow nodes to notify clients of relevant events such as the end of the monitoring. Several provided command-line tools act as clients, performing one or several monitoring runs. We elaborate on THEMIS command-line tools in Section 7.8.

Modules. The architecture highlighting the main modules is presented in Figure 7.2, the remainder of this chapter is used to elaborate on the various modules. We illustrate how one can write specifications, decentralized monitoring algorithms, measures, and use the provided data structures and backend.

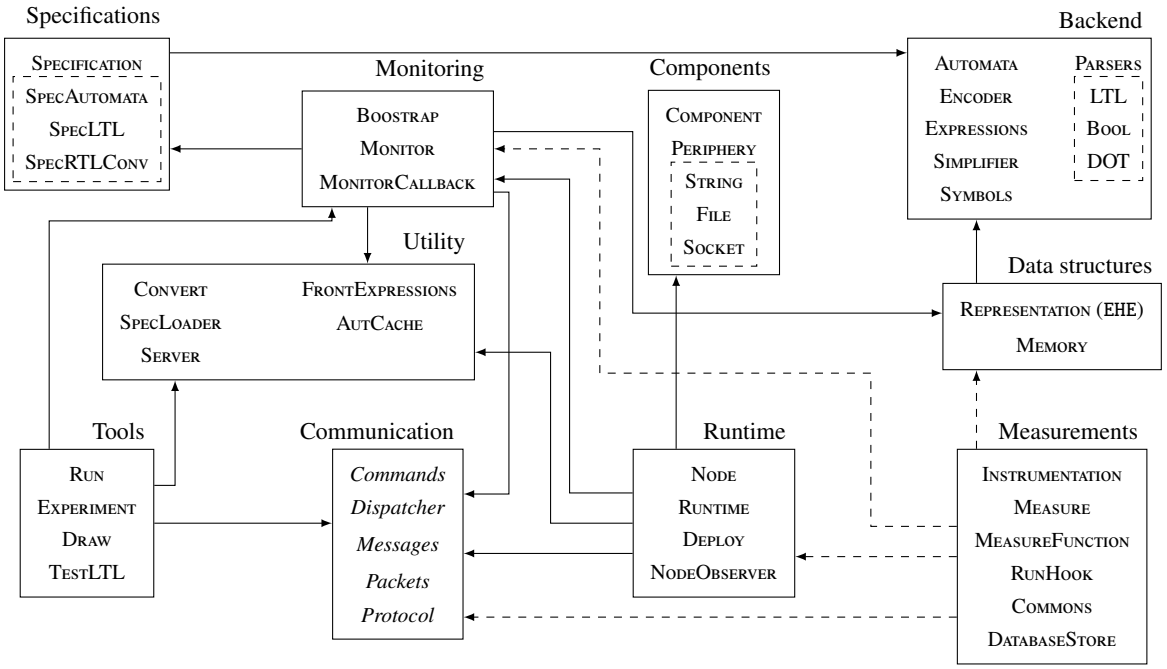


Figure 7.2: General architecture of the THEMIS framework. Arrows indicate dependency, while dashed arrows indicate instrumentation (during Java class loading).

7.2 Writing Decentralized Specifications

THEMIS is designed to manage decentralized specifications, as such the expected input specification is by default a collection of specifications. We elaborate on how to load specifications and interface with other tools for the purpose of synthesis.

7.2.1 Loading Specifications

Specifications are loaded and passed to a monitoring algorithm as a Map, where each specification is assigned a unique key. Specifications are loaded from an *xml* file using SpecLoader by calling loadSpec(File). Each specification must provide two attributes: an id and a class name. The id is a string name for the specification, it is used by the algorithm during the *setup* phase. The class name is a string representing a full class name of the specification class.

Example 27 (Writing a light-switch specification) Listing 2 shows an example of a decentralized specification written in THEMIS. It is used to express the property that ensures a light bulb is turned on after a switch is pressed, until the switch is no longer pressed. This specification is used when monitoring the smart home in Chapter 9 and detailed in Example 36. We see it consists of 3 subspecifications, that are written in LTL. The first two specifications are used to check when a light or switch is turned on. They are given the ids *light* and *switch*, respectively. The third specification (with id *checklight*) references the other two, to ensure the property described. The @ sign is used to indicate references to other specifications. These specifications are converted to automata during the setup phase of an algorithm, and a decentralized specification is formed (see Section 7.5.2). Let us consider specification *switch1* to illustrate how THEMIS loads it. In this case, an object of class SpecLTL is instantiated, on which method `void setLTL(String)` is called with the parameter "office_switch". *

Listing 2 Decentralized specification for a light-switch.

```
<specifications>
  <specification id="switch1"     class="uga.corse.themis.monitoring.specifications.SpecLTL">
      <setLTL><![CDATA[(office_switch)]]></setLTL>
  </specification>
  <specification id="light1"      class="uga.corse.themis.monitoring.specifications.SpecLTL">
      <setLTL><![CDATA[(office_light)]]></setLTL>
  </specification>
  <specification id="checklight"  class="uga.corse.themis.monitoring.specifications.SpecLTL">
      <setLTL><![CDATA[@switch1 -> X(@light1 U !@switch1))]]></setLTL>
  </specification>
</specifications>
```

Listing 3 Template for light-switch.

```
1  <specifications>
2    <template id="lightswitch">
3        <arg>room</arg>
4        <arg>roomid</arg>
5        <specification id="switch%roomid%" class="uga.corse.themis.monitoring.specifications.SpecLTL">
6            <setLTL><![CDATA[(%room%_switch)]]></setLTL>
7        </specification>
8        ...
9    </template>
10   <instanciate template="lightswitch">
11     <room>office</room>
12     <roomid>1</roomid>
13   </instanciate>
14   ...
15 </specifications>
```

7.2.2 Templates

Writing complex specifications often involves writing similar specifications multiple times. For example, consider instantiating the *checklight* specification for multiple rooms. THEMIS makes it possible to define *templates* which are pre-processed into specifications. Templates define arguments, which are then bound when instantiated. The pre-processing involves replacing a variable reference with the passed argument. The tag *template* defines a template, it is given an identifier *id*. It is then followed by multiple *arg* tags used to specify the name of the arguments for the template. A template may contain one or more specifications. For each specifications all text (id, class, arguments to method calls), detects arguments and replaces them. A variable reference is surrounded by the % sign. The template is instanitated with the tag *instanciate* with an attribute *template* indicating the template *id* to instanciate. The instantiation tag then contains a tag for each argument providing its value.

EXAMPLE 28 (SPECIFICATION TEMPLATE) Listing 3 shows an example of using templates in the specifications file. It defines the template *lightswitch* (lines 2-9) which takes two arguments: *room* and *roomid*, which determine the name of the room and its index. The template defines the three specifications explained in Example 27. For simplicity, we show only one, and focus on the id of the specification "switch%roomid%" (line 5). Lines 10-13 instantiate the template with the value of 1 for argument *roomid*. When the specification is loaded, this will append the specification in the template with an *id* of "switch1". *

7.2.3 Integration with Monitor Synthesis Tools

We recall that the decentralized specification semantics defined in Section 6.1 relies on automata for monitoring. In general, we allow specifications to be arbitrarily defined and passed to the *setup* phase of an algorithm to manage them. As such, it is possible to even write custom specifications and algorithms that do not rely on automata (e.g. rewrite-based). However, for the purpose of the thesis we focus on automata-based specifications. For convenience, the THEMIS framework provides a conversion utility, used to convert arbitrary specifications into automata. Class CONVERT is designed to assist developers in converting specifications by automatically calling monitor synthesis tools (such as `ltl2mon` [BLS11], and LamaConv [Ins]). It also provides additional functionality for completing

Listing 4 Specification expressed in SALT stating that the entrance door must alternate between open and closed at least twice. The event *%empty%* serves as a symbol to be replaced by an arbitrary event to allow for partial observations (by extending the alphabet).

```
1  <specifications>
2    <specification id="showup" class="uga.corse.themis.smarthome.specs.SpecRTLConv">
3        <arg>-salt</arg>
4        <arg>-ltl</arg>
5        <arg>-moore</arg>
6        <arg>-min</arg>
7        <line>event entrance_door,%empty%</line>
8        <line>assert occurring[>=2] entrance_door</line>
9    </specification>
10 </specifications>
```

Listing 5 Using LamaConv to convert SALT specifications.

```
1  Convert.Converter rtlconv = new Convert.Converter() {
2      @Override
3      public SpecAutomata toAutomata(Specification spec) {
4          SpecRTLConv rtlspec = (SpecRTLConv) spec;
5          try {
6              Automata aut = RTLConv.RTLConv(rtlspec.getInput(), rtlspec.getArgs());
7              Convert.completeEvents(aut);
8              return new SpecAutomata(aut);
9          }catch(Exception e) {
10             log.fatal(e.getLocalizedMessage(), e);
11         }
12         return null;
13     }
14 };
15 Convert.registerConvert("uga.corse.themis.smarthome.specs.SpecRTLConv", rtlconv);
```

automata, and simplify the automata transitions (to normalize them, see Remark 1). To convert an arbitrary Specification to a SpecAutomata, class Convert provides the method `SpecAutomata makeAutomataSpec(Specification)`. By default `THEMIS` provides a converter for LTL that uses `ltl2ltmon`. A converter is a simple interface that defines the method `SpecAutomata toAutomata(Specification spec)`, which generates a SpecAutomata from an arbitrary Specification. It is possible to customize the conversion by providing a custom converter to Convert using the method `void registerConvert(String classname, Converter conv)`.

Example 29 (Synthesis of SALT specifications) We illustrate a specification written in Smart Assertion Language for Temporal Logic (SALT) [BL11], an imperative-like temporal specification language suited for software engineers. It is possible to synthesize monitors written in SALT using LamaConv [Ins]. Listing 4 displays the custom SpecRTLConv specification written to call LamaConv with a SALT specification. Lines 3-6 pass the necessary command-line arguments for LamaConv to indicate that the specification is a SALT specification (as LamaConv supports multiple specification formalisms). Lines 7-8 describe the SALT specification. Line 7 defines the alphabet of events, the event *entrance_door* indicates the opening of the door, while the special event *%empty%* is used to expand the alphabet, as to account for partial observations, since SALT is event based and we need to specify atomic propositions. Line 8 defines the assertion that the event *entrance_door* must alternate at least twice (that is hold $\top$ then $\bot$ then $\top$ again). Listing 5 shows the code needed to implement the converter. The class RTLConv is used to abstract the call to the LamaConv process, and parse its output. The converter calls LamaConv (line 6), then completes the provided automaton (line 7). Line 15 shows the code necessary to register the converter with the class SpecRTLConv. *

Now that we defined specifications, we present in the next section (Section 7.3) the remaining input to a decentralized monitoring algorithm: traces and observations.

7.3 Managing Components and Traces

In this section, we discuss the provided mechanisms to manage decentralized traces (Definition 9), the main input to a monitoring algorithm. We recall from Section 1.3, that our view of a system is a set of components. One or more monitors can be attached to a component. The component provides one or more observations to the attached monitors. In THEMIS a component consists in a logical unit that groups multiple observation sequences (called *peripheries*). We discuss the mechanism to define a *periphery* in Section 7.3.1. Then, we elaborate on grouping peripheries to form a component, and discuss the deployment of components in Section 7.3.2.

7.3.1 Peripheries

A *periphery* is a two-way link between the THEMIS framework and an observed entity. Interface PERIPHERY provides the main form of input for the THEMIS framework. It consists of three methods: `next`, `notify` and `stop`. Method `Memory<Atom> next()` returns a memory containing all observations observed since the last call to `next`. Method `void notify(Control)` allows the THEMIS framework to send control signals to the periphery. This is designed for enabling interaction with the periphery, in the case the periphery is a process or thread, or remote system. Finally, `void stop()` signals that observations are no longer necessary. This is used to notify peripheries of the end of the monitoring. Upon initialization by the THEMIS framework, a periphery is typically passed a configuration string in its constructor to set it up.

THEMIS provides various implementations for a periphery. The shared main format for observations consist of a string representing an atomic proposition, followed by a column, followed by a `t` or `f` to indicate respectively $\top$, and $\bot$. Events are defined as comma-separated observations (see Definition 5 for details). Typically, a call to method `next` returns the next available event. For example the event: `a:t,b:f` consists of two observations `a` and `b` associated with $\top$ and $\bot$ respectively. The provided implementations are as follows:

TRACESTRING A periphery that reads a sequence of events delimited by the character ">". This is rarely used, but provided for debugging purposes.

TRACEFILE A periphery that reads a sequence of events from a file, each line constitutes an event.

TRACENETPLAIN A periphery that reads a sequence of events from a network socket, allowing events to be streamed over a network. Events are delimited by lines. This makes it easy to allow input streams to come from networked processed or systems, the plain format allows utilities like `cat` or `telnet` to easily transmit remote log files to THEMIS.

EXAMPLE 30 (TRACEFILE PERIPHERY) Listing 6 shows the implementation of TRACEFILE. We notice that the constructor (Line 4) configuration string consists of the file path to open. Method `next` (Lines 8-21) provides the observations by reading the file. It reads the next line in the file (Line 12), and parses the event (Line 15-19), adding to the memory the atomic propositions encountered (Line 18). Method `stop` (Lines 22-26) closes the files, as the monitoring has completed. *

A component is associated with multiple peripheries, and it controls the process of reading observations provided by each periphery. We next elaborate on components in Section 7.3.2.

7.3.2 Managing Components

Building Components. A component constitute a logical unit which groups and communicates with one or more peripheries. A component is identified by a unique identifier (its *name*), and can be assigned one or more peripheries. The component is initialized and invoked by a THEMIS node, to return observations, that will be fed to monitors. The abstract class COMPONENT provides the main functionality for managing the names of components. The necessary interface for managing, fetching observations is provided in Listing 7. The provided implementation for a component is ASYNCOBSERVATIONS, method `observe` calls method `next` of each periphery asynchronously, and awaits the event. Once *all* peripheries have returned their events, it merges them into one memory and returns them.

Listing 6 The TraceFile periphery implementation.

```
public class TraceFile implements Periphery {
    BufferedReader trace;
    FileReader fr;
    public TraceFile(String fname) throws FileNotFoundException {
        fr = new FileReader(new File(fname));
        trace = new BufferedReader(fr);
    }
    public Memory<Atom> next() throws IOException {
        Memory<Atom> mem = new MemoryAtoms();
        String line = null;
        //Read next event
        line = trace.readLine();
        if (line == null || line.isEmpty()) return mem;
        //Parse Event
        String[] aps = line.split(",");
        for (String ap : aps) {
            String[] obs = ap.split(":");
            mem.add(new AtomString(obs[0]), obs[1].equals("t"));
        }
        return mem;
    }
    public void stop() {
        try {
            trace.close(); fr.close();
        } catch (IOException e) {}
    }
}
```

Listing 7 Abstract Component methods.

```
public abstract void addInput(Periphery per);
public abstract Collection<? extends Periphery> getInput();
public abstract  Memory<Atom> observe() throws Exception;
```

Configuration and deployment. It is important to note that components are deployed on the THEMIS node as opposed to being instantiated on the client. This means that components execute on the system where the node is deployed. As such, it is important to account for the configuration string to be correct, in the presence of paths. Clients typically only define the necessary information needed for the node to create components.

Trace convention. The default tools bundled with THEMIS illustrated in Section 7.8 rely on a simple trace convention to load components. While the THEMIS framework allows clients to manage components as they please, we present a simple trace format used by both the Run and Experiment tool to avoid defining components, peripheries and traces for executing algorithms. The trace format defines a single input folder, where multiple files are placed. Components are given letter names starting from the letter "a", and traces are given an *id*. For example, for the trace with id 0, with 3 components, the input folder contains 3 files named `0-a.trace`, `0-b.trace`, and `0-c.trace`. Each component is then associated with a periphery of type TraceFile.

With the input to a decentralized monitoring algorithm defined, we shift the focus to the data structures useful for writing decentralized monitoring algorithms. We present them in Section 7.4.

7.4 Data Structures Implementations

THEMIS provides a modular implementation of the two data structures presented in this thesis: Memory (Definition 6) and EHE (Chapter 5). These data structures are used as building blocks to write and assess decentralized algorithms. Both data structures implement Cloneable and Serializable, as are to be communicated through nodes.

Listing 8 MEMORY interface.

```
1 void add(Atom a, boolean observed);
2 Verdict get(Atom a);
3 boolean isEmpty();
4 void merge(Memory<K> m);
5 void clear();
```

7.4.1 Memory

The *memory* data structure is at the core of any decentralized monitoring algorithm, as it is used to store observations, and the result of other monitors. The generic MEMORY interface is parameterized by the type of its content but must always be able to process atoms. This allows any memory implementation to process atoms, and receive observations as atoms, while keeping its internal storage type abstracted.

Atoms. At the core of the memory are *atoms*, designed internally by interface ATOM. The main responsibility of an implementation of ATOM is to provide comparison with other atoms, as it is important to establish a total order between atoms for eventual consistency (Section 4.2). Another important method to implement is `observe` which is supposed to return a unique string representing the atom, as this is used to encode the atom when transmitted to simplifiers. `THEMIS` currently provides 4 implementations of ATOM:

ATOMSTRING A string representing an atomic proposition.

ATOMOBLIGATION A pair of timestamp and an ATOMSTRING, associating a timestamp for an atomic proposition.

ATOMNEGATION An atom indicating the negation of another included atom (string or obligation).

ATOMEMPTY A special atom indicating the absence of any atomic proposition.

Operations. The main operations performed on a memory are the addition of atoms, merging of another memory, and checking if an atom is included in the memory. They are shown in in Listing 8. When adding atoms to the memory using method `add`, it is necessary to associate it with whether or not it has been observed. The memory is responsible for managing the internal representation by either negating it or using its own internal representation.

> **REMARK 2 (MEMORY OF EVENTS)** Our implementation allows memory to be also used to store events as in [CF16a], atoms implement a specific method `group`, which returns a key for each atomic proposition. Atomic propositions sharing the same key are grouped to form an event. Event matching follows the semantics of subset matching, with special consideration for negation and empty events. *

Memory implementations. The default memory implementation provided is MEMORYATOMS which follows the structure and semantics used in this thesis. An additional memory implementation MEMORYRO is a wrapper that contains an arbitrary MEMORY inside of it. It adds the *read-only* constraint, such that it is not possible to modify the enclosed memory through it. MEMORYRO is useful when managing collected observations or resulting observations, due to the multithreaded nature of `THEMIS`, ensuring no concurrent access or change to underlying observations. When dealing with long traces, memories tend to get big, as such it is important to perform garbage collection. The last implementation MEMORYINDEXED stores only ATOMOBLIGATION, and is used to group all obligations by timestamp, such that it is easy to remove all observations past a certain timestamp.

7.4.2 Execution History Encoding

The `EHE` data structure is implemented as REPRESENTATION, it supports all operations used in Chapter 5 with the addition of other useful operations. An `EHE` is instantiated with an automaton (for which it encodes the execution), and a simplifier, which is used for performing boolean simplification of its entries. In addition to containing the entries encoding the execution as explained in Definition 13, the `EHE` implementation keeps track of (1) the minimal timestamp (`start`), (2) the maximal timestamp (`end`), (3) the last timestamp for which an expression

simplified to $\top$ for a given state (`lastResolved`), and (4) similar to (3) but the state was labeled with a final verdict (`lastVerdict`). The extra information is useful for developers designing algorithms, as it allows them to optimize the usage of the EHE.

Initializing and expanding. The initial state of an EHE is by default assigned timestamp 0, and is set to the initial state of the associated automaton. However, it may become necessary to change the timestamp, when for example having to run the same monitor from a different timestamp. Method `void addInitRule(int)` takes a timestamp, and adds the initial state of the automaton but at the specified timestamp. However, this does not reset the state of the EHE. For resetting the EHE state, it is possible to use `void reset(int)` which performs the similar function but also clears the EHE entries and its state. Now that we can initialize the EHE, method `void tick()`, performs a mov by 1 timestamp (see Section 5.1.1) from the end. To merge EHEs, the method `void mergeForward(Representation)` merges all entries that start from at least the timestamp `start`.

Resolving. Method `boolean update(Memory, int)` performs the main logic of the EHE data structure, it simplifies and rewrites each expression of the entries using a memory, starting from a given timestamp (by default it uses `lastResolved + 1`), upon resolving a state it stops and returns $\top$, if no state is resolved it returns $\bot$. For performance improvement, method `update` stores the verdict associated with the resolved state to be queried later. As such, when it returns $\top$, it is possible to use method `VerdictTimed scanVerdict()` to return the last found verdict. Class VerdictTimed contains the verdict and its associated timestamp.

Garbage collection. It is possible to remove all entries inferior to a given timestamp using method `int drop(int)` which takes a timestamp and removes all entries with an earlier timestamp. The method returns the number of timestamps removed. For easier management, method `int dropResolved()` removes all entries before the `lastResolved` timestamp, which are essentially no longer needed.

Improving performance. The adopted EHE implementation relies on the simplifier as it mostly performs simplifications. By changing the simplifier it is possible to obtain better performance. This is because EHE is built iteratively, by starting with smaller expressions, the resulting expressions for the next timestamp are smaller. By doing so, we keep the size from growing too large. For our implementation we rely on `ltlfilt` from SPOT [DL13], which also simplifies LTL expressions. Let us consider two profiles: Quick and Aggressive. The profile Quick uses `ltlfilt` with default parameters for simplification. Furthermore, it calls the simplifier for every expression in the EHE. We note that this does not perform the most thorough boolean simplification. The profile Aggressive utilizes the flag `--boolean-to-isop` which rewrites boolean formulas as irredundant sum of products, as such ensuring a more aggressive simplification strategy. This strategy is more costly in terms of memory and computation. Therefore, we make calls to the simplifier only for complex simplifications, and implement the basic Boolean simplification while traversing the expression to replace atomic propositions by looking up the memory (in the operation rw). Figure 7.3 shows the data transferred for the two variants of the migration algorithms, which is associated with the size of the EHE. The x-axis indicates the algorithm's variant and the number of components, where Migr (resp. Migrr) stands for earliest obligation (resp. round-robin) variant. The y-axis is presented in logarithmic scale, it illustrates the size of the EHE. The size is presented as an abstract metric by counting the size necessary to encode expressions. We notice that the algorithms can be compared similarly for any given profile. However, for the more aggressive simplification, we observe a significant drop in the size of the EHE.

Now that we defined the necessary data structures and input for monitoring, we can proceed in the next section (Section 7.5) to detail the process of writing a decentralized monitoring algorithm.

7.5 Writing Decentralized Monitoring Algorithms

Designing a decentralized monitoring algorithm consists of two phases. As such, when implementing a decentralized monitoring algorithm, THEMIS provides two classes to manage the phases: Bootstrap and Monitor. They

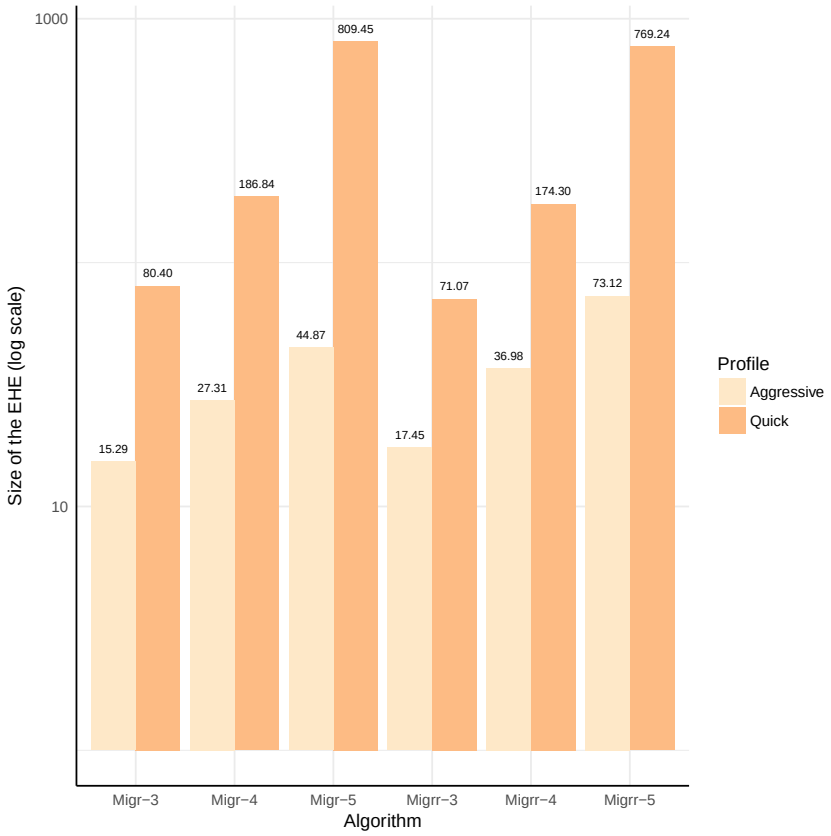


Figure 7.3: Size of EHE for different algorithms and components.

correspond respectively to the two phases: *setup* and *monitor*. Section 7.5.1 illustrates writing a BOOTSTRAP, Section 7.5.2 details the structure of MONITOR.

7.5.1 Setup Phase: Writing the Boostrap

The BOOTSTRAP interface presents the minimal necessary implementation for a monitoring algorithm to perform the *setup* phase. It consists of a single method implementation: `Set<? extends Monitor> getMonitors(Topology)`. The method is passed a TOPOLOGY object containing information about the system graph. It displays the number of components and the connections between components (similar to that for *compatibility* in Section 6.2.2). The method must return a set of objects implementing MONITOR, this set contains all monitors to deploy. We elaborate on the MONITOR interface in the next section (Section 7.5.2).

While interface BOOTSTRAP presents the minimal necessary requirements to be fullfilled, it does not pass the specification to the algorithm, and this needs to be done manually. The interface STANDARDMONITORING (which extends BOOTSTRAP) adds the additional requirement of managing the specification. It adds the following method: `void setSpecification(Map<String, Specification>)`, which is tasked with initializing the specifications. As such, all algorithms that are implemented in this thesis, and all tools, will focus on STANDARDMONITORING. We illustrate the *setup* phase of algorithm Orch in Example 31.

EXAMPLE 31 (ORCHESTRATION SETUP) The orchestration algorithm (Orch) consists in setting up a main monitor which will be in charge of monitoring the entire specification. However since that monitor cannot access all observations on all components, orchestration introduces one monitor per component to forward the observations to the main monitor. Therefore, for our setup, we consider the case of a main monitor (labeled m_0) placed on component c_0 which monitors the specification and $|C| - 1$ forwarding monitors that only send observations to m_0 (labeled m_k with $k \in [1, |C| - 1]$).

Listing 9 Setup phase for orchestration.

```
1  public class Orchestration implements StandardMonitoring {
2      protected Automata aut;
3      public void setSpecification(Map<String, Specification> decentSpec) {
4          Specification main = decentSpec.get("root");
5          if(main == null) throw new IllegalArgumentException("No Root Spec found");
6          setMainSpec(Convert.makeAutomataSpec(main));
7      }
8      public void setMainSpec(Specification spec) {
9          if(!(spec instanceof  SpecAutomata))
10             throw new IllegalArgumentException("Specification must be an automaton!");
11         aut = ((SpecAutomata) spec).getAut();
12         if(aut == null) throw new IllegalArgumentException("Automaton is not valid");
13         aut = Convert.simplifyTransitions(aut);
14     }
15     public Set<? extends Monitor> getMonitors(Topology topology) {
16         int c = topology.getCountComponents();
17         Set<Monitor> mons = new HashSet<>(c+1);
18         if(c <= 0) return mons;
19         if(aut == null) throw new IllegalStateException("No Automaton provided");
20         Set<String> comps = topology.getGraph().keySet();
21         Iterator<String> iter = comps.iterator();
22         int i = 0;
23         MonOrchMain main = new MonOrchMain(i);
24         main.setSpec(aut);
25         main.setComponentKey(iter.next());
26         mons.add(main);
27         for(i = 1; i < c; i++) {
28             MonOrchSlave slave = new MonOrchSlave(i);
29             slave.setComponentKey(iter.next());
30             mons.add(slave);
31         }
32         return mons;
33     }
34 }
```

To implement the *setup* phase of Orch, we create a class ORCHESTRATION that implements STANDARDMONITORING. The class is shown in Listing 9. Then, we begin by implementing the `setSpecification` method (lines 3-7). Since Orch performs decentralized monitoring of a centralized specification, we use as convention the key *root* to denote the centralized specification (Line 4). Then, we convert the specification to an automata-based specification (Line 6) as explained in Section 7.2.3, and pass it to the `setMainSpec` method, which extracts the automaton from the specification and simplifies the expressions on its transitions. Now that the main automaton is generated, we implement the call to `getMonitors`, which deploys monitors (Lines 15-33). To do so, we first get the number of components in the system using the TOPOLOGY class (Line 16). If there is more than one component, we retrieve the iterator on components (Line 20). Then, we instantiate the main Orch monitor with *id* 0 (Line 23). The main monitor is passed the automaton (Line 24), and is deployed on the first available component (Line 25). For each remaining component, we create a forwarding monitor (Line 28), and associate it with the component (Line 29).∗

7.5.2 Monitor Phase: Writing Monitors

In Section 7.5.1 we explained how decentralized monitoring algorithms using a specification and a topology are able to perform their *setup* phase and deploy monitors. We now elaborate on writing monitors with the help of the data structures presented in Section 7.4. Listing 10 presents the most relevant methods of the API. We classify the methods into 3 groups: deployment, communication, and monitoring.

Deployment. Monitors are first instantiated in a client and configured using the BOOTSTRAP implementation of a given algorithm. Their data is transferred to the node on which they will perform monitoring. Therefore, monitors implement SERIALIZABLE as they are transferred as objects to a THEMIS node. Upon receipt on a node, a monitor is capable to initialize its state using method `setup`. At the end of monitoring, the method `cleanup` is called so that monitor can perform its necessary cleanup code. A special method `reset` is used to "soft reset" the monitor, that is,

Listing 10 Relevant methods for interface MONITOR.

```
1  void setup();
2  void cleanup();
3  void reset() throws Exception;
4
5  void setNetwork(Network net);
6  void send(int to, Message msg);
7  Message recv(boolean consume);
8
9  void monitor(MemoryRO<Atom> observations, MonitorCallback callback) throws Exception;
10 void communicate();
11 Verdict getCurrentVerdict();
```

it informs the monitor that it only needs to reset to its initial state. Method `reset` is used to indicate to the monitor that the specification is still the same. This method is useful for monitoring multiple traces on the same monitor, or recovering in a given node, when necessary.

Communication. Monitors communicate with each others using a NETWORK object provided with method `setNetwork`. The communication is abstracted for the monitor, as such it is only able to send a message to another monitor knowing its *id* (using method `send`), or read any messages sent to it stored in a queue (using method `recv`). Method `recv` by default consumes the message received, it can be passed a boolean ($\perp$) to keep the message. THEMIS provides the class GENERALMONITOR that implements the basic communication functionality. As such, it suffices to extend it and focus on implementing the monitoring logic.

Monitoring. The monitor main logic is contained in method `monitor`. An additional method `communicate` is provided when it is necessary to separate the monitoring from communication, for example, when simulating certain models of computation (such as the Bulk Synchronous Parallel model [Val90]). Method `monitor` receives a read-only memory containing observations, and a handler MONITORCALLBACK for interacting with the node. The handler contains two methods `notifyVerdict` and `notifyAbort`. Monitors can use those methods to respectively notify the node when reaching a verdict, or request that monitoring be aborted. Method `notifyVerdict` takes as parameter the monitor *id* of the monitor that reached the verdict, the verdict reached, and additional data to send to the node (anything that extends Object). Method `ntofiyAbort` takes as parameter the monitor *id* and an enum representing the type of abort: whether to abort all monitoring on the given node, or to issue a global abort on all participating nodes. Finally, method `getCurrentVerdict` returns the current verdict of the monitor, this includes non-final verdicts.

EXAMPLE 32 (MAIN MONITOR FOR ORCH) Following the *setup* phase of Orch presented in Example 31, we now illustrate the main monitoring logic of the main (central) monitor. Listing 11 shows the partial source of MONORCHMAIN implementing the methods mentioned in this section. Lines 1-6 show method `setup`, which is called when the monitor is deployed on the node. We see here that it begins by creating an `EHE` (Line 2), a memory (Line 3), initiates the simplifier process (Line4), and initializes the current verdict to ?. For method `reset`, we simply clear the memory (Line 8) and re-initialize the `EHE` (Line 9). For method `cleanup`, we simply stop the simplifier process (Line 11). The main monitoring logic is provided by implementing method `monitor` (Lines 12-36). First, we merge the observations local to the component we attached the monitor (Line 13). Second, we process all incoming messages (Lines 16-19), and merge all received memories (Line 18). Third, we increment the `EHE` state (Line 24), rewrite and simply it with the existing memory (Line 25), and get the last resolved verdict (Line 26). Details on these operations is presented in Section 7.4.2. For the scope of our experimentation, we had a delay of 1 timestamp, so it was safe to clear the memory (Line 27). However, for an arbitrary fixed delay, it is best to use a memory of type MEMORYINDEXED and manage the garbage collection. Fourth, we verify if the obtained verdict is final. If the verdict is not final, we continue monitoring. Otherwise, we use the MONITORCALLBACK to notify the node that we have reached a final verdict (Line 30). Then since we know this is the main monitor, we initiate an abort (Line 31).∗

After elaborating on the sturcture and design of decentralized monitoring algorithms, we discuss in (Section 7.6) writing measures to assess their performance.

Listing 11 Main monitor of Orch.

```
public void setup() {
    autRep = new Representation(spec);
    memory = new MemoryAtoms();
    simplifier = SimplifierFactory.spawnDefault();
    verdict = Verdict.NA;
}
public void reset() {
    memory.clear();
    autRep = new Representation(spec, simplifier);
}
public void cleanup() { simplifier.stop(); }
public void monitor(MemoryRO<Atom> observations, MonitorCallback callback) throws Exception {
    memory.merge(observations);
    Message m;
    //Handle incoming messages
    while ((m = recv()) != null) {
        MemoryPacket packet = (MemoryPacket) m;
        memory.merge((Memory<Atom>) packet.mem);
    }
    //Nothing observed: Either first timestamp or end of trace
    if (memory.isEmpty())
        return;
    else
        autRep.tick();
    autRep.update(memory, -1);
    VerdictTimed v = autRep.scanVerdict();
    memory.clear(); //Since we know the maximum delay is 1 we can remove all elements
    if (v.isFinal()) {
        this.verdict = v.getVerdict();
        callback.notifyVerdict(id, v.getVerdict(), new VerdictTimed(v.getVerdict(), v.getTimestamp()));
        callback.requestAbort(id, MonitorCallback.ABORT_TYPE.ABORT_RUN);
        return;
    }
    autRep.dropResolved();
    return ;
}
```

7.6 Writing Measures

THEMIS uses AspectJ [KHH+01a] (introduced in Section 10.3) to record measures of a metric for a given algorithm. Writing a measure for an algorithm consists in using AspectJ's aspects to intercept the points in the execution. In this section, we illustrate the support provided by THEMIS to simplify writing (Section 7.6.1), storing, and analyzing measures (Section 7.6.2).

7.6.1 Measurements API

Measures in THEMIS are implemented as aspects, written in AspectJ. The details of AspectJ instrumentation are provided in Section 2.2. Since THEMIS provides interfaces for data structures (Section 7.4) and the monitoring (Section 7.5), designing measures is highly modular, as measures are instrumented for any class that implements the interface. For example, measures related to communication are instrumented on any class implementing the NETWORK interface. Additionally, THEMIS provides classes to simplify writing measures for users unfamiliar with AspectJ.

Aspect-oriented programming. We recall the basic principles. A *joinpoint* is a well-defined point in program execution where additional code is to be injected. A *pointcut* is a syntactic element which refers to a set of *joinpoints* and execution context information. Basic pointcuts can be composed and identified (referred to as *named pointcuts*) so as to increase re-usability. Pointcuts can refer to compile-time information such as function signature, a variable name, and a module that needs to be matched. Additionally, pointcuts are able to specify dynamic execution constraints, such as a function being invoked while inside another function (e.g. `cflow` pointcut in AspectJ). An

Listing 12 Named pointcuts for aspect COMMONS, and their usage.

```
1  //Starting Monitoring, call happens before exec
2  pointcut monitoringStartCall(Map<String, Serializable> tags)
3  pointcut monitoringStartExec(Map<String, Serializable> tags)
4  //Stopping Monitoring, Exec happens before call
5  pointcut monitoringStopExec()
6  pointcut monitoringStopCall()
7  //During monitoring
8  pointcut inMonitor(Monitor mon)                        //Capture the monitor object
9  pointcut monitorStep(Monitor mon, MemoryRO<Atom> obs) //Call to Monitor.monitor
10 pointcut monitoringStep(Integer t)                     //Timestamp when performing round-based monitoring
11 pointcut sendMessage(Integer to, Message m)            //Capture sent message
12 pointcut onVerdict(Integer monid, Verdict verdict)     //Capture emitted verdict
13 //Usage
14 after(Integer mid, Verdict v) : Commons.onVerdict(mid, v) {
15   System.out.println("Verdict found by monitor " + mid + ": " + v);
16 }
```

advice defines the additional code to be executed at each specific joinpoint selected by a pointcut. An *aspect* serves as the modular unit that encapsulate advices, pointcuts, and additional behavior.

Common pointcuts. To simplify the process of writing measures, THEMIS provides the COMMONS aspect which only introduces named pointcuts on the relevant parts of monitoring. These named pointcuts can be used when defining measures by the user. The main named pointcuts of aspect COMMONS are presented in Listing 12. These pointcuts are related to the main flow when monitoring with a decentralized monitoring algorithm as explained in Section 7.5.2. The start of the monitoring (Lines 2-3) pointcut includes the tags passed to the node when monitoring begins (detailed in Section 7.7). Tags could include information as to the name of run or objects useful for benchmarking, this allows measures to capture them. The end of the monitoring (Lines 5-6) pointcut is used by measures when cleanup is needed after monitoring. Multiple pointcuts are provided during monitoring to (1) get the underlying monitor context (Line 8) when for example counting evaluations, (2) capture every monitor call (Line 9) for measures relying on analyzing observations or monitor progress, (3) capture the round number (Line 10) for round-based monitoring algorithms, (4) message exchange (Line 11), and (5) capturing the emitted verdict. Finally, we show the use a pointcut from COMMONS to print the obtained verdict after it is reached (Lines 14-16).

> **REMARK 3 (HOOKS)** While we provide the named pointcuts in COMMONS to write measures, they can also be used to write hooks to execute additional logic when monitoring. Specifically for hooks, we introduce the class RUNHOOK which simplifies the task by providing three abstract methods to implement: `prestart`, `start` and `end`. The first two methods allow a hook to run before executing monitoring where `prestart` is executed before `start`. The last method allows a hook to run at the end of the run of a monitoring algorithm. ∗

Measures. Measures are defined as objects of class MEASURE. A measure groups (1) a string representing the measure name (similar to a variable identifier) (*key*), (2) a small string describing the measure (*description*), (3) an object which represents the value of the measure, and (4) a variable arguments update function invoked to update the value of the measure (defined as interface MEASUREFUNCTION). A collection of update functions is provided by class MEASURES which include computing sum, max, min of integers and floats. Invoking `update` on a measure calls its update function. To initialize, group and update measures we extend the abstract aspect INSTRUMENTATION. Aspect INSTRUMENTATION provides the base functionality for managing measures. It uses the named pointcuts in COMMONS and associates with each pointcut introduced in Listing 12 an empty advice function. Programmers can then override the empty function to implement their own logic. As such, it is possible to write measures with no knowledge of AspectJ, and use AspectJ only when advanced measures are necessary. We illustrate writing two communication measures in Example 33

EXAMPLE 33 (WRITING COMMUNICATION MEASURES) We consider counting the number of messages exchanged between monitors and their size. Furthermore, we count the number for each message type exchanged (which we store in a HASHMAP shown at Line 2). First, we define an aspect that extends INSTRUMENTATION (Line 1). Second, we override method `setup` (Lines 3-9) to define and initialize the measures. Line 4 defines the description for the

Listing 13 Writing measures for communication.

```
1  public aspect Communication extends Instrumentation {
2      HashMap<String, Long> comms = new HashMap<String, Long>();
3      protected void setup(Node n, Map<String, Serializable> tags) {
4          setDescription("Communication");
5          addMeasures(
6                new Measure("msg_num" , "Number of Messages", 0L, Measures.addLong)
7              , new Measure("msg_data", "Data Exchanged"    , 0L, Measures.addLong)
8          );
9      }
10     protected synchronized void printAll() {
11         super.printAll();
12         Commons.printSection("Messages Details");
13         for (Entry<String, Long> entry : comms.entrySet())
14             Commons.printMeasureln(entry.getKey(), Commons.FMT_NUM, entry.getValue());
15         comms.clear();
16     }
17     after(Integer to, Message m): Commons.sendMessage(to, m)
18     {
19         synchronized(comms) {
20           String s = m.getClass().getSimpleName();
21           Long n = 0l;
22           if (comms.containsKey(s))
23               n += comms.get(s);
24           n++;
25           comms.put(s, n);
26           update("msg_num", 1L);
27           update("msg_data", Config.sizeMessage(m, new Config.SizeDefault()));
28         }
29     }
30 }
```

group of measures we are creating (the entire aspect). Lines 5-8 add two measures, the number of messages and the message data exchanged. The number of messages (Line 6) is given the *key* "msg_num" to identify it, it is initialized to 0l, and uses the update function MEASURES.ADDLONG, which sums long numbers. Since we added the HASHMAP to count detailed messages counts, we override the `printAll` (Lines 10-16) which by default prints all added mesaures, and we simply print the map entries. Using the named pointcut `sendMessage` in COMMONS, we define the advice to increment our measures (Lines 17-30). First, we capture the message classname (Line 20), and increment the number of messages of that classname (Lines 20-25). We note that since the node runs multithreaded code, it is important to synchronize appropriately on the HASHMAP. Second, we update the number of messages by calling `update("msg_num", 1L)`, this calls the update function of the measure with the key "msg_num" and passes it "1" as argument. Since the update function adds long numbers, it will basically increment the count. Third, we perform a similar update on the data transferred by invoking a helper function which determines the size of the message. Thus, we have defined an aspect that computes the sum of messages and data transferred across an entire run of a monitoring algorithm. *

In the next section (Section 7.6.2), we discuss how to enable, disable, and store measures in THEMIS.

7.6.2 Managing Measures

Measures in THEMIS are aspects, as such they are instrumented into the system using the AspectJ compiler through a process called *weaving*. It is possible to *weave* aspects at compile time by recompiling the code with the aspects, or at load-time when classes are loaded in the JVM. THEMIS utilizes AspectJ load-time weaving (LTW) to incorporate measures during load-time.

Enabling and disabling measures. It is possible to enable or disable aspects by configuring the AspectJ load-time weaver. The weaver is configured using an xml file called `aop.xml` found under `META-INF` on the classpath. The file contains the aspects to weave, by overriding the file it is possible to specify which measures to use. It is also possible to exclude certain aspects using the xml tag `<exclude within="uga.corse.themis.somepackage.*"/>`.

Listing 14 Default weaver configuration.

```
1  <aspectj>
2    <aspect name="uga.corse.themis.utils.AutomataCache"/>
3
4    <aspect name="uga.corse.themis.measurements.Commons"/>
5    <aspect name="uga.corse.themis.measurements.RunHook"/>
6    <aspect name="uga.corse.themis.measurements.Instrumentation"/>
7    <aspect name="uga.corse.themis.measurements.DatabaseStore"/>
8
9    <aspect name="uga.corse.themis.measurements.basic.ExecutionHistory"/>
10   <aspect name="uga.corse.themis.measurements.basic.GlobalRun"/>
11   <aspect name="uga.corse.themis.measurements.basic.Simplifications"/>
12   <aspect name="uga.corse.themis.measurements.basic.Evaluations"/>
13
14   <aspect name="uga.corse.themis.algorithms.orchestration.MsgSize"/>
15   <aspect name="uga.corse.themis.algorithms.migration.MsgSize"/>
16   <aspect name="uga.corse.themis.algorithms.choreography.MsgSize"/>
17 </aspectj>
```

```
1  SELECT alg, comps, avg(msg_num), avg(msg_data), count(*)
2  FROM bench WHERE alg in ('Migration', 'MigrationRR')
3  GROUP BY alg, comps
```

	alg	comps	avg(msg_num)	avg(msg_data)	count(*)
1	Migration	3	2.04226336011177	267.8458714635	572600
2	Migration	4	2.16402472527473	668.129401098901	364000
3	Migration	5	3.33806822465134	3954.09705050886	530600
4	MigrationRR	3	32.7222301781348	482.572275585051	572600
5	MigrationRR	4	31.8533351648352	932.708425824176	364000
6	MigrationRR	5	19.2345269506219	4361.30746324915	530600

Figure 7.4: Querying the database to summarize and retreive measures using `sqlitebrowser`. Here we summarize the measures by grouping by algorithm and number of components. We report the message number and data transferred as well as the total number of executions summarized.

Listing 14 shows the default `aop.xml` file provided by THEMIS. Aspect AUTOMATACACHE is used to cache calls to monitor synthesis tools, while aspect DATABASESTORE (Line 7) is used to store measures in a database. Lines 4-6 introduce the aspects necessary to implement the rest of the measures or hooks. Lines 9-12 introduce the measures that are later used when comparing algorithms (Chapter 8). Lines 14-16 are used to define hooks that register handlers to compute the message size for each algorithm.

Storing measures. Aspect DATABASESTORE intercepts other aspects of type INSTRUMENTATION to retrieve their measures automatically and store them in a SQLite database [Hwa]. The name of the database file is provided in an environment variable `THEMIS_BENCH_DB`. In the database, a table is created (called `bench`) that has all measures as columns, the name of each column is associated with the *key* of the given measure. Aspect DATABASESTORE manages the creation and update of the table automatically to include measures. At the end of every monitoring run, a new record is added to the table with the value of each measure at the end of the run. Measures are also automatically printed at the end of a monitoring run, as such it is also possible to disable the database and rely on log files.

Processing measures. Measures are processed by either querying the database directly, or exporting the database as CSV and then providing it as input to other software like R. This allows third-party tools to visualize, plot and analyze the result of the measures. Figure 7.4 illustrates querying the database to summarize measures given unique algorithms and number of components in the system. In practice, we avoid summarizing information before the complete analysis. As such, we typically export the information as *csv* and analyze and plot it in R.

Decentralized monitoring algorithms are run on THEMIS nodes. Measures are typically woven on the THEMIS node, running the algorithm. In the next section (Section 7.7), we describe nodes and their design considerations.

7.7 Nodes and Runtime

At the backbone of the THEMIS framework lies the *node*. A node receives necessary information to deploy components (Section 7.3), and monitors (Section 7.5). The main task of a THEMIS node is to execute the monitoring logic. Therefore, it manages (1) gathering observations from components and sending them to monitors, (2) communication between monitors on the same node or other nodes, and (3) manages control signals to start and stop monitoring. In Section 7.7.1, we introduce a general overview of nodes. In Section 7.7.2, we show how to write a node. Finally, in Section 7.7.3 we illustrate how to execute nodes.

7.7.1 Overview of Nodes

A THEMIS node implements the main logic of monitoring with the addition of control logic to manage the monitoring. It is ultimately responsible of deploying monitors and components, and executing the monitoring logic of the monitors.

Monitoring logic. Nodes encode the high-level assumptions on the system, as they manage how observations are retrieved and passed to the monitors. We illustrate the default node implementation in Example 34.

Example 34 (NodeRounds monitoring logic.) Our node implementation NodeRounds manages the monitoring by rounds. For every round, it submits tasks to gather observations for each component asynchronously. As soon as observations are ready, it submits one task per monitor associated with the component to call method `monitor`, passing the observations. It waits for all monitors to execute their `monitor` method. This constitutes a monitoring round. The node then increments the timestamp counter and initiates a new round. The process continues until a monitor notifies the node of a verdict, aborts or a limit on the number of rounds is reached (timeout). *

Management. In addition to executing the monitoring logic, a node keeps track of the *platform*. A platform is a collection of addresses of all monitors and components. Since multiple nodes are able to coordinate to execute a monitoring algorithm in a distributed fashion, nodes ultimately manage the communication between the various monitors across nodes. We recall that monitors only communicate by providing the *id* of the monitor they wish to send a message to. Nodes are controlled via *commands*. A command is a control message sent to the node instructing it to perform a specific operation. The logic of a command is implemented via the interface CmdRuntime. Component and monitor deployment, as well as, starting, stopping and restarting monitoring are all implemented as commands. Communication between monitors across nodes is implemented with commandMonitorPayload that wraps a monitor message. Implementing a THEMIS node allows programmers to fine tune the assumptions on the monitoring execution and simulation they wish to have, without modifying the underlying monitoring code (for the algorithms) or dealing with the trace.

In the next section (Section 7.7.2), we illustrate the interface, and the main methods needed to implement a node.

7.7.2 Writing a Node

The interface describing the necessary behavior to be implemented by a node is presented in Listing 15. We break it down 3 groups: management objects, monitoring control, and observers.

Listing 15 Interface of a THEMIS NODE.

```
1  public interface Node  {
2      //Information managed by a node (setters)
3      void        setAddress(String addr);
4      void        addComponent(Component comp);
5      boolean     attachMonitor(String comp, Monitor mon);
6      boolean     changePlatform(Platform platform);
7      void        setDispatcher(Dispatcher disp);
8      void        setNetwork(Network net);
9      //... (getters)
10     //Retreive monitoring related information
11     Integer                 getMonitorCount();
12     Integer                 getComponentCount();
13     Component               getComponentForMonitor(Integer integer);
14     Set<Monitor>            getMonitorsForComponent(String componentKey);
15     Map<String, Component>  getManagedComps();
16     Map<Integer, Monitor>   getManagedMonitors();
17     //Main monitoring functionality
18     void        start(Map<String, Serializable> tags);
19     void        stop();
20     void        reset();
21     boolean     isStarted();
22     //Observers on nodes used for executing additional code unrelated
23     //to the monitoring logic
24     int         observerRegister(NodeObserver obs);
25     void        observerRemove(int id);
26 }
```

Management objects. The first group (Lines 2-16) consists of methods for managing the data that surrounds the monitoring logic. It includes the address given to the node, the functionality to add components, attach monitors, and manage the platform, the dispatcher, and the network. A dispatcher is responsible for sending and receiving commands (explained in Section 7.7.1). The default dispatcher (SOCKETS) notification utilizes sockets and serializes objects to communicate across nodes. The network represents the abstract communication model for managing communication between monitors. The default network (NETWORKDISTRIBUTED) models communication by associating a queue for each monitor. As such when sending a message to another monitor, the message is appended to its queue, when receiving a message the queue is popped. When a monitor is not found on the local node, the network queries the node to retrieve the address of the node hosting the monitor, and issues a command to the node with the message payload.

Monitoring control. The second group (Lines 17-21) contains the main logic for monitoring a node must implement. It contains methods `start`, `stop`, `restart`, and `isStarted`. Method `start` is the method that initiates the monitoring. It contains the logic to gather observations and invoke monitors, until completion. Method `stop` is used to interrupt the monitoring on a node (for example, upon receiving an abort), when stopping the node does not clear its state. Method `reset` is used to stop monitoring and clear the node state. When clearing the state, the node removes all information about the deployed components and monitors, and will be available to receive a new deployment and monitoring algorithm. It is important to note that upon normal completion of the monitoring algorithm, the node performs a reset.

Observers. The third group (Lines 24-25) consists of registering and removing *observers*. Node observers manage additional event handling code to be performed on the node. It allows clients to interact with the node as its state changes. For example, the observer DISPATCHEND notifies a client when a node stops. This allows a client to know that the monitoring has ended. Observers also provide an extension to the node logic that is independent from the monitor logic or assumptions.

7.7.3 Running a Node

Once a node is written it consistutes the server-side of THEMIS. To initialize a node, we invoke the RUNTIME tool, using `java -javaagent:/path/to/themis.jar uga.corse.themis.Runtime host port nodeclass` or with

the script `themis-node host port nodeclass`. The Java agent includes THEMIS on the classpath and utilizes the AspectJ weaver to weave necessary measures at load-time during the execution (see Section 7.6.2). The parameters passed include the host and port to use for communicating with the node, and the class (implementation) to use as a node. The third argument is optional and by default references NodeRounds. To run a custom node, it suffices to add its implementation on the classpath, and provide its full class name to the Runtime tool.

In the current section, we elaborated on how to design and deploy nodes. In the next section (Section 7.8), we elaborate on clients that interact with nodes to generate and deploy monitors.

7.8 Using Tools to Perform Monitoring

After elaborating on the approach to design algorithms (Section 7.5), and measures to assess them (Section 7.6), we now discuss the THEMIS tools designed to execute the decentralized monitoring algorithms. We first introduce performing a single monitoring run in Section 7.8.1. Then we elaborate on the Experiment client useful for creating reproducible experiments in Section 7.8.2. Finally, we elaborate on some other tools provided to help design, visualize and debug algorithms in Section 7.8.3. The tools presented in this section are used in both Chapter 8 to compare the algorithms presented in Section 1.3.1, and Chapter 9 to monitor a smart apartment.

7.8.1 Running a Monitoring Algorithm

We recall that THEMIS operates in client-server mode, where nodes (Section 7.7) are the servers, and tools are built using clients that interact with nodes. Additionally, clients are tasked with executing the *setup* phase of a monitoring algorithm, wherein the network of monitors is created.

General use-case. Clients typically have to manage and deploy components and monitors on nodes. To that end, multiple helper classes are provided to aid in that task. We introduced in Section 7.2.3 the class SpecLoader, which helps manage loading and initializing decentralized specifications. Upon loading the specification, the client must load the component information. The client never initializes components, but summarizes information about the platform and components (their class and peripheries). After determining the platform and the specification, the client invokes the *setup* phase of an algorithm (Section 7.5.1), which initializes a set of monitors. The client relays the information of the platform, components, and monitors to one or more THEMIS node(s). Once nodes are configured, the client sends the nodes a signal to begin monitoring and waits on notifications.

Deployment. Since most clients rely on the same code to communicate with nodes, the helper class Deploy is provided to simplify interaction with nodes. It provides three static methods to deploy components, monitors, and begin monitoring. In addition to sending data to nodes, clients often have to wait on the node to notify them. This is the case when the client is waiting on the monitoring to finish. A simple server (SemaphorePingback) is provided, that takes a semaphore and releases it when the node reports completion or failure of the monitor. This semaphore is usually used to block the main thread, until the node completes its execution.

Basic client. THEMIS provides a basic tool that accomplishes the general use-case. The Run tool performs one or more execution of the same decentralized monitoring algorithm with the same decentralized specification on multiple traces. The decentralized trace follows the convention explained in Section 7.3.2. The tool is called using:

```
themis-run -nc <INT> -nr <INT> -in <PATH> \
  -spec <PATH> -alg <CLASS> -tid <INT> -tmax <INT> \
  -node <NODE> -listen <NODE>
```

Its arguments are as follows:

- `nc` indicates the number of components in the system;
- `nr` indicates the length of the run (after which to timeout);

- `in` indicates the input path of the folder containing the decentralized trace;
- `spec` indicates the XML file containing the decentralized specification;
- `alg` provides the full classname of the BOOTSTRAP class of the algorithm to execute;
- `tid` id of the trace to run;
- if `tmax` is provided, then all traces with id in the range [`tid`, `tmax`] are executed;
- `node` contains the node address as a string "host:port" to deploy on; and
- `listen` contains the address of the server listening to the completion message.

7.8.2 Experiments

When multiple runs are required with fixed specifications and traces, `THEMIS` provides the EXPERIMENT tool. An *experiment* is a bundle of configuration parameters and a collection of traces, decentralized specifications, and decentralized monitoring algorithms. It provides a uniform way to execute a full experiment with `THEMIS` to compare, gather metrics, and explore the behavior of decentralized monitoring algorithms. The experiment is a two step process. The first step is a verification phase that runs a test on each provided decentralized specification. The second step is an execution phase that runs the provided algorithms on the traces for each specification.

Verification phase. During the verification phase, the client loads each specification (specified as a list in a text file), and runs a test on it. The test is defined using interface TESTORACLE, which provides one method `verify`. Method `verify` takes as input a decentralized specification and returns a boolean indicating the specification has passed or failed the test. If all specifications pass the test, the client moves to the next phase. If at least one specification fails the test, two new files are generated with the extensions `.keep` and `.discard`. They indicate respectively the specifications that passed and failed the test. An example test to verify synthesis and monitorability of LTL formulae is provided in Example 35.

EXAMPLE 35 (TESTLTL) Oracle TESTLTL considers a centralized LTL specification, it extracts the formula, synthesizes the automaton, and verifies if it is monitorable. It returns $\perp$ if the automaton cannot be synthesized (monitor synthesis tool times out), or the automaton is not monitorable (Section 6.2.1). $*$

Execution phase. After verifying that all specifications comply to the test, the client begins executing the traces for each algorithm for every specification. The client EXPERIMENT is capable of using multiple nodes to run the experiment, it splits the traces to monitor for one algorithm evenly among all nodes.

Configuration. To configure an experiment, we rely on a main (`.properties`) configuration file. Algorithms and specifications are stored each in a text file, where each line indicates respectively the BOOTSTRAP classname of an algorithm to run (Section 7.5.1), and the path to a decentralized specification to load (Section 7.2). Listing 16 shows a configuration file for an experiment. We first state that we want to execute the verification phase (Line 1), and generated the files with the specifications that passed and failed the test (Line 2). Then, we provide the test oracle (Line 3). To configure traces, we define the trace directory (Line 4), and the number of traces to read (Line 5). By setting the number of traces to 100, we read all traces with id in the range $[0, 99]$. For more details on the convention adopted for traces see Section 7.3.2. To configure the run, we specify the length of the run (Line 6), after which to timeout. The number of components to initialize (Line 7), this must be inferior or equal to the number of components for the trace. The file containing the list of specifications (Line 8), and algorithms (Line 9). To configure the node information, we first specify the nodes to use for the experiment (Line 10). In this case, we use two nodes. Then, we specify the notification address (Line 11), to use for the node to notify the client when monitoring completes. Finally, we can add any additional environment variables that could be useful. For this experiment, we specify the variable `THEMIS_CACHE_LTL` (Line 12), which defines the path of the cache for the synthesized automata from LTL (Section 7.3.2).

Listing 16 Experiment configuration file.

```
tests.skip=false
tests.discard=true
tests.oracle=uga.corse.themis.tools.experiment.TestLTL
traces.dir=/path/to/traces/
traces.maximum=100
run.length=110
run.components=3
run.specs=specs.txt
run.algorithms=../algs.txt
node.targets=localhost:8056,localhost:8057
node.host=localhost:8091
THEMIS_CACHE_LTL=cache
```

Bundling. Experiments are bundled in a simple folder, which can be compressed to form a bundle of traces, algorithms, and specifications. We note that the measures are defined on the node, and not in the experiment. While this may seem counter-intuitive, we recall that the node is instrumented at load-time Section 7.6.2. This means that measures are defined when the node is executed. In which case, it suffices to provide a simple `Makefile` that initializes the node. The command could add to the classpath necessary custom measures, and define its own `aop.xml` which includes which measures to use and exclude. The necessary sources of the added measures or algorithms could be bundled as a `jar` in the experiment folder, and added to the classpath appropriately. Since all measures are stored in a database, the experiment folder can also include necessary scripts to query, dump, and process the outcome of the measures. For example, it can include additional `R` scripts to analyze and plot the data.

Re-using and extending. Experiments can be re-used and re-run easily as they are self-contained in a folder. Furthermore, it is easy to add additional algorithms to test, it amounts to ensuring its source is on the classpath, and its BOOTSTRAP implementation is added as a new entry to the text file listing the algorithms. Similarly, by instrumenting new measures on the node itself, an experiment can be run to investigate or compare the existing algorithms under the new measures. This ensures that minimal effort is needed to re-run existing experiments with minor modifications.

While writing algorithms, measures, and designing experiments is error-prone, additional tools are provided to ease the tasks. We introduce them in the next section (Section 7.8.3).

7.8.3 Utility Tools

In addition to providing the core functionality for performing monitoring, the `THEMIS` framework provides additional tools to assist programmers in designing algorithms, or setting up experiments. In this section, we cover three tools: visualization of automata, generation of random traces, and generation of random specifications.

Automata visualization. Since an automaton is a graph, we utilize the Graphstream [PDGO08] graph library to draw automata. The interface provided is found under the `themis-ui` package, using the tool DRAW, and the helper class UI. The drawing tool can be directly invoked from the command-line to draw the synthesized automata specifications found in a decentralized specification file (Section 7.2.3). The tool provides additional options to simplify the expressions, select a subset of the specifications for drawing, and synthesize and draw directly a provided LTL formula. The helper class can be also directly used from within a program using the static method `draw` which takes an arbitrary automaton and draws it. Figure 7.5 illustrates an example output of the tool with simplified expressions.

Trace generation. `THEMIS` provides the tool GENERATOR used to generate random traces using COLT [CER99]. COLT provides a set of open source libraries for high performance scientific and technical Computing in Java. In particular, COLT provides implementation for random engines and probability distributions. The generator tool relies on a *COLT profile* file to decide which engine and probability distribution to use in COLT, along with the parameters to initialize the distribution. An example COLT profile is shown in Listing 17. We use the DRAND

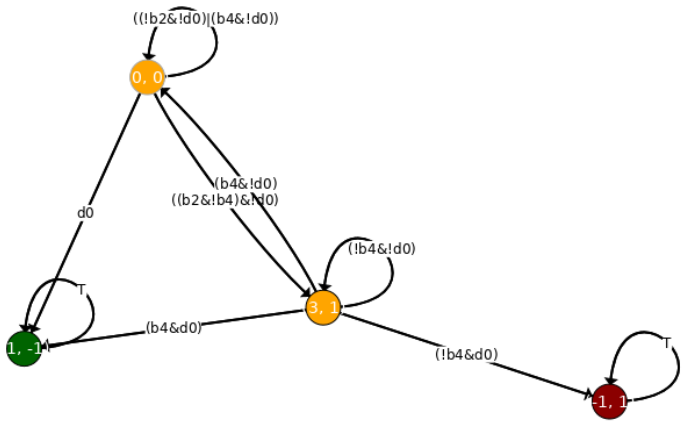


Figure 7.5: A drawn automaton using Graphstream.

Listing 17 COLT profile for binomial distribution.

```
<generator>
        <engine seed="" type="NOW">cern.jet.random.engine.DRand</engine>
        <distribution cls="cern.jet.random.Binomial">
                <param type="INT">100</param>
                <param type="DOUBLE">0.3</param>
        </distribution>
</generator>
```

engine from COLT, seeded with the current time (NOW). We define the distribution to be BINOMIAL with the two parameters $n = 100$ and $p = 0.3$. Using the COLT library, observations are generated by setting the $\bot$ verdict threshold at 0.5, values less or equal to 0.5 are associated with $\bot$ while those greater than are associated with $\top$. The environment variable VERDICT_INVERT can be set to "true" to invert the verdicts. We invoke the generator tool using the following script:

`themis_gentrace ntraces trace_length ncomps nobspercomp profile out [keep]`

The parameters are as follows:

- `ntraces` indicates the number of traces to generate;
- `trace_length` indicates the length of the trace;
- `ncomps` denotes the number of components in the trace;
- `nobspercomp` denotes the number of observations to generate per component;
- `profile` is a path to the COLT profile file;
- `out` is the output folder under which to store the trace; and
- `keep` is an optional argument used to keep the folder as by default the output folder is compressed and deleted.

Specification generation. `THEMIS` provides scripts to generate random LTL specifications by invoking the `randltl` tool from SPOT [DL13]. The command to generate multiple specifications is as follows:

`themis_genspec nspecs ncomps nobs template out [keep]`

The parameters are as follows:

- `nspecs` denotes the number of specifications to generate;
- `ncomps` denotes the number of components to reference in the specification;
- `nobs` is the number of observations per component to reference;
- `template` is the template file, a decentralized specification xml file which contains the string "%form%", the string is replaced with the generated formula by `randltl`;
- `out` is the output folder where specifications are written; and
- `keep` is an optional argument used to keep the folder as by default the output folder is compressed and deleted.

THEMIS Module	Functionality	Tool/Library
Specifications	Monitor synthesis	LamaConv [Ins]
Specifications	Monitor synthesis	`ltl2mon` [BLS11]
`EHE`	Boolean and LTL simplification	`ltlfilt` (SPOT) [DL13]
Measures	Storage/Querying	SQLite3 [Hwa]
Measures	Instrumentation	AspectJ [KHH+01a]
Random specification generation	Random LTL	`ltlrand` (SPOT) [DL13]
Random trace generation	Randomness	COLT [CER99]
Automata visualization	Graph visualization	`Graphstream` [PDGO08]

Table 7.1: Tools and libraries used by `THEMIS`.

Conclusion

We presented `THEMIS`, a modular framework for the design, analysis, and simulation of decentralized monitoring algorithms. We introduced an overview of the architecture and details to write decentralized specifications, algorithms that utilize the provided data structures. We also elaborated on the various possibilities to incorporate observations to feed them to monitors. Futhremore, we presented in detail the client-server of `THEMIS` by defining nodes and clients that deploy components and monitors on nodes. In particular, we explained the client EXPERIMENT which allows developers to write and bundle experiments that utilize `THEMIS`. Finally, we illustrated some of the additional tools for generating traces, specifications and visualizing automata. The modular aspect of `THEMIS` allows it to interact with existing tools (summarized in Table 7.1) to provide a common workflow for their integration into one monitoring framework.

In the following two chapters, we illustrate how to utilize `THEMIS` to evaluate existing decentralized monitoring algorithms (Chapter 8), and second to apply decentralized specifications in the context of smart homes (Chapter 9).

Part II

Applications

CHAPTER 8

Comparing Decentralized Monitoring Algorithms

"How can we use decentralized specifications to compare existing decentralized monitoring algorithms?"

Contents

Chapter abstract

In this chapter, we aim to compare different decentralized monitoring algorithms in terms of computation, communication, and memory overhead. We elaborate on the general phases of decentralized monitoring algorithms and illustrate the approach to analyze them by adapting algorithms from [CF16a] as examples. The chosen algorithms are examples of different strategies used for decentralizing monitoring. In brief, we analyze the (worst-case) behavior of each algorithm by looking at its usage of the data structures (Memory and `EHE`). Then, using `THEMIS` (Chapter 7), we look at the usefulness of simulations to determine the advantages or disadvantages of certain algorithms in two scenarios. The first scenario explores synthetic benchmarks, that is, we consider random traces and specifications. This allows us to explore various levels of coverage. The second scenario explores a common pattern in programming: publish-subscribe. For that, we consider a publish-subscribe system, where multiple publishers subscribe to a channel (or topic), the channel publishes events to the subscribers. We use the Chiron user interface example [ACD+99, Tea99], along with the specifications formalized for it [DAC99a].

Introduction

Motivation. In Chapter 3, we introduced the various approaches of decentralized monitoring. Decentralized monitoring algorithms are primarily designed to address one issue at a time and are typically experimentally evaluated by considering runtime and memory overheads. However, such algorithms are difficult to compare as they may combine multiple approaches at once. For example, algorithms that use LTL rewriting [BF12, CF16a, RH05] (Section 3.1) not only exhibit variable runtime behavior due to the rewriting, but also incorporate different monitor synthesis approaches that separate the specification into multiple smaller specifications. Such techniques start from a global specification and then synthesize local monitors with either a copy of the global specification [BFRT16] or a completely different specification to monitor (typically a subformula of the original formula) [BF12, FCF14]. In this thesis, we referred to the former as a centralized specification and to the latter as a decentralized specification. These different approaches of synthesis are separate from monitoring and their evaluation is of interest.

Utility. Decentralized specifications (Chapter 6) allow us to split the problem of generating equivalent decentralized specifications from a centralized one (synthesis) from the problem of monitoring by defining two phases for decentralized monitoring: the *setup*, and *monitor* phases. By examining the topology of the monitors created after the *setup* phase, the usage of the data structures Memory and `EHE`, and the *monitor* phase of algorithms, we are able to analyze and identify scenarios that advantage certain strategies over others. In Chapter 7, we introduced the `THEMIS` tool which allows us to execute the different algorithms and gather different metrics. By adapting the existing algorithms in `THEMIS`, we extends the work of [CF16a] and look at the usefulness of simulations to determine the advantages or disadvantages of certain algorithms in two scenarios. The first is based on randomly generated traces and specifications, and the second utilizes the Chiron user-interface application with a provided formal specification.

Chapter organization. In Section 8.1, we present an analysis of the behavior of three different decentralized monitoring algorithms by looking at their usage of the data structures Memory and `EHE`. We first consider the parameters and the cost for the basic functions of the `EHE` and *memory* data structures in Section 5.2. Then, we analyze each of the presented decentralized monitoring algorithms in Section 8.1. In Section 8.2, we elaborate on the metrics and experimental setup used to simulate the three algorithms. Then, we compare the decentralized monitoring algorithms using the synthetic benchmark (Section 8.2.3) and Chiron (Section 8.2.4). Finally, we conclude in Section 8.2.4 and present future perspectives.

Key contributions. The key contributions of this chapter can be summarized as follows:

1. Adapting existing decentralized monitoring algorithms to our approach (API and data structures), in a uniform manner for the purpose of analysis and simulation;

2. Parametrizing the analysis of decentralized monitoring algorithms with information delay incurred by partial observations;

3. Performaing worst-case analysis of decentralized monitoring algorithms relying on automata-based approaches, utilizing boolean rewriting (with a known minimal form) instead of LTL rewriting;

4. Simulating three decentralized monitoring algorithms (and comparing variants of the same algorithm) in a synthetic benchmark and a user interface application to validate the analysis;

5. Exploring the effect of using different random probability distributions for trace generation, and their impact on coverage; and

6. Exploring the performance of various decentralized monitoring algorithms for different patterns of specifications [DAC99a] of the same application.

8.1 Analyzing Existing Algorithms

We aim to compare decentralized monitoring algorithms in terms of computation, communication, and memory overhead. We recall that the EHE and *memory* data structures are used to abstract the behavior of a monitoring algorithm. Their size and parameters have been identified in Section 5.2. We elaborate on the general phases of decentralized monitoring algorithms and illustrate the approach to analyze them by adapting algorithms from [CF16a] as examples.

8.1.1 Overview and General Approach

We shift the focus to the algorithms and their usage of the data structures. We first present an overview of the abstract phases performed by decentralized monitoring algorithms. We then elaborate on our approach to model their behavior. Finally, as an example, we present the analysis for each of the algorithms adapted from [CF16a] (presented in Section 1.3.1): Orchestration (Orch), Migration (Migr), and Choreography (Chor). The algorithms contain both multiple monitors monitoring the same specification (Orch, and Migr), and a decentralization algorithm which splits one global specification to multiple subspecifications distributed on monitors (Chor). We later explore the trends provided by the analysis by benchmarking in Section 8.2.

Overview. A decentralized monitoring algorithm consists of two phases: setting up the monitoring network, and monitoring. In the first phase, an algorithm initializes the monitors, defines their connections, and attaches them to the components. We represent the connections between the various monitors using a directed graph $\langle \text{Mons}, E \rangle$ where $E \subseteq 2^{\text{Mons} \times \text{Mons}}$ defines the edges describing the sender-receiver relationship between monitors. For example, the network $\langle \{\mathrm{m}_0, \mathrm{m}_1\}, \{\langle \mathrm{m}_1, \mathrm{m}_0 \rangle\} \rangle$ describes a network consisting of two monitors m_0 and m_1 where m_1 sends information to m_0. In the second phase, an algorithm proceeds with monitoring, wherein each monitor processes observations and communicates with other monitors.

We consider the existing three algorithms: Orchestration, Migration and Choreography [CF16a] adapted to use EHE. We note that these algorithms operate over a global clock, therefore the sequence of phases can be directly mapped to the timestamp. We choose an appropriate encoding of *Atoms* to consist of a timestamp and the atomic proposition ($\mathit{Atoms} = \mathbb{N} \times \mathit{AP}$). These algorithms are originally presented using an LTL specification instead of automata, however, it is possible to obtain an a Moore automaton equivalent to the specification as described in [BLS11].

Approach. A decentralized monitoring algorithm consists of one or more monitors that use the EHE and memory data structures to encode, store, and share information. We recall from Section 5.2 that the cost of utilizing the data structures and their size depends on (1) information delay (δ_t) which parametrize the time needed to resolve partial observations, (2) the number of states in the automaton ($|Q|$), and (3) the maximum size of an expression labeling a transition in the automaton (L). We recall that the notation $|\mathcal{I}^t|$ denotes the size of the EHE at timestamp t. Therefore, by studying δ_t, we derive the size of the EHE and the memory a monitor would use. Knowing the sizes, we determine the computation overhead of a monitor, since we know the bound on the number of simplifications a monitor needs to make ($\delta_t|Q|$), and we know the bounds on the size of the expression to simplify ($\delta_t L$). Once the cost per monitor is established, the total cost for the algorithm can be determined by aggregating the costs per monitors. This can be done by summing to compute total cost or by taking the maximum cost in the case of concurrency following the Bulk Synchronous Parallel (BSP) [Val90] approach.

8.1.2 Orchestration

As explained in Section 7.5.1, the orchestration algorithm (Orch) consists in setting up a main monitor which will be in charge of monitoring the entire specification. However since that monitor cannot access all observations on all components, orchestration introduces one monitor per component to forward the observations to the main monitor. Therefore, for our setup, we consider the case of a main monitor m_0 placed on component c_0 which monitors the specification and $|C| - 1$ forwarding monitors that only send observations to m_0 (labeled m_k with $k \in [1, |C| - 1]$). We consider that the reception of a message takes at most d rounds. The information delay δ_t is then bounded by a

constant, $\delta_t \leq \mathrm{d}$. The number of messages sent at each round is $|C| - 1$, i.e., the number of forwarding monitors sending their observations. The size of a message is linear in the number of observations for the component, for a forwarding monitor labeled with m_k, the size of the message is $|\mathrm{tr}(t, c_k)| \times (\mathrm{s}_{\mathbb{N}} + \mathrm{s}_{AP} + \mathrm{s}_{\mathbb{B}_2})$, where $|\mathrm{tr}(t, c_k)|$ represents the number of observations for the component on which a monitor is attached, and $\mathrm{s}_{\mathbb{N}} + \mathrm{s}_{AP} + \mathrm{s}_{\mathbb{B}_2}$ denotes the size of an observation associated with a given timestamp (i.e. memory entry).

8.1.3 Migration

The migration algorithm (Migr) initially consists in rewriting a formula and transferring the result of the rewriting from one or more component to other components to fill in missing observations [CF16a]. We call the monitor rewriting the formula the active monitor. Our EHE encoding guarantees that two monitors receiving the same information are in the same state (see Section 5.1.2). Therefore, monitoring with Migration amounts to rewriting the EHE and migrating it across components. Since all monitors can send the EHE to any other monitor, the monitor network is a strongly-connected graph. In Migr, the delay depends on the choice of function choose, which determines which component to migrate to next upon evaluation. By using a simple function choose, which causes a migration to the component with the atom with the smallest timestamp, it is possible to view the worst case as an expression where for each timestamp we depend on information from all components, therefore $|C| - 1$ rounds are necessary to get all the information for a timestamp ($\delta_t = |C| - 1$). We parametrize Migr by the number of active monitors at a timestamp m. Function choose in [CF16a] selects exactly one component if there is a need for communication and zero if there is no need. Therefore, after the initial choice of m, subsequent rounds can have at most m active monitors.

We illustrate Migr in Algorithm 2. The state of a migration monitor consists of a variable isActive that determines whether or not the monitor is active, and $\mathcal{I}$ that is an EHE encoding the same automaton shared by all monitors. At each round the monitor receives a timestamp t and a set of observations o. Line 2 displays the memory update with observations for that round. Lines 3 to 9 describe the reception of EHEs from other monitors. Upon receiving an EHE, the monitor state is set to active (Line 7). An active monitor will then update its EHE by first ensuring that it is expanded to the current timestamp using mov (Line 11), then rewriting and evaluating each entry (Lines 12-17). The number of entries in the EHE depends on δ_t. If any of the entries is evaluated to a final verdict (Line 14), then the verdict is found and we terminate. While the verdict is not found, the migration algorithm first removes all unnecessary entries in the EHE (Line 18). Unnecessary entries are entries for which the state is known, the last known state is only kept, entries for all previous timestamps are removed. After removing unnecessary entries, we determine a new monitor to continue monitoring using the function choose (Lines 19-22). The initial choice of active monitors is bounded by $m \leq |C|$. Since at most $m - 1$ other monitors can be running, there can be $(m - 1)$ merges. In the worst-case the information delay requires the formula to be passed around to all components, as such $\delta_t = |C| - 1$. The size of the resulting EHE is $m \times |\mathcal{I}_t^{\delta}| = m(|C| - 1)^2|Q|L$. In the worst case, the upper bound on the size of EHE is $(|C| - 1)^3|Q|L$. The number of messages is bounded by the number of active monitors m. The size of each message is however the size of the EHE, since Migr requires the entire EHE to be sent.

8.1.4 Choreography

Choreography (Chor) presented in [CF16a] splits the initial LTL formula into subformulas and delegates each subformula to a component. Thus Chor can illustrate how it is possible to monitor decentralized specifications. We now consider the setup and monitor phases of Chor.

Setup. Choreography begins by taking the main formula, then deciding to split it into subformulae. Each monitor will monitor the subformula, notify other monitors of its verdict, and when needed *respawn* . Recall from the definition of Δ' (see Definition 17), that monitoring is recursively applied to the remainder of a trace starting at the current event. That is, initially we monitor from e_0 to e_n and then from e_1 to e_n and so forth. To do so, it is necessary to reset the state of a monitor appropriately, this process is called in [CF16a] a *respawn*. Once the subformulae are determined, we generate an automaton per subformula to monitor it. Then, we construct the network of monitors in the form of a tree, in which the root is the main monitor. Verdicts for each subformula are then propagated in the hierarchy until a verdict can be reached by the root monitor.

A choreography monitor is a tuple $\langle id, \mathcal{A}_{\varphi_{id}}, ref_{id}, coref_{id}, respawn_{id} \rangle$ where:

Algorithm 2 Migration

```
1:  procedure MIGRATION(t, o)
2:      M ← M †2 memc(o, ts_t)                          ▷ Add observations to memory
3:      while Received I′ do                            ▷ Received an EHE from another monitor
4:          if isActive then                            ▷ If the monitor is active, the monitor EHE has information
5:              I ← I †∨ I′                             ▷ Merge information with existing information
6:          else
7:              I ← I′; isActive ← ⊤                    ▷ Monitor becomes active after it receives an EHE
8:          end if
9:      end while
10:     if isActive then
11:         t′ ← getEnd(I); I ← mov(I, t′, t)           ▷ Build EHE up to current timestamp
12:         for each t_v ∈ dom(I) do                    ▷ Go through all EHE timestamps
13:             v ← verAt(I, M, t_v)                    ▷ Evaluate the entries associated with timestamp t_v
14:             if v ∈ B_2 then                         ▷ Found a final verdict
15:                 Report v and terminate
16:             end if
17:         end for
18:         I ← dropResolved(I)                         ▷ Purge EHE of non-needed entries
19:         c_k ← choose(I)                             ▷ Determine the next component
20:         if c_k ≠ c then                             ▷ Is the next component not local
21:             isActive ← ⊥; Send I to m_k             ▷ Send to relevant monitor, stop monitoring
22:         end if
23:     end if
24: end procedure
```

- id denotes the monitor unique identifier (label);
- $\mathcal{A}_{id}$ the automaton that monitors the subformula;
- $ref_{id} \subseteq 2^{\text{Mons}}$ the monitors that this monitor should notify of a verdict;
- $coref_{id} \subseteq 2^{\text{Mons}}$ the monitors that send their verdicts to this monitor;
- $respawn_{id} \in \mathbb{B}_2$ specifies whether the monitor should *respawn*;

To account for the verdicts from other monitors, the set of possible atoms is extended to include the verdict of a monitor identified by its id. Therefore, $Atoms = (\mathbb{N} \times AP) \cup (\text{Mons} \times \mathbb{N})$. Monitoring is done by replacing the subformula by the id of the monitor associated with it.

Before splitting a formula, it is necessary to determine the component that hosts its monitor. The component score is computed by counting the number of atomic propositions associated with a component in the subformula [CF16a].

$$
\begin{aligned}
&\text{scor}(\varphi, c) : LTL \times C \to \mathbb{N} \\
&\quad = \texttt{ match } \varphi \texttt{ with} \\
&\quad \mid a \in AP \quad \to \begin{cases} 1 & \text{if lu}(a) = c \\ 0 & \text{otherwise} \end{cases} \\
&\quad \mid \text{ op } \phi \quad \to \text{scor}(\phi, c) \\
&\quad \mid \phi \text{ op } \phi' \quad \to \text{scor}(\phi, c) + \text{scor}(\phi', c)
\end{aligned}
$$

The chosen component is determined by the component with the highest score [CF16a], using $\text{chc} : LTL \to C$:

$$\text{chc}(\varphi) = \underset{c \in C}{\text{argmax}}(\text{scor}(\varphi, c))$$

In order to setup the network of monitors, firstly the LTL expression is split into subformulae and the necessary monitors are generated to monitor each subformula. The tree of monitors is generated by recursively splitting the formula at the binary operators. We present the *setup* phase as a tree traversal of the LTL formula to generate the monitor network, merging nodes at each operator, which is a different flavor of the recursive generation procedure in [CF16a]. Given the two operands, we choose which operands remains in the host component, and (if necessary) which would be placed on a different component. Therefore, we add the constraint that at least one part of the LTL expression must still remain in the same component. Given two formulas φ and φ' and an initial base component c_b

we determine the two components that should host φ and φ' with the restriction that one of them is c_b:

$$c_1 = \text{chc}(\varphi), \quad c_2 = \text{chc}(\varphi')$$
$$s_1 = \text{scor}(\varphi, c_b), \quad s_2 = \text{scor}(\varphi', c_b)$$
$$\text{split}(\varphi, \varphi', c_b) = \begin{cases} \langle c_b, c_b \rangle & \text{if } c_1 = c_2 = c_b \\ \langle c_1, c_b \rangle & \text{if } (c_1 \neq c_b) \\ & \quad \wedge (c_2 = c_b \vee s_2 > s_1) \\ \langle c_b, c_2 \rangle & \text{otherwise} \end{cases}$$

Algorithm 3 displays the procedure to split the formula. We define the needed information to create monitors: a unique identifier, and an LTL formula. Monitor data is a pair $\langle id, spec \rangle$ that represents respectively the id of the monitor and the formula that it monitors. We next construct a tree of monitor data. For each binary operator, we determine which of the operands needs to be hosted in a new component. The result is a tuple: $\langle \text{root}, \text{N}, \text{E} \rangle$ where:

- root is the root of the tree;
- N is the set of generated monitor data nodes;
- E is the set of edges between the various nodes (using only ids).

We use the dot notation root.spec to refer to the formula associated with the root node. The generation procedure is as follows.

- First, chc determines the `host` component where the root monitor resides (Line 3).
- Second, the AST of the *LTL* formula is traversed using netx, which splits on binary operators (Line 4).
  - When the formula is an atomic proposition, we simply generate a monitor that monitors the atomic proposition (Lines 8-10).
  - When both formulae can be monitored with the same monitor, it does not split (Line 17-21).
  - Otherwise
    1. We recurse on the side kept, to further split the formula (Lines 24,31);
    2. We recurse on the side split, with a new *host* and *id* (Lines 25,30);
    3. We merge the subnetworks by: (Lines 27,33)
       (a) Generating the host monitor with the formula resulting from the recursion;
       (b) Connecting the split branch's root monitor to the current host monitor;
       (c) Adding the split branch's root monitor to the set of additional monitors;
       (d) Merging the set of additional monitors and edges from both branches.

Once the monitor data tree is created, monitors are created accordingly, generating an automaton for the subformula, where some of its atomic propositions have been replaced with monitor ids. Each monitor is initialized with the refs and corefs sets based on the edges setup.

Remark 4 (Compacting the network) The monitor network can further be compacted as follows; monitors with the same subformula are merged into one, and their refs and corefs will be the result of the set union. However, one or more merged monitor will have to replace all occurrences of the id of the other monitors in all subformulae of all monitors. *

Monitor. Once the subformulae are determined by splitting the main formula, we adapt the algorithm to generate an automaton per subformula to monitor it. To account for the verdicts from other monitors, the set of possible atoms is extended to include the verdict of a monitor identified by its id. Therefore, $Atoms = (\mathbb{N} \times AP) \cup (\text{Mons} \times \mathbb{N})$. Monitoring is done by replacing the subformula by the id of the monitor associated with it. Therefore, monitors are organized in a tree, the leaves consisting of monitors without any dependencies, and dependencies building up throughout the tree to reach the main monitor that outputs the verdict. Since each monitor is in charge of

Algorithm 3 Setting up the monitor tree

procedure $NET\_CHOR(\varphi, C)$
  $id \leftarrow 0$ ▷ Start with id 0.
  $c_h \leftarrow \text{chc}(\varphi)$ ▷ Choose the starting component.
  $\langle root, mons, edges\rangle \leftarrow netx(\varphi, id, c_h)$ ▷ Split the formula recursively.
  return $\langle \{root\} \cup mons, edges\rangle$
end procedure
procedure $netx(\varphi, id_c, c_h)$ ▷ Split a formula, given placed on a component
  if $\varphi \in AP$ **then** ▷ Reached a leaf, don't split.
    $m \leftarrow \langle \varphi, id_c\rangle$
    return $\langle m, \emptyset, \emptyset\rangle$ ▷ No additional nodes or edges needed.
  else if φ matches op e **then** ▷ Unary operator requires no split.
    $\langle \text{root}, \text{N}, \text{E}\rangle \leftarrow \text{netx}(e, id_c, c_h)$ ▷ Process the subformula recursively.
    $m \leftarrow \langle \text{op root.spec}, id_c\rangle$ ▷ Replace root monitor to apply the unary operator to the resulting formula.
    return $\langle m, \text{N}, \text{E}\rangle$ ▷ Propagate back to parent, with the resulted edges and nodes.
  else if φ matches e op e' **then** ▷ We decide to split only on binary operators.
    $\langle c_1, c_2\rangle \leftarrow split(e, e', c_h)$ ▷ Determine host components for subformulae.
    if $c_1 = c_2 = c_h$ **then** ▷ No Split, both components are the same (host component)
      $\langle \text{root}_\text{l}, \text{N}_\text{l}, \text{E}_\text{l}\rangle \leftarrow netx(e, id_c, c_h)$ ▷ Process left sub-branch
      $\langle \text{root}_\text{r}, \text{N}_\text{r}, \text{E}_\text{r}\rangle \leftarrow netx(e', id_c, c_h)$ ▷ Process right sub-branch
      $m \leftarrow \langle \text{root}_\text{l}\text{.spec op root}_\text{r}\text{.spec}, id_c\rangle$ ▷ Merge results
      return $\langle m, \text{N}_\text{l} \cup \text{N}_\text{r}, \text{E}_\text{l} \cup \text{E}_\text{r}\rangle$
    else if $c_1 = c_h$ **then** ▷ Split right branch. Keep left subformula on the host component.
      $id_n \leftarrow \text{newid}()$ ▷ Generate a new ID.
      $\langle \text{root}_\text{l}, \text{N}_\text{l}, \text{E}_\text{l}\rangle \leftarrow netx(e, id_c, c_h)$ ▷ Recurse on the subformula hosted on the same component
      $\langle \text{root}_\text{r}, \text{N}_\text{r}, \text{E}_\text{r}\rangle \leftarrow netx(e', id_n, c_2)$ ▷ Recurse on the subformula moved to a new component
      $m \leftarrow \langle \text{root}_\text{l}\text{.spec op } id_n, id_c\rangle$ ▷ Replace right side with the reference
      return $\langle m, (\text{N}_\text{l} \cup \text{N}_\text{r} \cup \{\text{root}_\text{r}\}), (\text{E}_\text{l} \cup \text{E}_\text{r} \cup \{\langle id_n, id_c\rangle\})\rangle$ ▷ Merge nodes and edges, add a dependency edge.
    else ▷ Split left branch, analogous to the right split.
      $id_n \leftarrow \text{newid}()$
      $\langle \text{root}_\text{l}, \text{N}_\text{l}, \text{E}_\text{l}\rangle \leftarrow netx(e, id_n, c_1)$
      $\langle \text{root}_\text{r}, \text{N}_\text{r}, \text{E}_\text{r}\rangle \leftarrow netx(e', id_c, c_h)$
      $m \leftarrow \langle id_n \text{ op root}_\text{r}\text{.spec}, id_c\rangle$
      return $\langle m, (\text{N}_\text{l} \cup \text{N}_\text{r} \cup \{\text{root}_\text{l}\}), (\text{E}_\text{l} \cup \text{E}_\text{r} \cup \{\langle id_n, id_c\rangle\})\rangle$
    end if
  end if
end procedure

Table 8.1: Scalability of existing algorithms. $|C|$: number of components, $|AP_c|$: number of observations per component, |tr|: size of the trace, $|E|$: number of edges between monitors, m: active migration monitors, and depth(rt): depth of the monitor network.

Algorithm	δ_t	**# Msg**	**|Msg|**
Orchestration	$\Theta(1)$	$\Theta(\|C\|)$	$O(\|AP_c\|)$
Migration	$O(\|C\|)$	$O(m)$	$O(m\|C\|^2)$
Choreography	$O(\text{depth(rt)} + \|\text{tr}\|)$	$\Theta(\|E\|)$	$\Theta(1)$

evaluating a subformula, the monitors communicate the evaluation of the formula as a verdict in $\mathbb{B}_2$ when it is resolved. Furthermore, monitors may instruct other monitors to stop monitoring as they are no longer necessary. The two messages are referred to as msg_{ver} and msg_{kill}, respectively. For each monitor labeled $\ell \in AP_{\text{mons}}$, we determine the set $coref_\ell \in 2^{AP_{\text{mons}}}$ which contains the labels of monitors that send their verdicts to monitor $\mathcal{A}_\ell$. The information delay for a monitor depends on its depth in the network tree. The depth of a monitor labeled ℓ that depends on the set of monitors $coref_\ell$, is computed recursively as follows:

$$\text{depth}(\ell) = \begin{cases} 1 & \text{if monitorable}(\mathcal{A}_\ell) \wedge coref_\ell = \emptyset, \\ 1 + \max(\{\text{depth}(\ell') \mid \ell' \in coref_\ell\}) & \text{if monitorable}(\mathcal{A}_\ell) \wedge coref_\ell \neq \emptyset, \\ \infty & \text{otherwise,} \end{cases}$$

A monitor synthesized from a non-monitorable specification will never emit a verdict (Section 6.2.1), therefore its depth is ∞. A leaf monitor has no dependencies, its depth is 1. Since the depth controls the information delay (δ_t), it is possible in the case of choreography to obtain a large `EHE` depending on the specification. In the worst case, the size of the `EHE` can be linear in the size of the trace $\delta_t = |\text{tr}|$, as it will be required to store the `EHE` until the end of the trace. As such, properties of the specification such as monitorability (see Section 6.2.1) impact greatly the delay, and thus performance. In terms of communication, the number of monitors generated determines the number of messages that are exchanged. By using the splitting function (presented in [CF16a]), the number of monitors depends on the size of the LTL formula. Therefore, we expect the number of messages to grow with the number of atomic propositions in the formula. Denoting by $|E|$ the number of edges between monitors, we can say that the number of messages is linear in $|E|$. The size of the messages is constant, it is the size needed to encode a timestamp, id and a verdict in the case of msg_{ver}, or only the size needed to encode an id in the case of msg_{kill}.

8.1.5 Discussion

Choosing an algorithm. We summarize the main parameters that affect the algorithms in Table 8.1. This comparison could serve as a guide to choose which algorithm to run based on the environment (architectures, networks etc). For example, on the one hand, if the network only tolerates short message sizes but can support a large number of messages, then Orch or Chor is preferred over Migr. On the other hand, if we have heterogeneous nodes, as is the case in the client-server model, we might want to offload the computation to one major node, in this scenario Orch would be preferable as the forwarding monitor require no computation. This choice can be further impacted by the network topology. In a ring topology for instance, one might want to consider using Migration (with $m = 1$), as using Orch might impose further delay in practice to relay all information, while in a star topology, using Orch might be preferable. In a more hierarchical network, Chor can adapt its monitor tree to the hierarchy of the network.

Explaining earlier results. In [CF16a], the authors conducted experiments to compare the various algorithms. In particular, they broke down the metrics by delay, message count, message size and number of progressions (rewrites to the formula). We focus on the first three ignoring the last, since we do not monitor by rewriting. First, the authors rank the algorithms from lowest to highest as follows: Orch, Chor, and Migr. This is consistent with our analysis, since Orch has a constant delay, Chor a delay depending on the depth of the network, and Migr on the number of components. However, we note that there are (small) cases where Chor will have a worst-case delay of the size of the trace, this is not reflected in [CF16a]. Second, the authors discuss the number of messages sent by each algorithm (assuming a round-based scheme). The lowest algorithm is Migr (with $m = 1$), followed by Chor, which is followed by Orch. We note that in our analysis, the number of messages in migration depends on the number

of active monitors, in choreography on the number of edges in the network, and in orchestration on the number of components. Considering that the monitors are organized in a DAG, there will be generally less edges than components (using the splitting strategy), this contributes to choreography outperforming orchestration. This can be seen as the authors state that exceptions allow Orch to preform better than Orch. In particular "for [..] randomly generated formulae of size 5, or when the depth of the network is greater than or equal to 3.". A larger formula and a deeper network require a more complex network organization, which could result in more edges. We recall that the experiment ranged on a number of components between 3 and 5, where not all components were referenced at all times. Third, the authors discuss the size of the messages, ranking the algorithms from short to long messages as follows: Orch, Chor, and Migr. We recall that for Chor the message size is constant, while for Orch the message size depends on the number of observations per component. Either way both are significantly smaller than Migr which sends the full monitoring information. While the analysis shows that in theory, Chor should perform better than Orch with respect to size of the message, we are unsure how the size was captured. Since in the experiments in [CF16a], the size for Chor grows linearly with the size of the formula, while by definition the message is defined with a fixed size (ids + verdict). As such, we believe the encoding scheme has an impact on the size of the message.

Since we perform a worst-case analysis, we investigate in the next section (Section 8.2) the trends shown by simulating the behavior of the algorithms on a benchmark consisting of randomly generated specifications and traces. Furthermore, we use a real example in Section 8.2.4 to refine the comparison by looking at six different specifications.

8.2 Comparing Algorithms using Simulation in `THEMIS`

We use `THEMIS` to compare the adapted versions of existing algorithms (Orch, Migr, and Chor), introduced as examples to validate the trends presented in the analysis in Section 8.1. Furthermore, since the analysis presented worst-case scenarios, we look at the usefulness of simulations to determine the advantages or disadvantages of certain algorithms. The `THEMIS` tool, the data for both scenarios used, the scripts used to process the data, and the full documentation for reproducing the experiments is found at [EHF18c].

8.2.1 Overview of Scenarios

We additionally consider a round-robin variant of Migr, Migrr, and use that for analyzing the behavior of the migration family of algorithms as it has a predictable heuristic (function choose). We compare the algorithms under two scenarios. The first scenario explores synthetic benchmarks, that is, we consider random traces and specifications. This allows us to account for different types of behavior. The second scenario explores a specific example associated with a common pattern in programming. For that, we consider a publish-subscribe system, where multiple publishers subscribe to a channel (or topic), the channel publishes events to the subscribers. We use the Chiron user interface example [ACD+99, Tea99], along with the specifications formalized for it [DAC99a].

8.2.2 Monitoring metrics

Using information delay. The first considered metric is that of information delay (δ_t) (Section 5.2). The information delay impacts the size of the `EHE` and therefore the computation, communication costs to send an `EHE` structure, and also the memory required to store it. We recall that the information delay is the number of timestamps it took a `EHE` starting from a known state, to find a new known state. That is, it indicates the timestamps needed to acquire sufficient (partial) information to determine the global state (Section 5.2). To compute the average information delay (*average delay*), we sum all the timestamps across the entire run, and count the number of resolutions. By doing so, we acquire the average number of timestamps stored in an `EHE`. We note that this metric does not depend on the number of monitors, as it is normalized by the number of resolutions. Furthermore, it is possible for delay to fall below 1, as some traces can cause some monitors to emit a verdict at the very first timestamp. By considering our analysis in Section 8.1, we split our metrics into two main categories: computation and communication. The `EHE` structure requires the evaluation and simplification of a Boolean expression which is costly (see Section 5.1).

Computation, To measure computation, we can count the number of expressions evaluated (using memory lookup), and the number of calls to the simplifier. For this experiment, we consider the calls to the simplifier, counting the number of simplifications needed. Since algorithms may have more than one monitor active, we consider for a given round the monitor with the most simplifications. We sum the maximum number of simplifications per round across all the rounds, and then normalize by the number of rounds. This allows us to determine the slowest monitor per round, as other monitors are executing in parallel. Therefore, we determine the bottleneck. We refer to this metric as critical simplifications. This can be similarly done by considering the number of expressions evaluated.

Load-balance. Since monitors can execute in parallel, we also want to account for load-balance in the computation. As such, we introduce *convergence* as a metric to capture load balancing across a run of length n, where:

$$\text{conv}(n) = \frac{1}{n}\sum_{t=1}^{n}\left(\sum_{c\in C}\left(\frac{s_c^t}{s^t} - \frac{1}{|C|}\right)^2\right), \text{ with } s^t = \sum_{c\in C} s_c^t.$$

At a round t, we consider all simplifications performed on all components s^t and for a given component s_c^t. The ideal scenario is when computations have been spread evenly across all components. Thus, the ideal ratio is $\frac{1}{|C|}$. We compute the ratio for each component ($\frac{s_c^t}{s^t}$), then its distance to the ideal ratio. Distances are added for all components across all rounds then normalized by the number of rounds. The higher the convergence the further away we are from having all computations spread evenly across components. Convergence can also be measured similarly on evaluated expressions.

Communication. We consider communication using three metrics: number of messages, total data transferred, and the data transferred in a given message. The number of messages is the total messages sent by all monitors throughout the entire run. The data transferred consists of the total size of messages sent by all monitors throughout the entire run. Both the number of messages and the data transferred are normalized using the run length. Finally, we consider the data transferred in a given message to verify the message sizes. To do so, we normalize the total data transferred using the number of messages.

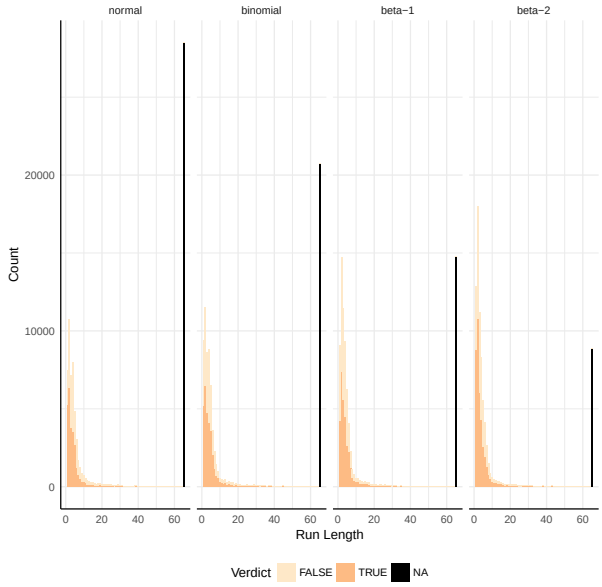
8.2.3 Synthetic Benchmark

Experimental Setup

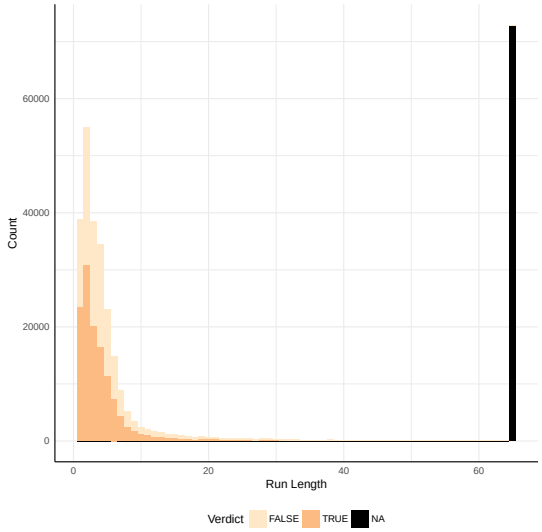
We generate the specifications as random LTL formulas using `randltl` from `Spot` [DL13] then converting the LTL formulae to automata using `ltl2mon` [BLS11]. We generate traces by using the `Generator` tool in `THEMIS` which generates synthetic traces using various probability distributions (see Section 7.8.3). For all algorithms we considered the communication delay to be 1 timestamp. That is, messages sent at t are available to be received at most at $t+1$. In the case of migration, we set the active monitors to 1 ($m = 1$). For our experiment, we vary the number of components between 3 and 5, and ensure that for each number we have 100 formulae that reference all components. We were not able to effectively use a larger number of components since most formulae become sufficiently large that generating an automaton from them using `ltl2mon` fails. The generated formulae were fully constructed of atomic propositions, there were no terms containing $\top$ or $\bot$[1]. We use 200 traces of 60 events per component, we associate with each component 2 observations. Traces are generated using 4 probability distributions (50 traces for each probability distribution). The used distributions [PH66] include *normal* ($\mu = 0.5, \sigma^2 = 1$), *binomial* ($n = 100, p = 0.3$), and two *beta* distributions: *beta-1* ($\alpha = 2, \beta = 5$), and *beta-2* ($\alpha = 5, \beta = 1$). The varied distributions provide different probability to assign $\top$ and $\bot$ to observations in the traces, as such we achieve varied coverage[2]. Figure 8.1a shows the outcome of runs for different probability distributions. We notice that, by varying the distributions, we obtain different distributions of verdicts across all runs for all given specifications. The trace length is chosen to be 60, based on the consideration that random formulae usually cause monitor verdicts to either be returned very early or timeout (Figure 8.1b). The percentage of runs that lasted more than 60 timestamps and returned a final verdict is less than 0.1% of total runs. When computing sizes, we use a normalized unit to

[1] To generate formulae with basic operators, string "true=0,false=0,xor=0,M=0,W=0,equiv=0,implies=0,ap=6,X=2,R=0" is passed to `randltl`.

[2] An observation is assigned $\top$ if the generated number is strictly greater than 0.5, and is otherwise $\bot$. For the binomial distribution, we consider $p = 0.3$ the probability of obtaining $\top$.



(a) Verdicts for traces based on the different probability distributions.



(b) Verdicts for all runs, across all traces of all probability distributions, using all random generated formulae.

Figure 8.1: Verdicts emitted by different run lengths.

separate the encoding from actual implementation strategies. Our assumptions on the sizes follow from the bytes needed to encode data (for example: 1 byte for a character, 4 for an integer). We normalized our metrics using the length of the run, that is, the number of rounds taken to reach the final verdict (if applicable) or timeout, as different algorithms take different numbers of rounds to reach a verdict. In the case of timeout, the length of the run is 65 (length of the trace, and 5 additional timestamps to timeout).

Increasing Coverage

In Table 8.2, we examine the variance by observing metrics wrt probability distributions used to generate the traces. To exclude the variance due to the number of components, we fix $|C| = 6$, as it provides the highest variance. For each metric, we present the mean and the standard deviation (between parentheses). All metrics are normalized by the length of the run. The metrics in order of columns are: average information delay (δ_t), average number of messages (#Msgs), total data transferred (Data), average maximum simplifications per monitor (S), and convergence based on expressions evaluated (Conv_E). We observe that by changing the probability distribution, the metrics vary significantly. This is particularly prominent for the information delay (especially in the case of Chor), and data transferred. As such, by varying the trace generation, we are able to increase our benchmarking coverage.

Comparing Algorithms

Figures 8.2 and 8.3 display the metrics for each of the algorithms. Figure 8.2a shows the average information delay. As expected, orchestration never exceeds a delay of 1. For migration, the delay depends on the heuristic used, as mentioned in Section 8.1, its worst case is the number of components. Migration can still have a lower delay than orchestration in some cases (as observed for $|C| \geq 4$). This observation is due to the initial monitor placement, as in our case we chose the first component always to be where we place the main orchestration monitor (component A), while for migration, the heuristic function (choose) decides which monitor starts. As such, in a specification where the verdict can be resolved at the first timestamp, migration has an advantage. For Chor, the delay is at least 1, as the network depth affects the delay. Furthermore, we notice that the delay for Chor is not particularly affected with the number of components. We know that its worst-case will depend on traces in cases of non-monitorability, we inspect that further in Section 8.2.4. Figure 8.2b shows the average maximum computation done by a monitor for a given round. By looking at computation, we notice that Orch performs no simplifications.

Table 8.2: Variation of average delay, number of messages, data transfer, critical simplifications and convergence with traces generated using different probability distributions for each algorithm. Number of components is $|C| = 6$. Table cells include the mean and the standard deviation (in parentheses).

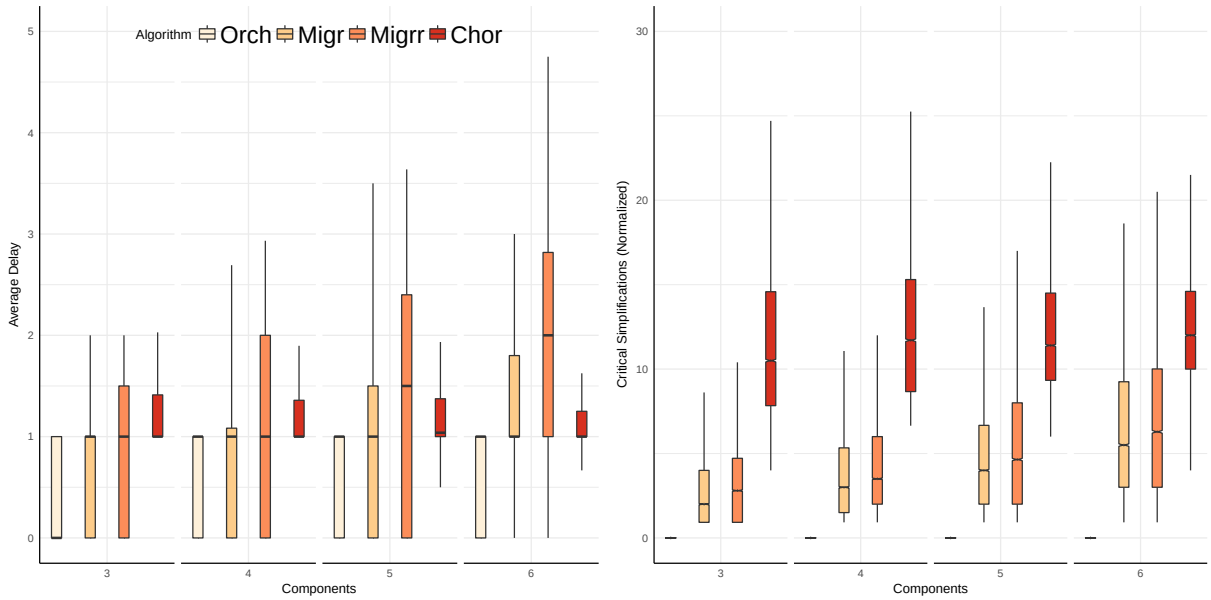
Alg.	Trace	δ_t	#Msgs	Data	S_{crit}	$Conv_E$
Orch	normal	0.69 (0.46)	7.13 (1.38)	94.87 (18.46)	0.00 (0.00)	0.83 (0.01)
	binomial	0.69 (0.46)	7.15 (1.39)	93.10 (18.13)	0.00 (0.00)	0.83 (0.01)
	beta-1	0.70 (0.46)	6.98 (1.47)	96.11 (20.17)	0.00 (0.00)	0.83 (0.02)
	beta-2	0.69 (0.46)	6.91 (1.71)	83.37 (20.66)	0.00 (0.00)	0.82 (0.02)
Migr	normal	1.72 (1.42)	0.50 (0.32)	110.13 (276.43)	7.01 (5.01)	0.82 (0.03)
	binomial	1.67 (1.40)	0.49 (0.32)	95.44 (221.65)	6.86 (4.91)	0.82 (0.04)
	beta-1	1.82 (1.44)	0.53 (0.32)	133.15 (313.44)	7.14 (5.16)	0.82 (0.04)
	beta-2	1.53 (1.36)	0.47 (0.38)	56.48 (114.54)	5.95 (4.24)	0.82 (0.03)
Migrr	normal	2.64 (1.93)	0.70 (0.35)	177.48 (358.61)	7.50 (5.18)	0.83 (0.03)
	binomial	2.59 (1.95)	0.69 (0.36)	171.64 (318.94)	7.49 (5.21)	0.83 (0.03)
	beta-1	2.82 (1.94)	0.74 (0.34)	210.02 (452.83)	7.49 (5.23)	0.82 (0.03)
	beta-2	2.55 (2.08)	0.66 (0.41)	162.28 (287.28)	6.93 (4.90)	0.82 (0.02)
Chor	normal	2.02 (1.97)	5.92 (1.60)	52.54 (14.23)	12.68 (3.63)	0.13 (0.10)
	binomial	1.93 (1.86)	5.95 (1.61)	52.76 (14.33)	12.55 (3.70)	0.13 (0.10)
	beta-1	2.59 (4.58)	5.80 (1.64)	51.54 (14.48)	13.29 (4.33)	0.14 (0.12)
	beta-2	2.95 (7.26)	5.81 (1.79)	51.52 (15.93)	13.50 (9.91)	0.14 (0.14)

This is the case as expressions in the EHE do not become sufficiently complex to require simplification. We recall that for orchestration, the memories of all local observations are sent to the main monitor within one timestamp. And as such, by memory lookup, the expression is immediately evaluated without the need to simplify. We notice that for the average case, Migr performs a small number of simplifications, and Chor still executes a reasonable number of simplifications. Figure 8.2c shows the convergence for the algorithms. Since Chor is the only algorithm that performs computations at different components in a given round, we notice that the convergence is much lower.

For communication, we first consider the number of messages transferred normalized by the length of the run. We notice that for algorithms Orch and Chor, the number of messages increases with the number of components. Since Chor depends on the edges that connect monitors, it scales better with the number of components than Orch (the depth of the network is usually smaller than the number of components). In contrast, we notice that for Migr and Migrr, the number of messages is independent from the number of components, as it depends on the number of active monitors. Figure 8.3a presents the total data transferred normalized by the run length. We notice by examining algorithm Orch that sending all observations can be costly. Algorithms Migr and Migrr are capable of sending much less data on average, but have variable behavior, and scale poorly, we notice an increase as $|C|$ increases. Algorithm Chor performs better than Orch, and scales much better with component size. While Migr and Migrr send fewer messages than the other algorithms, and have better scaling in the number of messages transferred, they can still, in total, send more data depending on the traces and specification. We notice that the 75% quartile for Migrr still exceeds that of Orch. Since total data transferred includes both the number of messages and their sizes, we present the size of the message in Figure 8.3b by dividing the total data transferred by the number of messages. We observe that for Orch and Chor the size of a message is constant, not very variable and does not depend on $|C|$, while for Migr and Migrr we observe a significant increase as $|C|$ increase. We recall from Section 8.1 that the migration algorithms send the EHE which expressions grow exponentialy in the size of the information delay.

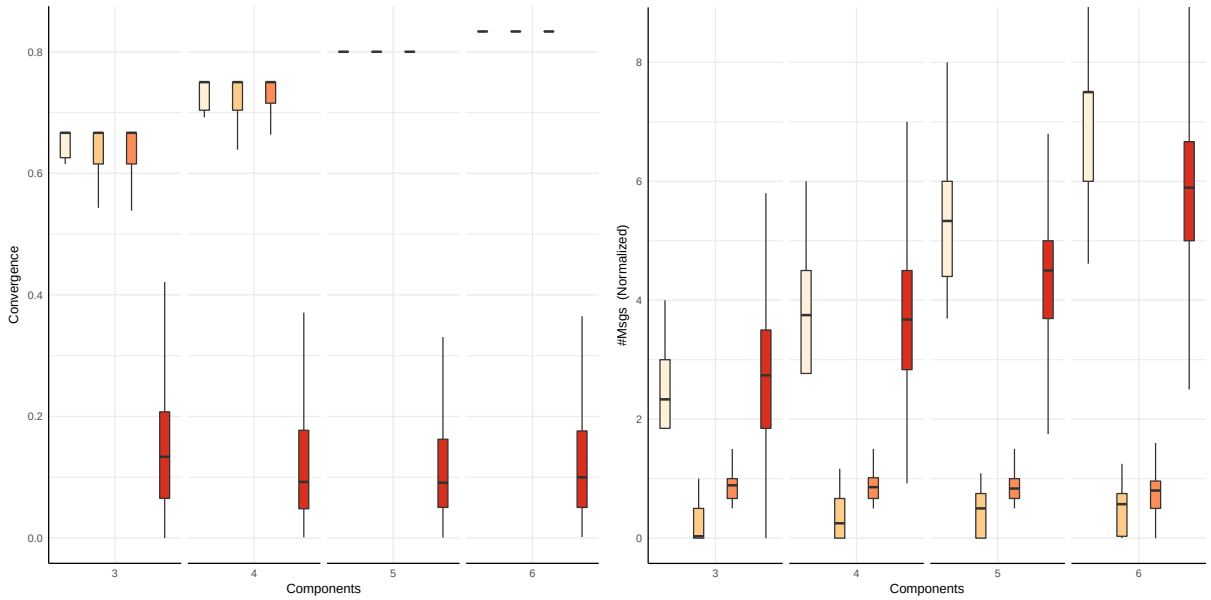
Comparing Variants

Using the same dataset, we look at another use case of THEMIS; that of comparing variants of the same algorithm. In this case, we focus on differences between Migr and Migrr. The heuristic of Migr improves on the round-robin heuristic of Migrr by choosing to transfer the EHE to the component that can observe the atomic proposition with the earliest timestamp in the EHE (referred to as earliest obligation [CF16a]). Using the simple heuristic, we notice a drop in the delay starting from $|C| > 4$ (Figure 8.2a). The simple heuristic of earliest obligation seems to reduce on average the delay of the algorithm, interestingly, it maintains a mean of 1. Furthermore, we observe a drop in both messages transferred (Figure 8.3a) and size of messages (Figure 8.3b). Consequently, this constitutes a drop in



(a) Average delay

(b) Critical simplifications

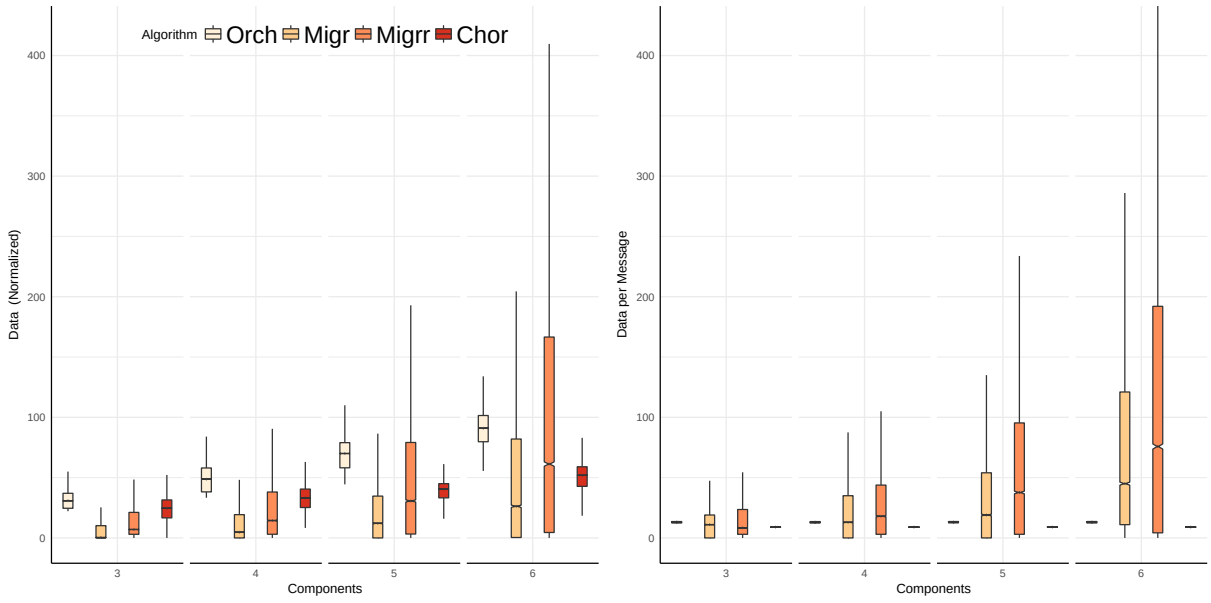


(c) Convergence

(d) Number of messages

Figure 8.2: Comparison of delay, computation and number of messages. Algorithms are presented in the following order: Orch, Migr, Migrr, Chor.

the total data transferred (Figure 8.3a). We note that the message size is also the size of the `EHE`. The drop in the number of messages sent is explained by the decision not to migrate when the soonest observation can be observed by the same component, while for Migrr, the round-robin heuristic causes the `EHE` to always migrate. However, this does not lead to a much lower number of simplifications (Figure 8.2b). Using `THEMIS` to compare the variants shows us that the earliest obligation heuristic reduces the size of the `EHE`, and thus, the size of the message, but also the number of messages sent. However, it does not seem to impact computation as the number of simplifications remains similar.



(a) Total data transferred

(b) Data per message

Figure 8.3: Data Transfer

Discussion

The observed behavior of the simulation aligns with the initial analysis described in Section 8.1. We observe that the EHE presents predictable behavior in terms of size and computation. The delay presented for each algorithm indeed depends on the listed parameters in the analysis. With the presented bounds on EHE, we can determine and compare the algorithms that use it. Therefore, we can theoretically estimate the situations where algorithms might be (dis)advantaged. However, both Figures 8.2 and 8.3 show that for most metrics, we observe a large variance (as evidenced by the inter-quartile difference). Furthremore, Table 8.2 shows that metrics vary for the different probability distributions. As such, we caution that while the analysis presents trends where algorithms have the advantage, it is still necessary to address exceptions, hence the need for simulation.

To account for exceptions, we explore the differences in the algorithms in Section 8.2.4 by considering real examples with existing formalized specifications.

8.2.4 The Chiron User Interface

Overview

Moving away from synthetic benchmarks, we consider properties that apply to patterns of programs and specifications. In this section, we compare the algorithms by looking at a real example that uses the publish-subscribe pattern. To that extent, we consider the Chiron user interface example [ACD+99]. Chiron consists of artists responsible of rendering parts of a user interface, that register for various events via a dispatcher. A dispatcher receives events from an abstract data type (ADT) and forwards them to the registered artists. We chose Chiron for two practical reasons. Firstly its example source code (in ADA), and its specifications are available online [Tea99]. The specification is completely formalized and utilizes various LTL patterns described in [DAC99a, DAC99b]. Thus, it covers a multitude of patterns for writing specifications. Secondly, the Chiron system can be easily decomposed into various components, we consider four components, the dispatcher (A), the two artists (B,C) and the main thread (D). The main thread is concerned with observing termination of the program.

Table 8.3: List of Chiron atomic propositions and components.

C	Name	Original (AP)	AP	Comments
A	Dispatcher	registered_event_a1_e1	a0	True after an artist has completed registration
		registered_event_a1_e2	a1	
		registered_event_a2_e1	a2	
		registered_event_a2_e2	a3	
		notify_a1_e1	a4	True when starting to dispatch an event to an artist
		notify_a1_e2	a5	
		notify_a2_e1	a6	
		notify_a2_e2	a7	
		lst_sz0_e1	a8	Tracking the size of the list (state)
		lst_sz0_e2	a9	
		lst_gt2_e1	a10	
		lst_gt2_e2	a11	
B	Artist1	notify_client_a1_e1	b0	Artist receives a notification
		notify_client_a1_e2	b1	
		register_event_a1_e1	b2	Artist requests to register for event
		register_event_a1_e2	b3	
		unregister_event_a1_e1	b4	Artist requests to unregister
		unregister_event_a1_e2	b5	
C	Artist2	notify_client_a2_e1	c0	See Artist1
		notify_client_a2_e2	c1	
		register_event_a2_e1	c2	
		register_event_a2_e2	c3	
		unregister_event_a2_e1	c4	
		unregister_event_a2_e2	c5	
D	Main	term	d0	Main program terminates

Experimental Setup

We broke down the Chiron system based on analysis of the examples provided in [Tea99, DAC99a], using the various specifications rewritten in [DAC99c]. Table 8.3 displays various associations we used to generate our traces and events. Column C assigns an ID to the component. Column Name lists the logical module of the system we considered as a component. Column Original (AP) lists the atomic propositions provided by the authors of Chiron, and then edited by [DAC99a]. Column AP maps the atomic propositions to our traces. Column Comments includes comments on the atomic propositions.

Table 8.4 lists the subset of the Chiron specification we considered. For each property, column ID references the original property name in [Tea99], column $\mathbb{B}_3$ references the expected verdict at the end of the trace[3], and column pattern identifies the LTL pattern corresponding to the formula. We modify the Chiron example program [Tea99] to output a trace of the program, and consider the specifications listed in Table 8.4[4]. For example, we consider the specification 15b shown in Listing 8.1. It states that the first artist always unregisters before the program terminates. Since an artist cannot register for the same event twice, we need only check that an unregister event responds to a register event.

Since we monitor offline, we generate the trace by inserting a global monitor that contains information about all relevant atomic propositions. The program is then instrumented to notify the monitor of events. Specifications and traces are then provided as input to `THEMIS` to process with the existing algorithms. We randomized the events dispatched in the Chiron example, and generated 100 traces of length 279. We targeted generating traces under 300 events. This corresponds to the ADT dispatching 91 events, in addition to events to register, and unregister artists.

[3] In the case where the expected verdict is ?, the specification is designed to falsify the property, as such if no falsification is found, we will terminate with verdict ?.

[4] We exclude specification 7 as we were unable to generate an automaton using `ltl2mon` for it. This is due to the formula either being too complex, or non-monitorable.

Table 8.4: Monitored Chiron specifications. CRC stands for Constrained Response Chain.

ID	$\mathbb{B}_3$	Pattern	Description
1	?	Absence	An artist never registers for an event if she is already registered for that event, and an artist never unregisters for an event unless she is already registered for that event.
2	?	CRC (2-1)	If an artist is registered for an event and dispatcher receives this event, it will not receive another event before passing this one to the artist.
3	$\top$	Precedence	Dispatcher does not notify any artists of an event until it receives this event from the ADT.
5	?	Absence	Dispatcher does not block ADT if no one is registered (this means that if no artists are registered for events of kind 1, dispatcher does nothing upon receiving an event of this kind from the ADT).
7*	?	CRC (3-1)	The order in which artists register for events of kind 1 is the order in which they are notified of an event of this kind by the dispatcher. In other words, if artist1 registers for event2 before artist2 does, then once dispatcher receives event2 from the ADT, it will first send it to artist1 and then to artist2.
15a	?	Universal	The program never terminates with an artist registered.
15b	$\top$	Response	An artist always unregisters before the program terminates. Given that you can't register for the same event twice, we need only check that unregisters respond to registers

Listing 8.1: Chiron specification (15b): "An artist always unregisters before the program terminates"

```
(register_event_a1_e1 -> (!term U unregister_event_a1_e1)) U (term | [](!term))
```

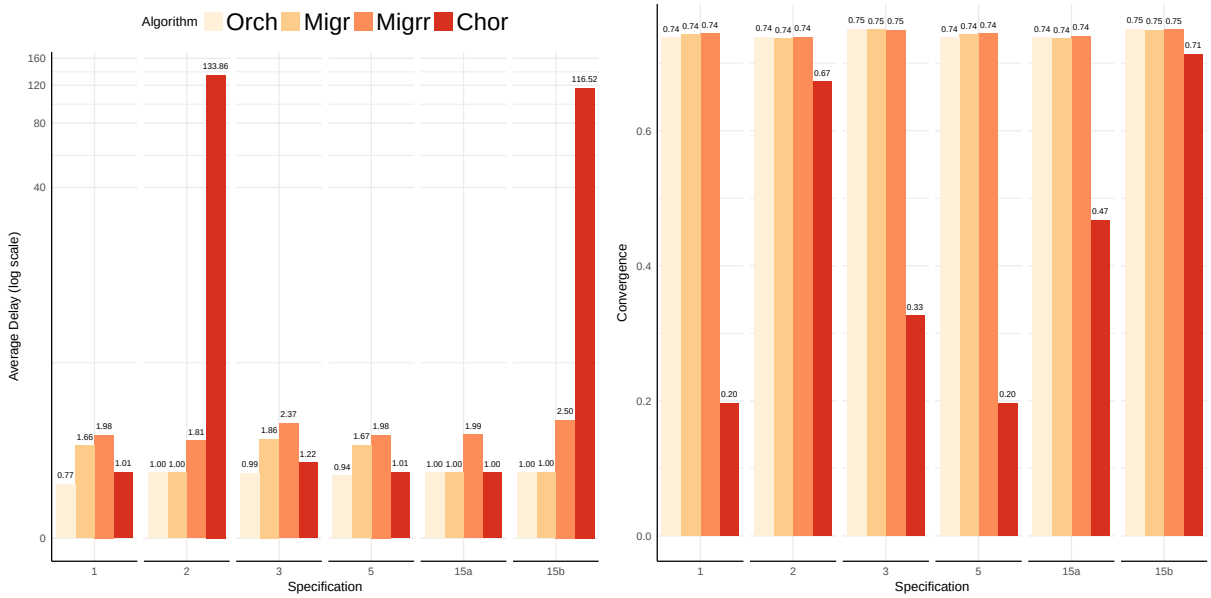
Comparing algorithms

Figure 8.4 presents the means for the metrics of average delay, convergence, and both critical and maximum simplifications[5]. We immediately notice for Chor the high average delay for specifications 2 and 15b (133.86, and 116.52, respectively). In these two cases, the heuristic to generate the monitor network for choreography has split the network inefficiently, and introduced a large delay due to dependencies. We recall that the heuristic used for choreography consider LTL formulae. For a given formula it counts the number of references to atomic propositions of a given component. The monitor tasked with monitoring the formula will then be associated with the highest component. To generate a decentralized specification, the heuristic starts with an LTL formula and splits it into two subformulas, then one of the subformulas is chosen to remain on the current component while the other is delegated to the component with the most references to atomic propositions. We see in this case that simply counting references and breaking ties using the lexicographical order of the component name can yield inefficient decompositions. Furthermore, we notice that while Orch maintains the lowest delay, other algorithms can still yield comparable delays. In the case of specification 15a, we observe that Orch, Migr, and Chor have similar delay (1.0). While Migr may outperform Chor for specifications 2 and 15b, it is the opposite for specifications 1, 3 and 5. *This highlights that the network decomposition of monitors (i.e., the setup phase) is an important consideration when designing decentralized monitoring algorithms.*

Figure 8.4b presents the convergence (computed using the number of expressions evaluated). We see that the poor decomposition also yields non-balanced workloads on the monitors. In the case of specifications 2 and 15b, we observe a convergence of 0.67 to 0.71 for Chor, respectively. The observed convergence is comparable with that of Orch and Migr. Furthermore, it is still possible to improve on the load balance for specifications 3 and 15a, as the convergence is high (0.33 and 0.47).

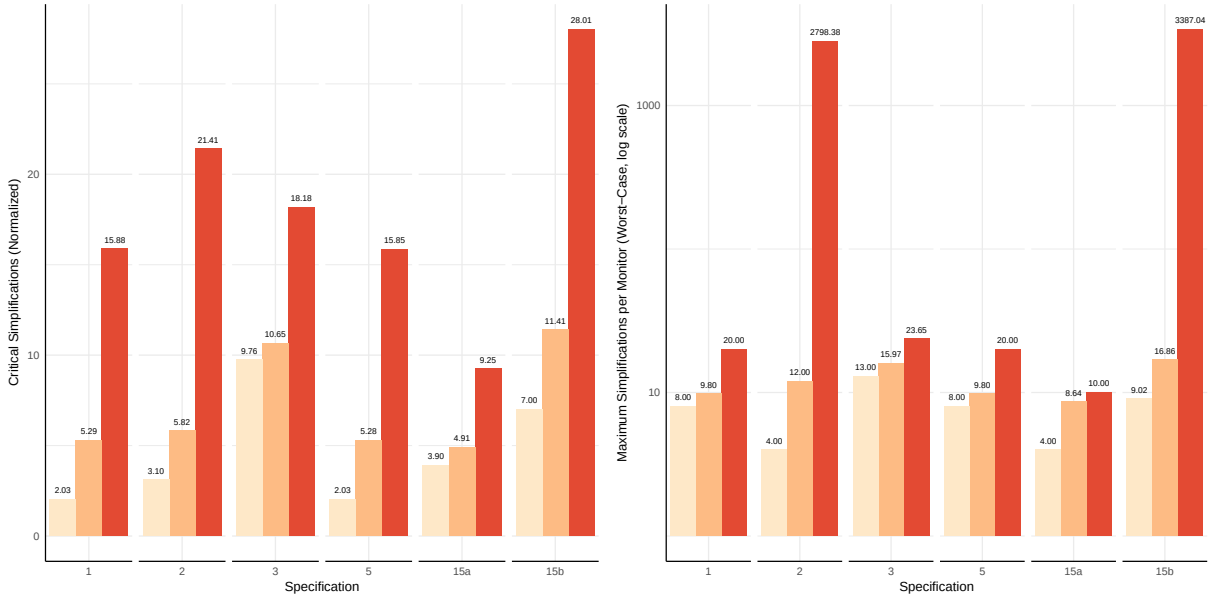
Figure 8.4c illustrates critical simplifications, we see that Chor has a higher cost compared to Migr in terms of computation. We also notice that Migr performs better than Migrr for all specifications. The heuristic of migrating the formula based on the atomic proposition with the earliest timestamp (earliest obligation) does indeed improve computation costs. More importantly, we notice that the highest delay for Chor is for specifications 2 and 15b. To

[5]We note that since we broke down the metrics per specification, we have little variation in the data, for details and standard deviations refer to Appendix B.1.



(a) Average delay

(b) Convergence



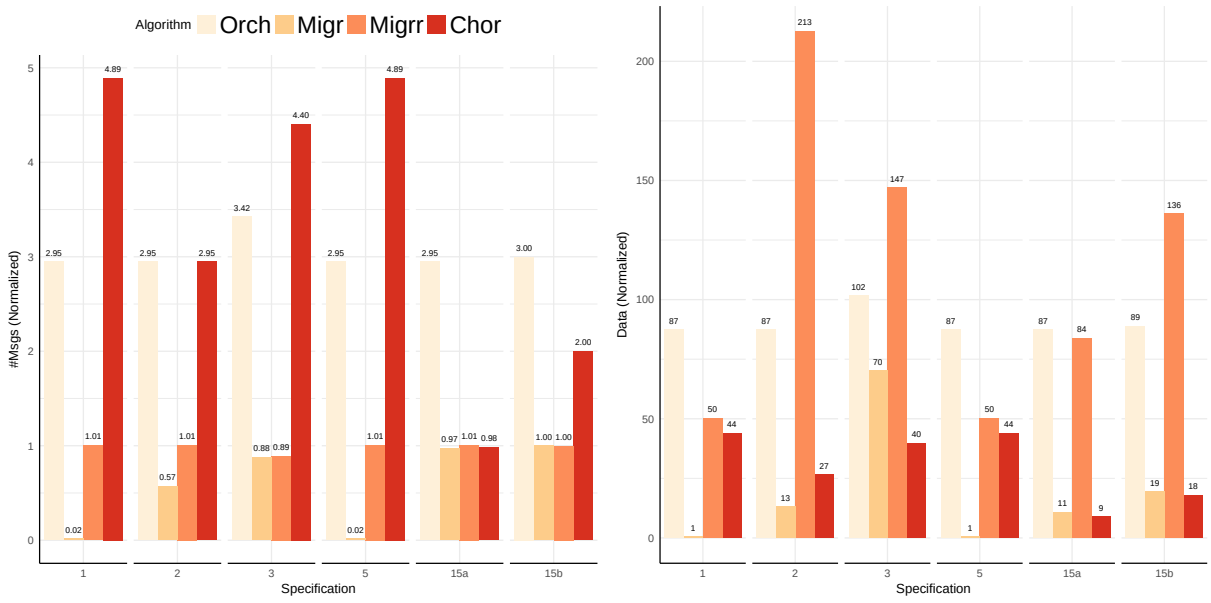
(c) Critical simplifications

(d) Maximum simplifications (worst case)

Figure 8.4: Comparison of delay, convergence and number of simplifications. Algorithms are presented in the following order: Orch, Migr, Migrr, Chor. Orch is omitted in the simplifications count as it is zero.

inspect that, we look at the maximum delay induced in a given monitor for an entire run, and consider the mean across all traces to obtain the worst-case maximum simplifications in Figure 8.4d. Indeed, we notice a peak in the maximum number of simplification in a given round for specifications 2 and 15b. Particularly, we notice that while comparable in other specifications (e.g., for specification 15a, we have 10 max simplifications for Chor as opposed to 8.64 and 10.00 for Migr and Migrr, respectively), the maximum number of simplifications for Chor increases to 2,798 (compared to 12 for Migrr), and 3,387 (compared to 16.86 for Migrr) for specifications 2 and 15b, respectively. In this particular case, we see how delay can impact the maximum number of simplifications.

We now consider communication costs by observing the number of messages transferred in Figure 8.5a. We see that Migr and Migrr perform well compared to the other two algorithms, with Migr performing consistently better



(a) Number of messages (normalized)

(b) Total data transferred (normalized)

Figure 8.5: Data Transfer

than Migrr. We note that the analysis of Migr indicates that the number of messages per round will be in the worst case the number of active monitors (in our case that is 1). One can see in specification 5 that Migr sends only 0.02 messages on average per round, compared to Migrr with 1.01, followed by Orch with 2.95, and finally Chor with 4.89. We see that Orch outperforms Chor in the case of specifications 1, 2, 3 and 5, where generally Orch sends 1-2 messages less. We note that this pattern is in line with the trends shown in Figure 8.2d. We see for $|C| = 4$ that Orch and Chor overlap, with Migrr outperforming both, and Migr outperforming all other algorithms. Interestingly, we find that in the case of specification 15a in Figure 8.5a, Chor sends a number of messages (0.98) slightly higher than Migr (0.97), and lower than Migrr (1.01). The difference in the number of messages is consistent with the lower whiskers in Figure 8.2d. Similarly, when considering the total data transferred in Figure 8.5b, we see as a trend across specifications Migr being particularly good, while still being slightly outperformed by Chor in specifications 15a and 15b. Furthermore, we notice that Migrr performs poorly and indeed sends more data than Orch in most cases, indicating that a heuristic can indeed be instrumental in the success of designing the family of migration algorithms. We notice also that the trends from Figure 8.3a seem to apply in most cases, Orch sends a lot more data than Migr and Chor, with Migrr possibly surpassing Orch.

Conclusion and Perspectives

Conclusion. We have shown how decentralized specifications can be used to aid in the analysis and simulation of decentralized monitoring algorithms. Particularly by providing a general description of a decentralized monitoring algorithm in two phases: *setup* and *monitor*, and two data structures which account for partial information: Memory and EHE. We then mapped three existing algorithms: Orchestration, Migration and Choreography to our approach using our data structures. Using the THEMIS tool and its metrics (introduced in Chapter 7), we implement three algorithms (and one additional variant) in THEMIS under our model and data structures: orchestration (Orch), migration using earliest obligation (Migr), migration using round-robin (Migrr), and choreography (Chor). We analyze their behavior by running simulations. Using THEMIS and the designed metrics, we explore simulations of the algorithms with various metrics for delay, computation, and communication costs, on two scenarios and validate the trends observed in the analysis. In general, we observe for the three algorithms in both the simulation and the Chiron example, the similar trends predicted in the analysis. For information delay, orchestration proves to be the most efficient with a constant delay, as there is no complex messaging and processing. In the case of migration this delay can grow significantly with the number of components depending on the heuristic. In the

case of choreography the delay depends on the depth of the network, which is maintained lower than the number of components. However, in extreme cases, the delay for choreography could be tied to the length of the trace which makes it impossible to use with large traces. The low delay of orchestration does not make it efficient in all situations however. Orchestration requires that all information be always communicated, which generates a large amount of messages of small size. Migration sends the least amount of messages at the cost of sending large messages. Choreography distributes the messages to be sent in a hierarchical setting, which allows it to send less messages than orchestration, while still maintaining constant size. In terms of computation, orchestration requires all computation to be performed on a single node. For migration, monitors need to be able to perform the computations on all nodes, as a monitor may migrate to any. For choreography, the subspecification of each monitor determines the computational cost of that monitor. Knowing how these algorithms behave proves important when performing decentralized runtime verification, as the system being monitored can exhibit constraints either in terms of computation or communication.

Perspectives. Using analysis and simulation to decide which decentralized algorithm to use to monitor specific situations proves useful, as it can characterize key advantages of such algorithms over others. For example, if we are constrained to monitor on a system that supports high throughput, one can utilize algorithms that communicate a large number of small messages. However, for systems with high latency, it is preferable to communicate large messages but less frequently. In the future, one could consider creating new metrics for `THEMIS` to analyze more aspects of decentralized monitoring algorithms. In particular to assess and characterize the *setup* phase of an algorithm and the topology of monitors. We see that this is important, as in two specifications out of the five when using Chiron traces (Section 8.2.4), the choreography algorithm, using a simple heuristic, generated an inefficient decentralized specification. New metrics would be automatically instrumented on all existing algorithms and experiments could be easily replicated to compare them.

CHAPTER 9

Bringing Runtime Verification Home: Hierarchical Monitoring of Smart Homes

"Are decentralized specifications advantageous to monitor smart homes?"

Contents

Chapter abstract

We use decentralized specifications to check various specifications in a smart apartment. The specifications can be broken down into three types: behavioral correctness of the apartment sensors, detection of specific user activities (known as activities of daily living), and composition of specifications of the previous types. The context of the smart apartment provides us with a complex system with a large number of components with two different hierarchies to group specifications and sensors: geographically within the same room, floor or globally in the apartment, and logically following the different types of specifications. We illustrate how decentralized specifications allow us to re-use specifications, and combine them to: (1) scale beyond existing centralized RV techniques, and (2) greatly reduce computation and communication costs.

Introduction

Smart homes. Sensors and actuators are used to create "smart" environments which track the data across sensors and human-machine interaction. One particular area of interest consists of homes (or apartments) equipped with a myriad of sensors and actuators, called *smart homes* [CC15]. Smart homes are capable of providing added services to users. These services rely on detecting the user behavior and the context of such activities [BCR09], typically detecting activities of daily living (ADL) [TIL04, CHN+12] from sensor information. Detecting ADL allows to optimize resource consumption (such as electricity [APS+17]), improve the quality of life for the elderly [MAN+17] and users suffering from mild impairment [TNM18].

RV in the smart home context. Relying on information from multiple sources and observing behavior is not just constrained to activities. It is also used with RV techniques that verify the correct behavior of systems. RV techniques have been used for instance in the context of automotive [CFB+12] and medical [LSàT16] systems. In both cases, RV is used to verify communication patterns between components and their adherence to the architecture and their formal specifications. While RV can be used to check that the devices in a smart home are performing as expected, we believe it can be extended to monitor ADL, and complex behavior on the activities themselves. We identify three classes of specifications for applying RV to a smart home. The first class pertains to the system behavior. These specifications are used to check the correct behavior of the sensors, and detect faulty sensors. Ensuring that the system is behaving correctly is what is generally checked when performing RV. However, it is also possible to use RV to verify other specifications. The second class consists of specifications for detecting ADL, such as detecting when the user is cooking, showering or sleeping. The third class pertains to user behavior. These specifications can be seen as meta-specifications for both system correctness and ADL, they can include safety specifications such as ensuring that the user does not sleep while cooking, or ensuring that certain activities are only done under certain conditions.

Motivation. However, standard RV techniques are not directly suitable to monitor the three classes of specifications. This is mainly due to scalability issues arising from the large number of sensors, as typically RV techniques rely on a large formula to describe specifications. Synthesizing centralized monitors from certain large formulas considered in this paper is not possible using the current tools. Instead, we make use of RV with decentralized specifications (Chapter 6) as it allows monitors to reference other monitors in a hierarchical fashion. The advantage of this is twofold. First, it provides an abstraction layer to relate specifications to each other. This allows specifications to be organized and changed without affecting other specifications, and even to be expressed with different specification languages. Second, it leverages the structure and layout of the devices to organize the hierarchies. On the one hand, we have a geographical hierarchy resulting from the spacial structure of the apartment from a given device, to a room, a floor, or the full apartment. On the other hand, we have a logical hierarchy defined by the interdependence between specifications, i.e. ADL, specifications that use other ADL specifications, and specifications that combine sensor safety with ADL specifications. For example, informally, consider checking two activities: sleeping and cooking, which can be expressed using formulae φ_s and φ_c respectively. A monitor that checks whether the user is sleeping and cooking requires to check $\varphi_s \wedge \varphi_c$ and as such will replicate the monitoring logic of another monitor that checks φ_s alone, instead of re-using the output of that monitor. The formula will be written twice, and changing the formula for detecting sleeping requires changing the formula for the monitor that checks both specifications. An artifact [EHF] that contains data, documentation, and software, is provided to replicate and extend on the work of this chapter.

Chapter organization. We begin by introducing the smart apartment context and elaborating on properties for RV in Section 9.1.1. Then we introduce decentralized specifications and their advantages for monitoring the smart apartment Section 9.1. In Section 9.3, we illustrate how trace data acquired from the Orange4Home dataset is processed and adapted to `THEMIS`. In Section 9.4, we evaluate using decentralized specifications for monitoring the apartment, by considering scalability, and portability to other datasets. Futhermore, we overview possible approaches for ADL monitoring in Section 9.5.

Key contributions. The key contributions of this chapter can be summarized as follows:

Figure 9.1: Suggested Schedule (Tuesday, Jan 31 2017)

1. We apply decentralized RV to analyze traces of over 36,000 timestamps spanning 27 sensors in a real smart apartment (Section 9.1.1).
2. We show how to go beyond system properties, to specify ADL using RV, and more complex interdependent specifications defined on up to 27 atomic propositions (Section 9.1.2).
3. We leverage the hierarchies, modularity and re-use afforded by decentralized specifications (Section 9.2) to both be able to synthesize monitors and to reduce overhead when monitoring complex interdependent specifications (Section 9.4.1).
4. We use RV to effectively monitor ADL and identifying some insights and limitations inherent to using formal LTL specifications to determine user behavior (Section 9.4.2).
5. We elaborate on the advantages of modularity by adapting parts of the specification to the ARAS [AEIE13] dataset (Section 9.4.3).

9.1 Writing Specifications for the Apartment

9.1.1 Devices and Organization

We consider a single actual apartment, with multiple rooms, where activities are logged using sensors. Amiqual4Home is an experimental platform consisting of a smart apartment, a rapid prototyping platform, and tools for observing human activity.

Overview of Amiqual4Home. The Amiqual4Home apartment is equipped with 219 sensors and actuators spread across 2 floors [LLR+17]. Amiqual4Home uses the OpenHab 6 integration platform for all the sensors and actuators installed. Sensors communicate using KNX, MQQT or UPnP protocols sending measurements to OpenHab over the local network, so as to preserve privacy. The general layout of the apartment consists of 2 floors: the ground and first floor. On the ground floor (resp. first floor), we have the following rooms: `entrance`, `toilet`, `kitchen`, and `livingroom` (resp. `office`, `bedroom`, and `bathroom`). Between the two floors, there is a connecting `staircase`. This layout reveals a geographical hierarchy of components, where we can see the rooms at the leaves, grouped by floors then the whole apartment. While in effect all device data is fed to a central observation point, it is reasonable to consider the hierarchy in the apartment as a simpler model to consider hierarchies in general, as one is bound to encounter a hierarchy at a higher level (from houses, to neighborhoods, to smart cities, etc.). Furthermore, hierarchies appear when integrating different providers for devices in the same house.

Reusing the Orange4Home dataset. Amiqual4Home has been used to generate multiple datasets that record all sensor data, this includes an ADL recognition dataset [LLR+17] (ContextAct@A4H), and an energy consumption dataset [CLRC17] (Orange4Home). In this paper, we reuse the dataset from [CLRC17]. The case study involved a

Table 9.1: Specifications considered in this paper. (*) indicates added ADL specifications. G indicates specification group: system (S), ADL (A), and meta-specifications (M). $|AP|^d$ (resp. $(|AP|^c)$): atomic propositions needed to specify specification in decentralized (resp. centralized) specifications. d is the maximum depth of monitor dependencies.

G	Scope	Name	Description	$\|AP\|^d$	$\|AP\|^c$	d
S	Room	`sc_light`(i)	light switch turns on light ($i \in [0..3]$).	2	2	1
M	House	`sc_ok`	All light switches are ok.	4	8	2
A	Toilet	`toilet`*	Toilet is being used.	1	1	0
A	Bathroom	`sink_usage`	Sink is being used.	1	2	1
A	Bathroom	`shower_usage`	Shower is being used.	1	2	1
A	Bedroom	`napping`	Tenant is sleeping on the bed.	1	1	1
A	Bedroom	`dressing`	Tenant is dressing, using the closet.	2	3	1
A	Bedroom	`reading`	Tenant is reading.	3	5	2
A	Office	`office_tv`	Tenant is watching TV.	1	1	1
A	Office	`computing`	Tenant is using the computer.	1	1	1
A	Kitchen	`cooking`	Tenant is cooking food.	2	2	1
A	Kitchen	`washing_dishes`	Tenant is cleaning dishes.	2	3	1
A	Kitchen	`kactivity`*	Using cupboards and fridge.	4	9	1
A	Kitchen	`preparing`	Tenant is preparing to cook food.	2	11	2
A	Living	`livingroom_tv`	Tenant is watching TV.	2	2	1
A	Floor 0	`eating`	Tenant is eating on the table.	2	2	1
M	Floor 0	`actfloor`(0)	Activity triggered on floor 0.	6	16	3
M	Floor 1	`actfloor`(1)	Activity triggered on floor 1.	7	11	3
M	House	`acthouse`	Activity triggered in house	2	27	4
M	House	`notwopeople`	No 2 simultaneous activities on dif. floors.	2	27	4
M	House	`restricttv`	No watching TV for more than 10s.	2	3	3
M	House	`firehazard`	No cooking while sleeping.	2	3	2

person living in the home and following (loosely) a schedule of activities spread out across the various rooms of the house, set out by the authors. Figure 9.1 displays the suggested schedule of activities for Tuesday, Jan 31 2017. This allows us to nicely reconstruct the schedule from the result of monitoring the sensors. Furthermore, the person living in the home provided manual annotations of the activities done, which helps us assess our specifications. We chose to use [CLRC17] over [LLR+17] as it involves only one person living in the house at a time which simplifies specifying and validating specifications.

Monitoring environment. In total, we formalize 22 specifications that make use of up to 27 sensors, and evaluate them over the course of a full day of activity in the apartment. That is, we monitor the house (by replaying the trace) from 07:30 to 17:30 on a given day, by polling the sensors every 1 second, creating a trace of a total of 36,000 timestamps. Specifications are elaborated in Section 9.1.2 and expressed as decentralized specifications (introduced in Chapter 6). Traces are replayed using the `THEMIS` tool (Chapter 7) which supports decentralized specifications and provides a wide range of metrics. We elaborate on the trace replay in Section 9.3.

9.1.2 Specification Groups

We now specify specifications that describe different behaviors of components in the smart apartment. Specifications can be subdivided into 3 groups: system-behavior specifications, user-behavior specifications, and meta specifications on both system and user behavior. The specifications we considered are listed in Table 9.1.

System behavior. The first group of specifications consists in ensuring that the system behaves as expected. That is, verifying that the sensors are working properly. These properties are the subject of classical RV techniques [FHR13, BLS11] applied to systems. For the scope of this case study, we verify light switches as system

properties. We verify that for a given room i, whenever the switch is toggled, then the light must turn on until the switch is turned off. We verify the property at two scopes, for a given room, and the entire apartment. While this property appears simple to check, it does highlight issues with existing centralized techniques applied in a hierarchical way. We develop the property in Section 9.2.1, and show the issues in Section 9.2.2.

ADL. The second group of specifications is concerned with defining the behavior of the user inferred from sensors. The sensors available in the apartment provide us with a wealth of information to determine the user activities. The list of activities of interest is detailed in [Kat83] and includes activities such as cooking and sleeping. By correctly identifying activities, it is possible to decide when to interact with the user in a smart setting [APS+17], provide custom care such as nursing for the elderly [MAN+17], or help users who suffer from mild impairment [TNM18]. Inferring activities done by the user is an interesting problem typically addressed through either data-based or knowledge-based methods [CHN+12]. The first method consists in learning activity models from preexisting large-scale datasets of users' behaviors by utilizing data mining and machine learning techniques. The built models are probabilistic or statistical activity models such as Hidden Markov Model (HMM) or Bayesian networks, followed by training and learning processes. Data-driven approaches are capable of handling uncertainty, while often requiring large annotated datasets for training and learning. The second method consists in exploiting prior knowledge in the domain of interest to construct activity models directly using formal logical reasoning, formal models, and representation. Knowledge-driven approaches are semantically clear, but are typically poor at handling uncertainty and temporal information [CHN+12]. We elaborate on such limitations in Section 9.4.2. Writing specifications can be seen as a knowledge-based approach to describe the behavior of sensors. As such, we believe that runtime verification is useful to describe the activity as a specification on sensor output. We formalize a specification for the following ADL activities described in [CLRC17] (see Table 9.1). We re-use the traces to verify that our detected activities are indeed in line with the schedule proposed. Figure 9.2 displays the reconstructed schedule after detecting ADL with runtime verification. Each specification is represented by a monitor that outputs (with some delay) for every timestamp (second) verdicts $\top$ or $\bot$. To do this, the monitor finds the verdict for a timestamp t then respawns to monitor $t + 1$. Verdict $\top$ indicates that the specification holds, that is, the activity is being performed. The reconstructed schedule shows the eventual outcome of a specification for a given timestamp ignoring delay. In reality some delay happens based on the specification itself, and the dependencies on other monitors.

Meta-specifications. Specifications of the last group are defined on top of the other specifications. That is, we refer to a meta specification as a specification that defines the interactions between various specifications. While one can easily define specifications by defining predicates over existing ones, such as checking that the light switch specification holds in all rooms or whether or not detecting an activity was performed on a specific floor or globally in the house, we are interested more in specifications that relate to each other. We consider a meta specification that reduces fire hazards in the house. In this case, we specify that the tenant should not cook and sleep at the same time, as this increases the risk of fire. In addition to mutually excluding specifications, we can also constrain the behavior of existing specifications. For example, we can specify a specification regulating the duration of watching TV to be at most 10 timestamps.

9.2 Monitoring the Apartment

We show how we monitor the apartment using decentralized specifications, while highlighting their advantages.

9.2.1 Monitor Implementation

To monitor the apartment, we rely on LTL3 monitors [BLS11] (Section 2.1.2). We recall that an LTL3 monitor is a complete and deterministic Moore automaton where states are labeled with the verdicts $\mathbb{B}_3 = \{\top, \bot, ?\}$. Verdicts $\top$ and $\bot$ respectively indicate that the current execution complies and does not comply with the specification, while verdict ? indicates that the verdict has not been determined yet. Verdicts $\top$ and $\bot$ are called final, as once the monitor outputs $\top$ or $\bot$ for a given trace, it cannot output a different verdict for any suffix of that trace. Using LTL3 monitors for representing specifications allows us to take advantage of the multiple RV tools that convert different

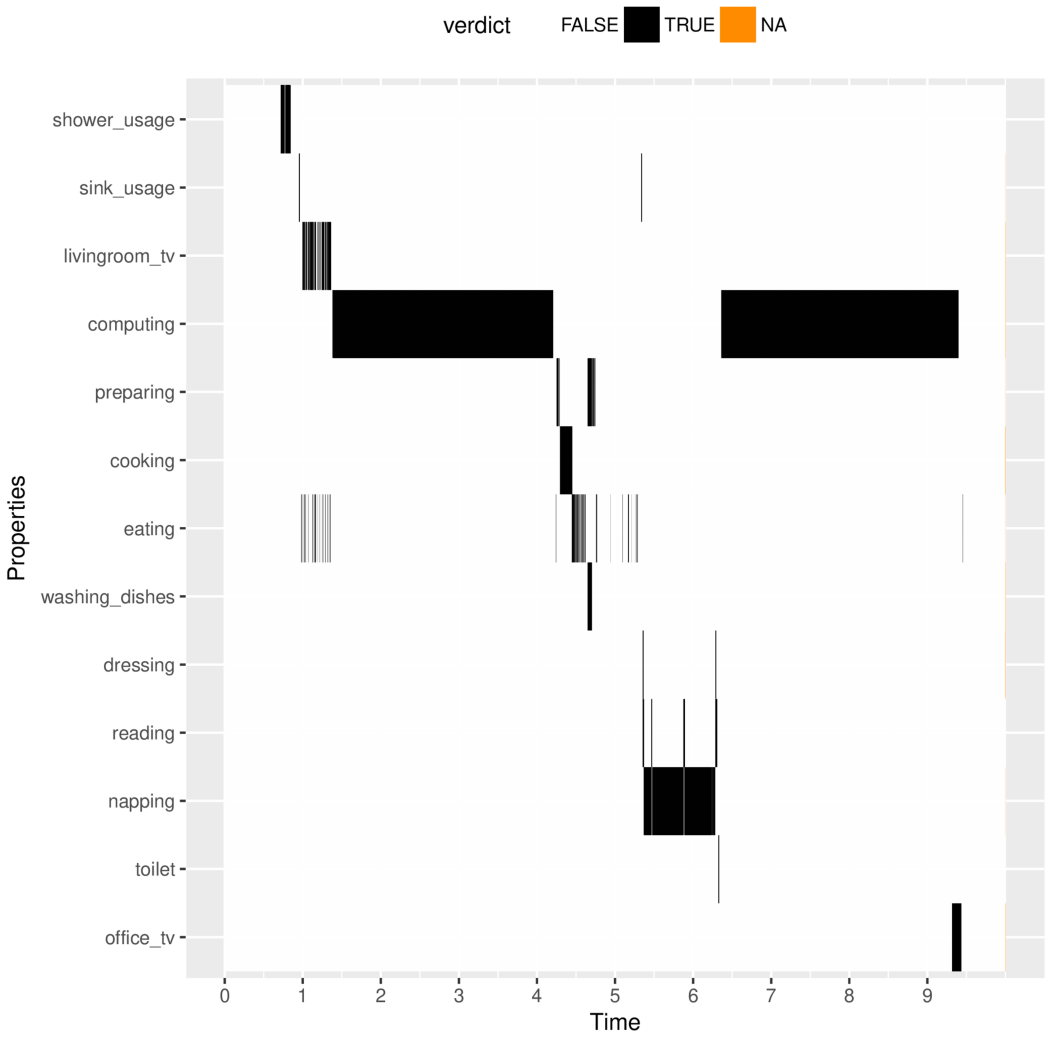


Figure 9.2: Detected ADL for Tuesday, Jan 31 2017. Time is in hours starting from 7:30.

specification languages to LTL3 monitors. For our monitoring, we use the `THEMIS` tool (Chapter 7) which is able to use both `ltl2mon` [BLS11] and `LamaConv` [Ins] to generate monitors.

Example 36 (Check light switch) Let us consider property `sc_light`(i) (sensor check light): "Whenever a light switch is triggered in a room i at some timestamp t, then the light must turn on at $t+1$ until the switch is turned off again". We recall the LTL_3 monitors for the property in Example 22. We define the monitors over the set of atomic propositions extended with the room index, as we consider multiple rooms. Each monitor checks whether the property is falsified (as it is a safety property). ∗

For the scope of this chapter and for clarity, we use LTL extended with two (syntactic) operators, mostly to strengthen and relax time constraints. We consider the operator *eventually within t* ($\mathbf{F}_{\leq t}$) which considers a disjunction of next operators. It is defined as: $\mathbf{F}_{\leq t}\, ap \stackrel{\text{def}}{=} ap \vee \mathbf{X}ap \vee \mathbf{XX}ap \vee ...\mathbf{X}^t ap$, where ap is an atomic proposition. Intuitively, the eventually within states that ap holds within a given number of timestamps. Operator $\mathbf{F}_{\leq t}$ allows us to relax the time constraints for a given atomic proposition. Similarly, we consider the operator *globally within t* ($\mathbf{G}_{\leq t}$) which is the dual of the previous operator. Operator $\mathbf{G}_{\leq t}$ is a conjunction of next operators. $\mathbf{G}_{\leq t}\, ap \stackrel{\text{def}}{=} ap \wedge \mathbf{X}ap \wedge \mathbf{XX}ap \wedge \mathbf{X}^t ap$.

Example 37 (Check light switch modalities) The property expressed in Example 36 can be expressed in LTL as: `sc_light`$(i) \stackrel{\text{def}}{=} \mathbf{G}(\text{s}_i \implies \mathbf{X}(\ell_i \,\mathbf{U}\, \neg \text{s}_i))$. The property can be modified with the extra operators to relax or constrain

the time on the light. The relaxed property $\texttt{sc\_light}'(i) \stackrel{\text{def}}{=} \mathbf{G}(\mathrm{s}_i \implies \mathbf{F}_{\leq 3}(\ell_i \,\mathbf{U}\, \neg \mathrm{s}_i))$ allows the right-hand side of the implication to hold within any of the next 3 timestamps instead of immediately after. The bounded property $\texttt{sc\_light}''(i) \stackrel{\text{def}}{=} \mathbf{G}(\mathrm{s}_i \implies \mathbf{G}_{\leq 3}(\ell_i))$ states that the light is on starting from the timestamp the switch is turned on and the subsequent two (for a total of 3). An example of such a property is the restriction on watching TV for a specific duration (Table 9.1) where $\texttt{restricttv} \stackrel{\text{def}}{=} \mathbf{G}(\texttt{tv} \implies \mathbf{F}_{\leq 10}\, \neg \texttt{tv})$. *

9.2.2 Using Decentralized Specifications

While simple specifications can be expressed with both LTL and automata, it quickly becomes a problem to scale the formulae or account for hierarchies (see Sect. 9.2.3). As such, we use decentralized specifications (Chapter 6).

Overview. We recall from Chapter 6 that a decentralized specification considers the system as a set of components, defines a set of monitors, additional atomic propositions that represent references to monitors, and attaches each monitor to a component. A decentralized trace is a partial function that assigns to each component and timestamp an event. Each monitor is a Moore automaton as described in Section 9.2.1 where the transition label is restricted to only atomic propositions related to the component on which the monitor is attached, and references to other monitors. A monitor reference is evaluated as if it were an oracle. That is, to evaluate a monitor reference m_i at a timestamp t, the monitor referenced ($\mathcal{A}_i$) is executed starting from the initial state on the trace starting at t. The atomic proposition m_i at t takes the value of the final verdict reached by the monitor.

Example 38 (Decentralized light switch) Figure 6.2b shows the decentralized specification for the check light property from Example 36. We have two monitors $\mathcal{A}_{\texttt{sc\_light}_i}$ and $\mathcal{A}_{\ell_i}$. They are respectively attached to the light switch, and light bulb components. In the former, the atomic propositions are either related to observations on the component (s_i, switch on), or references to other monitors (m_{ℓ_i}). The light switch monitor first waits for the switch to be on to reach q_1. In q_1, at some timestamp t, it needs to evaluate reference m_{ℓ_i} by running the trace starting from t on monitor $\mathcal{A}_{\ell_i}$. *

Assumptions. The assumptions of decentralized specifications are sufficient to monitor smart homes. We recall the assuptions: no monitors send messages that contain wrong information; no messages are lost, they are eventually delivered in their entirety but possibly out-of-order; all components share one logical discrete clock marked by round numbers indicating relevant transitions in the system specification. While security is a concern in the smart apartment setting, the first two assumptions are met in this case study as the apartment sensor network operates on the local network, and we expect monitors to be deployed by the sensor providers, and users of the apartment. The last assumption is also met in the smart setting, as all sensors share a global clock.

Hierarchical dependencies. Decentralized specifications allow us to analyze the dependencies between various monitors, and organize them in logical hierarchies represented as directed acyclic graphs (DAGs). The DAGs help us relate specifications to other specifications and analyze the inter-dependent behavior of monitors. We elaborate on the benefits of the hierarchical dependencies in Section 9.2.3.

Example 39 (Hierarchical dependencies) Figure 9.3 presents the dependency DAG of specification `preparing`. We can see that specification `preparing` depends directly on both specifications `kactivity` and `cooking`. Specification `kactivity` depends on specifications `cubpoard`, `sink_water`, `presence`, and `fridge_door`, as it depends on the tenant being present in the kitchen, opening or closing cupboards or the fridge, or using the sink. The later specifications do not depend on other specifications but on direct observations from the components. We note that while `presence` is not used in this case study to determine the cooking activity, since a tenant can start cooking and leave the kitchen. One could imagine that specifications can share dependencies, as such the hierarchy is indeed best represented as a DAG. Let us consider the monitor checking specification `cupboard`. Since we have 5 cupboard doors, we have 5 sensors in total (1 for each door). The monitor observing the 5 different observations simply checks if one is open and relays its verdict upwards, transmitting only the summary of observations instead of the totality. In this example, the hierarchy can be seen starting from different sensors on the same component, and expanding geographically to the different components in the room (kitchen). *

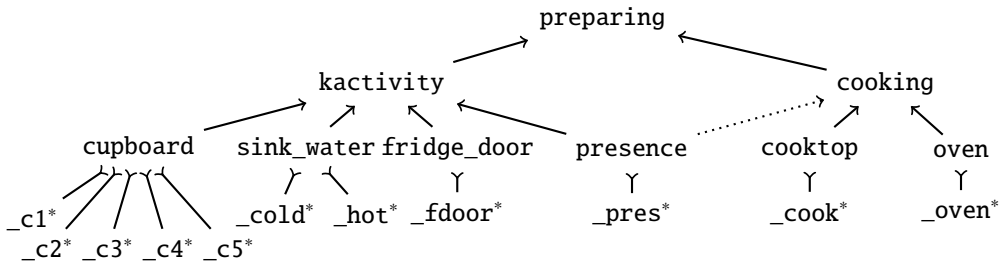


Figure 9.3: Dependencies for `preparing`. * indicates an atomic proposition of a component.

Specifications list. The formulae associated with each specification from Table 9.1 are listed in Table 9.2. The formulae are designed to be representative of behavior, we did not aim for a formula to encompass all possible user behavior.

9.2.3 Advantages of Decentralized Specifications

Modularity and re-use. Monitor references in decentralized specifications allow specifications writers to modularize behavior. Given that a monitor represents a specific specification, this same monitor can be re-used to define more complex specifications at a higher level, without consideration for the details needed for this specification. This allows specification writers to reason at various levels about the system specification.

Let us consider the ADL specification `cooking` (resp. `sleeping`) which specifies whether the tenant is cooking (resp. sleeping) in the apartment. One can reason about the meta-specification `firehazard` using both `cooking` and `sleeping` specifications without considering the lower level sensors that determine these specifications, that is

$$\texttt{firehazard} \stackrel{\text{def}}{=} \mathbf{G}(\texttt{sleeping} \implies \neg\texttt{cooking}).$$

While we can define `cooking` as

$$\texttt{cooking} \stackrel{\text{def}}{=} \texttt{kitchen\_presence} \wedge \mathbf{F}_{\leq 5}(\texttt{kitchen\_cooktop} \vee \texttt{kitchen\_oven}).$$

Additionally, any specification that requires either `sleeping` or `cooking` specifications can re-use the verdict outputted by their respective monitors. For example the specifications `actfloor`(0) and `actfloor`(1) require the verdicts from monitors associated with `cooking` and `sleeping`, respectively, since cooking happens on the ground floor while sleeping on the first floor. Furthermore, we can disjoin `actfloor`(0) and `actfloor`(1) to easily specify that an activity has happened in the house, $\texttt{acthouse} \stackrel{\text{def}}{=} \texttt{actfloor}(0) \vee \texttt{actfloor}(1)$. While specification `acthouse` can be seen as a quantified version of `actfloor`(i), we can use modular specifications for behavior, for example we can verify the triggering of an alarm in the house within 5 timestamps of detecting a fire hazard, i.e. $\texttt{checkalert} \stackrel{\text{def}}{=} \texttt{firehazard} \implies \mathbf{F}_{\leq 5}(\texttt{firealert})$.

In addition to providing a higher level of abstraction and reasoning about specifications, the modular structure of the specifications present three additional advantages. The first allows the sub-specifications to change without affecting the meta-specifications, that is if the sub-specification `cooking` is changed (possibly to account for different sensors), no changes need to be propagated to specifications `firehazard`, `actfloor`(0), `acthouse`, and `checkalert`. The second advantage is controlling duplication of computation and communication, as such sensors do not have to send their observations constantly to all monitors that verify the various specifications. The specification `cooking` requires knowledge from the kitchen presence sensor, the kitchen cooktop (being enabled) and the kitchen oven. Without any re-use these three sensors (presence, cooktop, and oven) need to send their information to monitors checking: `firehazard`, `actfloor`(0), `acthouse`, and `checkalert`. The third advantage is a consequence of modeling explicitly the dependencies between specifications. This allows the monitoring to take advantage of such dependencies and place the monitors that depend on each other closer depending on the hierarchy, either geographically (i.e., in the same room or floor) or logically (i.e., close to the monitors of the dependent sub-specifications). Furthermore, knowing the explicit dependencies between specifications allows the user to choose a placement for their monitors, adjusting the placement to the system architecture. In the case a placement is not possible, it is possible to create intermediate specifications that simply relay verdicts of other monitors, to transitively connect all components that are not connected.

Table 9.2: List of specifications. A prefixed atomic proposition with m_ indicates that the monitor is deployed on the component.

Name	Formula
sc_light(i)	$\mathbf{G}(\text{switch}_i \implies \mathbf{X}(\text{light}_i\, \mathbf{U}\, \neg\text{switch}_i), i \in [0..3]$
sc_ok	$\bigwedge_{i\in[0..3]}$ sc_light(i)
m_toilet	toilet_water
sink_usage	$\mathbf{G}_{\leq 3}$(m_bathroom_sink_water)
m_bathroom_sink_water	bathroom_sink_cold $\vee$ bathroom_sink_hot
shower_usage	$\mathbf{G}_{\leq 2}$(m_bathroom_shower_water)
napping	$\mathbf{G}_{\leq 25}$(m_bedroom_bed_pressure)
dressing	$\mathbf{F}_{\leq 4}$(m_bedroom_closet_door $\vee$ m_bedroom_drawers))
reading	m_bedroom_light $\wedge$ $\mathbf{F}_{\leq 4}$($\neg$dressing $\wedge$ $\neg$napping)
office_tv	$\mathbf{F}_{\leq 3}$(m_office_tv)
computing	$\mathbf{F}_{\leq 3}$(m_office_deskplug)
cooking	$\mathbf{F}_{\leq 5}$(m_kitchen_cooktop $\vee$ m_kitchen_oven)
washing_dishes	$\mathbf{F}_{\leq 3}$(m_kitchen_dishwasher $\vee$ m_kitchen_sink_water)
kactivity	m_kitchen_presence $\wedge$ $\mathbf{F}_{\leq 3}$(m_kitchen_sink_water $\vee$ m_kitchen_fridgedoor $\vee$ m_kitchen_cupboard)
preparing	kitchen_activity $\wedge$ $\neg$cooking
livingroom_tv	$\mathbf{F}_{\leq 3}$(m_livingroom_tv $\wedge$ m_livingroom_couch)
eating	$\neg$m_kitchen_presence $\wedge$ $\mathbf{G}_{\leq 6}$(m_livingroom_table)
actfloor(0)	cooking $\vee$ preparing $\vee$ eating $\vee$ washing_dishes $\vee$ livingroom_tv $\vee$ m_toilet
actfloor(1)	computing $\vee$ dressing $\vee$ napping $\vee$ office_tv $\vee$ reading $\vee$ shower_usage $\vee$ sink_usage
acthouse	actfloor(0) $\vee$ actfloor(1)
notwopeople	$\neg$(actfloor(0) $\wedge$ actfloor(1))
restricttv_office	office_tv $\implies$ $\mathbf{F}_{\leq 10}$($\neg$office_tv)
restricttv_living	livingroom_tv $\implies$ $\mathbf{F}_{\leq 10}$($\neg$livingroom_tv)
restricttv	restricttv_living $\wedge$ restricttv_office
firehazard	napping $\implies$ $\neg$cooking

Abstraction from implementation. Decentralized specifications define modular specifications that can be composed together to form bigger and more complex specifications. One setback for learning-based techniques to detect ADL is their specificity to the environment. That is, the training set is specific to a house layout, user profile (i.e., elderly versus adults) [vKEK10].

By using references to monitors, we leave the implementation of the specification to be specific for the house or user profile. Using our existing example, cooking is implemented based on the available sensors in the house, which would change for different houses. However, the meta-specifications such as firehazard can be defined independently from the implementation of both cooking and sleeping.

Furthermore, using monitor references, which are treated as oracles, opens the door to utilizing existing techniques in the literature for non-automata based monitors. That is, as a reference is expected to eventually evaluate to $\top$ or $\bot$, any implementation of a monitor that can return a final verdict for a given timestamp can be incorporated to form more complex specifications. For example, one can use the various machine learning techniques [BCR09, vKEK10, TIL04] to define monitors that detect specific ADLs, then reference them in order to define more complex specifications.

Scalability. Decentralized specifications allow for a higher level of scalability when writing specifications, and also when monitoring. By using decentralized specifications, we restrict a given monitor to atomic propositions local to the component on which it is attached, and references to other monitors (see Section 9.2.2). This greatly reduces the number of atomic propositions to consider when synthesizing the monitor and reduces its size, as the

sub-specifications are offloaded to another monitor.

For example, let us consider writing specifications using LTL formulae. The classical algorithm that converts LTL to Moore automata is doubly exponential in the size of the formula including all permutations of atomic propositions (to form events) [BLS11]. As such reducing both the size of the formula and the number of atomic propositions used in the formula helps significantly when synthesizing the monitors, allowing us to scale beyond the limits of existing tools. For a large formula, it becomes impossible to generate a central monitor using the existing synthesis techniques. Decentralized specifications provide a way to manage the large formula by subdividing it into subformulae. The decomposition ensures that the formula evaluates to the same verdict given the same observations, at the cost of added delay.

Example 40 (Synthesizing check light) Recall the system property $\texttt{sc\_light}(i)$ in Example 37 responsible for verifying that in a room i a light switch does indeed turn a light bulb on until the switch is turned off. We recall the LTL specification $\texttt{sc\_light}(i) \stackrel{\text{def}}{=} \mathbf{G}(\mathrm{s}_i \implies \mathbf{X}(\ell_i \,\mathbf{U}\, \neg \mathrm{s}_i))$. To verify the property across n rooms of the house, we formulate a property $\texttt{sc\_ok} \stackrel{\text{def}}{=} \bigwedge_{i\in[0..n]} \texttt{sc\_light}(i)$. In the case of a decentralized specification the formula will reference each monitor in each room, leading to a conjunction of n atomic propositions. However, in the case of a centralized specification, the specification needs to be written as: $\texttt{sc\_ok}^{\text{cent}} \stackrel{\text{def}}{=} \bigwedge_{i\in[0...n]} \mathbf{G}(\mathrm{s}_i \implies \mathbf{X}(\ell_i \,\mathbf{U}\, \neg \mathrm{s}_i))$, which is significantly more complex as a formula consisting of $4n$ operators (to cover the sub-specification), along n conjunctions, and defined over each sensor and light bulb atomic propositions ($2n$). Given that monitor synthesis is doubly exponential, both `ltl2mon` [BLS11] and `lamaconv` [Ins] require significant resources and time to generate the minimal Moore automaton (in our case[1], both tools where unable to generate the monitor for $n = 3$ after a timeout of one hour). ∗

9.3 Trace Replay with THEMIS

To perform monitoring we use `THEMIS` (Chapter 7) which is a tool for defining, handling, and benchmarking decentralized specifications and their monitoring algorithms. For replaying the trace, we perform monitoring by defining a start time, an end time and a polling interval. For this case study, for a given date, we use 07:30 as start time, 17:30 as an end time, and a 1 second polling interval.

We first recall important aspects of monitoring with `THEMIS` in Section 9.3.1 Then, in Section 9.3.2, we elaborate on the trace format provided in the public dataset, and our adaptation for replay to perform the monitoring. In brief, the process consists of extracting each sensor data converting it to observations (atomic propositions and verdicts), and passing the observation to a logical component for multiple related sensors. Finally, in Section 9.3.3, we introduce extra considerations when using `THEMIS` for monitoring large traces.

9.3.1 Using THEMIS for Monitoring

Overview. We recall from Chapter 7 that `THEMIS` is a tool to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ. It consists of a library and command-line tools. `THEMIS` provides an API, data structures, and measures for decentralized monitoring. These building blocks can be reused or extended to modify existing algorithms, design new algorithms, and elaborate new approaches to assess existing algorithms. `THEMIS` encompasses existing approaches [BKZ15, CF16a] that focus on presenting one global formula of the system from which they derive multiple specifications, and in addition supports any decentralized specification.

Monitoring. `THEMIS` defines two phases for a monitoring algorithm: setup and monitor. In the first phase, the algorithm creates and initializes the monitors, connects them to each other so they can communicate, and attaches them to components so they receive the observations generated by components. In the second phase, each monitor receives observations at a timestamp based on the component it is attached to. The monitor can then perform some computation, communicate with other monitors, abort monitoring or report a verdict. The two distinct phases

[1] On an Intel(R) Core(TM) i7-6700HQ CPU, using 16GB RAM, and running openjdk 1.8.0_172, with ltl2mon 0.0.7.

separate the monitor generation (monitor synthesis) problem from the monitoring, giving algorithms the freedom to generate monitors and deploy them on components, while integrating with existing tools for monitor synthesis such as [BLS11, Ins]. The monitors used in this case study use similar logic than *choreography* [CF16a], as they are defined over a shared global clock. All monitors start monitoring at $t = 0$. A monitor monitors the compliance of the specification for a given timestamp t, which could take a fixed delay d to check. After reaching the delay at $t + d$, the monitor reports the verdict for t to all other monitors that depend on it, and starts monitoring the specification again for $t + 1$ (i.e., it *respawns*). As such, the communication between monitors consists of sending verdicts for given timestamps.

Data structures. `THEMIS` provides two main data structures for monitoring: *memory* and *execution history encoding* (EHE), they are detailed in Chapter 5. The *memory* buffers all observations the monitor received, either from being attached to a component or from other monitors. The EHE encodes the execution of the underlying automaton, keeping track of potential states when receiving partial observations. In brief, an EHE can be modeled as a partial function that associates a timestamp t and a state q of the automaton with a boolean expression e, whenever e holds, we are sure that the automaton is in state q at timestamp t. As such, EHE relies on boolean simplification to determine the state of the automaton. The memory footprint for monitors consists of the sizes of their *memory* and *EHE*.

9.3.2 Generating the Trace

Provided trace. The trace from [CLRC17] is given as a database with a table for each sensor. We extract each table as a *csv* file for each sensor. The provided sensor data is stored as entries of values associated with timestamps, representing the changes in the sensor data across time. Typically, a new entry is provided whenever a change in the sensor data occurs. The data provided either consists of Boolean domains or numbers such as integers or reals (double).

Generating atomic propositions. The sensor data needs to be processed to create observations, as LTL3 monitors (see Section 9.2.1) operate on atomic propositions. Each sensor is implemented as an input (*Periphery* in `THEMIS`) to a logical component. For example, for the shower water, we use both cold and hot water sensors but define only a single component (“shower water”), from an RV perspective, “hot” and “cold” are multiple observations passed to the “shower water” component. To process different sensor data, we implemented two peripheries: *SensorBool* and *SensorThresh*. The first periphery parses Boolean values from the *csv* file associated with timestamps. The processing associates Boolean values $\top$ (resp. $\bot$) based on sensor data such as: "ON" (resp. "OFF"), and "OPEN" (resp. "CLOSED"). The second periphery reads real (double) values, and returns a Boolean based on whether the number is below or above a certain threshold. Both peripheries associate the generated Boolean with a given atomic proposition to generate an observation.

Synchronizing traces. The provided dataset only provides sensor updates, that is, the data only contains timestamps and values for a sensor when the value changes. Our monitoring strategy, however, requires polling the devices at given fixed time intervals. Since the system has a global clock, to synchronize observations, our periphery implementations synchronize on a date at the start and an increase (in our case 1 second) and a default Boolean value for the observation. When polled, the periphery returns the default value if nothing is observed yet, or the last value observed otherwise. The last value observed is updated when changes occur in the *csv* file. In short, we interpolate values between changes to return the oldest value before a change.

Determining the polling rate. We take advantage of the system’s global clock to evaluate the specification synchronously for all components. As such, we need a fixed interval to poll the monitors in order to evaluate the specification, that is, take the necessary transition in each of the automata. We refer to this interval as the *polling rate*. The *polling rate* determines the frequency of evaluation of the specification; the higher the rate, the more rounds, and the more monitors process and communicate. To determine the minimal rate, we consider the rate of change for all sensors involved in the specification. We are interested in ensuring that no sensor changes twice in between the evaluation of the specification. To do so, we write a simple program that processes the trace files for

Listing 18 Rates of change for sensor data. The highlighted sensors are skipped since their data never change.

```
livingroom_table              SensorBool    28.csv    Min: 3000       Max: 230704000    (ms) [OK]
kitchen_dishwasher            SensorThresh  167.csv   Min: 2190810000 Max: 2190810000   (ms) [SKIP]
office_deskplug               SensorThresh  119.csv   Min: 6000       Max: 231159000    (ms) [OK]
office_tv                     SensorBool    283.csv   Min: 420000     Max: 343980000    (ms) [OK]
livingroom_couch              SensorBool    45.csv    Min: 3000       Max: 247031000    (ms) [OK]
kitchen_presence              SensorBool    269.csv   Min: 2000       Max: 230702000    (ms) [OK]
kitchen_c1                    SensorBool    300.csv   Min: 1000       Max: 259080000    (ms) [OK]
kitchen_c2                    SensorBool    315.csv   Min: 1000       Max: 431493000    (ms) [OK]
kitchen_c3                    SensorBool    316.csv   Min: 1000       Max: 259095000    (ms) [OK]
kitchen_c4                    SensorBool    317.csv   Min: 1000       Max: 259051000    (ms) [OK]
kitchen_c5                    SensorBool    355.csv   Min: 1000       Max: 779361000    (ms) [OK]
kitchen_sink_hotwater         SensorThresh  184.csv   Min: 12000      Max: 260085000    (ms) [OK]
kitchen_sink_coldwater        SensorThresh  189.csv   Min: 12000      Max: 260501000    (ms) [OK]
bedroom_closet_door           SensorBool    339.csv   Min: 7000       Max: 605093000    (ms) [OK]
bedroom_luminosity            SensorThresh  120.csv   Min: 1000       Max: 254250000    (ms) [OK]
kitchen_cooktop               SensorThresh  36.csv    Min: 7000       Max: 260333000    (ms) [OK]
bathroom_shower_coldwater     SensorThresh  22.csv    Min: 12000      Max: 345139000    (ms) [OK]
bathroom_shower_hotwater      SensorThresh  201.csv   Min: 12000      Max: 345066000    (ms) [OK]
kitchen_fridge_door           SensorBool    314.csv   Min: 1000       Max: 260749000    (ms) [OK]
livingroom_tv                 SensorBool    282.csv   Min: 840000     Max: 344040000    (ms) [OK]
toilet                        SensorThresh  254.csv   Min: 12000      Max: 518222000    (ms) [OK]
bathroom_sink_coldwater       SensorThresh  86.csv    Min: 12000      Max: 260437000    (ms) [OK]
bathroom_sink_hotwater        SensorThresh  264.csv   Min: 25000      Max: 25000        (ms) [SKIP]
kitchen_oven                  SensorThresh  232.csv   Min: 2191235000 Max: 2191235000   (ms) [SKIP]
bedroom_drawer_1              SensorBool    357.csv   Min: 1000       Max: 345825000    (ms) [OK]
bedroom_drawer_2              SensorBool    358.csv   Min: 2000       Max: 515617000    (ms) [OK]
bedroom_bed_pressure          SensorThresh  349.csv   Min: 1000       Max: 342361000    (ms) [OK]

                               (Detected Rate)        Min: 1000       Max: 779361000    (ms)
```

each sensor in an input specification, to determine the rate of change. Listing 18 shows an example output on the 27 sensors used for ADL detection. It shows the atomic proposition associated with the sensor, the sensor type, the trace file, the fastest change rate (min), and the slowest change rate (max), and whether or not it is skipped. The rates are provided in milliseconds. Then, we aggregate over all sensors by computing the fastest and slowest. Sensors are not included in the aggregate computation (i.e., skipped) if no change appears in their entire trace file. In this case, we choose 1 second as our polling rate, as no sensor will change twice within a second.

9.3.3 Considerations for Large Traces

Managing the trace length (36,000) is an issue for decentralized specifications. They rely on eventual consistency and will wait on input for the length of the trace, which requires a lot of memory. This was not an issue for the small traces (of length 100) used to compare algorithms. One can see that utilizing the data structures and monitors as presented in Section 9.3 poses a challenge due to the large trace length and specific specifications, as delay could grow to be the size of the trace.

Garbage collection. For processing large traces we utilize implementation MemoryIndexed of the data structure *memory* that is used to store observations to add garbage collection. Observations are indexed by timestamp. When the monitor concludes with a final verdict for timestamp t, and respawns to monitor timestamp $t+1$, all observations associated with a timestamp less than or equal to t are removed from the memory.

Delay considerations. The *EHE* data structure is designed to be as general as possible, and keeps expanding while it has not detected the state the automaton is in. For large trace sizes, this can cause an *EHE* to grow quickly to consume all available memory and prevents the monitoring from completion. This is prominently the case when monitoring safety properties. Safety properties such as $\mathtt{p} \stackrel{\text{def}}{=} \mathbf{G}(\mathtt{ap})$ will only conclude when $\mathtt{ap}$ is $\bot$. So long as $\mathtt{ap}$ is $\top$, the monitor checking $\mathtt{p}$ does not reach a final verdict, and does not report it to its parent. Consequently, a monitor that checks a safety property that is never violated, incurs a delay that is as long as the trace size. To alleviate this problem, we carefully crafted the specifications to apply operators **G** and **F** on subspecifications that can be evaluated within a very small delay. Another approach is to limit the expansion of the *EHE* to a fixed length (assuming a fixed maximal delay), and use a sliding window to maintain the limit. This approach, however, may cause monitoring not to conclude in some cases.

9.4 Assessing the Monitoring of the Appartment

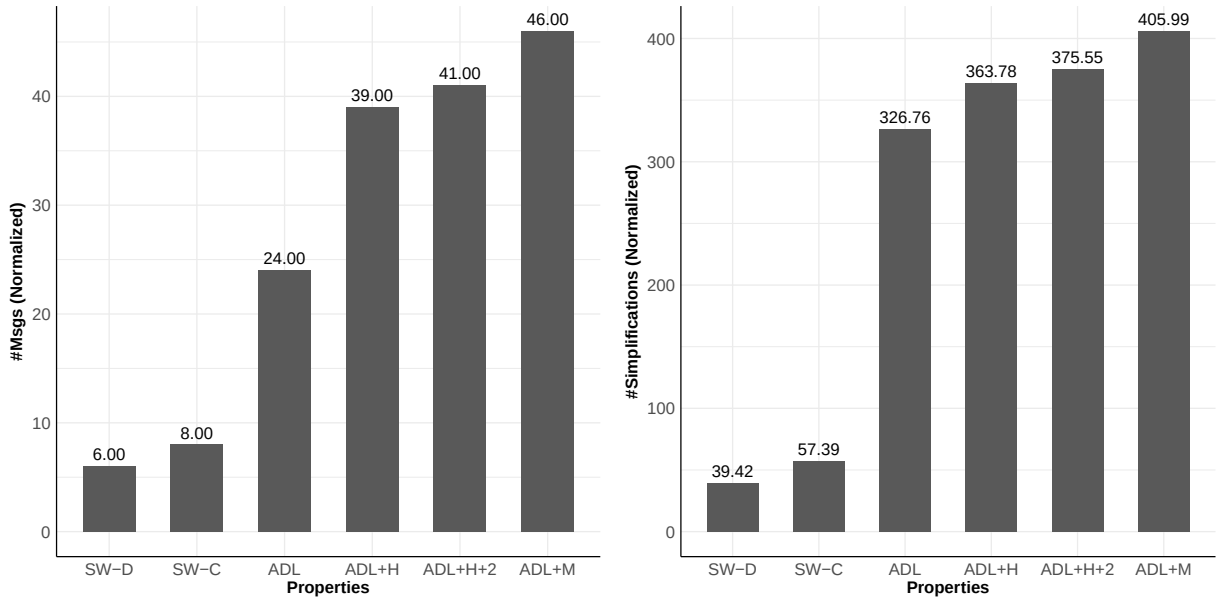
Monitoring the smart apartment requires leveraging the interdependencies between specifications to be able to scale, beyond monitoring system properties, to more complex meta-specifications (as detailed in Section 9.1.2). We assess using decentralized specifications to monitor the apartment by conducting three scenarios. The first scenario (Section 9.4.1) evaluates the advantages of using decentralized specifications presented in Section 9.2.3 (modularity, scalability, and re-use) by looking at the complexity of monitor synthesis, and communication and computation costs when adding more complex specifications that re-use sub-specifications. The second scenario (Section 9.4.2) evaluates the effectiveness of detecting ADL by looking at various detection measures such as precision and recall. The third scenario (Section 9.4.3) portrays the advantages of modularity by (i) adapting specification `napping` to use different sensors without modifying dependencies, and (ii) porting specification `firehazard` to a completely different environment (using the ARAS dataset [AEIE13]).

9.4.1 Monitoring Efficiency and Hierarchies

Monitor synthesis. Table 9.1 displays the number of atomic propositions referenced by each specification for the decentralized ($|AP^d|$) and the centralized ($|AP^c|$) settings. Column d indicates the maximum depth of the directed acyclic graph of dependencies, so as to assess how many levels of sub-specifications need to be computed. When $d = 0$, it indicates that the specification can be evaluated directly by the monitor placed on the component, while $d = 1$ indicates that the monitor has to poll at most 1 monitor for its verdict (which typically relays the component observations). More generally, when $d = n$, it indicates that the specification depends on a monitor that has at most depth $n - 1$. The atomic propositions indicate either direct references to sensor observations (in the centralized setting) or references to either sensor observations or dependent monitors (in the decentralized setting). For certain specifications such as `toilet` which relies only on the water sensor in the toilet to be detected, there is no difference between using a centralized or decentralized specification, as it resolves to the observations. Reduction becomes more pronounced when specifications re-use other specifications as sub-specifications. For example, specification $\texttt{acthouse} \stackrel{\text{def}}{=} \texttt{actfloor}(0) \vee \texttt{actfloor}(1)$, when decentralized, uses only 2 references (for each of the sub-specifications). However, when expanded, it references all 27 sensors used to detect activities. Additionally, specification $\texttt{notwopeople} \stackrel{\text{def}}{=} \neg(\texttt{actfloor}(0) \wedge \texttt{actfloor}(1))$ does not re-use the sub-specifications and requires all sensors again. This greatly reduces the formula size and allows us to synthesize the monitors needed to check the formulae, as the synthesis algorithm is doubly exponential as mentioned in Section 9.2.3.

Assessing re-use and scalability. Reducing the size of the atomic propositions needed for a specification not only affects monitor synthesis, but also performance, as atomic propositions represent the information needed to determine the specification (Section 9.2.3). To assess re-use and scalability, we perform two tasks and gather two measures pertaining to computation and communication, and present results in Figure 9.4. The first task compares a centralized (SW-C) and a decentralized (SW-D) version of specification `sc_ok` presented in Example 40 using only 2 rooms. The second task introduces large meta-specifications on top of the ADL specifications to check scalability. Firstly, we measure the communication and computation for monitoring ADL specifications (ADL). Secondly, we introduce specifications $\texttt{actfloor}(0)$, $\texttt{actfloor}(1)$ and `acthouse` (ADL+H) as they require information about all sensors for ADL. Thirdly, we add specification `notwopeople` (ADL+H+2), as it re-uses the same sub-specifications as specification `acthouse`. Lastly, we show all measures for all meta-specifications in Table 9.1 (ADL+M). We re-use two measures from Section 8.2: the total number of simplifications the monitors are doing, and the total number of messages transferred. These measures are provided directly with `THEMIS`. The total number of simplifications (**#Simplifications**) abstracts the computation done by the monitors, as they attempt to simplify Boolean expressions that represent automaton states, which are the basic operations for maintaining the monitoring data structures (Section 5.2). The total number of messages abstracts the communication (**#Msgs**), as our messages are of fixed length, they also represent the total data transferred. Both measures are normalized by the number of timestamps in the execution (36,000). The resulting normalized measures represent the number of simplifications and messages per round.

Results. Figure 9.4a shows the normalized number of messages sent by all monitors. For the first task, we notice that the number of messages is indeed lower in the decentralized setting, SW-D sends on average 2 messages per



(a) Communication

(b) Computation

Figure 9.4: Scalability of communication and computations in decentralized specifications.

timestamp less than SW-C, which corresponds to the difference in the number of atomic propositions referenced (6 for SW-D and 8 for SW-C). For the second task, we notice that on the baseline for ADL, we observe 24 messages per timestamp, a smaller number than the sensors count (27). This is because some ADL like `toilet` are directly evaluated on the sensor without communicating, and other ADL like `preparing`, re-use other ADL specifications like `kactivity`. By introducing the 3 meta-specifications stating that an activity occurred on a floor or globally in a house, the number of messages per round only increases by 15. This also coincides with the number of atomic propositions for the specifications (6 for `actfloor`(0), 7 for `actfloor`(1), and 2 for `acthouse`) as those monitors depend in total on 15 other monitors to relay their verdicts. This costs much less than polling 16 sensors to determine `actfloor`(0), 11 sensors to determine `actfloor`(1), and 27 (a total of 54) to determine `acthouse`. To verify this, we notice that the addition of `notwopeople` (ADL+H+2) that needs information from all 27 sensors, only increases the total number of messages per timestamp by 2. The specification `notwopeople` reuses the verdicts of the two monitors associated with each `actfloor` specification. After adding all the meta-specifications (ADL+M), the total number of messages per timestamp is 46, which is less than the number needed to verify adding `actfloor`, and `acthouse` in a centralized setting (54). We notice a similar effect for computation (Figure 9.4b).

9.4.2 ADL Detection using RV

Measurements. Table 9.3 displays the effectiveness of using RV to monitor all ADL specifications on the trace of three days with different schedules. To assess the effectiveness, we considered the provided self-annotated data from [CLRC17], where the user annotated the start and end of each activity. We measure precision, recall and F1 (the geometric mean of precision and recall). To measure precision, we consider a true positive when the verdict $\top$ of a monitor for a given timestamp fell indeed in the self-annotated interval for the activity. To measure recall, we measure the proportion of the intervals that have been determined $\top$ using RV. This approach is more fine-grained than the approach used in [LLR+17] where the precision and recall are computed for the start and end of intervals.

Results. The effectiveness of detection depends highly on the specification. Our approach performs well for the specifications `computing`, `cooking`, `office_tv`, as it exhibits high precision and high recall. The second group of specifications contains specifications such as `shower_usage`, and `livingroom_tv`. It exhibits high precision but medium recall, that is, we were able to determine around 40 to 50% of all the timestamps where the specifications held according to the person annotating, without any false positives. The third group is similar to the second group

Table 9.3: Precision, Recall, and F1 of monitoring all ADL specifications on three days with different schedules.

	Tuesday, Jan 31 2017			Monday, Feb 20 2017			Tuesday, Feb 21 2017		
Specification	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
`computing`	0.98	0.99	0.99	0.94	0.99	0.96	0.99	0.99	0.99
`office_tv`	1.00	0.80	0.89	1.00	0.94	0.97		-	
`cooking`	0.88	0.88	0.88	0.90	0.93	0.92		-	
`shower_usage`	1.00	0.50	0.67		-		1.00	0.63	0.77
`washing_dishes`	1.00	0.47	0.64	0.93	0.63	0.75		-	
`livingroom_tv`	1.00	0.43	0.60		-		1.00	0.47	0.64
`dressing`	1.00	0.41	0.58	1.00	0.31	0.47		-	
`toilet`*	1.00	0.18	0.30		-		0.75	0.24	0.36
`sink_usage`	1.00	0.13	0.23	1.00	0.24	0.35	0.003	0.16	0.01
`eating`	0.61	0.35	0.44	0.70	0.73	0.71		-	
`napping`	0.43	0.95	0.60	0.38	0.94	0.54		-	
`preparing`	0.23	0.77	0.35	0.21	0.79	0.34		-	
`reading`	0.37	0.04	0.06	0.02	0.10	0.03		-	

but has very low recall (13-18%) and contains the specifications `toilet` and `sink_usage`. We notice that for `sink_usage` specific user behavior can throw it off, as seen for the trace of Feb 21, we elaborate on the limitations in the next paragraph. The fourth group, which includes the specifications `napping` and `preparing`, shows high recall but a high rate of false positives. And finally, specification `reading` is not properly detected, as it has a high rate of false positives and covers almost no annotated intervals.

Limitations of RV for detecting ADL. The limitations of using RV to detect ADL are due to the modeling. As mentioned in Section 9.1.2, RV can be seen as a knowledge-based approach to activity detection, as such it suffers from similar weaknesses and limitations [CHN+12]. The activity is described as a rigid formal specification over the sensor data, and this has two consequences. Firstly, since RV relies purely on sensor data, activities which cannot be inferred from existing sensors will be poorly detected or not detected at all. This is the case for `reading`, as there are no sensors to indicate that the tenant is reading. We infer reading by checking that the light is on in the room and no other specified activity holds. Secondly, given that specifications are rigid, we expect the user to behave exactly as specified for the activity to be detected, any minor deviation results in the activity not being detected (as seen in on Feb 21). To illustrate this point, the specification `computing` relies on the power consumption of the plug in the office. Had the tenant been charging his phone instead of computing, the recall would have suffered greatly. Another great example of this is the `shower_usage` specification, that is captured by inspecting the water usage of the shower. The time the tenant spends getting into the shower and out of the shower will not be considered, which greatly impacts recall. The above issues are further compounded by the annotation being carried out by a person. The annotator can for example take a few seconds to annotate some events which could impact recall, especially for short intervals of activity. However, even with the inherent limitations of using knowledge-based approaches, our observed groups and results fall within the expected range, of knowledge-based approaches such as [LLR+17], and also have similar effectiveness as model-based SVM approaches such as [CFC15]. We elaborate on how the introduced modularity from decentralized specifications can alleviate some of these issues in Section 9.4.3.

9.4.3 Specification Adaptation for ADL Detection

Decentralized specifications introduce numerous advantages (see Section 9.2.3) for monitoring hierarchical systems that can change. We illustrated in Section 9.4.1 the scalability of decentralized specifications with hierarchies. Decentralized specifications allows specifications to be written with references to other specifications. The references allow specifications to be modular, changing the referenced specification can be done transparently with no modification to the specifications that depend on it. In this section, we illustrate the advantages of modularity in two cases. In the first case, we improve the detection of the activity `napping` by adding relevant sensors. The

Table 9.4: Modifying the decentralized specification to improve detection, and adapt to new environment.

(a) Refining `napping` using the bedroom sensors: bed pressure (`weight`), presence (`pres`), and light (ℓ).

Formula	Precision	Recall	F1
$\mathbf{G}_{\leq 25}(\texttt{weight})$	0.43	0.95	0.60
$\mathbf{G}_{\leq 3}(\texttt{weight})$	0.43	0.99	0.60
$\mathbf{F}_{\leq 3}(\texttt{weight})$	0.43	1.0	0.60
$\mathbf{G}_{\leq 3}(\texttt{pres} \wedge \texttt{weight})$	0.34	0.14	0.20
$\mathbf{G}_{\leq 3}(\neg\ell \wedge \texttt{weight})$	1.00	0.97	0.99

(b) Modifications to detect `firehazard` in ARAS.

Specification	Formula
`preparing`	$\mathbf{F}_{\leq 3}(\texttt{m\_kdrawer} \vee \texttt{m\_fridge} \vee \texttt{m\_cupboard})$
`cooking`	`preparing`
`beds`	$\texttt{bed1} \vee \texttt{bed2}$
`beds'`	$\texttt{bed1} \wedge \texttt{bed2}$
`napping`	$\mathbf{G}_{\leq 25}(\texttt{beds})$
`firehazard`	$\texttt{napping} \implies \neg\texttt{cooking}$

change only requires changing the monitor for `napping`, and no change is necessary for the remaining dependent specifications. In the second case, we apply the specification `firehazard` and all its dependencies on a completely different environment using the ARAS dataset [AEIE13].

Improving activity detection. We modify the specification `napping` to better capture the activity. This requires no change to specifications that depend on `napping`. Table 9.4a shows the changes in precision and recall, for various versions of the specification `napping`. We modify the formula to relax the time constraints on the output of the bed pressure sensor. We notice, that while this could slightly improve recall (0.95 to 1), it does not translate to any precision improvement (it remains at 0.43). We explore using additional sensors in the room to capture the specification better. Using the presence sensor proves to be detrimental as it reduces precision to 0.34 and recall to 0.14. This is reasonable, as the presence sensor is a motion detector, and when someone is sleeping there may be no motion at all. However, people typically tend to turn the lights off when sleeping. Using the additional light sensor to detect lights are off, helps us increase precision to 1 and recall to 0.99. One could see that the effect of ADL detection is behavior specific, a tenant that sleeps with lights on will have undetected sleep using our specification. Being able to change the specific specification without impacting the rest of the specification provides the flexibility to tune the ADL detection to specific users and behaviors.

Adapting to new environments. In Section 9.1.2 we mentioned that ADL can be challenging as the detection of the specification does not only depend on the user behavior, but also on the environment in which it is monitored. In the context of learning techniques, using information learned from one environment to apply it to detection of ADL in other environments is discussed in [vKEK10]. Since decentralized specifications provide both a hierarchical and modular approach to designing specifications, it is possible to adapt specifications to new environment, by only changing the relevant parts or dependencies, and reasoning at the appropriate level. For instance, while specifications specifying ADL may change depending on the sensors and user behavior, meta-specifications do not necessarily change. We adapt specification `firehazard` and all its dependencies in the ARAS [AEIE13] dataset. The ARAS dataset features contact, pressure, distance, and light sensors, recording the interactions of two tenants with the sensors over a period of 30 days.

Table 9.4b shows the changes in the decentralized specification compared with that of Amiqual4Home found in Table 9.2. For activity `preparing`, we follow a similar pattern, looking at the usage of cubpoards, fridge, and kitchen drawers. Thus, we adapt the formula to reflect the available sensors in the kitchen. However, the ARAS dataset does not provide any electricity sensors for appliances, nor any way to detect heat being turned on. As such it is impossible to detect cooking using any sensors. Since we cannot tell `preparing` and `cooking` apart, we define `cooking` to simply be equivalent to `preparing`. Notice how in this case, we inverted the dependency from Figure 9.3 (in ARAS, `cooking` depends on `preparing`). The ARAS dataset records the behavior of *two* people, instead of just *one*. As such, activity `napping` needs to be adjusted for the *two beds*. There are two ways to do so, the first assumes either one of the tenants is napping (`beds`), and the second assumes both are napping simultaneously (`beds'`). We notice that the meta-specification `firehazard` remains unchanged. However, it has two different interpretations. If we use `beds`, then it is possible to trigger `firehazard` when one tenant is cooking while the other is sleeping. We verify that, and notice that it is indeed falsified in 8 days (7, 9, 16, 17-19, 24, 27).

Using `beds'`, allows us to only capture `firehazard` when both tenants are sleeping. It is then possible to refer `napping` to `allnapping` and `anynapping`, then using `firehazard` on `allnapping`, which would apply in both scenarios.

Discussion. We see that modularity provides several advantages. It allows us to make local change to specifications that do not need to be propagated upwards. It also makes it possible to generalize and abstract the specification to adapt to multiple environments. Decentralized specifications allow specifications to be written in a modular and adaptable fashion, allowing specifications to be adapted to target changes in user behavior and environment. It can be seen much like component-based design [SGM02], which separate the implementation of each component in software, from its interaction with other components.

9.5 Related Work

We present similar or useful techniques for detecting ADL in a smart apartment that use log analysis and complex event processing. Then, we present techniques from stream-based RV that can be extended for monitoring smart apartments.

ADL detection using log analysis. Detecting ADL can be performed using trace analysis tools. The approach in [LLR+17] defines parametric events using Model Checking Language (MCL) [MT08] based on the modal mu-calculus (inspired by temporal logic and regular expressions). Traces are read and transformed into actions, then actions are matched against the specifications to determine locations in the trace that match ADL. Five ADL (sleep, using toilets, cooking, showering, and washing dishes) are specified and checked in the same smart apartment as our work. While this technique is able to detect ADL activities, it amounts to checking traces offline, and a high level of post-processing is required to analyze the data. In [BCE+16], the authors describe an approach for log analysis at very large scale. The specification is expressed using Metric First Order Temporal Logic (MFOTL), and logs are expressed as a temporal structure. The authors develop a *MapReduce* monitoring algorithm to analyze logs generated by more than 35,000 computers, producing approximately 1 TB of log data each day. While this approach is designed for distributed systems, does not map dependencies, and works offline, it could be used to process and monitor rich specifications over sensor data seen as log files.

ADL detection using Complex Event Processing. Reasoning at a much higher level of abstraction than sensor data, the approach in [HGB16] attempts to detect ADL by analyzing the electrical consumption in the household. To do so, it employs techniques from Complex Event Processing (CEP), in which data is fed as streams and processed using various functions to finally output a stream of data. In this work, the ADL detection is split into two phases, one which detects peaks and plateaus of the various electrical devices, and the second phase uses those to indicate whether or not an appliance is being used. This illustrates a transformation from low-level data (sensor signal) to a high-level abstraction (an appliance is being used). The use of CEP for detecting ADL is promising, as it allows for similar scalability and abstraction. However, CEP's model of named streams makes it hard to analyze the specification formally, making little distinction between specification and implementation of the monitoring logic.

ADL detection using Runtime Verification. Similarly to CEP but focusing on Boolean verdicts, various stream-based RV techniques have been elaborated such as LOLA [DSS+05] which are used to verify correctness properties for synchronous systems such as the PCI bus protocol and a memory controller. A more recent approach uses the Temporal Stream-Based Specification Language (TeSSLa) to verify embedded systems using FPGAs [DDG+18]. Stream-based RV is particularly fast and effective for verifying lengthy parametric traces. However, it is unclear how these approaches handle monitor synthesis for a large number of components and account for the hierarchy in the system.

Discussion. Stream-based systems such as stream-based RV and CEP are bottom-up. Data in streams is eventually aggregated into more complex information and relayed to a higher level. Decentralized specifications also support top-down approaches, which would increase the efficiency of monitoring large and hierarchical systems. To

illustrate the point, consider the decentralized specification in Figure 6.2b. In the automaton $\mathcal{A}_{\texttt{sc\_light}_i}$, the evaluation of the dependent monitor $\mathcal{A}_{\ell_i}$ only occurs when reaching q_1, so long as the automaton is in q_0, no interaction with the dependent monitor is necessary. This top-down feedback can be used to naturally optimize dependencies and increase efficiency. Because of the oracle-based implementation of decentralized specifications, it is possible to integrate any monitoring reference that eventually returns a verdict. One could imagine integrating other stream-based monitors or even data-driven ADL detection approaches. The integration works both ways, as monitors can be considered a (blocking) stream of verdicts for the other techniques.

Conclusion and Future Work

Conclusion. Monitoring a smart apartment presents RV with interesting new problems as it requires a scalable approach that is compositional, dynamic, and able to handle a multitude of devices. This is due to the hierarchical structure imposed by either limited communication capabilities of devices across geographical areas or the dependencies between various specifications. Attempting to solve such problems with centralized specifications is met with several obstacles at the level of monitor synthesis techniques (as we are presented with large formulae), and also at the level of monitoring as one needs to model interdependencies between formulae and re-use the sub-specifications used to build more complex specifications. We illustrate how decentralized specifications tackle such systems by explicitly modeling of interdependencies between specifications. Furthermore, we illustrate monitoring specifications that detect ADL in addition to system properties and even more specifications defined over both types of specifications.

Future work. We believe that the use of decentralized specifications could be further extended to bring monitoring closer to data (collected on sensors), and make RV a suitable verification technique for edge computing. One challenge of the case study was to determine the correct sampling period for monitor to operate. Further investigation is required to layout the tradeoffs between the sampling period, communication overhead, and energy consumption. Also, decentralization is only supported by specifications based on the standard (point-based) LTL3 semantics. We believe that the use and decentralization of richer specification languages are desirable. For instance, we consider (i) using a counting semantics able to compute the number of steps needed to witness the satisfaction or violation of a specification [BBNR18] (ii) using techniques allowing to deal with uncertainty (e.g., in case of message loss) [BG13] (iii) using spatio-temporal specifications (e.g. [HJK+15]) to reason on physical locations in the house, and (iv) using a quantitative semantics possibly with time [BFMU17]. Finally, we consider using runtime enforcement [Fal10, FMFR11, FMRS18] techniques (especially those for timed specifications [FJMP16]) to guarantee system properties and improve safety in the house (e.g., disabling cooking equipment whenever specification `firehazard` is violated). This requires to define the foundations for decentralized runtime enforcement on the theoretical side, and provide houses and monitors with actuators on the practical side.

Part III

Instantiation for Multithreaded RV

CHAPTER 10

Challenges for Multithreaded RV

Contents

Chapter abstract

In this chapter, we review some of the main RV approaches and tools that handle multithreaded Java programs to highlight the challenges RV faces when targeting multithreaded programs. We recall that programs are instrumented to extract necessary information from the execution and feed it to monitors tasked with checking the properties. Parallel programs generally introduce an added level of complexity on the program execution due to *concurrency*. A concurrent execution of a parallel program is best represented as a partial order. A large number of RV approaches generate monitors using formalisms that rely on total order, while more recent approaches utilize formalisms that consider multiple traces. We discuss their assumptions, limitations, expressiveness, and suitability when tackling parallel programs such as *producer-consumer* and *readers-writers*. By analyzing the interplay between specification formalisms and concurrent executions of programs, we identify *four* questions RV practitioners may ask themselves to classify and determine the situations in which it is sound to use the existing tools and approaches.

Listing 10.1: A shared queue for *producer-consumer*.

```
1 public class SynchQueue {
2   private LinkedList<Integer> q = new LinkedList<Integer>();
3   public void    produce(Integer v)    { q.add(v); }
4   public Integer consume()      { return q.poll(); }
5 }
```

Thread 0 (Producer)	*Thread 1 (Consumer)*
① `sq.produce(0);`	② `sq.consume(); //0`
③ `sq.produce(1);`	④ `sq.consume(); //1`

Figure 10.1: Operations for a single producer and a single consumer thread operating on a shared queue (`sq`). Shaded circles specify a given number associated with the statement.

Introduction

Traces typically contain operations and events that a program executes. They are versatile: they serve to analyze, verify and characterize the behavior of a program. A single trace records information of a program execution. Information serves to profile the run of a program [ABF+10] so as to optimize its performance. Alternatively, a trace abstracts a single program execution, to verify behavioral properties expressed using formal specifications. A collection of traces model the program behavior as it allows to reason about possible executions or states. As such, multiple traces serve to check for concurrency properties [LFKV18] such as absence of data races [HMR14, SWYS11] and deadlock freedom [HR04].

In RV, Programs are instrumented to extract necessary information from the execution and feed it to monitors (Section 1.2). This information is typically referred to as the *trace* [RH16]. Monitors are synthesized from behavioral properties, they check if the trace complies with the properties. From the monitor perspective, the system is a black box; the trace is the *sole* system information provided. Therefore, for any RV technique, providing traces with correct and sufficient information is necessary for sound and expressive monitoring[1].

Parallel programs introduce an added level of complexity because of concurrency. The introduction of concurrency can result in the collected trace not being representative of the actual *concurrent execution* of a parallel program. A concurrent execution is best modeled as a partial order over *actions* executed by the program. The actions can represent function calls, or even instructions executed at runtime. The order typically relates actions based on time, it states that some actions happened before other actions. Actions that are incomparable are typically said to be *concurrent*. This model is compatible with various formalisms that define the behavior of concurrent programs such as weak memory consistency models [AG96, ANB+95, MPA05], Mazurkiewicz traces [Maz86, GK10], parallel series [LW01], Message Sequence Charts graphs [MR04], and Petri Nets [NPW81]. We introduce a text-book example of a multithreaded program, *producer-consumer* in Example 41.

Example 41 (Producer-consumer) We consider the classical *producer-consumer* example where a thread pushes items to a shared queue (generating a `produce` event), and another thread consumes items (one at a time) from the queue for processing (generating a `consume` event). We specify that consumers must not remove an item unless the queue contains one, and all items placed on the queue must be eventually consumed. Figure 10.1 illustrates the statements executed by two different threads: thread 0, and thread 1, representing respectively a producer and a consumer. Each statement is given a number for clarity. Both the producer and consumer use a shared queue shown in Listing 10.1. Statements in different threads can execute concurrently. We illustrate some correct and incorrect executions. Two correct executions have the following orders: ①②③④ and ①③②④; they comply with the specification. The execution with the order: ②①③④ is incorrect, as a `consume` attempts to retrieve an element from an empty queue. The execution with only the statements: ①③② is incorrect, as there remains an element to be consumed. The execution with the order: ②④①③ violates both conditions in the specification, since two `consume` events happen when the queue is empty, and after the executions there are two elements left to be consumed. ∗

[1] By soundness, we refer to the general principle of monitors detecting specification violation or compliance only when the actual system produces behavior that respectively violates or complies with the specification.

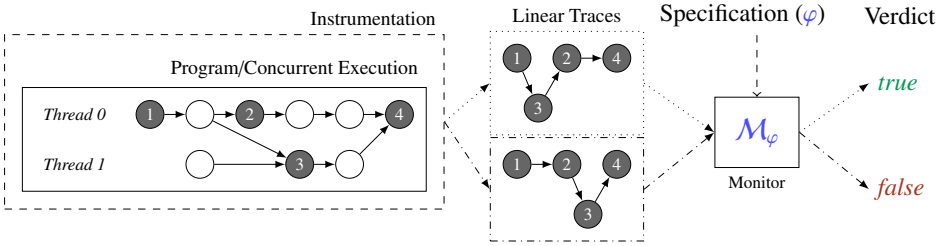


Figure 10.2: RV flow and the impact of linearizing traces. Before runtime, RV is applied to a program with a concurrent execution (dashed): a monitor $\mathcal{M}_\varphi$ is synthesized from a property φ, and the program is instrumented to retrieve its relevant events. At runtime, we observe two possible linear traces that could lead to verdicts (*true* or *false*) when processed by the same monitor.

Monitoring multithreaded programs. RV has initially focused on utilizing totally ordered traces, as it uses formalisms inspired from Linear Temporal Logic (LTL) or finite-state machines as specifications [LS09b, BHL+10, RCR15, MJG+12], until recently with the introduction of stream-based RV [DSS+05, HKG17, DGH+17], decentralized monitoring [BF12], and RV of hyperproperties [CS10]. Most of the top[2] existing tools for the online monitoring of Java programs rely on totally ordered traces and provide multithreaded monitoring support using one or more of the *three* modes. The first mode allows *per-thread* monitoring. The *per-thread* mode specifies that monitors are only associated with a given thread, and receive all events of the given thread. Monitors are unable to check properties that involve events across threads. This boils down to doing classical RV of single-threaded programs, assuming each thread is an independent program. When examining each thread or process while excluding others, one ignores the inter-thread dependencies, and it is therefore insufficient. For example, it is impossible to monitor *producer-consumer* illustrated in Example 41, as events happen on separate threads. In that setting, a specification cannot express behavior involving events across threads. The second mode allows for *global* monitoring. It spawns global monitors, and ensures that events are fed to the monitor *atomically*, by utilizing locks. As such, a program execution is *linearized* so that it can be processed by the monitors. Locks force events to be totally ordered across the entire execution, which oversimplifies and ignores concurrency, as illustrated in Example 42.

Example 42 (Linearization) Figure 10.2 illustrates the typical RV flow for some property φ with a monitor $\mathcal{M}_\varphi$, where during the execution, an instrumented parallel program feeds a trace to a monitor. Filled circles represent the events relevant to the RV specification, and are numbered simply to distinguish them. We notice that, in the case of a concurrent execution, the trace could differ based on the linearization strategy which influences the observation order. In the first trace, event 3 precedes event 2, while in the second trace, we have the opposite. This could potentially impact the verdict of the monitor if the specification relies on the order between events 2 and 3. We recall *producer-consumer* from Example 41: if the program is not properly synchronized, linearizing the concurrent events could lead two different traces: ①②③④, and ②①③④. The first trace complies with the specification while the second violates it. ∗

The third mode allows monitors to receive events concurrently. This is typically done by providing a flag *unsynchronized*. In this mode, practitioners should handle the concurrency on their own, and in some cases specify their own monitoring logic. Writing additional concurrency logic, and managing concurrency has *three* disadvantages. First, by writing the monitors manually, we defeat the purpose of automatically generating monitors from a given formalism. Second, the manual monitors created may miss key information needed for managing concurrency. This extra information may require to implement additional instrumentation outside the tool. Third, the process is complicated due to concurrency, and is error-prone. We elaborate on the complications in Section 10.3. As such, we first ask if monitors are to be generated from a formalism.

Q0: Is the developer using the tool to automatically generate monitor logic?

For the scope of this chapter, we focus on the formalisms from which monitors could be synthesized. As such, we consider the answer to **Q0** is *yes*.

[2]Based on the first three editions of the Competition on Runtime Verification [BBF14, FNRT15, RHF16, BFB+17b].

Chapter Overview. In this chapter, we explore RV tools that explicitly handle multithreaded programs. We illustrate the problem of monitoring a parallel program using existing techniques. In doing so, we overview the related approaches, some of the existing tools, and their shortcomings. We discuss their assumptions, advantages, limitations, and suitability when tackling two textbook parallel programs: producer-consumer and readers-writers. In particular, we use manually written monitors using AspectJ [KHH+01b, The18], Java-MOP [CR05, CR07, MJG+12], and RVPredict [HMR14] to explore the challenges to monitoring multithreaded programs. By analyzing the interplay between specification formalisms and concurrent executions of programs, we propose *four* questions RV practitioners may ask themselves to classify and determine the situations in which it is reliable to use the existing tools and approaches as well as the situations where we believe more work is needed.

An online tutorial [EF18] is provided with the programs, tools, and an interactive guide to reproduce and experiment with the examples provided in this chapter. The examples included in the online tutorial are marked in the rest of the chapter with the dagger sign (†).

Key Contributions. This chapters explores challenges for multithreaded RV.

Overall, the challenges of monitoring multithreaded programs stem from the following facts:

- events in a concurrent program follow a partial order;
- most formalisms used by RV do not account for partial orders, but specify behavior over sequences of events (i.e., events are totally ordered); and
- an instrumented program must capture the order of events as it happens during the execution to pass it to monitors.

Moreover, we explore the situations where:

- a linear trace does not represent the underlying program execution;
- a linear trace hides some implicit assumptions which affect RV; and
- it is insufficient to use a linear trace for monitoring multithreaded programs.

10.1 Exploring Tools and Their Supported Formal Specifications

Runtime Verification approaches typically automatically synthesize monitors by relying on a formal specification of the expected behavior. A specification formalism allows to express properties that partition the system behaviors into correct and incorrect ones. As such, for a multithreaded program, we must first check the available properties that we can verify. We first classify the various approaches by considering the specification formalism alone.

10.1.1 Approaches Relying on Total-Order Formalisms

The first pool consists of tools and approaches where the specification language itself relies on a total order of events, as the input to monitors consists of words. We consider the tools commonly used for RV using those found in the RV competitions [BFB+17b, FNRT15, RHF16].

Java-MOP. Java-MOP [CR05, CR07, MJG+12] follows the design principle that specifications and programs should be developed together. Java-MOP provides *logic plugins* to express the specifications in several formalisms. Logic plugins include: finite-state machines, extended regular expressions, context-free grammars, past-time linear temporal logic, and string rewriting systems.

Tracematches. Tracematches [AAC+05, BHL+10] is another approach that uses regular expressions over user-specified events as specifications. Tracematches defines points in the execution where events occur, and specifies the actions to execute upon matching. Tracematches considers the semantics of such matching on large programs or multiple program runs, while binding the context associated with each event to the sequence. For example, it considers when a pattern matches multiple times, or matches multiple points in the program.

MarQ. MarQ [RCR15] is designed for monitoring properties expressed as Quantified Event Automata (QEA) [BFH+12]. MarQ focuses on performing highly optimized monitoring, by providing full control of monitors lifecycles and garbage collection. Furthermore, it introduces quantification and distinguishes quantified from free variables in a specification, this allows finer control over the monitoring procedure by managing the replication of monitoring (slicing). MarQ relies on the developer to instrument the program with AspectJ to send the events to the QEA.

LARVA. LARVA [CPS09b] uses dynamic automata with timers and events [CPS09a]. LARVA focuses on monitoring real-time systems where timing is of importance. LARVA specifications feature timeouts and stopwatches. LARVA is also capable of verifying large programs by storing events in a database and allowing the monitors to "catch up" with the system as it executes [CPA10].

> **Remark 5 (Unsynchronized monitors)** While we focus on formalisms capable of automatically generating monitors, we note that it is still possible to write unsynchronized monitors manually. We explain in Section 10.3 the difficulties that make the process error-prone. Java-MOP provides the *unsynchronized* flag to specify that no additional locks should be added, thus allowing monitors to receive events concurrently. *Logic plugins* would no longer be used to automatically synthesize monitors. MarQ by default is not thread safe [RCR15]. The developer must pre-process the events before passing them to the QEA monitor. ∗

10.1.2 Approaches Focusing on Detecting Concurrency Errors

The second pool of tools is concerned with specific behavior for concurrent programs. We consider absence of data races and deadlock freedom. Tools used that can verify specific properties related to concurrency errors include RVPredict [HMR14] and Java PathExplorer (JPaX) [HR04]. Further discussion on concurrency errors and additional tools are discussed in [LFKV18].

RVPredict. RVPredict relies on Predictive Trace Analysis (PTA) [HMR14, SWYS11]. PTA approaches model the program execution as a set of traces corresponding to the different orderings of a trace. As such, they encode the trace minimally, then restrict the set of valid permutations based on the model that is allowed. The approach in [HMR14] describes a general sound and complete model to detect data races in multithreaded programs and implement it in RVPredict. Traces are ordered permutations containing both control flow operations and memory accesses, and are constrained by axioms tailored to data race and sequential consistency. While [HMR14] can, in theory, model behavioral properties, RVPredict monitors only data races, but does so very efficiently.

JPaX. Similar to RVPredict, Java PathExplorer (JPaX) [HR04] is a Java tool designed for multithreaded programs. JPaX uses bytecode-level instrumentation to detect both race conditions and deadlocks in a multithreaded program execution. To do so, JPaX tracks information on locks and variables accessed by various threads during an execution. JPaX supports standard formalisms such as LTL and finite-state machines. However, it separates those from the two mentioned concurrency properties, and defaults to providing an event stream to the monitors similar to automata-based approaches.

10.1.3 Approaches Utilizing Multiple Traces

The third pool consists of approaches specifying behavior that spans multiple traces.

Stream-based techniques. Stream-based techniques introduced in Section 3.4 include LOLA [DSS+05], Beep-Beep [HKG17], and more recently, the Temporal Stream-Based Specification Language [DGH+17, CHL+18, DDG+18]. Stream-based specifications rely on named streams to provide events. These streams are then aggregated using various functions that modify the timing, filter events, and output new events.

Decentralized monitoring. Decentralized monitoring introduced in Section 3.1.2 considers the system as a set of components sharing a logical timestamp. It uses monitoring algorithms and communication strategies to monitor one specification over components by avoiding synchronization and with the aim of minimizing the communication costs. Algorithms manage a decentralized trace associating each event with a component and a timestamp; essentially managing for each component a totally ordered trace. DecentMon [BF16, CF16b] is a tool capable of simulating the behavior of decentralized monitoring algorithms.

Hyperproperties. *Hyperproperties* [CS10] are specified over sets of traces. They are used to relate properties over different multiple runs of a program. Notably, these include security policies as they regulate how a program can be used, and those cannot be inferred by only considering one execution. Typically, hyperproperties make use of variables that are quantified over multiple traces. Examples of hyperproperties include secure information flow which regulates what information can be learned by users when interacting with the system. RV approaches have been implemented to verify hyperproperties using rewriting [BSB17], and using model checking and automata [FRS15]. RVHyper [FHST18] is a tool capable of verifying hyperproperties on sets of traces.

10.1.4 Outcome: A First Classification

Since concurrent executions exhibit a partial order between events, formalisms that rely on total order require that the partial order be coerced into a total order. Our first consideration for monitoring concurrent programs relies solely on the specification formalism.

> **Q1:** Are the models of the specification formalism based on a total order?

If the answer to **Q1** is *yes*, then we are concerned with the first pool of tools. We elaborate on further considerations for total order approaches in Section 10.2. Otherwise, we verify whether or not we are checking very specific properties on partial orders, such as data race or deadlock freedom.

> **Q2:** Are we only concerned with the absence of data races or deadlock freedom?

If the answer to **Q2** is *yes*, then we are concerned with the second pool of tools, keeping in mind that they are unable to handle arbitrary specifications. Otherwise, we are concerned with the third pool, we elaborate on the potential of using these approaches in Section 10.4.2.

10.2 Linear Specifications for Concurrent Programs

In this section, we are concerned with RV approaches that rely on total-order formalisms (e.g., automata, LTL, regular expressions). We refer to specifications that use total-order formalisms to describe the behavior of the system as *linear specifications*. We explore the assumptions and outcomes of checking properties specifying total-order behavior.

10.2.1 Per-thread Monitoring

Overview. A simple approach to monitor multithreaded programs is to consider each thread in the program execution independent. That is, the monitoring technique assumes that each thread is a separate serial program to monitor. A monitor is assigned to each thread and receives only events pertaining to that thread. This is called

Table 10.1: Monitoring 10,000 executions of 2 variants of *producer-consumer* using global monitors. Reference (REF) indicates the original program. Column **V** indicates the variant of the program. Column **Advice** indicates intercepting *after* (A) and *before* (B) the function call, respectively. Columns **True** and **False** indicate the number of executions (#) and the percentage over the total number of executions (%) for which the tool reported these verdicts.

V	Consumers	Tool	Advice	True		False		Timeout	
				#	%	#	%	#	%
1	1-2	REF			-			0	(0%)
		JMOP	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
		MarQ	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
		LARVA	A	10,000	(100%)	0	(0%)	0	(0%)
			B	10,000	(100%)	0	(0%)	0	(0%)
2	1	REF			-			631	(6.3%)
		JMOP	A	4,043	(40.43%)	5,957	(59.57%)	0	(0%)
			B	7,175	(71.75%)	6	(0.06%)	2,819	(28.19%)
		MarQ	A	4,404	(44.04%)	5,583	(55.83%)	13	(0.13%)
			B	9,973	(99.73%)	16	(0.16%)	11	(0.11%)
		LARVA	A	4,755	(47.55%)	5,245	(52.45%)	0	(0%)
			B	9,988	(99.88%)	2	(0.02%)	10	(0.10%)
2	2	REF			-			4,785	(47.85%)
		JMOP	A	128	(1.28%)	9,220	(92.20%)	652	(6.52%)
			B	1,260	(12.60%)	7,617	(76.17%)	1,123	(11.23%)
		MarQ	A	33	(0.33%)	9,957	(99.57%)	10	(0.10%)
			B	432	(4.32%)	9,530	(95.30%)	38	(0.38%)
		LARVA	A	250	(2.50%)	9,488	(94.88%)	262	(2.62%)
			B	5,823	(58.23%)	4,131	(41.31%)	46	(0.46%)

per-thread monitoring. Java-MOP and Tracematches support flag *perthread* [AAC+05, For18] to monitor a property on each thread independently. It is also possible to use MarQ by quantifying over the threads, to monitor each thread independently for a given property.

Example 43 (Per-thread iterator†) We use for example the classical property described in [CR05] "An iterator's method `hasNext` must always be called at least once before a call to method `next`". Monitoring *per-thread* proves useful, when we are concerned about the usage of iterators in a given thread, and not across threads. Using Java-MOP, we can monitor a simple program that has two threads processing a shared list of integers concurrently. Each thread creates an iterator on the shared list, the first finds the minimum, while the second finds the maximum. In this case, it is sufficient to check that the iterator usage is correct for each thread independently. ∗

Limitations. Since *per-thread* monitoring performs RV on a single thread, and all events in a given thread are totally ordered, it follows that monitoring is sound in such situations. However, in most cases, we may be interested in monitoring events across threads. This is the case with *producer-consumer* detailed in Example 41. To monitor the program we need to keep track of produces and consumes. By considering threads separately, one is not able at all to monitor the correct behavior, as producer and consumer are separate threads. Monitoring *per-thread* is not useful in this setting. Therefore, it becomes important to distinguish between properties for which events are shared across threads.

Q3: Does there exist a model of the specification where events are generated by more than a single thread?

We addressed in this section the tools and limitations when the answer to **Q3** is *no*. When the answer to **Q3** is *yes*, a developer has to consider *global* monitoring, explained in Section 10.2.2.

10.2.2 Global Monitoring

Overview.

Whenever the specification formalism relies on events across threads, the existing approaches that use a total-order formalism typically define global monitors. This is the default mode for Java-MOP, MarQ, and for Tracematches this is called "*global tracematch*". This is the only mode for LARVA. Furthermore, these tools typically include synchronization guards on such monitors. For example, LARVA synchronizes events passed to the monitors, such that a monitor cannot receive two events concurrently, while MarQ requires the developer to specify synchronization when needed, and Java-MOP offers an *unsynchronized* flag, to disable locking on monitors.

We discussed the implications of using *unsynchronized* in Section 5. We next present Example 44 in which we monitor *producer-consumer* (Example 41) using Java-MOP, LARVA, and MarQ[3].

Example 44 (Monitoring producer-consumer†) The property can be expressed as a context-free grammar (CFG) using the rule: `S -> S produce S consume | epsilon`. We specify the property for each tool and associate events `produce` and `consume` with adding and removing elements from a shared queue, respectively. We first verify this example using *per-thread* monitoring using Java-MOP, and notice quickly that the property is violated, as the first monitor is only capable of seeing produces, and the second only consumes. Using global monitoring, we monitor a large number of executions (10,000) of two variants of the program, and show the result in Table 10.1. For each execution, the producer generates a total of 8 `produce` events, which are then processed using up to 2 consumers. The first variant is a correctly synchronized *producer-consumer*, where locks ensure the atomic execution of each event. The second variant is a non-synchronized *producer-consumer*, and allows the two events to be fed to the monitors concurrently. In both cases, the monitor is synchronized to ensure that the monitor processes each event atomically. Additional locks are included by Java-MOP and LARVA, we introduce a lock for MarQ, as it is not thread-safe. This is consistent as to check the CFG (or the automaton for LARVA and MarQ), we require a totally ordered word, as such traces are eventually linearized.

In the first variant, the monitor outputs verdict *true* for all executions. This is consistent with the expected behavior as the program is correctly synchronized, as such it behaves as if it were totally ordered. However, with no proper synchronization, `produce` and `consume` happen concurrently, we obtain one of two possible traces:

$$tr_1 = \texttt{produce} \cdot \texttt{consume} \quad \text{and} \quad tr_2 = \texttt{consume} \cdot \texttt{produce}.$$

While tr_1 seems correct and tr_2 incorrect, `produce` and `consume` happen concurrently. After doing 10,000 executions of the second variant, monitoring is unreliable: we observe verdict *true* for some executions, while for others, we observe verdict *false*. Even for the same tool, and the same number of consumers, we notice that the reported verdicts vary depending on whether or not we choose to intercept before or after the function call to create the event. For example, even when using a single consumer with Java-MOP, we see that the verdict rate for verdict *false* goes down from 60% when intercepting before the function call, to almost 0% when intercepting after the function call. We note that selecting to intercept before or after a method call can depend on the specification. For consistency reasons, we chose to intercept both events in the same way. Either choice produces inconsistent verdicts when concurrency is present, due to context switches.

In the second variant, the consumer must check that the queue has an element, and then poll it to recover it. Since it is badly synchronized, it is possible to deadlock as the check and the poll are not atomic. In this case, the program cannot terminate. To distinguish deadlocked executions, we terminate the execution after 1 second, and consider it a timeout, since a non-deadlocked execution takes less than 10 milliseconds to execute. It is important to note that when the specification detects a violation the execution is stopped, this could potentially lower the rate of timeouts. The rate of timeout of the original program (REF) is given as reference. We notice that the tools interfere with the concurrency behavior of the program in two ways. First, the locking introduced by the global monitoring can actually force a schedule on the program. We observe that when a single consumer is used and locks are used before the function call. In this case, the rate of getting verdict *true* is higher than when introduced after the call (72% for Java-MOP, 99.7% for MarQ, and 99.8% for LARVA). When the locks are applied naively, they can indeed correct the behavior of the program, as they force a schedule on the actions `produce` and `consume`. This, of course, is coincidental, when 2 consumers are used, we stop observing this behavior. Second, we observe that changing

[3] On Java openjdk 1.8.0_172, using Java-MOP version 4.2, MarQ version 1.1 commit 9c2ecb4 (April 7, 2016), and LARVA commit 07539a7 (Apr 16, 2018). The equivalent specifications are presented in in Appendix B.2

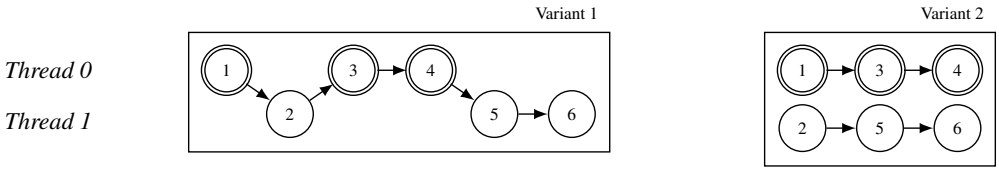


Figure 10.3: Concurrent execution fragments of *producer-consumer* variants. Double circle: `produce`, normal: `consume`. Events are numbered to distinguish them.

Listing 19 RVPredict (partial) output for *producer-consumer* variant 2.

```
1  ----------------Instrumented execution to record the trace-----------------
2  [RV-Predict] Log directory: /tmp/rv-predict2523508450121758452
3  [RV-Predict] Finished retransforming preloaded classes.
4  main Complete in 28
5  Data race on field java.util.LinkedList.$state:
6      Read in thread 14
7        > at SynchQueue.consume(SynchQueue.java:24)
8          at Consumer.run(Consumer.java:14)
9      Thread 14 created by thread 1
10         at java.util.concurrent.ThreadPoolExecutor.addWorker(Unknown Source)
11
12     Write in thread 13
13       > at SynchQueue.produce(SynchQueue.java:18)
14         at Producer.run(Producer.java:19)
15     Thread 13 created by thread 1
16         at java.util.concurrent.ThreadPoolExecutor.addWorker(Unknown Source)
```

the interception from before to after the function call modifies the rate of timeout. For example, when using 1 consumer, the reference rate is 6% (REF). When using Java-MOP (B), the rate goes up to 28%, while for LARVA (B) it goes down to 0.1%. It is possible to compare the rate of timeout of Java-MOP (B) and LARVA (B) since the monitor is not forcing the process to exit early, as the rate of reaching verdict *false* is low for both ($< 0.1\%$). We elaborate more on the effect of instrumentation on concurrency in Section 10.3. ∗

To understand the inconsistency in the verdicts, we look at the execution fragments of each variant in Figure 10.3. In the first variant, the program utilizes locks to ensure the queue is accessed atomically. This allows the execution to be a total order. For the second variant, we see that while we can establish order between either `produce`, or `consume`, we cannot establish an order between events. During the execution, multiple total orders are possible, and thus different verdicts are possible.

Limitations.

It is now possible to distinguish further situations where it is reliable to use global monitors. We notice that to evaluate a total order formalism, we require a trace which events are totally ordered. When dealing with a partial order, tools typically use locks and ensure that the partial order will be coerced into a total order. We see that the monitoring of the second variant failed since the program was not properly synchronized. One could assume that it is necessary to first check that the program is properly synchronized, and perhaps deadlock-free as well. To do so, one could use RVPredict or JPaX to verify the absence of data race (as shown in Example 45). Upon verifying that the program is synchronized, one could then run global monitors.

Example 45 (Detecting data race†) Let us consider the second variant of *producer-consumer* as described in Example 44. Listing 19 displays the (partial) output of executing RVPredict on the program. Particularly, we focus on one data race report (out of 4). We notice that in this case, lines 7 and 13 indicate that the data race occurs during those function calls. Yet, these are the calls we used to specify the `produce` and `consume` events. In this case, we can see that the data race occurs at the level of the events we specified. Upon running RVPredict on the first variant, it reports no data races, as it is properly synchronized. ∗

While checking the absence of data race is useful for the case of *producer-consumer*, it is not enough to consider a

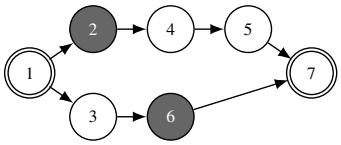


Figure 10.4: Concurrent execution fragment of *1-Writer 2-Readers*. Double circle: `write`, normal: `read`. Events are numbered to distinguish them. Events 2 and 6 are an example of concurrent events as there is no order between them.

properly synchronized program to be safe when using global monitors. This is due to the possible existence of concurrent regions independently from data race. We illustrate the case of concurrent regions in Example 46.

Example 46 (1-Writer 2-Readers†) Figure 10.4 illustrates a concurrent execution fragment of *1-Writer 2-Readers*, where a thread can write to a shared variable, and two other threads can read from the variable. The threads can read concurrently, but no thread can write or read while a write is occurring. In this execution, the first reader performs 3 reads (events 2, 4, and 5), while the second reader performs 2 reads (events 3 and 6). We notice that indeed, no reads happen concurrently. In this case, we see that the program is correctly synchronized (it is data-race free and deadlock-free). However, we can still end up with different total orders, as there still exists concurrent regions. By looking at the concurrent execution, we notice that we can still have events on which we can establish a total order[4]. ∗

On the one hand, a specification relying on the order of events found in concurrent regions (i.e., "*the first reader must always read before the second*") can still result in inconsistent monitoring, similarly to *producer-consumer*. On the other hand, a specification relying on events that can always be totally ordered (i.e., "*there must be at least one read between writes*") will not result in inconsistent monitoring. We notice that to distinguish these two cases, we rely (i) on the order of the execution (concurrent regions), and (ii) the events in the specification. Two events that cannot be ordered are therefore called *concurrent events*. For example, the events 2 and 6 are concurrent, as there is no order relation between them. Instrumenting the program to capture *concurrent events* may also be problematic as we will explain in Section 10.3.

10.2.3 Outcome: Refining the Classification

We are now able to formulate the last consideration for totally ordered formalisms.

Q4: Is the satisfaction of the specification sensitive to the order of concurrent events?

If the answer to **Q4** is *no*, then it is possible to linearize the trace to match the total order expressed in the specification. Otherwise, monitoring becomes *unreliable* as the concurrency can cause non-determinism, or even make it so the captured trace is not a representation of the execution as we explain in Section 10.3.

Remark 6 (Expressiveness) We noticed that utilizing linear specifications for monitoring multithreaded programs works well when the execution of the program can be reduced to a total order. On the one hand, we see *per-thread* monitoring (Section 10.2.1) restricting events to the same thread. On the other hand, we see *global* monitoring restricting the behavior to only those that can be linearized. As such, in these cases, the interplay between trace and specification constrains the expressiveness of the monitoring to either the thread itself, or the segments in the execution that can be linearized. ∗

[4] This is similar to the notion of *linearizability* [HW90].

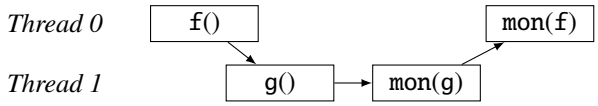


Figure 10.5: Advice execution (`mon`) with context-switches leading to incorrect trace capture.

```
1   g       f_trace
2   f       g_trace
3   f       f_trace
4   g       g_trace
5   f       f_trace
6   g       g_trace
7   f       f_trace
8   g       g_trace
9   f       f_trace
10  g       f_trace
11  f       g_trace
12  f       f_trace
13  g       g_trace
14  g       g_trace
```

(a) Comparison between the system trace (left) and the trace collected by the monitor (right).

Tool	Advice	Sync	Identical	Different
AspectJ	A	✓	4,912	5,088
	B		9,170	830
Java-MOP	A	✓	1,737	8,263
	B		9,749	251
LARVA	A	✓	8,545	1,455
	B		9,992	8
Java-MOP	A	✗	2,026	7,974
	B		9,517	483

(b) Comparing traces collected with AspectJ, LARVA, and Java-MOP across 10,000 executions. The column **Advice** indicates respectively intercepting *after* (A) and *before* (B) the function call.

Figure 10.6: Comparison of collected traces using instrumentation and the system trace.

10.3 Instrumentation: Advice Atomicity

Generally, trace collection is done after instrumentation of the program using AspectJ, or other techniques (such as bytecode instrumentation). As mentioned in Section 5, it is still possible to specify *unsynchronized* monitors and handle concurrency without the tool support. We note that using AspectJ for instrumentation is found in Java-MOP, Tracematches, MarQ, and LARVA [BFB+17b]. In this section, we show that instrumentation may lead to unreliable traces in concurrent regions.

10.3.1 Extracting Traces

Extracting a trace from a program execution often requires executing additional code at runtime. For example, to capture a method call, one could insert a print statement before or after the method call. This extra running code is referred to as *advice* by AspectJ. When an action is executed, *the code responsible for gathering the trace will not, in general, execute atomically with the action.* For multithreaded programs, the execution order may be incorrectly captured due to context switches between threads. To illustrate the issues caused by context switches, we have two threads with a race condition on a call to function `f` and `g` respectively, we match the call and execute the advice right after the call. We show this by adding a call to the advice code `mon`(), right after the function call. We see in Figure 10.5 that in the execution the call to function `f` precedes the call to function `g`, however, due to context switches, the advice associated with `g` (`mon(g)`) executes before that associated with function `f` (`mon(f)`). In this case the order perceived by the monitors is `g` · `f` while the order of the execution is `f` · `g`. In this scenario, the generated trace is not representative of the execution, and thus the check performed by the monitor is unreliable.

Example 47 (Advice Atomicity†) For this example, we create two threads such that each calls a unique function (`f` and `g`, respectively) an equal number of times. Each function consists of a single print statement (to `stdout`) indicating the function name. We create a simple monitor that prints (to `stderr`) the same function name while appending "_trace". Then, we verify that the traces are identical, that is the prints from within the functions follow the same order as those in the monitor. Figure 10.6a shows a fragment of a trace that is different. We see at lines (1-2) that the trace of the monitor starts with `f` · `g` while in the program execution the order is `g` · `f`. Figure 10.6b shows the difference between the captured trace by the monitor and the trace of the system, using monitors created manually with AspectJ, and automatically with Java-MOP and LARVA. The monitor created manually with AspectJ is also representative of MarQ as MarQ relies on the user writing the event matching in AspectJ, then calling the

QEA monitor. Column **Sync** distinguishes the case when using *unsynchronized* in Java-MOP. We notice that the traces differ from the actual program execution for AspectJ, Java-MOP and LARVA. Traces appear to differ more when intercepting after the function call. In AspectJ, the rate of identical traces drops from 91% (B) to 49% (A). This drop is also visible for LARVA and Java-MOP. This is not surprising as Java-MOP and LARVA use AspectJ for instrumentation while introducing some variation as each tool has some additional computation performed on matching. The rate change could be associated with either the specific program or the virtual machine in this case, as the added computation from the monitors and AspectJ could affect the schedule. More importantly, we notice that even when the monitors are synchronized, the captured trace is not guaranteed to be identical to that of the execution. ∗

This problem can only be solved if atomicity for the granularity level can be guaranteed. In general, source-level instrumentation of method calls with AspectJ, or even bytecode instrumentation at the `INVOKE` level will still not be atomic. Adding a lock not only increases overhead, but can also introduce deadlocks if the method invocation is external to the code being instrumented (e.g., calls to libraries). However, by adding locks one can modify the behavior of the program as illustrated in Example 47, as such one needs to minimize the area to which the lock is applied.

10.3.2 Discussion

In certain conditions, capturing traces can still be done in the case of concurrent events. First, a developer must have full knowledge of the program (i.e., it must be seen as a *white box*), this allows the developer to manually instrument the locks to ensure atomic capture, avoiding deadlocks and managing external function calls carefully. Second, we require that the instrumented areas tolerate the interference, and therefore must prove that the interference does not impact significantly the behavior of the program, by modifying the schedule. In this case, one could see that global monitoring (Section 10.2.2) reports correct verdicts *for the single execution*.

> **Remark 7 (Monitor placement)** An additional important aspect for tools pertains to whether the monitors are inlined in the program or execute separately. For multithreaded programs, instrumentation can place monitors so that they execute in the thread that triggers the event, or in a separate thread, or even process. These constitute important implementation details that could limit or interfere with the program differently. However, for the scope of the thesis, we focus on issues that are relevant for event orders and concurrency. ∗

10.4 Reasoning About Concurrency

Section 10.2 shows that approaches relying on total order formalisms are only capable of reliably monitoring a multithreaded program when the execution boils down to a total order. Therefore, it is important to reason about concurrency when designing monitoring tools, while still allowing behavioral properties. We present GPredict [HLR15] in Section 10.4.1, a concurrency analysis tool that can be used for specifying behavior over concurrent regions. We discuss in Section 10.4.2 the potential of multitrace approaches, first introduced in Section 10.1.3. In Section 10.4.3, we present certain approaches from outside RV that may prove interesting and provide additional insight.

10.4.1 Generic Predictive Concurrency Analysis

Concurrent behavior as logical constraints solving. The more general theory behind RVPredict (Section 10.1.2) develops a sound and maximal causal model to analyze concurrency in a multithreaded program [HMR14]. In this model, the correct behavior of a program is modeled as a set of logical constraints, thus restricting the possible traces to consider. The theory supports any logical constraints to determine correctness, it is possible to encode a specification on multithreaded programs as a set of logical constraints. However, allowing for arbitrary specifications to be encoded while supports in the model, is not supported in the provided tool (RVPredict).

Listing 10.2: GPredict specification depicting atomic regions.

```
1 AtomicityViolation (Object o){
2   event begin before(Object o) : execution(m());
3   event read  before(Object o) : get(* s) && target(o);
4   event write before(Object o) : set(* s) && target(o);
5   event end   after(Object o)  : execution(m());
6
7   pattern: begin(t1, <r1) read(t1) write(t2) write(t1) end(t1,>r1)
8 //pattern: read(t1) || write(t2)
9 }
```

GPredict. Using the same sound and maximal model for predictive trace analysis [HMR14] discussed in Section 10.1.2, GPredict [HLR15] extends the specification formalism past data-races to behavior. Specifications are able to include behavioral, user-specified events, and are extended with thread identifiers, atomic regions, and concurrency. Events are defined similarly to Java-MOP using AspectJ for instrumentation. Atomic regions are special events that denote either the start or end of an atomic region. Each atomic region is given an ID. The specification formalism uses regular expressions extended with the concurrency operator "||" which allows events to happen in parallel.

Example 48 (Specifying concurrency) Listing 10.2 shows a specification for GPredict written for a multithreaded program, we re-use the example from [HLR15]. The program consists of a method (`m`) of an object which reads and writes to a variable (`s`). Lines 2 and 5 specify the events that denote respectively reaching the start and end of method (`m`). Line 3 and 4 specify respectively the `read` and `write` events. Lines 7 and 8 illustrate respectively specifications for atomic regions and concurrency. The events in the specification can be parametrized by the thread identifier, and a region delimiter. To specify an atomic regions, an event can indicate whether it is the start or end of a region using the characters > and < respectively. The delimiter is followed by a region identifier, which is used to distinguish regions in the specification. In this case, we see that the `begin` and `end` events emitted by thread `t1` delimit an atomic region in which a read by thread `t1` must be followed by a write by thread `t2`, which is followed by a write by thread `t1`. The specification is violated if any of the events happen in a different order or concurrently. To specify concurrent events, one must utilize "||" as shown on Line 8. In this case, the specification says that a read in thread `t1` can happen in parallel with a write in thread `t2`. ∗

Limitations. While GPredict presents a general approach to reason about behavioral properties in concurrent executions, and hence constitutes a solution to monitoring when concurrency is present, it still requires additional improvements for higher expressiveness and usability. Notably, GPredict requires specifying thread identifiers explicitly in the specification. This requires specifications with multiple threads to become extremely verbose, and cannot handle a dynamic number of threads. For example, in the case of *readers-writers*, adding extra readers or writers requires rewriting the specification and combining events to specify each new thread. The approach behinds GPredict can also be extended to become more expressive, e.g. to support counting events to account for fairness in a concurrent setting. Furthermore, GPredict relies on recording a trace of a program before performing an offline analysis to determine concurrency errors [HLR15]. *Ideally, we prefer to be able to detect the error during the execution and not postmortem.*

10.4.2 Multi-trace Specifications: Possible Candidates?

RV approaches and tools that utilize multiple traces include approaches that rely on streams, decentralized specifications, and hyperproperties (as described in Section 10.1.3).

Thread events as streams. Stream-based RV techniques deal with synchronized streams in general, the order of the events is generally total. It is possible to imagine that ordering could be performed by certain functions that aggregate streams. For example, it is possible to create a stream per event per thread, and then aggregate

them appropriately to handle the partial order specifications. However, as is, either specifying or adding streams to multithreaded programs remains unclear, but presents an interesting possible future direction.

Thread-level specifications as references. Decentralized specifications present various manners to implicitly deal with threads, but do not in particular deal with multithreaded programs. Since monitors are merely references, and references can be evaluated as oracles at any point during the execution. Monitors are triggered to start monitoring, and are required to eventually return an evaluation of a property. Even when specifications are totally ordered, in the sense that they are automata-based, the semantics that allow for eventual evaluation of monitors make it so monitors on threads can evaluate local specifications and explicitly communicate with other threads for the additional information. In the next chapter (Chapter 11), we introduce a two-level decentralized specification to monitor multithreaded programs *online*.

Concurrent executions as multiple serial executions. Hyperproperties are properties defined on a set of traces. Generally used for security, they allow for instance to check different executions of the same program from multiple access levels. By executing a concurrent program multiple times, we can obtain various totally ordered traces depending on the concurrent regions. As such, a possible future direction could explore how to express concurrency specifications as hyperproperties, and the feasibility of verifying a large set of totally ordered traces.

10.4.3 Inspiration From Outside RV

Static Techniques

Static techniques for checking concurrency errors often reason on the memory operations performed by threads. We discuss applications of separation logics, fence inference, and model checking to verify multithreaded programs.

Separation logic approaches. Separation logic [OP99, Rey02] is an extension to Hoare logic (described in Section 1.1). It extends obligations to apply to the program memory by reasoning independently (in separation) about small program states (called frames). The concurrent version of separation logic is referred to as *concurrent separation logic* (CSL) [O'H07]. Implicit Dynamic Frames (IDF) [SJP09, SJP12] extends separation logic with first-order expressions over memory. In [ABH16], the authors present a layered approach combining both CSL and IDF to reason about the program states at three layers. The first layer reasons about data race freedom of multiple threads independently modifying a shared object. The second layer adds a local state for each thread, to allow expressing consistency between the local state and the global state of a shared object. The third layer captures all operations performed on a shared object to form a history. The history is captured by tracing local operations, then when threads synchronize they share their information about operations. The three-layered approach is not only able to verify lock-based programs but also lock-free programs.

Fence Inference. Other approaches similar to RVPredict (Section 10.1.2) perform automatic verification by performing fence inference under relaxed memory models [BLP15, KVY11]. Fences are special instructions that enforce ordering constraints on the operations in a program. They are typically used to explicitly disallow a compiler or a CPU to re-order operations in certain regions. These approaches are mainly concerned with the automatic detection of locations in the program to add fences, as doing so manually by the developer may prove incorrect. Fence inference can be seen as determining concurrency segments in a program of interest with respect to the memory operations.

Reasoning on execution graphs. Approaches for model checking concurrency often rely on exploring interleaving operations which occur in the program execution. In [KLSV18], the authors focus on execution graphs directly for verifying weak memory models, in particular the RC11 (repaired C++[5]) model [LVK+17]. That is, instead of having the algorithm consider the set of all thread interleavings accounting for weak memory models and determining equivalence classes, they instead only explore consistent execution graphs of a program. The problem

[5]Called repaired as it fixes soundness of the C++ concurrency model presented by [BOS+11].

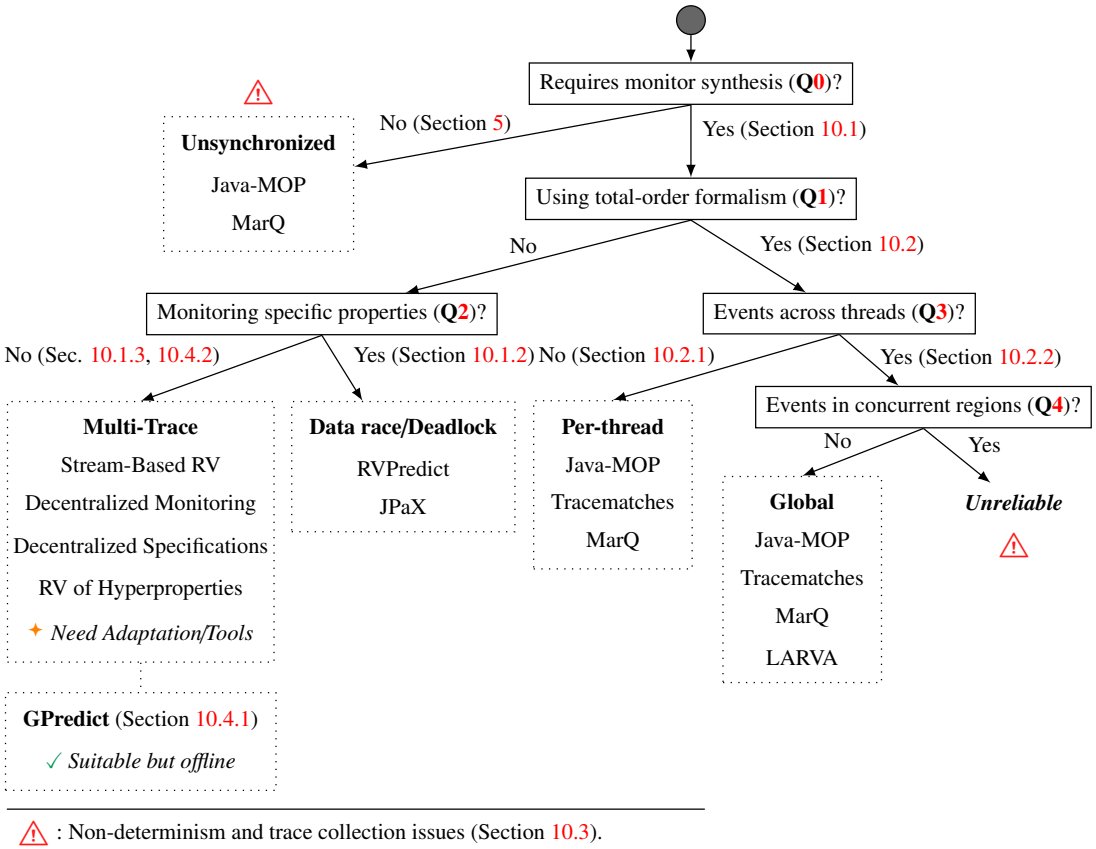


Figure 10.7: RV approaches and considerations for monitoring multithreaded programs.

becomes to effectively enumerate all consistent execution graphs of a program. As such, they reason using partial orders directly, relying on the principle that RC11 consistency is prefix-closed.

Dynamic Techniques

We present two interesting dynamic techniques that rely on capturing the trace probabilistically, and testing rare schedules.

Relying on heuristics. Determining exact concurrency regions is costly during execution or may interfere with the execution. An interesting direction is to utilize heuristics to determine concurrent regions. BARRACUDA [EPP+17] detects synchronization errors on GPUs by instrumenting CUDA applications and performing binary-level analysis. BARRACUDA avoids large overhead as it uses heuristics to approximate the synchronization in linear traces.

Testing schedules. PARROT [CSL+13] is a *testing* framework that explores the interleavings of possible threads to test concurrent programs. PARROT analyzes the possible schedules of threads, and forces the application to explore them, thus exposing concurrency issues. The motivation behind PARROT is the realization that certain schedules occur in low probability under very specific circumstances.

Conclusion

We overviewed RV approaches that support multithreaded programs. By considering the various specifications formalisms, we are able to classify the tools by looking at whether or not they rely on total-order formalisms. We investigated the limitations of linear traces in the case of RV tools relying on formalisms that use total order, and noted the situations where linear traces lead to inconsistent verdicts. After presenting tools capable of checking specific properties, we mentioned various recent RV techniques using properties over multiple traces, and discussed their potential for monitoring multithreaded programs. Figure 10.7 summarizes the decisions a developer must consider when choosing RV tools for multithreaded monitoring, and the limitations of the existing approaches. We caution users of tools that using a formalism in which events are specified as a total order is not reliable when monitoring concurrent events (as we cannot reliably answer **Q4**). It is possible to monitor multithreaded programs that exhibit concurrency using GPredict (Section 10.4.1). However, there are still limitations with writing specifications easily and expressively, and also GPredict performing offline analysis. Furthermore, RV techniques capable of specifying properties over multiple traces prove to be interesting candidates to extend to monitor multithreaded programs.

In the next chapter (Chapter 11), we introduce a two-level decentralized specification approach to monitoring multithreaded programs *online*.

CHAPTER 11

Opportunistic RV for Multithreaded Programs using a Two-Level Decentralized Specification

Contents

Chapter abstract

Decentralized monitoring consists in deploying multiple monitors to focus on monitoring a given specification. Typically monitors need to communicate to relay important information determined by their local monitoring. In this chapter, we introduce the decentralized monitoring of multithreaded programs using the main idea behind decentralized specifications. In our setting, monitors are deployed to monitor specific threads, and only exchange information upon reaching synchronization regions defined by the program itself. That is, they use the opportunity of a lock in the program, to evaluate information across threads. As such, we refer to this approach as *opportunistic* RV. By using the existing synchronization, our approach reduces additional overhead and interference to synchronize at the cost of adding a delay to determine the verdict. We utilize a textbook example of *readers-writers* as it contains concurrent regions, and show how opportunistic RV is capable of expressing specifications on concurrent regions, without incurring significant delay. We present a manual monitoring implementation for *readers-writers*, and show that the overhead of our approach scales particularly well.

Introduction

Motivation. In Chapter 10, we overviewed the challenges for RV of multithreaded programs. We noted that RV tools relying on total order and global monitoring may linearize the trace wrongly when concurrency is present in the program. We also pointed out that perthread monitoring is not expressive enough to include specifications with events defined across threads. The solution that stood out for multithreaded monitoring included GPredict (Section 10.4.1). However we noted its limitations. First, it requires that thread identifiers be explicitly modeled in the specification, without accounting for the dynamic number of threads. Second, its expressiveness only accounts for regular expressions, and as such does not account for counting. Third, its analysis is performed offline, and we would like that the monitoring technique detects the error during the execution.

Approach. In this chapter, we present a generic approach to monitor lock-based multithreaded programs. Our approach consists of a two-level monitoring technique that relies on existing locks in the program. At the first level, a thread-local specification checks a given property on the thread itself, where events are totally ordered. At the second level, we define *scopes* which delimit concurrency regions. Scopes rely on operations in the program guaranteed to follow a total order. The guarantee is ensured by the platform itself, either the program model, the execution engine (JVM in our case), or the compiler. In this chapter, we utilize lock acquires to delimit scopes. Upon reaching the totally ordered operations, a scope monitor utilizes the result of all thread-local monitors that executed in the concurrent region to construct a scope state, and perform monitoring on a sequence of such states. This approach heavily relies on existing totally ordered operations in the program. However, it incurs minimal interference and overhead as it does not add additional locks.

An instantiation of decentralized specifications. Our approach can be seen as an instantiation of decentralized specifications targeting multithreaded programs. Decentralized specifications as presented in Chapter 6 rely on references to evaluate subspecifications. This requires that subspecifications eventually return a result (i.e. eventual consistency) to be able to progress, all while abstracting delay and time as each reference could be linked to a different automaton. Determining a common time for references to be evaluated is necessary only to relate atomic propositions to each other. However, using the same concept of decentralized specifications, we instantiate semantics that are useful for multithreaded programs. In this approach to decentralized multithreaded monitoring, local properties denote the leaves of the dependency tree of specifications, while scope properties depend on the local properties. Local properties are processed for each concurrency region to generate atomic propositions for the scope property, which triggers a step in the monitoring at the level of the scope.

Key contributions. The key contributions of this chapter can be summarized as follows:

1. Introducing a two-level decentralized specification for the sound online monitoring of multithreaded programs;
2. Ensuring that the approach is sufficiently generic to be used by existing RV techniques, including streams and techniques which formalism uses total order;
3. Extending expressiveness to specify behavior over an arbitrary number of threads, and include arbitrary transformations to the result of local monitoring; and
4. Expressing properties for *readers-writers* that cannot be expressed using existing approaches, and monitoring them efficiently.

11.1 Modeling The Program Execution

We are concerned with an abstraction of a program execution, we focus on a model that can be useful for monitoring behavioral properties of programs. Furthermore, we seek not to constrain the model to a specific program execution model, but encompass existing ones. We first choose the smallest observable *execution step* done by a program. We refer to this step as an *action*. We are interested in determining the *granularity* and the *order* of these actions.

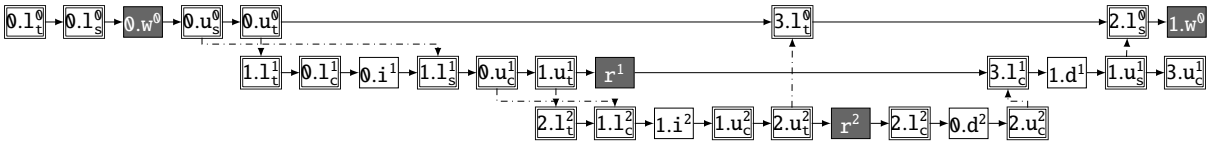


Figure 11.1: Concurrent execution fragment of 1-Writer 2-Readers. The action labels l, u, w, r, ci, cd indicate respectively the following: lock, unlock, write, read, increment readers counter and decrement readers counter. The lock ids t, s, c indicate the following locks respectively: test for readers, service, and readers counter. Actions with a double border indicate actions pertaining to locks. The reading and writing actions are filled to highlight them.

Definition 20 (Action) An action is a tuple ⟨lbl, id, ctx⟩, where: lbl is a label, id is a unique identifier, and ctx is the context of the action.

An action consists in the runtime information of the smallest step executed by the program, and is usually tied to a syntactic element lbl (label). The label captures the instruction name, the opcode, the function name, or the specific task information depending on the granularity of actions. Since the action is a runtime object, the same syntactic element could execute more than once, as such, we use id to distinguish two executions of the same element. Finally, the context (ctx) is a set containing all necessary annotations for the assumptions of the model. For example, it can contain thread id, process id, lock identifier, or memory addresses. Since we will be focusing on multithreaded examples of actions in the rest of the chapter, we include the threadid and possible resource labels (such as lock identifiers) in the context. We use the notation $\texttt{id.lbl}^{\texttt{threadid}}_{\texttt{resource}}$ to denote an action, omit `resource` when it is absent, and omit `id` when there is no ambiguity. Furthermore, we use the notation a.threadid for a given action a to retreive the threadid in the context.

Definition 21 (Concurrent Execution) A concurrent execution is a partially-ordered set of actions $(\mathbb{A}, \rightarrow)$.

We capture the order of actions as a partial order, as it is general enough to represent various formalisms and models of concurrent systems. This model is compatible with various formalisms that define the behavior of concurrent programs such as weak memory consistency models [AG96, ANB+95, MPA05], Mazurkiewicz traces [Maz86, GK10], parallel series [LW01], Message Sequence Charts graphs [MR04], and Petri Nets [NPW81].

Example 49 (Concurrent fragment for 1-Writer 2-Readers.) Figure 11.1 shows a concurrent execution fragment for *1-Writer 2-Readers* introduced in Example 46. We recall that a thread can write to a shared variable, and two other threads can read from the variable. The threads can read concurrently, but no thread can write or read while a write is occurring. The concurrent execution fragment contains all actions performed by all threads, along with the partial order inferred from locks. We included redundant information to show all order information that can be obtained. We have three locks: test for readers (t), service (s), and readers counter (c). Lock t checks if any reader is currently reading, this lock gives preference to writers. Lock s is used to regulate access to the shared resource, it can be either obtained by readers or one writer. Lock c is used to regulate access to the readers counters, it only synchronizes readers. In this concurrent execution, we have three synchronizations on the resource. First, the writer acquires the resource and writes. Second, the readers acquire the resource and read in parallel. Third, the writer acquires the resource and writes. Action $1.\texttt{l}^1_\texttt{s}$ represents the execution of a lock acquire instruction by a thread with threadid 1 on the shared resource lock (`s`). ∗

11.2 Opportunistic Multithreaded RV

Overview. Opportunistic RV considers thread-local monitors that check local properties, and dedicated monitors for concurrency regions defined over additional synchronization events. First, we introduce events that account for the dynamic number of threads in Section 11.2.1. Second, we elaborate on the notion of scopes in Section 11.2.2. Third, we define the semantics to evaluate scope properties in Section 11.2.3. Fourth, we present the procedure to monitor scopes in Section 11.2.4.

11.2.1 Dynamic Events and Threads

Goal. Threads are typically created at runtime, each thread has a given identifier, we refer to it as a *threadid*. We denote the set of all threadids by TID. However, *threadids* are subject to change from one execution to another, and it is not known in advance how many threads will be spawned during the execution. Therefore, the goal for the specification language is to allow for properties to be assigned to threads generically, regardless of their number and changes in *threadid*. For the remainder of this section we fix the concurrent execution $\langle \mathbb{A}, \rightarrow \rangle$.

Distinguishing threads. To allow for dynamic number of threads, we first define thread types $\mathbb{T}$, to distinguish threads that are relevant to the specification. For example, thread types for *readers-writers* are in the set $\mathbb{T}_{rw} \stackrel{\text{def}}{=} \{\text{reader}, \text{writer}\}$. Thread types allows us to express a property over all threads of the same type. In order to assign a type to a thread in practice, we define a set of actions $\mathbb{S} \subseteq \mathbb{A}$ called "spawn" actions. For example in *readers-writers*, we can assign the spawn action of a reader (resp. writer) to be the method invocation of `Reader.run` (`Writer.run`). Function $\text{spawn} : \mathbb{S} \rightarrow \mathbb{T}$, assigns a thread type to a spawn action. The threads that match a given type are determined based on the spawn action(s) present during the execution. We note that a thread is able to have multiple types. To reference all threads assigned a given type, we use function $\text{pool} : \mathbb{T} \rightarrow 2^{\text{TID}}$. That is, given a type t, a thread with threadid $tid \in \text{pool}(t)$ iff $\exists a \in \mathbb{A} : \text{spawn}(a) = t \wedge a.\text{threadid} = tid$. This approaches allows a thread to have multiple types, as it is possible for different properties to operate on different events in the same given thread.

Events. Thread types are used to determine threads that are relevant for specific properties. For example, we are interested in verifying properties that apply to readers only. Properties are evaluated by processing a sequence of events. We note that actions ($\mathbb{A}$) illustrate information about the program execution at runtime. Actions need to be converted to events, that will be processed by the specification. As such, we define for each thread type $type \in \mathbb{T}$, the alphabet of events: $\mathbb{E}_{type}$. The set $\mathbb{E}_{type}$ contains all the events that can be generated from actions for the particular thread type $type \in \mathbb{T}$. The empty event $\mathcal{E}$ is a special event that indicates that no events are matched. Then, we provide a total function $\text{ev}_{type} : \mathbb{A} \rightarrow \{\mathcal{E}\} \cup \mathbb{E}_{type}$. The implementation of ev relies on the specification formalism used, it is capable of generating events based on the action itself, its label, id or context. For example, the conversion can utilize runtime context of actions to generate parametric events when needed. We illustrate a simple event alphabet containing atomic propositions, and a function ev that matches using the label of an action in Example 50.

Example 50 (Events.) We identify for *readers-writers* (Example 49) two thread types: $\mathbb{T}_{rw} \stackrel{\text{def}}{=} \{\text{reader}, \text{writer}\}$. We are interested in the events $\mathbb{E}_{\text{reader}} \stackrel{\text{def}}{=} \{\text{read}\}$, and $\mathbb{E}_{\text{writer}} \stackrel{\text{def}}{=} \{\text{write}\}$. For a specification at the level of a given thread, we have either a reader or a writer, the event associated with the reader (resp. writer) is read (resp. write).

$$\text{ev}_{\text{reader}}(\text{a}) \stackrel{\text{def}}{=} \begin{cases} \text{read} & \text{if a.lbl} = \text{“r”}, \\ \mathcal{E} & \text{otherwise} \end{cases} \qquad \text{ev}_{\text{writer}}(\text{a}) \stackrel{\text{def}}{=} \begin{cases} \text{write} & \text{if a.lbl} = \text{“w”}, \\ \mathcal{E} & \text{otherwise.} \end{cases}$$

While we illustrated a simple case that is sufficient to monitor *readers-writers*, we can also consider more complex cases where events are generated using the action context. As such, let us consider the new more complex type reader′. Consider that read actions operate on an array of size n, and their context contain the index read (stored in the key ind), the reader will read all values in the array starting from the index 0 to $n - 1$. We define for reading the start and end events: $\mathbb{E}_{\text{reader}'} \stackrel{\text{def}}{=} \{\text{startread}, \text{endread}\}$. We associate them with actions as follows:

$$\text{ev}_{\text{reader}'}(\text{a}) \stackrel{\text{def}}{=} \begin{cases} \text{startread} & \text{if a.lbl} = \text{“r”} \wedge \text{a.ctx(ind)} = 0, \\ \text{endread} & \text{if a.lbl} = \text{“r”} \wedge \text{a.ctx(ind)} = n - 1, \\ \mathcal{E} & \text{otherwise.} \end{cases}$$

*

11.2.2 Scopes: Properties Over Concurrent Regions

Overview. Multithreaded programs provide a complicated execution flow for all threads: threads can be sleeping, active, or running concurrently with other threads. This is challenging for verification techniques to manage the

order of events among threads, and relate it to other threads in the entirety of the execution. For our approach, we define the notion of *scope*. A *scope* defines a projection of the concurrent execution to delimit the concurrent regions to consider along with the properties that need to be checked. A scope allows verification to be performed at level of concurrent regions instead of the entire program. This provides a hierarchy at the level of concurrent regions, where properties checked independently on threads are aggregated for every concurrent region.

Synchronizing actions. A scope s is associated with a *synchronizing predicate* $\text{sync}_\text{s} : \mathbb{A} \to \mathbb{B}_2$ which is used to determine *synchronizing actions* (SAs). The set of synchronizing actions for a scope s is defined as: $\text{SA}_\text{s} = \{a \in \mathbb{A} \mid \text{sync}_\text{s}(a) = \top\}$. SAs constitute synchronization points in a concurrent execution for multiple threads, they are actions that do not occur concurrently. A valid set of SAs is such that there exists a total order on all actions in the set (i.e., no two SAs can occur concurrently). As such SAs are sequenced, and can be mapped to indices. Function $\text{idx}_\text{s} : \mathbb{A} \to \mathbb{N}^*$ returns the index of a synchronizing action. For convenience, we map them starting at 1, as 0 will indicate the initial state. The notation $|\text{idx}_\text{s}|$ indicates the length of the sequence.

Example 51 (Synchronizing predicate) While multiple scopes for *readers-writers* (Example 49) are possible due to the three locks types. In this example, we consider the resource lock (s) to be the one of interest, as it denotes the concurrent regions which allow either a writer to write or readers to read. We label the scope by res for the remainder of the chapter. The synchronizing predicate sync_res selects all actions such that their label is equal to l (lock acquire), and with the lock id s present in the context of the action. It is defined as follows: $\text{sync}_\text{res}(\langle \text{lbl}, \text{id}, \text{ctx}\rangle) \stackrel{\text{def}}{=} (\text{lbl} = \text{l} \wedge \text{ctx}(\text{resid}) = \text{s})$. The sequence of SAs obtained is $\mathtt{0}.\mathtt{l}^{\mathtt{0}}_{\mathtt{s}} \cdot \mathtt{1}.\mathtt{l}^{\mathtt{1}}_{\mathtt{s}} \cdot \mathtt{2}.\mathtt{l}^{\mathtt{0}}_{\mathtt{s}}$. The value of idx_res for each of the obtained SAs is respectively 1, 2 and 3. ∗

Scope region. A scope region is a projection of the concurrent execution delimited by two successive SAs. It delimits actions that fall between the two SAs. In the remainder of the chapter, we define two "special" synchronizing actions: $\text{begin}, \text{end} \in \mathbb{A}$ common to all scopes that are needed to evaluate the first and last region. The actions refer respectively to the beginning and end of a program execution.

Definition 22 (Scope region) A scope region for a scope s and an associated index function $\text{idx}_\text{s} : \mathbb{A} \to \mathbb{N}^*$ is a function $\mathcal{R}_\text{s} : \text{codom}(\text{idx}_\text{s}) \cup \{0, |\text{idx}_\text{s}| + 1\} \to 2^{\mathbb{A}}$:

$$\mathcal{R}_\text{s}(i) \stackrel{\text{def}}{=} \begin{cases} \{a \in \mathbb{A} \mid \langle a', a\rangle \in \!\!\rightarrow \wedge \langle a, a''\rangle \in \!\!\rightarrow \wedge\, \text{issync}(a', i-1) \wedge\, \text{issync}(a'', i)\} & \text{if } 1 \le i \le |\text{idx}_\text{s}|, \\ \{a \in \mathbb{A} \mid \langle a', a\rangle \in \!\!\rightarrow \wedge \langle a, \text{end}\rangle \in \!\!\rightarrow \wedge\, \text{issync}(a', i-1)\} & \text{if } i = |\text{idx}_\text{s}| + 1, \\ \{a \in \mathbb{A} \mid \langle \text{begin}, a\rangle \in \!\!\rightarrow \wedge \langle a, a''\rangle \in \!\!\rightarrow \wedge\, \text{issync}(a'', 1)\} & \text{if } i = 0 \\ \emptyset & \text{otherwise,} \end{cases}$$

where: $\text{issync}(a, i) \stackrel{\text{def}}{=} (\text{sync}_\text{s}(a) = \top \wedge \text{idx}_\text{s}(a) = i)$.

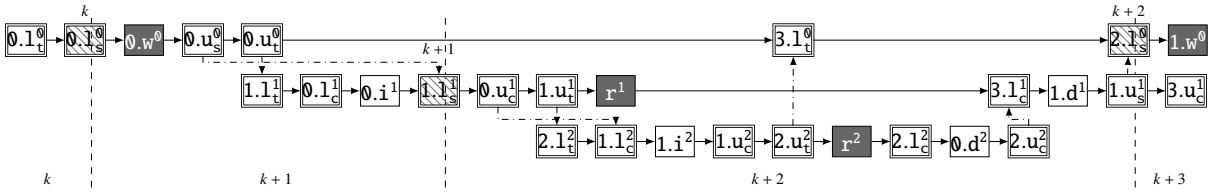
We illustrate scope regions on *readers-writers* in Example 52.

Example 52 (Scope regions) Figure 11.2a depicts the scope regions for scope res obtained when synchronizing on the resource lock (Example 51). Every lock acquire delimits parts of the concurrent execution. The region $k+1$ includes all actions between the two lock acquires $\mathtt{0}.\mathtt{l}^{\mathtt{0}}_{\mathtt{s}}$ and $\mathtt{1}.\mathtt{l}^{\mathtt{1}}_{\mathtt{s}}$. That is, $\mathcal{R}_\text{res}(k{+}1) = \{\mathtt{0}.\mathtt{w}^{\mathtt{0}}, \mathtt{0}.\mathtt{u}^{\mathtt{0}}_{\mathtt{s}}, \mathtt{0}.\mathtt{u}^{\mathtt{0}}_{\mathtt{t}}, \mathtt{1}.\mathtt{l}^{\mathtt{1}}_{\mathtt{t}}, \mathtt{0}.\mathtt{l}^{\mathtt{1}}_{\mathtt{c}}, \mathtt{0}.\mathtt{i}^{\mathtt{1}}\}$. The region $k+2$ contains two concurrent reads. ∗

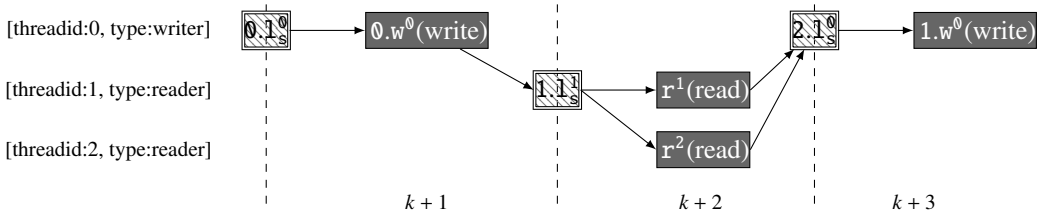
Local properties. For a given scope region, we assign a property for each thread type. A thread of a given type evaluates locally a given property on local events. We refer to those properties as *local properties*. These properties can be seen as the analogous of *per-thread* monitoring applied between two SAs. For a specific thread, we have a guaranteed total order on the local actions being formed.

Definition 23 (Local property) A local property is a tuple $\langle \text{type}, \text{EVS}, \text{RT}, \text{eval}\rangle$ with:

- $\text{type} \in \mathbb{T}$ is the thread type for which the local property applies;



(a) Scope regions using SAs that acquire lock s (indicated with a pattern). The index of each SA is displayed above it. The vertical dashed lines indicate the limits of a region. The region index is shown for each region.



(b) Projected actions using the scope and local properties, and their corresponding events.

Figure 11.2: View of a scope in a concurrent execution fragment of 1-Writer 2-Readers. The action labels l, u, w, r, ci, cd indicate respectively the following: lock, unlock, write, read, increment readers counter and decrement readers counter. The lock ids t, s, c indicate the following locks respectively: test for readers, service, and readers counter. Filled actions indicate actions for which function ev for the thread type returns an event. Actions with a pattern background indicate the SAs for the scope.

- $\text{EVS} \subseteq \mathbb{E}_{\text{type}}$ is a subset of (thread type) events relevant to the property evaluation;
- RT is the resulting type of the evaluation (called *return type*); and
- $\text{eval} : (\mathbb{N} \to \text{EVS}) \to \text{RT}$ is the evaluation function of the property, taking as input a sequence of events, and returning the result of the evaluation.

We use the dot notation: for a given property $\text{prop} = \langle \text{type}, \text{EVS}, \text{RT}, \text{eval} \rangle$ we use prop.type, prop.EVS, prop.RT, and prop.eval respectively.

Example 53 (At least one read) The property "at least one read", defined for the thread type reader, states that a reader must perform at least one read event. It can be expressed using LTL as $\varphi_{1\text{r}} \stackrel{\text{def}}{=} \mathbf{F}(\text{read})$. Let $\text{LTL3}^{\text{ap}}_{\varphi}$ denote the evaluation of LTL3 semantics on a set of atomic propositions ap and a formula φ. To check on readers, we specify it as the local property: $\langle \text{reader}, \{\text{read}\}, \mathbb{B}_3, \text{LTL3}^{\{\text{read}\}}_{\varphi_{1\text{r}}} \rangle$. Similarly we can define the local specification for at least one write. For more complex specifications, we recall from Example 50 the type reader′, which tracks the start and end of reading using the events startread and endread. In this case, we can modify the local specification to determine that a read happened in two ways: use only the start of reading ignoring the end, or we can consider reading to be the start followed by an end. In the first case, we have $\text{EVS}_0 = \{\text{readstart}\}$, and the property is $\mathbf{F}(\text{readstart})$. In the second case, we have $\text{EVS}_1 = \{\text{readstart}, \text{readend}\}$, and the property is $\mathbf{F}(\text{readstart} \wedge \mathbf{X}\text{readend})$. ∗

Scope trace. To evaluate a local property, we restrict the trace to actions local to a given thread contained within a scope region. A scope trace is analogous to acquiring the trace for *per-thread* monitoring [AAC+05, For18] in a given scope region (Section 10.2.1).

Definition 24 (Scope trace) A *scope trace* is a projection of the concurrent execution that determines the sequence of relevant events for a given local property $\text{p} = \langle \text{type}, \text{EVS}, \text{RT}, \text{eval} \rangle$, in a scope region $\mathcal{R}_{\text{s}}$ with

index i, for a given thread with tid $\in$ TID.

$$\text{proj}(\text{tid}, i, \text{p}, \mathcal{R}_\text{s}) \stackrel{\text{def}}{=} \begin{cases} \text{evproj}(a_0) \cdot \ldots \cdot \text{evproj}(a_n) & \text{if } i \in \text{dom}(\mathcal{R}_\text{s}) \wedge \text{tid} \in \text{pool(type)}, \\ \mathcal{E} & \text{otherwise,} \end{cases},$$

$$\text{with: } \forall \ell \in [0, n] : \text{evproj}(a_\ell) \stackrel{\text{def}}{=} \begin{cases} e & \text{if } \text{ev}_\text{type}(a_\ell) \in \text{EVS} \\ \mathcal{E} & \text{otherwise,} \end{cases}$$

where $\cdot$ is the sequence concatenation operator (such that $a \cdot \mathcal{E} = \mathcal{E} \cdot a = a$), with $(\forall j \in [1, n] : \langle a_{j-1}, a_j \rangle \in \rightarrow) \wedge (\forall k \in [0, n] : a_k \in \mathcal{R}_\text{s}(i) \wedge a_k.\text{threadid} = \text{tid}$.

To create the scope trace for a given local property and thread, we first consider all actions in a scope region (as we did in Example 52), then we filter the actions to include only actions that are associated with the threadid of the considered thread (i.e., a_k.threadid = tid), the considered thread has the correct type associated with the local specification (i.e., tid $\in$ pool(type)), and finally the action is associated with an event for the local property (i.e., $\text{ev}_\text{type}(a_\ell) \in$ EVS). We illustrate scope traces in Example 54.

Example 54 (Scope trace.) Figure 11.2b illustrates the projection on the scope regions defined using the resource lock (Example 52) for each of the 1 writer and 2 reader threads, where the properties "at least one write" or "at least one read" (Example 53) apply. We see the scope traces for region $k + 1$ are respectively write, $\mathcal{E}, \mathcal{E}$ for the threads with threadids 0, 1, and 2 respectively. For that region, we can now evaluate the local specification independently for each threads on the resulting traces. ∗

Scope state. A scope state aggregates the result of evaluating all local properties for a given scope region. To define a scope state, we consider a scope s, with a vector of local properties $\langle \text{prop}_0, \ldots, \text{prop}_n \rangle$ of return types respectively $\langle \text{RT}_0, \ldots, \text{RT}_n \rangle$. Since a local specification can apply to multiple threads dynamically, for each specification we create the type as a dictionary binding a threadid to the return type (represented as a total function). We use the type na to determine a special type indicating the property does not apply to the thread (as the thread type does not match the property). We can now define the return type of evaluating all local properties as $\text{RI} \stackrel{\text{def}}{=} \langle \text{TID} \rightarrow \{\text{na}\} \cup \text{RT}_0, \ldots, \text{TID} \rightarrow \{\text{na}\} \cup \text{RT}_n \rangle$. Function $\text{state}_\text{s} : \text{RI} \rightarrow \mathbb{I}_\text{s}$ processes the result of evaluating local properties to create a scope state of type $\mathbb{I}_\text{s}$.

Example 55 (Scope state) We illustrate the scope state by evaluating the properties "at least one read" (p_r) and "at least one write" (p_w) (Example 53) on scope region $k + 2$ in Figure 11.2b. We have TID $= \{0, 1, 2\}$, we determine for each reader the trace (being (read) for both), and the writer being empty (i.e. no write was observed). As such for property p_r (resp. p_w), we have the result of the evaluation $[0 \mapsto \text{na}, 1 \mapsto \top, 2 \mapsto \top]$ (resp. $[0 \mapsto ?, 1 \mapsto \text{na}, 2 \mapsto \text{na}]$). We notice that for property p_r, the thread of type writer evaluates to na, as it is not concerned with the property.

We now consider the state creation function state_s. We consider the following atomic propositions activereader, activewriter, allreaders, and onewriter that indicate respectively: at least one thread of type reader performed a read, at least one thread of type writer performed a write, all threads of type reader (|pool(reader)|) performed at least a read, and at most one thread of type writer performed a write. The scope state in this case is a vector of 4 boolean values indicating the each atomic proposition respectively. As such by counting the number of threads associated with $\top$, we can compute the Boolean value of each atomic proposition. For the region $k + 2$, we have the following state: $\langle \top, \bot, \top, \bot \rangle$. We can establish a total order of scope states. For $k + 1$, $k + 2$ and $k + 3$, we have the sequence $\langle \bot, \top, \bot, \top \rangle \cdot \langle \top, \bot, \top, \bot \rangle \cdot \langle \bot, \top, \bot, \top \rangle$. ∗

We are now able to define formally a scope by associating an identifier to a synchonizing predicate, a list of local properties, a spawn predicate, and a scope property evaluation function. We denote by SID the set of scope identifiers.

Definition 25 (Scope) A scope is a tuple $\langle \text{sid}, \text{sync}_\text{sid}, \langle \text{prop}_1, \ldots, \text{prop}_n \rangle, \text{state}_\text{sid}, \text{seval}_\text{sid} \rangle$, where:

- sid $\in$ SID is the scope identifier;
- $\text{sync}_{\text{sid}} : \mathbb{A} \to \mathbb{B}_2$ is the synchronizing predicate that determines SAs;
- $\langle \text{prop}_0, \ldots, \text{prop}_n \rangle$ is a list of local properties (Definition 23);
- $\text{state}_{\text{sid}} : \langle \text{TID} \to \{\text{na}\} \cup \text{prop}_0.\text{RT}, \ldots, \text{TID} \to \{\text{na}\} \cup \text{prop}_n.\text{RT} \rangle \to \mathbb{I}_s$ is the scope state creation function;
- $\text{seval}_{\text{sid}} : \mathbb{N} \times \mathbb{I}_s \to \mathbb{O}$ is the evaluation function of the scope property over a sequence of scope states.

In the next section (Section 11.2.3), we elaborate properties at the level of scopes (i.e., $\text{seval}_{\text{sid}}$). We define two ways to evaluate properties and present three scope properties for *readers-writers* (in Example 56).

11.2.3 Semantics for Evaluating Scopes

After defining scope states, we are now able to evaluate properties on the scope. We consider two evaluation schemes distinguished by the evaluation of local properties in concurrency regions. The first evaluation scheme evaluates the local property starting from the start of the region till its end. In this scheme local specifications are designed to reason only about the concurrent region. The second evaluation scheme evaluates the local property starting from the beginning of the program till the current end of a concurrent region, passing synchronization events to local monitors. The second scheme allows for more properties to be expressed at the cost of more complex semantics.

Evaluating scopes by reasoning on concurrency regions. To evaluate a scope property, we first evaluate each local property for each scope region, we then use $\text{state}_{\text{sid}}$ to generate the scope state for the region. After producing the sequence of scope states, function $\text{seval}_{\text{sid}}$ evaluates the property at the level of a scope.

Definition 26 (Evaluating a scope property) Given the concurrent execution, using the synchronizing predicate sync_{sid}, we obtain the regions $\mathcal{R}_{\text{sid}}(i)$ for $i \in [0, m]$ with $m = |\text{idx}_{\text{sid}}| + 1$. The evaluation of a scope property (noted res) for the scope $\langle \text{sid}, \text{sync}_{\text{sid}}, \langle \text{prop}_0, \ldots, \text{prop}_n \rangle, \text{state}_{\text{sid}}, \text{seval}_{\text{sid}} \rangle$ is computed as:

$$\begin{aligned} \text{res} &= \text{seval}_{\text{sid}}(\text{SR}_0 \cdot \ldots \cdot \text{SR}_m), \\ \text{where } \text{SR}_i &= \text{state}_{\text{sid}}(\langle \text{LR}_0, \ldots, \text{LR}_n \rangle) \\ \text{and } \forall tid \in \text{TID}, \forall j \in [0, n], \text{LR}_j &= \begin{cases} tid \mapsto \text{prop}_j.\text{eval}(\text{proj}(tid, i, \text{prop}_j, \mathcal{R}_{\text{sid}})) & \text{if } tid \in \text{pool}(\text{prop}_j.\text{type}) \\ tid \mapsto \text{na} & \text{otherwise} \end{cases} \end{aligned}$$

Example 56 (Evaluating scope properties) We use LTL to formalize three different scope properties based on the scope states form Example 55 operating on the alphabet {activereader, activewriter, allreaders, onewriter}:

1. Mutual exclusion between readers and writers: $\varphi_0 \stackrel{\text{def}}{=}$ activewriter **XOR** activereader.
2. Mutual exclusion between writers: $\varphi_1 \stackrel{\text{def}}{=}$ activewriter $\implies$ onewriter.
3. All readers must read a written value: $\varphi_2 \stackrel{\text{def}}{=}$ activereader $\implies$ allreaders.

We recall that a scope state is a vector of boolean values for the atomic propositions in the following order: activereader, activewriter, allreaders, and onewriter. The sequence of scope states form Example 55: $\langle \bot, \top, \bot, \top \rangle \cdot \langle \top, \bot, \top, \bot \rangle \cdot \langle \bot, \top, \bot, \top \rangle$ complies with the specification $\varphi_0 \wedge \varphi_1 \wedge \varphi_2$. *

Evaluating with concurrency information. We notice that in Definition 26 monitors on local properties are reset for each concurrency region. As such, they are unable to express properties that are local to threads but span multiple concurrency regions. The semantics of function res conceptually focus on treating concurrency regions independently. However, we can account for elaborating the expressiveness of local properties by extending the

alphabet for each local property with the atomic proposition sync which delimits the concurrency region. The atomic proposition sync denotes that the scope synchronizing action has occurred, and adds it to the trace. We need to take careful consideration that threads may sleep and in fact not receive any events during a concurrent region. For example, consider two threads waiting on a lock, when a thread gets the lock, the other will not. As such, to pass the sync event to the local specification of the sleeping thread requires we instrument very intrusively to account for that, a requirement we do not want to impose. Therefore, we add the restriction that local properties are only evaluated if at least one event relevant to the local property is encountered in the concurrency region (that is not the synchronization event). Using that consideration, we are able to define an evaluation that considers all events starting from concurrent region 0 up to i, and adding sync events between scopes. This allows local monitors to account for synchronization, either to reset or check more expressive specifications such as "*a reader can read at most n times every m concurrency regions*", and "*writers must always write a value that is greater than the last write*".

Definition 27 (Evaluating a scope property without resetting) Given the concurrent execution, using the synchronizing predicate sync_{sid}, we obtain the regions $\mathcal{R}_{\text{sid}}(i)$ for $i \in [0, m]$ with $m = |\text{idx}_{\text{sid}}| + 1$. The evaluation of a scope property without reset (noted $\text{res}_{\text{wsync}}$) for the scope $\langle \text{sid}, \text{sync}_{\text{sid}}, \langle \text{prop}_0, \ldots, \text{prop}_n \rangle, \text{state}_{\text{sid}}, \text{seval}_{\text{sid}} \rangle$ is computed as:

$$
\begin{aligned}
\text{res}_{\text{wsync}} &= \text{seval}_{\text{sid}}(\text{SR}_0 \cdot \ldots \cdot \text{SR}_m), \\
\text{where } \text{SR}_i &= \text{state}_{\text{sid}}(\langle \text{LR}_0, \ldots, \text{LR}_n \rangle) \\
\text{and } \forall tid \in \text{TID}, \forall j \in [0, n], \text{LR}_j &= \begin{cases} tid \mapsto \text{prop}_j.\text{eval}(\text{tr}_0 \cdot \text{sync} \cdot \ldots \cdot \text{sync} \cdot \text{tr}_i) & \text{if } tid \in \text{pool}(\text{prop}_j.\text{type}) \wedge \text{tr}_i \neq \mathcal{E} \\ tid \mapsto \text{na} & \text{otherwise} \end{cases} \\
\forall k \in [0..i], \text{tr}_k &= \text{proj}(tid, k, \text{prop}_j, \mathcal{R}_{\text{sid}})
\end{aligned}
$$

11.2.4 Monitoring Scopes

Overview. Our main concern when developing an algorithm for evaluating scope properties is to not introduce additional synchronization. We also seek to interfere the least we can in the program execution. As such we rely on the synchronization found in the program itself. We recall from Section 11.1, that the program execution model consists of actions and a partial order among those actions. While serving as a theoretical model, it is not efficient to capture all actions, and their order. In this section, we do not focus on instrumentation to capture actions. Actions can be captured using any instrumentation framework such as AspectJ or DiSL, or through custom instrumentation. The reason for this is that all actions local to a thread are already totally ordered, and synchronizing actions are guaranteed by the underlying model to be totally ordered. For example, synchronizing actions based on locking are guaranteed to be totally ordered by the JVM implementation. Instead, we focus on the assembly and evaluation of scope states.

Scope channel. Local properties are evaluated by thread local monitors, which result is stored in a scope state for a given scope region (see Section 11.2.2). We use the notion of a *scope channel* to store information about the various scope states during the execution. We associate each scope with *one* scope channel, and each scope channel with its own timestamp and scope monitor responsible of checking the scope property. The timestamp can only be incremented by the scope monitor, which is only invoked when the synchronizing action for the scope is reached. The timestamp is shared and can only be read by monitors of local properties. The scope channel provides for each timestamp and for each monitor an exclusive memory slot to write its result, such that it does not conflict with other monitors.

Example 57 (Scope channel) Figure 11.3 displays the channel associated with the scope monitoring discussed in Example 55. For each scope region, the channel allows each monitor an exclusive memory slot to write its result (if it chooses to participate). The slots marked with a dash (-) are not necessary. Furthermore, na indicates that the thread was given a slot, but it did not write anything in it (see Definition 26). ∗

For a timestamp t, local monitors no longer write any information for any scope state with timestamp inferior to t,

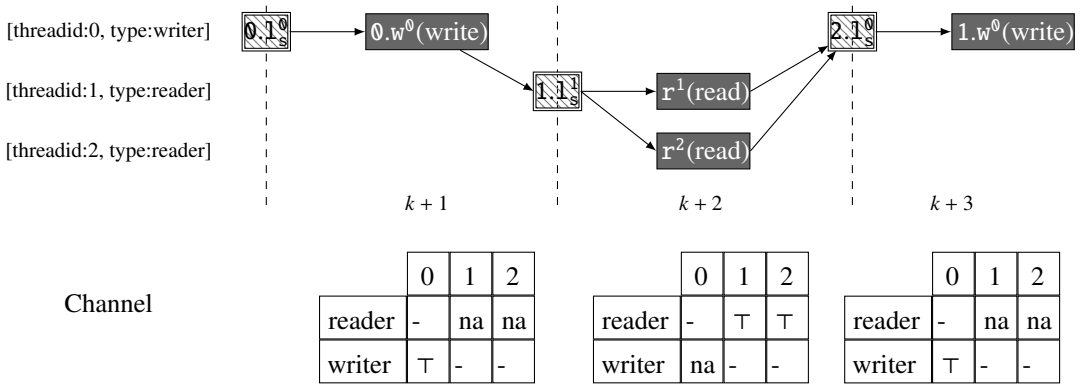


Figure 11.3: Example of a scope channel for 1-Writer 2-Readers.

this makes such states always consistent to be read by any monitor associated with the scope. While this is not in the scope of the paper, it allows monitors to effectively access past data of other monitors consistently.

Thread-local monitors. Thread-local monitors are responsible for monitoring a local property for a given thread. Multiple such monitors can exist on a given thread, depending on the needed properties to check, and the thread types the thread belongs to. A thread-local monitor is associated with a scope channel. It receives an event, performs checking and is able to write its result on the channel at the current timestamp (of that channel).

Scope monitors. As mentioned when defining scope channels, scope monitors are responsible for checking the property at the level of the scope. Upon reaching a synchronizing action by any of the threads associating with the scope, the given thread will invoke the scope monitor. The scope monitor relies on the scope channel (shared among all threads) to have access to all observations, additional memory can be allocated for its own state, but it has to be shared among all threads associated with the scope. The scope monitor is invoked atomically after reaching the scope synchronizing action. First, it compiles the scope state based on the results of the thread-local monitors stored in the scope channel. Second, it invokes the verification procedure on the generated state. Finally, before completing, it increments the timestamp associated with the scope channel.

In the next section (Section 11.3), we present our priliminary findings for monitoring *readers-writers* with the specifications discussed in this section (Example 55).

11.3 Preliminary Evaluation

In this section, we monitor *readers-writers* with our approach, using the specification found in Example 55. For the purpose of this example, we utilize the standard LTL_3 semantics defined over the $\mathbb{B}_3$ verdict domain. As such, we all the local and scope properties types are $\mathbb{B}_3$. We introduce the syntax needed to describe scope properties in Section 11.3.1, and preliminary results on hand-written monitors in Section 11.3.2.

11.3.1 Expressing Properties

We recall from Example 55, the properties that we check. Locally we check for an eventual read and write (resp. **F**(read) and **F**(write)), then we count the threads participating, to form the following atomic propositions: activereader, activewriter, allreaders, onewriter. They indicate respectively that: at least one reader, at least one writer, the number of readers that performed a read is equal to the number of reader threads in the program, and exactly one writer performed a write. We recall the scope properties from Example 56 they are: mutual exclusion between readers and writers, ensuring that all readers perform a read when at least one does, and mutual exclusion

Listing 11.1: Readers-writers specification.

```
1  selectors {
2    event AR
3      on    "ReentrantLock.Acquire"
4      when "%lockname == SharedArr.resourceLock"
5  }
6
7  types{
8    reader {
9      spawn on "Reader.Run"
10     event read on "SharedArr.read"
11   }
12   writer {
13     spawn on "Writer.Run"
14     event write on "SharedArr.write"
15   }
16 }
17
18 scope s0 (AR) {
19   property s0r on reader(read)  is "LTL=F(read)"
20   property s0w on writer(write) is "LTL=F(write)"
21
22   atom activereader : count(s0r, T) > 0,
23   atom activewriter : count(s0w, T) > 0,
24   atom onewriter    : count(s0w, T) == 1,
25   atom allreaders   : count(s0r, T) == size(reader)
26
27   check "LTL=G(
28          (activereader XOR activewriter)
29       && (activewriter => onewriter )
30       && (activereader => allreaders)
31     )"
32 }
```

between writers. These properties are all defined on the same scope, the one that alternates between readers and writers (Example 51).

Listing 11.1 illustrates the specification for writing local properties and a single scope with the three scope properties. We first define the synchronizing predicates using the *selectors* keyword (Lines 1-5). In this case we define the predicate AR by looking for an action labeled ReentrantLock.Acquire, and we match on its context using the *lockname* key.

After defining the synchronizing predicates, we define the thread types and their associated events (Lines 7-16). For each type we determine the spawn event (Lines 9,13), then all remaining events (Lines 10,14). In this case we match reading and writing based on the action label. Once thread types are defined, we determine scopes (Line 18-32).

We introduce one scope with the id s0, and specify that its synchronizing predicate is AR (Line 2-4). Then we determine the local properties, by giving a name for each property (Lines 19-20), determining the thread type and the events it applies on. In this case each property is expressed using the same string passed to LamaConv [Ins], which is used to synthesize the monitors. After defining the local property, we define the atomic propositions needed for the scope state (Lines 22-25). For this purpose, we use the helper function count to compute a count of how many monitors (on multiple threads) returned a certain verdict for a local property. Using the function count to summarize the result of local properties, we establish the atomic propositions needed for the scope state. Finally, we introduce the scope property (Lines 27-31), which is generated similarly to a local property but using the atomic propositions for the scope.

11.3.2 Preliminary Assessment of Overhead

Experiment setup. For the purpose of the experiment, we instrument *readers-writers* to insert our monitors, and compare our approach to global monitoring using a custom aspect written in AspectJ. In total, we have three scenarios: non-monitored, global, and opportunistic. In the first scenario (non-monitored), we do not perform monitoring. In the second and third scenario, we perform global and opportunistic monitoring. Using our approach we verify the three properties expressed in Section 11.3.1: mutual exclusion between readers and writers, mutual exclusion between the writers themselves, and ensuring all readers participate in reading a written value. For global

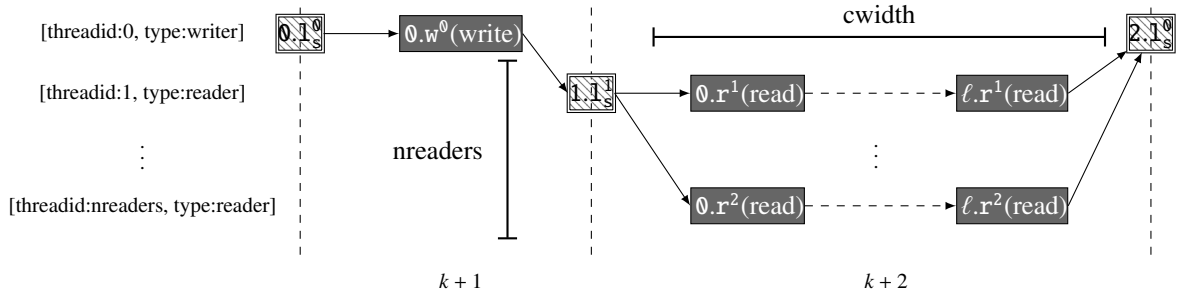


Figure 11.4: Parametrizing concurrency regions using number of participating parallel threads (nreaders) and width of the concurrency region (cwidth) with $\ell \stackrel{\text{def}}{=} \text{cwidth} - 1$.

monitoring, we are unable to check concurrency properties, as such we verify *only* the third property (ensuring all readers participate), as it amounts to counting the different threads of type reader that performed a read action between any two write actions, and ensuring the count matches the total number of readers. As such, the custom aspect simply locks and increments necessary counters to check a single scope property. We recall that global monitoring introduces additional locks for reads that occur concurrently. We perform the experiment on a NUMA machine consisting of 24 NUMA nodes containing each a 6-core Intel(R) Xeon(R) CPU E5-2620 v2 using Java HotSpot(TM) 64-Bit Server VM (build 25.0-b70, mixed mode) with the AspectJ compiler version 1.8.9.

Measures. In order to evaluate the overhead of our approach, we are interested in defining parameters to characterize concurrency regions found in *readers-writers*. We identify two parameters: the *number of readers* (nreaders), and the *width of the concurrency region* (cwidth). They are illustrated in Figure 11.4. On the one hand, nreaders determines the maximum parallel threads that are verifying local properties in a given concurrency region. On the other hand, cwidth determines the number of reads each reader performs concurrently. Parameter cwidth is measured in number of read events generated. By increasing the size of the concurrency regions, we increase lock contention when multiple concurrent events cause a global monitor to lock. We fix the number of writers to 1, $\text{nreaders} \in \{1, 3, 7, 15, 23, 31, 63, 127\}$ and $\text{cwidth} \in \{1, 5, 10, 15, 30, 60, 100, 150\}$. We perform a total of $100,000$ writes and $400,000$ reads, where reads are distributed evenly across readers. We measure the execution time (in ms) of 50 runs of the program for each of the parameters and scenarios.

Preliminary results. We report the results using the averages, while providing the scatter plots with linear regression curves in Figures 11.5, and 11.6. Figure 11.5 shows the overhead when varying the number of readers (nreaders). We notice that for the base program (non-monitored), the execution time increases from 1058 ms (when nreaders = 1) to 1377 ms (when nreaders = 127) as lock contention overhead becomes more prominent and the JVM is managing more threads. In the case of global monitoring, we notice a stable runtime ranging from 1215 ms (when nreaders = 1) to 1218 ms (when nreaders = 127), while for opportunistic monitoring, we see a much greater increase ranging from 1341 ms (when nreaders = 1) to 1816 ms (when nreaders = 127). We recall that for global monitoring, we utilize only one monitor for verifying the entire program, and the overhead is merely that of locking, while for opportunistic monitoring, we deploy monitors on each thread of the program, and evaluates the scope property using one monitor. Since we have more monitors running the more threads are executing, the increase in overhead is expected. Overall, the overall overhead is acceptable, as opportunistic is also checking two additional properties (for mutual exclusion). We now consider the width of the concurrency region.

Figure 11.6 shows the overhead when varying the width of the concurrency region (cwidth). We observe that for the base program (non-monitored), the execution time decreases from 1506 ms (when cwidth = 1) to 810 ms (when cwidth = 150), as the more reads can be performed concurrently. In the case of global monitoring, we notice a decrease from 1655 ms (when cwidth = 1) to 1166 ms (when cwidth = 150), while for opportunistic monitoring, we see a much greater decrease going from 2159 ms (when cwidth = 1) to 1018 ms (when cwidth = 150). By increasing the number of concurrent events in a concurrency region, we highlight the overhead introduced by locking the global monitor. We recall that a global monitor must lock to linearize the trace, and as such interferes with the concurrency of a program. This can be seen by looking at the two curves for global and opportunistic monitoring, we see that opportunistic follows closer the speedup of the non-monitored program, while global

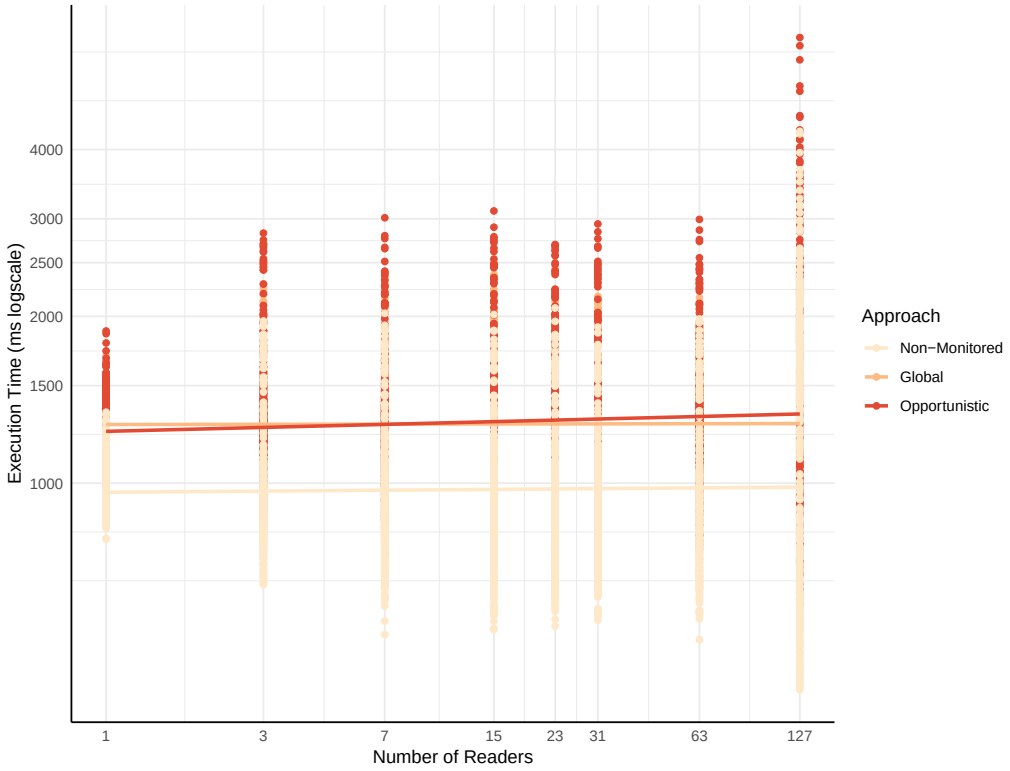


Figure 11.5: Execution time for *readers-writers* for non-monitored, global, and opportunistic monitoring. Parameter nreaders indicates the number of readers.

monitoring is much slower. For opportunistic monitoring, we expect a positive performance payoff when events in concurrency regions are dense.

Conclusion and Perspectives

Conclusion. We introduced a generic two-level decentralized specification to express properties on multithreaded programs. Our approach distinguishes between thread-local properties and properties that span concurrency regions, referred to as scopes. Using scopes, we are able to establish scope states that follow a total order, allowing any existing technique to verify scope-level properties. Our approaches relies heavily on existing totally-ordered operations in the program. However, by utilizing the existing synchronization, we are able to reduce overhead and remain sound when monitoring online. Finally, our preliminary evaluation suggests that monitoring with two-levels incurs a low overhead.

Perspectives. While the preliminary results appear promising, additional work needs to be invested to thoroughly investigate overhead, and complete the automatic synthesis and instrumentation of monitors. Furthermore, expressiveness of the specification can be increased by allowing for scopes to be contained in other scopes. This allows for properties that target not just thread-local properties, but also concurrent regions enclosed in other concurrent regions, utilizing the full hierarchical setting of decentralized specifications.

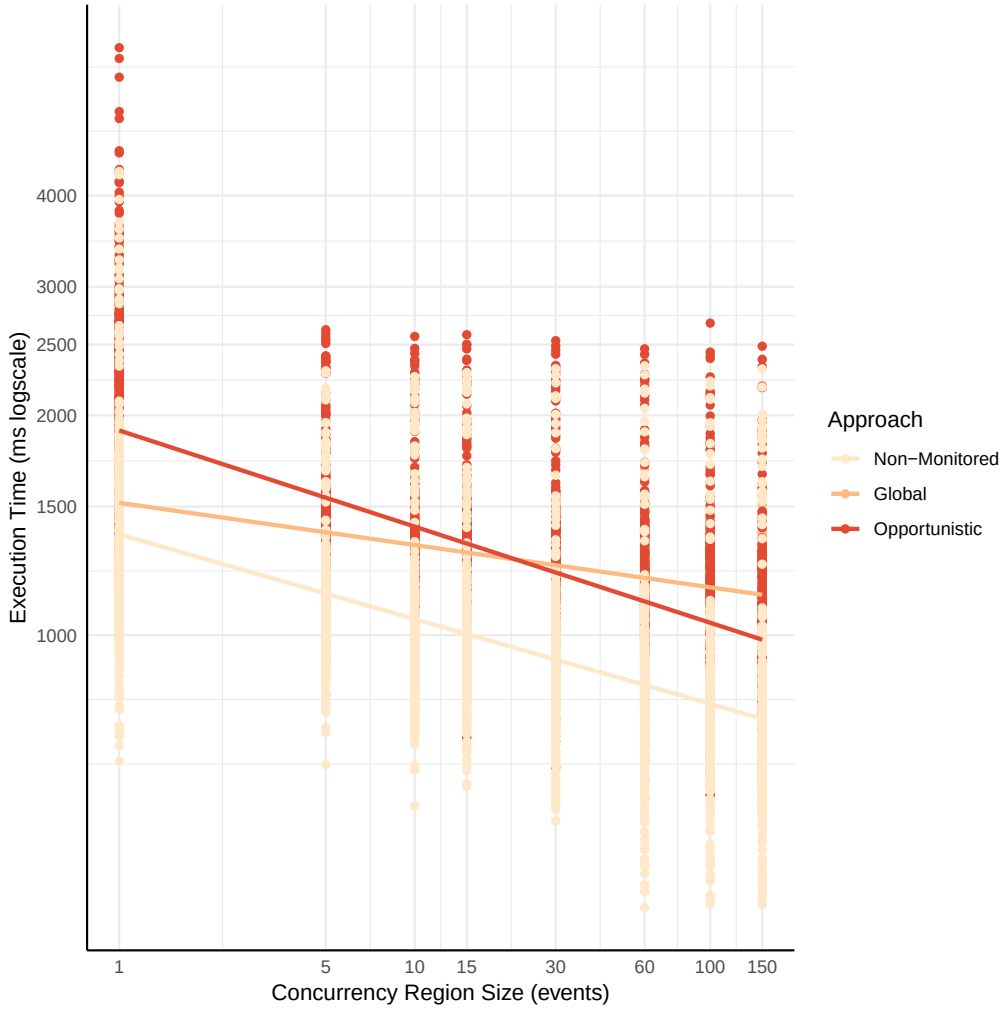


Figure 11.6: Execution time for *readers-writers* for non-monitored, global, and opportunistic monitoring. Parameter cwidth indicates the number of successive read events in a concurrency region.

Part IV

Conclusions and Perspectives

CHAPTER 12

Conclusion and Perspectives

This chapter concludes the thesis with a summary of contributions for all parts, and perspectives for future work.

Summary of Contributions

Focusing on decentralized RV, this thesis aims to improve on existing work by responding to three challenges: (1) managing partial information in decentralized monitoring, (2) separating the monitor deployment problem from the monitoring procedure itself, and (3) introducing decentralization in the specification semantics in a modular and hierarchical way.

Part one. To respond to these challenges, we introduced in the first part the `EHE` data structure (Chapter 5) and decentralized specifications (Chapter 6). We summarize the contributions:

1. We designed the `EHE` data structure (Chapter 5) to manage partial information. `EHE` replicates under strong eventual consistency while presenting two additional properties: determinism (Proposition 2) and soundness when encoding automata (Proposition 3). Furthermore, we have shown how `EHE` can be used for decentralized monitoring and presented the cost model for `EHE` as a generic model for analyzing algorithms with partial information (Section 5.2).

2. In Chapter 6, we introduced decentralized specifications which aim to answer the challenges (2) and (3). Decentralized specifications operates over multiple monitors, such that each monitor has its own specification limited to what is observable for the monitor, and references to other monitors. We elaborated on the semantics used to evaluate references while presenting a two-step view of decentralized monitoring, which consists in separating the monitor topology from the monitoring.

3. By separating between monitor deployment and execution and determining a hierarchy of specifications (based on dependency), we characterize and compute two properties of decentralized specifications (in Section 6.2): *compatibility* of a specification to a target system, and *monitorability* of a given decentralized specification (extended from monitorability of centralized specifications).

4. Furthermore, we introduced the design of `THEMIS` (Chapter 7), our framework for designing, monitoring, analyzing and simulating decentralized monitoring algorithms. `THEMIS` integrates with other tools (notably for monitor synthesis), and is able to provide an experimental environment for assessing existing decentralized monitoring algorithms, and new algorithms that rely on decentralized specifications.

Part two. After elaborating on the theory needed to tackle the challenges, we explored applications of decentralized specifications in two contexts: comparison of decentralized monitoring algorithms (Chapter 8), and monitoring smart homes (Chapter 9).

1. In Chapter 8, we adapted three existing decentralized monitoring algorithms (presented in Section 1.3.1) to our approach, using an API and dedicated data structures, in a uniform manner for the purpose of analysis and simulation. This allowed us to parametrize the analysis of decentralized monitoring algorithms with information delay incurred by partial observations.

2. We conducted a worst-case analysis of three decentralized monitoring algorithms relying on automata-based approaches, utilizing boolean rewriting (with a known minimal form) instead of LTL rewriting;

3. We simulated the three decentralized monitoring algorithms in a synthetic benchmark and a user interface application to validate the analysis while (1) exploring the effect of using different random probability distributions for trace generation, and their impact on coverage, and (2) studying the performance of various decentralized monitoring algorithms for different patterns of specifications of the same application.

4. Focusing on the strength of decentralized specifications, we explored their use in a hierarchical setting for monitoring smart homes in Chapter 9. We applied decentralized RV to analyze traces of over 36,000 timestamps spanning 27 sensors in a real smart apartment (Section 9.1.1). We showed how to go beyond system properties, to specify ADL using RV, and more complex interdependent specifications defined on up to 27 atomic propositions (Section 9.1.2).

5. We highlighted the modularity and scalability of decentralized specifications by being able to synthesize monitors (where existing approaches are unable to) and to reduce overhead when monitoring complex interdependent specifications (Section 9.4.1).

6. Furthermore, we identified limitations inherent to using formal LTL specifications to determine user behavior (Section 9.4.2).

Part three. After elaborating on direct applications of decentralized specifications, we instantiated a special form of decentralized specifications to monitor multithreaded RV. In our technique a specification for a multithreaded program can be formed of two levels, the first targeting thread-local properties, and the second focusing on concurrency regions.

1. In Chapter 10, we explored and identified the limitations of existing RV techniques when dealing with multithreaded programs. In particular the challenges of monitoring multithreaded programs stem from the following facts: (1) events in a concurrent program follow a partial order; (2) most formalisms used by RV do not account for partial orders, but specify behavior over sequences of events (i.e., events are totally ordered); and (3) an instrumented program must capture the order of events as it happens during the execution to pass it to monitors. As such, we explored situations where: (1) a linear trace does not represent the underlying program execution; (2) a linear trace hides some implicit assumptions which affect RV; and (3) it is insufficient to use a linear trace for monitoring multithreaded programs.

2. By considering all the challenges, we identified GPredict (Section 10.4.1) as the tool able to effectively monitor multithreaded programs. However, it does so with multiple limitations: (1) the analysis is performed offline, (2) the specifications do no take into account a variable number of threads, and (3) it cannot express succinctly important properties that rely on counting.

3. To account for the limitations of GPredict, we introduce in Chapter 11 a two-level decentralized specification which is used to monitor multithreaded programs. Our approach relies on existing operations in the program that follow a total order, and utilizes atomic regions in the program to avoid locking. By using the hierarchies of decentralized specifications, it defines two levels of monitoring. The first checks local properties on a given thread, the second verifies properties in a given concurrent region delimited by operations that follow a total order. In this manner, we have a total order that is sound on both levels.

4. We ensured that the approach is sufficiently generic to be used by existing RV techniques, including streams and techniques which formalism uses total order, and extended expressiveness to specify behavior over an arbitrary number of threads, and include arbitrary transformations to the result of local monitoring.

Perspectives

Whether to improve the underlying solutions or expand beyond them, the various challenges tackled in this thesis inspire us to tackle new problems in the future. We identify three tracks for future work relative to decentralized specifications: theoretical extensions, implementation extensions, and application scope expansions.

Theoretical Extensions

The first track for future work deals with theoretical extensions to decentralized specifications. Since decentralized specifications allow us to explicitly model dependencies between the various subspecifications of a given system, and separate the topology of monitors from the monitoring logic itself. Utilizing the additional relationships allows us to tackle more properties and better incorporate system knowledge.

Extending properties for decentralized specifications. By introducing decentralized specifications, we separate the monitor topology from the monitoring algorithm. We presented two properties: compatibility and monitorability (Section 6.2). *Monitorability* ensures that given a specification, monitors are able to eventually emit a verdict, for all possible traces. *Compatibility* ensures that a monitor topology can be deployed on a given system. This opens the way to reason about additional properties of the specifications (summarized in Section 6.3) such as equivalence of specifications, optimization of compatibility, and synthesis in general. Defining equivalence between decentralized specifications and computing it proves challenging, as delay needs to be taken into account when comparing various decentralized specifications. For example, it is possible to define equivalence as a relation between any two decentralized specifications that emit the same verdict for all possible traces. However, computing such a relation is not trivial and a challenging problem.

In Section 6.2.2, we discussed the *compatibility* of a decentralized specification with a given architecture. We defined the notion of a *compatible assignment* (Definition 19), which ensures that the communication dependency between monitors is respected when deployed on components. In brief, if two monitors must communicate, then they must be deployed on components capable of communicating. We recall that for monitors that solely depend on other monitors and not component observations, we have freedom to place them on multiple possible components. While compatibility provides the minimal requirement for monitors to be able to communicate, it does not account for performance. Knowing that we have a graph for the system, and a graph for the monitors dependencies, it is possible to annotate the system graph by labeling vertices by computation resources (such as compute power or memory limit of a given system component), and add weights to edges that represent communication resources (such as communication bandwidth or restrictions). Therefore, we are able to compute the optimal monitor placement that maximizes the usage of resources of a given system architecture. The problem becomes that of *optimal compatibility*.

We recall from Section 1.3.2 that we aimed in this thesis to separate the generation of the monitor topology from the monitoring itself. By separating the problems, we are now able to define extensions to the monitor synthesis problem. In particular, we can are no longer just interested in generating monitors, but also considering additional properties based on their inter-dependencies. All the properties discussed in this thesis and possible future properties become relevant when synthesizing decentralized specifications. For example, we are able to extend the synthesis problem as follows: *starting from a decentralized specification $\mathcal{D}$, can we generate a new decentralized specification $\mathcal{D}'$ that is verdict equivalent to $\mathcal{D}$, while optimizing the compatibility for memory (or communication) for a given system architecture?* Furthermore, it is possible to add constraints on specifications to manage delay, as currently we only consider eventual consistency. By analyzing the specification it is possible to devise stricter guarantees on delay, or transform it so as to reduce delay to improve performance when monitoring.

Incorporating system knowledge. Utilizing information about the system can prove useful when performing monitoring. Typically, the truth value of atomic propositions is assumed to be independent. However, this is not the case in some systems. For example, when branching, it is possible to know which events are possible for a given side of the branch and which are not. In [ZLD12], the authors utilize knowledge about the system (namely, the relationships between events) to predict the next elements in a trace allowing the monitoring to often conclude before observing the entire trace. As such, by assuming some relationships between atomic propositions, it is

possible to incorporate the same approach to decentralized specifications. This allows us to simplify the needed partial information required to progress with the monitoring, allowing for more performant decentralized monitoring. Furthermore, given that we model the dependencies between subspecifications, we are able to go beyond only considering relationships between atomic propositions, to determine the relationships between subspecifications themselves in a large decentralized specification. For example, if we consider the specifications tied to detecting the user activity in the smart apartment (Section 9.1.2), it is possible to define that activities need not be checked so long as the user has not entered the apartment. In large decentralized system, some monitors can be turned off when certain conditions are met, allowing better scalability and more customized verification.

Towards decentralized runtime enforcement. In this thesis, we have utilized the concept of decentralized specifications for runtime verification. Runtime verification does not interfere with the system being observed. However, it is possible to consider the usefulness of decentralized specifications for runtime enforcement [Fal10, FMFR11, FMRS18]. In runtime enforcement, monitors are capable of taking snapshots of the execution, rolling back, and suppressing events in a program. Performing runtime enforcement can be challenging since the delay of detecting violations in decentralized systems is not always clear. Furthermore, there are two notions of delay: one to verify and one to apply the correction. Decentralized specifications provide enforcers with additional information about relationships between specifications that is useful when enforcing specifications across a large system. The added information can be used to deploy decentralized enforcers, determine guarantees by analyzing the dependencies and the various subspecifications, and determine the communication and dependencies between enforcers in a similar fashion as would monitors in decentralized runtime verification.

Implementation Extensions

Possible improvement in the future could be focused on improving the `EHE` data structures as it is at the core of dealing with partial information, and improving the `THEMIS` metrics for a deeper analysis of decentralized monitoring algorithms.

Improving the `EHE`. Our implementation of the `EHE` can be seen as a proof-of-concept implementation simply to illustrate the presented approaches in this thesis. As such, it is not optimized for efficient monitoring. It is possible to improve the `EHE` by improving the internal representation of expressions, and performing additional simple simplifications before calling external simplifiers. For example, it is possible to use information about states that cannot be reached (i.e., the associated expression evaluates to $\bot$), to deduce additional information to deduce the reached state. Trivially, one can see that if n states are reachable in the `EHE`, and during the execution we determined that $n-1$ are non-reachable, we can deduce using determinism that the remaining state has been reached. In Section 7.4.2, we discussed the impact of choosing simplifiers and garbage collection on improving the performance of `EHE`. Since `EHE` is used as the unified representation data structure for partial information, any improvement to its performance, immediately reflects as an improvement on all algorithms utilizing it.

Improving `THEMIS` metrics. We also consider creating new metrics for THEMIS to analyze more aspects of decentralized monitoring algorithms. For example, in Section 8.2.2, we considered extended the metrics based on delay, simplifications, and messages size and frequency with convergence which measures load-balance. By identifying important aspects for decentralized specifications, we are able to detect the best scenario for a given decentralized monitoring algorithm. We see that this is important, as in two specifications out of the five in a real scenario when using Chiron traces (Section 8.2.4), the choreography algorithm using a simple heuristic generated an inefficient decentralized specification. A key advantage of `THEMIS`' design is that new metrics would be automatically instrumented on all existing algorithms and experiments could be easily replicated to compare and validate them.

Application Scope Expansions

The last track of future work focuses on improving expressiveness when using decentralized specifications for monitoring. In particular we look at monitoring user activities (Chapter 9) and hierarchies for multithreaded

monitoring (Chapter 11).

Expanding RV to monitor activities. Our assessment in Chapter 9 has shown that RV can be effective in monitoring user behavior in the addition of system properties. However, the level of confidence is still lower than one would expect from system behavior (for critical systems). The main limitation of RV is tied to the rigidity of formalizing behavior. The rigidity itself makes it so that users must perform exactly the expected behavior and not an approximation of it, for it to count as a detected activity. It is possible to improve the detection rate by improving the expressiveness of specifications and also using hybrid detection strategies.

Expanding the expressiveness of specifications helps in improving the detection rate, as it can describe more behavior. For example, when using MTL instead of LTL (see Section 3.1.1), it is possible to describe better the interaction with respect to time, as MTL is specifically designed to assign properties over intervals of time. More generally, we believe that the decentralization of richer specification languages is desirable. For instance, we consider (i) using a counting semantics able to compute the number of steps needed to witness the satisfaction or violation of a specification [BBNR18] (ii) using techniques allowing to deal with uncertainty (e.g., in case of message loss) [BG13] (iii) using spatio-temporal specifications (e.g. [HJK+15]) to reason on physical locations in the house, and (iv) using a quantitative semantics possibly with time [BFMU17].

Furthermore, since decentralized specifications rely on references to manage the monitoring of subspecifications (so long as they eventually return a verdict), they essentially reference other subspecifications as black-boxes. As such, it is possible to incorporate other detection techniques in different levels of the hierarchies to account for both strict and non-strict user behavior. For example, it is possible to use machine learning to detect that the user is sitting on a couch using a camera, when no pressure sensor exists for the couch. The machine learning technique could be easily integrated in the decentralized specification tree (but not analyzed), so long as it is modeled as a monitor that eventually returns a verdict when one is necessary. If we were to apply the same decentralized specification in a different house with an accurate couch sensor, we can then use a monitor on sensor data directly. From the point of view of all specifications depending on the reference to detecting whether or not the user on a couch, the way it is detected is not relevant.

Finally, we consider using runtime enforcement [Fal10, FMFR11, FMRS18] techniques (especially those for timed specifications [FJMP16]) to guarantee system properties and improve safety in the house (e.g., disabling cooking equipment whenever specification `firehazard` is violated). This requires to define the foundations for decentralized runtime enforcement on the theoretical side, and provide houses and monitors with actuators on the practical side.

Scope hierarchies for multithreaded monitoring. For monitoring multithreaded programs in Chapter 11, we introduced a two-level decentralized specification. At the first level we specified local properties, checked for each thread. Then, on the second level, we defined scope properties, which uses the local properties to determine a more complex property in a concurrent region. For this to work, we relied on the notion of scopes, which delimits concurrency regions using operations in the program guaranteed to follow a total order (such as acquiring locks). However, it is possible for scopes to be enclosed within other scopes. For example, consider in *readers-writers*, there exists a lock that manages the counter keeping track of the number of active readers. The readers counter lock is only accessed when the readers are active and have already taken the lock to the resource (thus forbidding writers to access it). Any scope defined on this lock will be enclosed in the scope that alternates between readers and writers defined on the resource lock. In the future, we hope to expand the two-level specification to account for multi-level concurrency regions, by considering not only local properties in a given scope, but also other scope properties for scopes enclosed within other scopes. By doing so, we are able to express more properties, and to define a fully hierarchical decentralized specification for multithreaded programs.

Bibliography

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 345–364. ACM, 2005.

[ABF+10] Laksono Adhianto, S. Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[ABH16] Afshin Amighi, Stefan Blom, and Marieke Huisman. Vercors: A layered approach to practical verification of concurrent software. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016*, pages 495–503. IEEE Computer Society, 2016.

[ACD+99] George S. Avrunin, James C. Corbett, Matthew B. Dwyer, Corina S. Pasareanu, and Stephen F. Siegel. Comparing finite-state verification techniques for concurrent software. Technical report, 1999.

[ACM+18] Hugo L. S. Araujo, Gustavo Carvalho, Morteza Mohaqeqi, Mohammad Reza Mousavi, and Augusto Sampaio. Sound conformance testing for cyber-physical systems: Theory and implementation. *Sci. Comput. Program.*, 162:35–54, 2018.

[AEIE13] Hande Özgür Alemdar, Halil Ertan, Özlem Durmaz Incel, and Cem Ersoy. ARAS human activity datasets in multiple homes with multiple residents. In *7th International Conference on Pervasive Computing Technologies for Healthcare and Workshops, PervasiveHealth 2013*, pages 232–235. IEEE, 2013.

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, December 1996.

[AGVY11] Edward Aftandilian, Samuel Z. Guyer, Martin T. Vechev, and Eran Yahav. Asynchronous assertions. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011*, pages 275–288. ACM, 2011.

[AMF14] Houssam Abbas, Hans D. Mittelmann, and Georgios E. Fainekos. Formal property verification in a conformance testing framework. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014*, pages 155–164. IEEE, 2014.

[ANB+95] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995.

[APS+17] Syeda Aimal, Komal Parveez, Arje Saba, Sadia Batool, Hafsa Arshad, and Nadeem Javaid. Energy optimization techniques for demand-side management in smart homes. In *Advances in Intelligent Networking and Collaborative Systems, The 9th International Conference on Intelligent Networking and Collaborative Systems, INCoS-2017.*, volume 8 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 515–524. Springer, 2017.

[Baa08] Sara Baase. *A Gift of Fire*. Prentice Hall, 2008.

[Bar13] Ezio Bartocci. Sampling-based decentralized monitoring for networked embedded systems. In Luca Bortolussi, Manuela L. Bujorianu, and Giordano Pola, editors, *Proceedings Third International Workshop on Hybrid Autonomous Systems, HAS 2013, Rome, Italy, 17th March 2013.*, volume 124 of *EPTCS*, pages 85–99, 2013.

[BBF14] Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2014.

[BBNR18] Ezio Bartocci, Roderick Bloem, Dejan Nickovic, and Franz Röck. A counting semantics for monitoring LTL specifications over finite traces. *CoRR*, abs/1804.03237, 2018.

[BCE+16] David A. Basin, Germano Caronni, Sarah Ereth, Matús Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring of temporal specifications. *Formal Methods in System Design*, 49(1-2):75–108, 2016.

[BCR09] Oliver Brdiczka, James L. Crowley, and Patrick Reignier. Learning situation models in a smart home. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 39(1):56–63, 2009.

[Bec04] Kent L. Beck. *JUnit - pocket guide: quick lookup and advice*. O'Reilly, 2004.

[BF12] Andreas Klaus Bauer and Yliès Falcone. Decentralised LTL monitoring. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

[BF16] Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48(1-2):46–93, 2016.

[BF18] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[BFB+17a] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. *International Journal on Software Tools for Technology Transfer*, pages 1–40, 2017.

[BFB+17b] Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer*, Apr 2017.

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 68–84, 2012.

[BFM+18] Patrick Baudin, Jean-Christophe Filliatre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language. Version 1.7., 2018. http://frama-c.com/download/acsl.pdf.

[BFMU17] Alexey Bakhirkin, Thomas Ferrère, Oded Maler, and Dogan Ulus. On the quantitative semantics of regular expressions over real-valued signals. In Alessandro Abate and Gilles Geeraerts, editors, *Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings*, volume 10419 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017.

[BFR+16] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A. Rosenblueth, and Corentin Travers. Decentralized asynchronous crash-resilient runtime verification. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016*, volume 59 of *LIPIcs*, pages 16:1–16:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[BFRT16] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. Challenges in fault-tolerant distributed runtime verification. In Margaria and Steffen [MS16], pages 363–370.

[BG13] Ezio Bartocci and Radu Grosu. Monitoring with uncertainty. In Luca Bortolussi, Manuela L. Bujorianu, and Giordano Pola, editors, *Proceedings Third International Workshop on Hybrid Autonomous Systems, HAS 2013, Rome, Italy, 17th March 2013.*, volume 124 of *EPTCS*, pages 1–4, 2013.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in EAGLE. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings*. IEEE Computer Society, 2004.

[BHL+10] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. *Journal of Logic and Computation*, 20(3):707–723, June 2010.

[BKZ15] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Failure-aware runtime verification of distributed systems. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015, December 16-18, 2015, Bangalore, India*, volume 45 of *LIPIcs*, pages 590–603. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[BL11] Andreas Bauer and Martin Leucker. The theory and practice of SALT. In *NASA Formal Methods - Third International Symposium, NFM 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 13–40. Springer, 2011.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[BLP15] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 367–385. ACM, 2015.

[BLS10] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[BMP18] Francesco Adalberto Bianchi, Alessandro Margara, and Mauro Pezzè. A survey of recent trends in testing concurrent software systems. *IEEE Trans. Software Eng.*, 44(8):747–783, 2018.

[BOS+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*, pages 55–66. ACM, 2011.

[BRH10] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3):675–706, 2010.

[BSB17] Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-Based Runtime Verification for Alternation-Free HyperLTL. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 77–93, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[BU08] David Buchfuhrer and Christopher Umans. The complexity of boolean formula minimization. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2008.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CC15] James L. Crowley and Joelle Coutaz. An ecological view of smart home technologies. In Boris De Ruyter, Achilles Kameas, Periklis Chatzimisios, and Irene Mavrommati, editors, *Ambient Intelligence*, pages 1–16, Cham, 2015. Springer International Publishing.

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2003.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CER99] CERN. http://dst.lbl.gov/acssoftware/colt/, 1999. http://dst.lbl.gov/ACSSoftware/colt/.

[CES97] Krzysztof Czarnecki, Ulrich W Eisenecker, and Patrick Steyaert. Beyond objects: Generative programming. In *CES97a [1 46]. The 23rd International Conference On Software Engineering*, pages 5–14. Citeseer, 1997.

[CF99] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[CF16a] Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.

[CF16b] Christian Colombo and Yliès Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.

[CFB+12] Sylvain Cotard, Sébastien Faucou, Jean-Luc Béchennec, Audrey Queudet, and Yvon Trinquet. A data flow monitoring service based on runtime verification for AUTOSAR. In Geyong Min, Jia Hu, Lei (Chris) Liu, Laurence Tianruo Yang, Seetharami Seelam, and Laurent Lefèvre, editors, *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 1508–1515. IEEE Computer Society, 2012.

[CFC15] Beichen Chen, Zhong Fan, and Fengming Cao. Activity recognition based on streaming sensor data for assisted living in smart homes. In *2015 International Conference on Intelligent Environments, IE 2015*, pages 124–127. IEEE, 2015.

[CHL+18] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: Temporal stream-based specification language. *CoRR*, abs/1808.10717, 2018.

[CHN+12] Liming Chen, Jesse Hoey, Chris D. Nugent, Diane J. Cook, and Zhiwen Yu. Sensor-based activity recognition. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 42(6):790–808, 2012.

[CLRC17] Julien Cumin, Grégoire Lefebvre, Fano Ramparany, and James L. Crowley. A dataset of routine daily activities in an instrumented home. In *Ubiquitous Computing and Ambient Intelligence - 11th International Conference, UCAmI 2017, Proceedings*, volume 10586 of *Lecture Notes in Computer Science*, pages 413–425. Springer, 2017.

[CPA10] Christian Colombo, Gordon J. Pace, and Patrick Abela. Compensation-aware runtime monitoring. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2010.

[CPS09a] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, pages 135–149, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[CPS09b] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 33–37. IEEE Computer Society, 2009.

[CR05] Feng Chen and Grigore Roşu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 546–550. Springer, April 2005.

[CR07] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 569–588. ACM, 2007.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CSL+13] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405. ACM, 2013.

[CSR08] Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for java. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering ICSE 2008*, pages 221–230. ACM, 2008.

[DAC99a] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420. ACM, 1999.

[DAC99b] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns project, 1999. http://patterns.projects.cs.ksu.edu/.

[DAC99c] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns project: List of specifications, 1999. http://patterns.projects.cs.ksu.edu/documentation/specifications/AFTER.raw.

[DDG+18] Normann Decker, Boris Dreyer, Philip Gottschling, Christian Hochberger, Alexander Lange, Martin Leucker, Torben Scheffel, Simon Wegener, and Alexander Weiss. Online analysis of debug trace data for embedded systems. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018*, pages 851–856. IEEE, 2018.

[DF14] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 201–211. IEEE Computer Society, 2014.

[DGH+17] Normann Decker, Philip Gottschling, Christian Hochberger, Martin Leucker, Torben Scheffel, Malte Schmitz, and Alexander Weiss. Rapidly adjustable non-intrusive online monitoring for multi-core systems. In Simone Cavalheiro and José Fiadeiro, editors, *Formal Methods: Foundations and Applications*, pages 179–196, Cham, 2017. Springer International Publishing.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DL13] Alexandre Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, volume 8172 of *Lecture Notes in Computer Science*, pages 442–445, Hanoi, Vietnam, October 2013. Springer.

[DL14] Volker Diekert and Martin Leucker. Topology, monitorable properties and runtime verification. *Theoretical Computer Science*, 537:29 – 41, 2014. Theoretical Aspects of Computing (ICTAC 2011).

[dOPSR16] Daniel Alfonso Gonçalves de Oliveira, Laércio Lima Pilla, Thiago Santini, and Paolo Rech. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. *IEEE Trans. Computers*, 65(3):791–804, 2016.

[Dor15] Frank Dordowsky. An experimental study using ACSL and frama-c to formulate and verify low-level requirements from a DO-178C compliant avionics project. In Catherine Dubois, Paolo Masci, and Dominique Méry, editors, *Proceedings Second International Workshop on Formal Integrated Development Environment, F-IDE 2015*, volume 187 of *EPTCS*, pages 28–41, 2015.

[DSS+05] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*, pages 166–174. IEEE Computer Society, 2005.

[Duf91] David A. Duffy. *Principles of automated theorem proving*. Wiley professional computing. Wiley, 1991.

[EF18] Antoine El-Hokayem and Yliès Falcone. RV for Multithreaded Programs Tutorial, 2018.

[EFJ16] Antoine El-Hokayem, Yliès Falcone, and Mohamad Jaber. Modularizing crosscutting concerns in component-based systems. In *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016. Proceedings*, pages 367–385, Berlin, Germany, 2016. Springer.

[EFJ18] Antoine El-Hokayem, Yliès Falcone, and Mohamad Jaber. Modularizing behavioral and architectural crosscutting concerns in formal component-based systems - Application to the Behavior Interaction Priority framework. *J. Log. Algebr. Meth. Program.*, 99:143–177, 2018.

[EH07] Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs, First International Conference, TAP 2007. Revised Papers*, volume 4454 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2007.

[EHF] Antoine El-Hokayem and Yliès Falcone. THEMIS Smart Home Artifact Repository. gitlab.inria.fr/monitoring/themis-rv18smarthome.

[EHF17a] Antoine El-Hokayem and Yliès Falcone. Monitoring decentralized specifications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 125–135, New York, NY, USA, 2017. ACM.

[EHF17b] Antoine El-Hokayem and Yliès Falcone. Themis: A tool for decentralized monitoring algorithms. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17-DEMOS), Santa Barbara, CA, USA, July 2017*, 2017.

[EHF17c] Antoine El-Hokayem and Yliès Falcone. Themis website, 2017. https://gitlab.inria.fr/monitoring/themis.

[EHF18a] Antoine El-Hokayem and Yliès Falcone. Bringing Runtime Verification Home. In *RV 2018 - 18th International Conference on Runtime Verification*, pages 1–17, Limassol, Cyprus, Nov 2018.

[EHF18b] Antoine El-Hokayem and Yliès Falcone. Can We Monitor All Multithreaded Programs? In *RV 2018 - 18th International Conference on Runtime Verification*, pages 1–24, Limassol, Cyprus, Nov 2018.

[EHF18c] Antoine El-Hokayem and Yliès Falcone. THEMIS Article Artifact, 2018. https://gitlab.inria.fr/monitoring/themis-artifact-article.

[EN501] CENELEC EN50128. Railway applications-communication, signalling and processing systems-software for railway control and protection systems. *European Committee for Electrotechnical Standardization*, 2001.

[EPP+17] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. BARRACUDA: Binary-level analysis of runtime races in CUDA programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 126–140. ACM, 2017.

[Fal10] Yliès Falcone. You should better enforce than verify. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.

[FCF14] Yliès Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and generalized decentralized monitoring of regular languages. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*, volume 8461 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2014.

[FFM12a] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[FFM12b] Alessandro Fantechi, Wan Fokkink, and Angelo Morzenti. Some trends in formal methods applications to railway signaling. *Formal Methods for Industrial Critical Systems*, pages 61–84, 2012.

[FHC97] Sara Fleury, Matthieu Herrb, and Raja Chatila. G^{en}om: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS'97*, pages 842–849, USA, 1997. IEEE.

[FHR13] Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron a. Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO science for peace and security series, d: information and communication security*, pages 141–175. ios press, 2013.

[FHST18] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. RVHyper : A Runtime Verification Tool for Temporal Hyperproperties. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 194–200. Springer, 2018.

[FJMP16] Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Sci. Comput. Program.*, 123:2–41, 2016.

[FJN+15] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. *Software and System Modeling*, 14(1):173–199, 2015.

[FKRT18] Yliès Falcone, Srdan Krstić, Giles Reger, and Dmitriy Traytel. A Taxonomy for Classifying Runtime Verification Tools. *To appear.*, 2018.

[FMFR11] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[FMRS18] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. Runtime failure prevention and reaction. In Bartocci and Falcone [BF18], pages 103–134.

[FNRT15] Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma. Second international competition on runtime verification CRV 2015. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015.

[For18] Formal Systems Laboratory. JavaMOP4 Syntax, 2018.

[FPS18] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime verification for decentralised and distributed systems. In Bartocci and Falcone [BF18], pages 176–210.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for Model Checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015.

[GA14] Stefan J. Galler and Bernhard K. Aichernig. Survey on test data generation tools - an evaluation of white- and gray-box testing tools for c#, c++, eiffel, and java. *STTT*, 16(6):727–751, 2014.

[GK10] Paul Gastin and Dietrich Kuske. Uniform satisfiability problem for local temporal logics over Mazurkiewicz traces. *Inf. Comput.*, 208(7):797–816, 2010.

[GP12] Gabriella Gigante and Domenico Pascarella. Formal methods in avionic software certification: The DO-178C perspective. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Proceedings, Part II*, volume 7610 of *Lecture Notes in Computer Science*, pages 205–215. Springer, 2012.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.

[Hal16] Sylvain Hallé. When RV Meets CEP. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 68–91, Cham, 2016. Springer International Publishing.

[HGB16] Sylvain Hallé, Sébastien Gaboury, and Bruno Bouchard. Activity recognition through complex event processing: First findings. In *Artificial Intelligence Applied to Assistive Technologies and Smart Environments, Papers from the 2016 AAAI Workshop*, volume WS-16-01 of *AAAI Workshops*. AAAI Press, 2016.

[HJK+15] Iman Haghighi, Austin Jones, Zhaodan Kong, Ezio Bartocci, Radu Gros, and Calin Belta. Spatel: A novel spatial-temporal logic and its applications to networked systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, pages 189–198, New York, NY, USA, 2015. ACM.

[HKB+18] Sylvain Hallé, Raphaël Khoury, Quentin Betti, Antoine El-Hokayem, and Yliès Falcone. Decentralized enforcement of document lifecycle constraints. *Inf. Syst.*, 74(Part):117–135, 2018.

[HKEF16] Sylvain Hallé, Raphaël Khoury, Antoine El-Hokayem, and Yliès Falcone. Decentralized enforcement of artifact lifecycles. In *20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, 2016*, pages 1–10. IEEE Computer Society, 2016.

[HKG17] Sylvain Hallé, Raphaël Khoury, and Sébastien Gaboury. Event stream processing with multiple threads. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 359–369, Cham, 2017. Springer International Publishing.

[HLR15] Jeff Huang, Qingzhou Luo, and Grigore Rosu. Gpredict: Generic predictive concurrency analysis. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1*, pages 847–857. IEEE Computer Society, 2015.

[HMR14] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 337–348, New York, NY, USA, 2014. ACM.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hof99] David Hoffman. I had a funny feeling in my gut, 1999. https://www.washingtonpost.com/wp-srv/inatl/longterm/coldwar/shatter021099b.htm.

[HR04] Klaus Havelund and Grigore Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, March 2004.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[Hwa] Hwaci. SQLite3 Home Page. https://www.sqlite.org.

[IBM15] IBM. Shrike technical overview, 2015. http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview.

[IML09] Juha Itkonen, Mika Mäntylä, and Casper Lassenius. How do testers do it? an exploratory study on manual testing practices. In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009*, pages 494–497. ACM / IEEE Computer Society, 2009.

[Ins] Institute for Software Engineering and Programming Languages. LamaConv - Logics and Automata Converter Library. www.isp.uni-luebeck.de/lamaconv.

[Kas18] Mohamad Kassab. Testing practices of software in safety critical systems: Industrial survey. In Slimane Hammoudi, Michal Smialek, Olivier Camp, and Joaquim Filipe, editors, *Proceedings of the 20th International Conference on Enterprise Information Systems, ICEIS 2018, Volume 2.*, pages 359–367. SciTePress, 2018.

[Kat83] Sidney Katz. Assessing self-maintenance: Activities of daily living, mobility, and instrumental activities of daily living. *Journal of the American Geriatrics Society*, 31(12):721–727, 1983.

[KHBZ15] Jorne Kandziora, Marieke Huisman, Christoph Bockisch, and Marina Zaharieva-Stojanovski. Runtime assertion checking of JML annotations in multithreaded applications with e-OpenJML. In Rosemary Monahan, editor, *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015*, pages 8:1–8:6. ACM, 2015.

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, 2001, Proceedings*, pages 327–353. Springer, 2001.

[KHJF18] Sean Kauffman, Klaus Havelund, Rajeev Joshi, and Sebastian Fischmeister. Inferring event stream abstractions. *Formal Methods in System Design*, 53(1):54–82, 2018.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[KLSV18] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018.

[KVB+99] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *11th Euromicro Conference on Real-Time Systems (ECRTS 1999), 9-11 June 1999, York, England, UK, Proceedings*, pages 114–122. IEEE Computer Society, 1999.

[KVY11] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence Abstractions for Relaxed Memory Models. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 187–198. ACM, 2011.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[LFKV18] João M. Lourenço, Jan Fiedor, Bohuslav Krena, and Tomás Vojnar. Discovering concurrency errors. In Bartocci and Falcone [BF18], pages 34–60.

[LLF+96] Jacques-Louis Lions, Lennart Lbeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. ARIANE 5 Flight 501 Failure Report by the Inquiry Board, 1996. http://www-users.math.umn.edu/~arnold/disasters/ariane5rep.html.

[LLO03] Karl J. Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *Comput. J.*, 46(5):542–565, 2003.

[LLR+17] Paula Lago, Frédéric Lang, Claudia Roncancio, Claudia Jiménez-Guarín, Radu Mateescu, and Nicolas Bonnefond. The ContextAct@A4H real-life dataset of daily-living activities - activity recognition using model checking. In *Modeling and Using Context - 10th International and Interdisciplinary Conference, CONTEXT 2017, Proceedings*, volume 10257 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2017.

[LS09a] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[LS09b] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, May 2009.

[LSàT16] Martin Leucker, Malte Schmitz, and Danilo à Tellinghusen. Runtime verification for interconnected medical devices. In Margaria and Steffen [MS16], pages 380–387.

[Luc05] David C. Luckham. *The power of events - an introduction to complex event processing in distributed enterprise systems*. ACM, 2005.

[LVK+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*, pages 618–632. ACM, 2017.

[LW01] Kamal Lodaya and Pascal Weil. Rationality in algebras with a series operation. *Inf. Comput.*, 171(2):269–293, 2001.

[MAN+17] Sumit Majumder, Emad. Aghayi, Moein Noferesti, Hamidreza Memarzadeh-Tehran, Tapas Mondal, Zhibo Pang, and M. Jamal Deen. Smart homes for elderly healthcare - recent advances and research challenges. *Sensors*, 17(11):2496, 2017.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.

[MB15] Menna Mostafa and Borzoo Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015*, pages 494–503. IEEE Computer Society, 2015.

[MC13] Richard Mayr and Lorenzo Clemente. Advanced automata minimization. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, 2013*, pages 63–74. ACM, 2013.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[MJG+12] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391. ACM, 2005.

[MR04] B. Meenakshi and Ramaswamy Ramanujam. Reasoning about layered message passing systems. *Computer Languages, Systems & Structures*, 30(3-4):171–206, 2004.

[MS16] Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, volume 9953 of *Lecture Notes in Computer Science*, 2016.

[MSG07] Neeraj Mittal, Alper Sen, and Vijay K. Garg. Solving computation slicing using predicate detection. *IEEE Trans. Parallel Distrib. Syst.*, 18(12):1700–1713, 2007.

[MT08] Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2008.

[MVZ+12] Lukás Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: a domain-specific language for bytecode instrumentation. In Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel, editors, *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, pages 239–250. ACM, 2012.

[MZA+15] Lukás Marek, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Aibek Sarimbekov, Walter Binder, and Petr Tuma. Introduction to dynamic program analysis with disl. *Sci. Comput. Program.*, 98:100–115, 2015.

[NCMG17] Aravind Natarajan, Himanshu Chauhan, Neeraj Mittal, and Vijay K. Garg. Efficient abstraction algorithms for predicate detection. *Theor. Comput. Sci.*, 688:24–48, 2017.

[NFBB17] Hosein Nazarpour, Yliès Falcone, Saddek Bensalem, and Marius Bozga. Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation. *Formal Asp. Comput.*, 29(6):951–986, 2017.

[Nic11] Thomas Nicely. Pentium FDIV flaw, 2011. http://www.trnicely.net/pentbug/pentbug.html.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theor. Comput. Sci.*, 13:85–108, 1981.

[OG07] Vinit A. Ogale and Vijay K. Garg. Detecting temporal logic predicates on distributed computations. In Andrzej Pelc, editor, *Distributed Computing, 21st International Symposium, DISC 2007, Proceedings*, volume 4731 of *Lecture Notes in Computer Science*, pages 420–434. Springer, 2007.

[O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[PDGO08] Yoann Pigné, Antoine Dutot, Frédéric Guinand, and Damien Olivier. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. *CoRR*, abs/0803.2093, 2008.

[PH66] E.S. Pearson and H.O. Hartley. *Biometrika tables for statisticians*. Cambridge Univ. Press, 1966.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, 1977*, pages 46–57. IEEE Computer Society, 1977.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[QS08] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of conurrent systems in cesar. In Grumberg and Veith [GV08], pages 216–230.

[RC17] Nisha Rathee and Rajender Singh Chhillar. A survey on test case generation techniques using UML diagrams. *JSW*, 12(8):643–648, 2017.

[RCR15] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. MarQ: Monitoring at Runtime with QEA. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 2015.

[RDCN18] Davi Silva Rodrigues, Márcio Eduardo Delamaro, Cléber Gimenez Corrêa, and Fátima L. S. Nunes. Using genetic algorithms in test data generation: A critical systematic mapping. *ACM Comput. Surv.*, 51(2):41:1–41:23, 2018.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[RH05] Grigore Rosu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Autom. Softw. Eng.*, 12(2):151–197, 2005.

[RH16] Giles Reger and Klaus Havelund. What Is a Trace? A Runtime Verification Perspective. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, Lecture Notes in Computer Science, pages 339–355, Cham, October 2016. Springer.

[RHF16] Giles Reger, Sylvain Hallé, and Yliès Falcone. Third international competition on runtime verification - CRV 2016. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016.

[RLL+13] Henrique Rebêlo, Ricardo Massa Ferreira Lima, Gary T. Leavens, Márcio Cornélio, Alexandre Mota, and César A. L. Oliveira. Optimizing generated aspect-oriented assertion checking code for JML using program transformations: An empirical study. *Sci. Comput. Program.*, 78(8):1137–1156, 2013.

[Run06] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[SGM02] Clemens A. Szyperski, Dominik Gruntz, and Stephan Murer. *Component software - beyond object-oriented programming, 2nd Edition*. Addison-Wesley component software series. Addison-Wesley, 2002.

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 148–172. Springer, 2009.

[SJP12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.

[SKV17] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, 2017*, volume 3 of *Kalpa Publications in Computing*, pages 164–173. EasyChair, 2017.

[SPBZ11] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.

[SRA03] Koushik Sen, Grigore Rosu, and Gul Agha. Generating optimal linear temporal logic monitors by coinduction. In *Advances in Computing Science - ASIAN 2003 Programming Languages and Distributed Computation, 8th Asian Computing Science Conference, Mumbai, India, December 10-14, 2003, Proceedings*, pages 260–275. Springer, 2003.

[SS14] Torben Scheffel and Malte Schmitz. Three-valued asynchronous distributed runtime verification. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2014, Lausanne, Switzerland, October 19-21, 2014*, pages 52–61. IEEE, 2014.

[SVAR04] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pages 418–427. IEEE Computer Society, 2004.

[SWYS11] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. *Generating Data Race Witnesses by an SMT-Based Analysis*, pages 313–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Tea99] The Chiron Team. Chiron user interface, 1999. http://laser.cs.umass.edu/verification-examples/chiron/index.html.

[The17] The Apache Software Foundation. Apache Commons BCEL, 2017. http://commons.apache.org/proper/commons-bcel/.

[The18] The Eclipse Foundation. The AspectJ project, 2018.

[TIL04] Emmanuel Munguia Tapia, Stephen S. Intille, and Kent Larson. Activity recognition in the home using simple and ubiquitous sensors. In *Pervasive Computing, Second International Conference, PERVASIVE 2004, Vienna, Austria, April 21-23, 2004, Proceedings*, volume 3001 of *Lecture Notes in Computer Science*, pages 158–175. Springer, 2004.

[TNM18] Himanshu Thapliyal, Rajdeep Kumar Nath, and Saraju P. Mohanty. Smart home environment for mild cognitive impairment population: Solutions to improve care and quality of life. *IEEE Consumer Electronics Magazine*, 7(1):68–76, 2018.

[TR05] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145 – 162, 2005.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[VGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In David A. Watt, editor, *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Proceedings*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2000.

[vKEK10] Tim van Kasteren, Gwenn Englebienne, and Ben J. A. Kröse. Transferring knowledge of activity recognition across sensor networks. In *Pervasive Computing, 8th International Conference, Pervasive 2010. Proceedings*, volume 6030 of *Lecture Notes in Computer Science*, pages 283–300. Springer, 2010.

[WB86] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. *Operating Systems Review*, 20(1):57–66, 1986.

[ZLD12] Xian Zhang, Martin Leucker, and Wei Dong. Runtime verification with predictive semantics. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 418–432. Springer, 2012.

Part V

Appendix

APPENDIX A

Proofs of Chapter 5

Proof (Proof of Proposition 2) : The proof is by induction on the number of timestamps in the EHE, i.e., $n = |\text{rounds}(\mathcal{I})|$. Without loss of generality, we can assume the automaton being encoded is normalized (see Remark 1), that is, all shared edges between any two states are replaced by one edge which is labeled by the disjunction of their labels.

One could see that the base case only contains the initial state of an automaton, i.e., $\mathcal{I}^0 = [0 \mapsto q_0 \mapsto \top]$, and as such the proposition holds.

Let us consider $n = 2$, we have $\mathcal{I}^1 = \text{mov}(\mathcal{I}^0, 0, 1)$. To compute mov, we first consider $\text{next}(\mathcal{I}^0, 0)$ which considers all states reachable from q_0 as the only tuple in $\mathcal{I}^0$ is $\langle 0, q_0 \rangle$, i.e., $\text{next}(\mathcal{I}^0, 0) = \{q' \in Q \mid \exists e \in Expr : \delta(q_0, e) = q'\}$, we know that only one such e can evaluate to $\top$ for any memory encoded with the identity encoder (idt), since the automaton is deterministic. Let us collect all such states and their expressions as $P = \{\langle q', e \rangle \in Q \times Expr \mid \exists e \in Expr : \delta(q_0, e) = q'\}$. We note that $\mathcal{I}^1(0, q_0) = \top$ is the only entry for timestamp 0. The property holds trivially for that entry. We now consider the entries in $\mathcal{I}^1$ for timestamp 1. Each of tuple $\langle q', e \rangle \in P$ corresponds to the expression $\mathcal{I}^1(1, q')$, constructed with $\text{to}(\mathcal{I}^0, 0, q', \text{ts}_1) = \mathcal{I}^0(0, q_0) \wedge \text{ts}_1(e)$. We note that ts_1 only adds the timestamp 1 to each atomic proposition. As such, for any given memory encoded with ts_1 only one such expression can be evaluated to $\top$.

Inductive step: We assume that the property holds on $\mathcal{I}^{n-1}$ for some $n \in \mathbb{N}$, that is:
$\forall \mathcal{M} \in Mem, \forall t \in \text{rounds}(\mathcal{I}^{n-1}), \exists q \in Q : (\text{eval}(\mathcal{I}^{n-1}(t, q), \mathcal{M}) = \top) \implies (\forall q' \in Q \backslash \{q\} \implies \text{eval}(\mathcal{I}^{n-1}(t, q'), \mathcal{M}) \neq \top)$. Let us prove that the property holds for $\mathcal{I}^n$.

The approach is similar to that of $n = 2$ using the recursive structure of the EHE to generalize. We decompose $\mathcal{I}^n$, using the definition of mov, as follows:

$$\mathcal{I}^n = \mathcal{I}^{n-1} \dagger_\vee \biguplus_{q' \in \text{next}(\mathcal{I}^{n-1}, n)} \{n \mapsto q' \mapsto \text{to}(\mathcal{I}^{n-1}, n-1, q', \text{ts}_n)\}$$

We know that $\text{rounds}(\mathcal{I}^n) = \text{rounds}(\mathcal{I}^{n-1}) \cup \{n\}$. The induction hypothesis states that the property holds for all entries in $\mathcal{I}^{n-1}$ (i.e. for $t \in \text{rounds}(\mathcal{I}^{n-1})$), we consider the entries for timestamp n only. Since $\biguplus$ applies $\dagger_\vee$ on the

entire set, and it is associative and commutative we consider the expression for a given state after all the merges, without consideration of order of merges. As such the states associated with timestamp n are computed using $\text{next}(\mathcal{I}^{n-1}, n)$. We have $\forall q' \in \text{next}(\mathcal{I}^{n-1}, n)$:

$$\begin{aligned}\mathcal{I}^n(n, q') &= \text{to}(\mathcal{I}^{n-1}, n-1, q', \text{ts}_n) && \text{(Def. mov)}\\ &= \bigvee_{\{\langle q,e'\rangle |\ \delta(q,e')=q'\}} (\mathcal{I}^{n-1}(n-1, q) \wedge \text{ts}_n(e')) && (1)\end{aligned}$$

(1) follows from the definition of to. If we examine the disjunction we notice using the induction hypothesis that there can only be a unique $q_\text{u} \in Q$ with $\mathcal{I}^{n-1}(n-1, q_\text{u})$ that evaluates to $\top$ at timestamp $n-1$. As such, the conjunction can only hold for one such q_u. Consequently, we can rewrite (1) by simplifying the disjunction and considering only states reachable from q_u, as the rest cannot evaluate to $\top$. Let us collect all such states and expressions in the set $P_\text{u} = \{\langle q', e'\rangle \mid q' \in \text{next}(\mathcal{I}^{n-1}, n) \wedge \exists e' \in \mathit{Expr}_{AP} : \delta(q_\text{u}, e') = q'\}$. The only entries that can still evaluate to $\top$ are:

$$\begin{aligned}\forall \langle q', e'\rangle \in P_\text{u} : \mathcal{I}^n(n, q') &= \mathcal{I}^{n-1}(n-1, q_\text{u}) \wedge \text{ts}_n(e')\\ &= \text{ts}_n(e')\end{aligned}$$

Since the automaton is deterministic, we know that we have one unique expression e_u that can evaluate to $\top$, given any memory encoded with idt. Since ts_n only adds the timestamp n to the atomic propositions without changing the expression, we deduce that only $\text{ts}_n(e_\text{u})$ evaluates to $\top$. As such, there is a unique expression that can evaluate to $\top$ for any given memory encoded with ts_n. Furthermore, we know that the expression has only been encoded with ts_n so when memories encoded with different timestamps or encoders are merged, they do not affect the evaluation of $\text{ts}_n(e_\text{u})$. As such, we have a unique entry $\mathcal{I}^n(n, q'_\text{u})$ s.t. $\delta(q_\text{u}, e_\text{u})$ that can evaluate to $\top$. Therefore:

$$\begin{aligned}\forall \mathcal{M} \in \mathit{Mem}, \forall t \in \text{rounds}(\mathcal{I}^n), \exists q \in Q :\\ (\text{eval}(\mathcal{I}^n(t, q), \mathcal{M}) = \top) \implies\\ (\forall q' \in Q : q' \neq q \implies \text{eval}(\mathcal{I}^n(t, q'), \mathcal{M}) \neq \top)\end{aligned}$$

Lemma (Evaluation modulo encoding) : Given a trace tr of length i and a reconstructed global trace $\rho(\text{tr}) = \mathit{evt}_1 \cdot \ldots \cdot \mathit{evt}_i$, we consider two memories $\mathcal{M}^i_\mathcal{A}$ and $\mathcal{M}^i$ generated under different encodings. We consider $\mathcal{M}^i_\mathcal{A} = \text{memc}(\mathit{evt}_i, \text{idt})$, and $\mathcal{M}^i = \biguplus^2_{t\in[1,i]}\{\text{memc}(\mathit{evt}_t, \text{ts}_t)\}$. We show that an expression encoded using different encodings evaluates the same for the memories, that is:

$$\forall e \in \mathit{Expr}_{AP} : \text{eval}(\text{idt}(e), \mathcal{M}^i_\mathcal{A}) \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}^i).$$ ∎

Proof (Proof of Lemma 1) : We first note that for the first evaluation $\text{eval}(\text{idt}(e), \mathcal{M}^i_\mathcal{A})$, we rely only on the event evt_i since $\mathcal{M}^i_\mathcal{A} = \text{memc}(\mathit{evt}_i, \text{idt})$. This is not the case for $\text{eval}(\text{ts}_i(e), \mathcal{M}^i)$ as $\mathcal{M}^i = \biguplus^2_{t\in[1,i]}\{\text{memc}(\mathit{evt}_t, \text{ts}_t)\}$. However, we notice that for the second evaluation we evaluate the expression $\text{ts}_i(e)$, that is, where the expression where all atomic propositions have been encoded by the timestamp i. Therefore, let us denote the memory with the timestamp i by $\mathcal{M}' = \text{memc}(\mathit{evt}_i, \text{ts}_i)$. We can rewrite $\mathcal{M}^i$ as follows:

$$\begin{aligned}\mathcal{M}^i &= \text{memc}(\mathit{evt}_i, \text{ts}_i) && \dagger_2\ \biguplus^2_{t\in[1,k]}\{\text{memc}(\mathit{evt}_k, \text{ts}_k)\}\\ &= \mathcal{M}' && \dagger_2\ \biguplus^2_{t\in[1,k]}\{\text{memc}(\mathit{evt}_k, \text{ts}_k)\}.\end{aligned}$$

We know that all entries $\langle k, a\rangle \in \text{dom}(\mathcal{M}^i)$ with $k < i$ do not affect at all the evaluation of an expression encoded with ts_i. As such we have:

$$\forall e \in \mathit{Expr}_{AP} : \text{eval}(\text{ts}_i(e), \mathcal{M}^i) \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}')$$ ∎

We now show that the two memories $\mathcal{M}^i_\mathcal{A}$ and $\mathcal{M}'$ contain simply an encoding of the same atomic propositions. We have by construction the following:

$$\begin{aligned}\forall a \in \text{dom}(\mathcal{M}^i_\mathcal{A}) : \langle i, a\rangle \in \text{dom}(\mathcal{M}^i) \wedge\ \mathcal{M}^i_\mathcal{A}(a) = \mathcal{M}'(\langle i, a\rangle)\\ \forall \langle i, a'\rangle \in \text{dom}(\mathcal{M}') : a' \in \text{dom}(\mathcal{M}^i_\mathcal{A}) \wedge\ \mathcal{M}'(\langle i, a'\rangle) = \mathcal{M}^i_\mathcal{A}(a')\end{aligned}$$

As such we have: $\forall e \in \mathit{Expr}_{AP} : \text{eval}(\text{idt}(e), \mathcal{M}^i_\mathcal{A}) \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}') \Leftrightarrow \text{eval}(\text{ts}_i(e), \mathcal{M}^i)$.

Proof (Proof of Proposition 3) : Given a trace tr of length i and a reconstructed global trace $\rho(\text{tr}) = evt_1 \cdot \ldots \cdot evt_i$, the proof is done by induction on the length of the trace $|\rho(\text{tr})|$. We omit the label ℓ for clarity.

Base case: $|\rho(\text{tr})| = 0, \rho(\text{tr}) = \epsilon$

$$\Delta^*(q_0, \epsilon) = q_0 = \text{sel}(\mathcal{I}^0, [\,], 0)$$
$$\mathcal{I}^0 = \text{mov}([0 \mapsto q_0 \mapsto \top], 0, 0) = [0 \mapsto q_0 \mapsto \top]$$ ∎

We only have expression $\top$ which is mapped to q_0 at $t = 0$. Expression $\top$ requires no memory to be evaluated.

Inductive step: We assume that the property holds for a trace of length i for some $i \in \mathbb{N}$, that is $\Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_i) = \text{sel}(\mathcal{I}^i, \mathcal{M}^i, i) = q_i$. Let us prove that the property holds for any trace of length $i + 1$.

We now consider the transition functions in the automaton:

$$\begin{aligned}
q_{i+1} &= \Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_{i+1}) \\
&= \Delta(\Delta^*(q_0, evt_1 \cdot \ldots \cdot evt_i), evt_{i+1}) && \text{(Definition 12)} \\
&= \Delta(q_i, evt_{i+1}) && \text{(Induction Hypothesis)} \\
&\Leftrightarrow \exists \mathsf{e} \in Expr_{AP} : \\
&\quad \delta(q_i, expr) = q_{i+1} \wedge \text{eval}(e, \mathcal{M}^{i+1}_{\mathcal{A}}) = \top && (1)
\end{aligned}$$

We note that, since the automaton is deterministic, there is a unique q_{i+1} such that $q_{i+1} = \Delta(q_i, evt_{i+1})$.

We now consider the EHE operations to reach q_{i+1} from q_i.

$$\begin{aligned}
q_i &= \text{sel}(\mathcal{I}^i, \mathcal{M}^i, i) \\
&\Leftrightarrow e = \mathcal{I}^i(i, q_i) \text{ with } \text{eval}(e, \mathcal{M}^i) = \top && (2) \\
&\quad \wedge \forall q'_i \in Q : q'_i \neq q_i \implies \text{eval}(\mathcal{I}^i(i, q'_i)) \neq \top && \text{(Proposition 2)} \\
&\Leftrightarrow \text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) = \top && (3)
\end{aligned}$$

(3) From the induction hypothesis, we know that $\mathcal{I}^i(i, q_i) = \top$, thus:

$$\begin{aligned}
&\text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) \\
&\quad = \bigvee_{\{\langle q, e' \rangle | \delta(q, e') = q'_{i+1}\}} (\mathcal{I}^i(i, q) \wedge \text{ts}_{i+1}(e')) \\
&\quad = \bigvee_{\{\langle q, e'' \rangle | \delta(q, e'') = q'_{i+1} \wedge q \neq q_i\}} (\mathcal{I}^i(i, q) \wedge \text{ts}_{i+1}(e'')) \\
&\quad \vee \bigvee_{\{\langle q_i, e''' \rangle | \delta(q_i, e''') = q'_{i+1}\}} (\text{ts}_{i+1}(e''')).
\end{aligned}$$

We split the disjunction to consider the expressions that only come from state q_i, we now show that one such expression evaluates to $\top$. We know from (1), that one such expression can be taken in the automaton:

$$\begin{aligned}
\exists e \in Expr_{AP} : \delta(q_i, e) = q_{i+1} \wedge \text{eval}(e, \mathcal{M}^{i+1}_{\mathcal{A}}) = \top && (1) \\
\Leftrightarrow \text{eval}(\text{ts}_{i+1}(e), \mathcal{M}^{i+1}) = \top && (4) \\
\Leftrightarrow \text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) = \top && (5)
\end{aligned}$$

(4) is obtained using Lemma 1 and $\text{idt}(e) = e$.
(5) follows from the disjunction.

Using the same approach, we can show that $\forall q' \in \text{next}(\mathcal{I}^i, i) : q' \neq q_{i+1} \implies \text{to}(\mathcal{I}^i, i, q', \text{ts}_{i+1}) \neq \top$, since the first part of the conjunction does not evaluate to $\top$, and we know that the second part cannot evaluate to $\top$ by (2). Finally, $\text{to}(\mathcal{I}^i, i, q_{i+1}, \text{ts}_{i+1}) = \top$ iff $\text{sel}(\mathcal{I}^{i+1}, \mathcal{M}^{i+1}, i + 1) = q_i$.

APPENDIX B

Additional Details

B.1 Detailed Data for Section 8.2

Tables B.1 and B.2 present the detailed comparison for the synthetic scenario and Chiron, respectively. The metrics presented are (in order of columns): average information delay (δ_t), normalized average number of messages (#Msgs), normalized data transferred (Data), maximum simplifications done by any given monitor per run, averaged across all runs (S_{max}), normalized critical simplifications (S_{crit}), and convergence based on expressions evaluated ($Conv_E$). For more details on the metrics, see Section 8.2.2.

Table B.1: Synthetic Benchmark. Cells contains mean and standard deviation in parentheses.

Alg.	$\|C\|$	δ_t	#Msgs	Data	S_{max}	S_{crit}	$Conv_E$
Orch	3	0.48 (0.50)	2.44 (0.61)	31.84 (8.07)	0.00 (0.00)	0.00 (0.00)	0.65 (0.02)
	4	0.53 (0.50)	3.85 (0.94)	50.05 (12.39)	0.00 (0.00)	0.00 (0.00)	0.74 (0.02)
	5	0.64 (0.48)	5.30 (1.16)	69.20 (15.55)	0.00 (0.00)	0.00 (0.00)	0.79 (0.02)
	6	0.69 (0.46)	7.04 (1.50)	91.86 (20.02)	0.00 (0.00)	0.00 (0.00)	0.83 (0.02)
Migr	3	0.58 (0.58)	0.27 (0.32)	8.46 (15.32)	4.72 (4.41)	3.08 (2.66)	0.65 (0.02)
	4	0.71 (0.67)	0.32 (0.34)	17.45 (35.87)	6.10 (6.17)	4.03 (3.75)	0.73 (0.03)
	5	0.96 (0.71)	0.43 (0.34)	30.41 (56.68)	7.41 (6.18)	4.97 (3.76)	0.79 (0.03)
	6	1.19 (0.86)	0.50 (0.34)	98.80 (244.94)	10.09 (8.32)	6.74 (4.87)	0.82 (0.04)
Migrr	3	0.76 (0.69)	0.78 (0.33)	14.51 (18.40)	5.62 (4.99)	3.51 (2.93)	0.65 (0.02)
	4	1.02 (0.90)	0.76 (0.36)	31.76 (51.55)	7.64 (7.16)	4.58 (4.04)	0.74 (0.03)
	5	1.39 (1.04)	0.75 (0.35)	62.83 (91.89)	9.70 (7.88)	5.70 (4.25)	0.79 (0.03)
	6	1.72 (1.19)	0.70 (0.37)	180.35 (360.25)	12.56 (9.76)	7.35 (5.14)	0.82 (0.03)
Chor	3	1.47 (1.99)	2.79 (1.10)	24.98 (9.85)	60.22 (242.88)	12.27 (6.55)	0.16 (0.12)
	4	1.36 (1.52)	3.84 (1.23)	34.36 (10.94)	44.71 (184.05)	12.95 (5.98)	0.13 (0.12)
	5	1.41 (1.55)	4.63 (1.37)	41.17 (12.16)	44.06 (223.15)	12.68 (6.06)	0.12 (0.11)
	6	1.29 (1.38)	5.87 (1.66)	52.09 (14.77)	38.35 (215.27)	13.01 (6.01)	0.13 (0.12)

Table B.2: Metrics for Chiron traces. Cells contains mean and standard deviation in parentheses.

Alg.	Spec	δ_t	#Msgs	Data	S_{max}	S_{crit}	$Conv_E$
Orch	1	0.77 (0.42)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	2	1.00 (0.00)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	3	0.99 (0.10)	3.42 (0.02)	101.62 (1.00)	0.00 (0.00)	0.00 (0.00)	0.75 (0.00)
	5	0.94 (0.24)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	15a	1.00 (0.00)	2.95 (0.00)	87.46 (0.00)	0.00 (0.00)	0.00 (0.00)	0.74 (0.00)
	15b	1.00 (0.00)	3.00 (0.01)	88.90 (0.16)	0.00 (0.00)	0.00 (0.00)	0.75 (0.00)
Migr	1	1.66 (0.03)	0.02 (0.00)	0.52 (0.00)	8.00 (0.00)	2.03 (0.00)	0.74 (0.00)
	2	1.00 (0.00)	0.57 (0.00)	13.09 (0.10)	4.00 (0.00)	3.10 (0.01)	0.74 (0.00)
	3	1.86 (0.00)	0.88 (0.01)	70.23 (1.00)	13.00 (0.00)	9.76 (0.05)	0.75 (0.00)
	5	1.67 (0.00)	0.02 (0.00)	0.52 (0.00)	8.00 (0.00)	2.03 (0.00)	0.74 (0.00)
	15a	1.00 (0.00)	0.97 (0.00)	10.71 (0.04)	4.00 (0.00)	3.90 (0.00)	0.74 (0.00)
	15b	1.00 (0.00)	1.00 (0.00)	19.36 (0.35)	9.02 (2.00)	7.00 (0.03)	0.75 (0.00)
Migrr	1	1.98 (0.00)	1.01 (0.01)	50.19 (0.33)	9.80 (1.37)	5.29 (0.07)	0.74 (0.00)
	2	1.81 (0.02)	1.01 (0.01)	212.55 (3.17)	12.00 (0.00)	5.82 (0.05)	0.74 (0.00)
	3	2.37 (0.03)	0.89 (0.02)	147.00 (3.97)	15.97 (0.22)	10.65 (0.09)	0.75 (0.01)
	5	1.98 (0.01)	1.01 (0.00)	50.16 (0.30)	9.80 (1.35)	5.28 (0.06)	0.74 (0.00)
	15a	1.99 (0.01)	1.01 (0.01)	83.80 (0.34)	8.64 (0.94)	4.91 (0.01)	0.74 (0.00)
	15b	2.50 (0.01)	1.00 (0.00)	136.05 (0.27)	16.86 (0.35)	11.41 (0.01)	0.75 (0.00)
Chor	1	1.01 (0.00)	4.89 (0.00)	44.02 (0.00)	20.00 (0.00)	15.88 (0.32)	0.20 (0.01)
	2	133.86 (0.17)	2.95 (0.00)	26.52 (0.00)	2798.38 (211.11)	21.41 (0.74)	0.67 (0.02)
	3	1.22 (0.04)	4.40 (0.13)	39.64 (1.16)	23.65 (0.87)	18.18 (0.93)	0.33 (0.02)
	5	1.01 (0.00)	4.89 (0.00)	44.02 (0.00)	20.00 (0.00)	15.85 (0.33)	0.20 (0.01)
	15a	1.00 (0.00)	0.98 (0.00)	8.84 (0.00)	10.00 (0.00)	9.25 (0.07)	0.47 (0.01)
	15b	116.52 (1.19)	2.00 (0.00)	18.00 (0.00)	3387.04 (316.16)	28.01 (1.13)	0.71 (0.00)

B.2 Specifications for Producer Consumer

We present the specifications used for monitoring *producer-consumer* using Java-MOP (Listing B.1), LARVA (Listing B.2), and MarQ (Listing B.3). The detailed findings and description is found in Section 10.2.2. The monitors were designed for *global monitoring*, to ensure the trace is fed to the corresponding formalism as a total order. As such, for MarQ locking was needed.

Listing B.1: Java-MOP specification and monitor for *producer-consumer*.

```
ProdCons() {
  event produce before() :
    call(* Queue.add(*))
    && cflow(execution(* SynchQueue.produce(*)))
    { }
  event consume before() :
    call(* Queue.poll())
    && cflow(execution(* SynchQueue.consume()))
    { }
  cfg : S -> S produce S consume | epsilon
  @fail {
    System.out.println("Failed!");
    System.exit(1);
  }
}
```

Listing B.2: LARVA specification and monitor for *producer-consumer*.

```
1  IMPORTS { import java.util.*; }
2  GLOBAL{
3    VARIABLES      {  int cnt = 0;}
4    EVENTS{
5      produce() = { Queue.add()  }
6      consume() = { Queue.poll() }
7    }
8    PROPERTY prodcons {
9      STATES{
10       BAD { bad {
11           System.err.println("Failed!");
12           System.exit(1);
13       }}
14       NORMAL    { ok }
15       STARTING  { starting }
16     }
17
18     TRANSITIONS{
19       starting -> bad [consume]
20       starting -> ok  [produce\          \cnt++;]
21       ok  -> ok       [consume\ cnt > 1  \cnt--;]
22       ok  -> starting [consume\ cnt == 1 \cnt--;]
23       ok  -> ok       [produce\          \cnt++;]
24       ok  -> bad      [consume\ cnt == 0 \      ]
25       bad -> bad      [produce]
26       bad -> bad      [consume]
27     }
28   }
29 }
```

Listing B.3: MarQ specification and monitor for *producer-consumer*.

```
public aspect MarQProdCon {
  //Events
  private final int PRODUCE = 1;
  private final int CONSUME = 2;
  //Produce Counter
  private int counter = 0;
  //Monitor + Lock
  Monitor monitor;
  private Object LOCK = new Object();

  before() : //Handle Event: Produce
    call(* Queue.add(*))
    && cflow(execution(* SynchQueue.produce(*)))
  {
    synchronized(LOCK){
      check(monitor.step(PRODUCE, counter));
      counter++;
    }
  }
  before() : //Handle Event: Consume
    call(* Queue.poll())
    && cflow(execution(* SynchQueue.consume()))
  {
    synchronized(LOCK){
      check(monitor.step(CONSUME, counter));
      counter--;
     }
  }
  private void check(Verdict verdict){
    if(verdict==Verdict.FAILURE){
      System.err.println("Failed!");
      System.exit(1);
    }
  }
  //Create QEA Specification
  public void init(){
    QEABuilder b = new QEABuilder("ProdCon");
    int ticket = 1;
    b.addTransition(1, PRODUCE, new int[] {ticket},
      Assignment.increment(ticket),          1);
    b.addTransition(1, CONSUME, new int[] {ticket},
      Guard.varIsGreaterThanVal(ticket, 0),
      Assignment.decrement(ticket),          1);
    b.addTransition(1, CONSUME, new int[] {ticket},
      Guard.varIsEqualToIntVal(ticket, 0), 2);
    b.addFinalStates(1);
    monitor = MonitorFactory.create(b.make());
  }
  public MarQProdCon(){ init(); }
}
```

APPENDIX C

Other Works

C.1 Aspect-Oriented Design for Component Based Systems

[EFJ18] Antoine El-Hokayem, Yliès Falcone, and Mohamad Jaber. Modularizing behavioral and architectural crosscutting concerns in formal component-based systems - Application to the Behavior Interaction Priority framework. J. Log. Algebr. Meth. Program., 99:143-177, 2018.

[EFJ16] Antoine El-Hokayem, Yliès Falcone, and Mohamad Jaber. Modularizing crosscutting concerns in component-based systems. In Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016. Proceedings, pages 367-385, Berlin, Germany, 2016. Springer.

Abstract. We define a method to modularize crosscutting concerns in Component-Based Systems (CBSs) expressed using the Behavior Interaction Priority (BIP) framework. Our method is inspired from the Aspect Oriented Programming (AOP) paradigm which was initially conceived to support the separation of concerns during the development of monolithic systems. BIP has a formal operational semantics and makes a clear separation between architecture and behavior to allow for compositional and incremental design and analysis of systems. We distinguish local from global aspects. Local aspects model concerns at the component level and are used to refine the behavior of components. Global aspects model concerns at the architecture level, and hence refine communications (synchronization and data transfer) between components. We formalize local and global aspects as well as their composition and integration into a BIP system through rigorous transformation primitives. We present `AOP-BIP`, a tool for Aspect-Oriented Programming of BIP systems, demonstrate its use to modularize logging, security, and fault tolerance in a network protocol, and discuss its possible use in runtime verification of CBSs.

Relevance. Runtime verification can be seen as a *crosscutting concern*. In fact, many approaches listed in Section 2.2 use AspectJ for instrumenting Java programs. RV frameworks for CBSs, and particularly for BIP systems (RV-BIP [FJN+15] and RVMT-BIP [NFBB17]) have been already developed. They define specific transformations to instrument components and insert monitors as components in the new system (RV-BIP for sequential systems and RVMT-BIP for multithreaded systems). However, since runtime verification is a crosscutting concern, it is possible to instrument a system with aspects (both global and local) to generate necessary events for monitoring in a generic manner for CBSs. When applying RV for BIP, we note that we are dealing with a grey box system (as normally, RV works on black box systems). As such, it knows more information about the structure and parts of the system. This allows to 1) have formal guarantees on the behavior of monitors, 2) allow for a more expressive RV, as it has access to more information about the system, such as the result of static analysis

performed on the system. We define two types of monitors, at the global and local level. At the global level, it is possible to monitor interactions by intercepting their ports and variable accesses. Thus, by describing global pointcuts, we can generate events that are global, and synthesize global aspects that implement monitors. At the local level, it is possible to monitor the component state by using local pointcuts. Thus we can generate events that are local to the component. Using local aspects, we can then describe local monitors that are embedded in the component to check for local events. Certain properties however require information from multiple local monitors, thus it is impossible to handle the synchronization with our current approach. Directly writing monitors as aspects is not handled for these types of properties. However, it is possible for each local monitor to print out an event, and a separate monitoring mechanism to verify the entirety offline. In this work, we do not tackle the automatic synthesis of monitors from a specification. However, we show how `AOP-BIP` can be used to write manual monitors for specific properties. The monitors are integrated in the system as BIP components and interactions. Properties are specified on the Dala robot [FHC97], a large and realistic modular interactive system, which [FJN+15] uses for the RV approaches to CBSs.

C.2 Hierarchical Decentralized Monitoring of Business Processes

[HKB+18] Sylvain Hallé, Raphaël Khoury, Quentin Betti, Antoine El-Hokayem, and Yliès Falcone. Decentralized enforcement of document lifecycle constraints. Inf. Syst., 74(Part):117-135, 2018.

[HKEF16] Sylvain Hallé, Raphaël Khoury, Antoine El-Hokayem, and Yliès Falcone. Decentralized enforcement of artifact lifecycles. In 20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, pages 1-10. IEEE Computer Society, 2016.

Abstract. Artifact-centric workflows describe possible executions of a business process through constraints expressed from the point of view of the documents exchanged between principals. A sequence of manipulations is deemed valid as long as every document in the workflow follows its prescribed lifecycle at all steps of the process. So far, establishing that a given workflow complies with artifact lifecycles has mostly been done through static verification, or by assuming a centralized access to all artifacts where these constraints can be monitored and enforced. We present in this paper an alternate method of enforcing document lifecycles that requires neither static verification nor single-point access. Rather, the document itself is designed to carry fragments of its history, protected from tampering using hashing and public-key encryption. Any principal involved in the process can verify at any time that the history of a document complies with a given lifecycle. Moreover, the proposed system also enforces access permissions: not all actions are visible to all principals, and one can only modify and verify what one is allowed to observe. These concepts have been implemented in a software library called Artichoke, and empirically tested for performance and scalability.

Relevance. In this work, the document is a set of values that are modified by various principals (referred to as "peers"). Peers may belong to one or more groups. Groups represent access levels. In a business environment, groups could represent different department, or levels in the managerial hierarchy. Peers interact with the document using *actions*. Actions are classified into two groups: altering and observation actions. Altering actions modify a certain value in the document, for example, it can be seen as filling a certain field in a form. Observation actions, can be seen as actions that do not modify the document, but indicate some action related to the process. An action contains the peer performing the action, the group on behalf the peer is doing the action, and data (such as a label, field change, and data to modify the document). In this paper, actions follow a total-order, the sequence of actions forms the trace. As such, verifying the trace against a specification is similar to performing RV. However, since access levels may prevent certain peers from observing what other peers do, traces defer based on the access group(s) the peer belongs to. To prevent peers access to the actions performed and their data, encryption is used, to authenticate peers we assume a public-private key scheme is used. Based on the decryption result, each peer observes a different trace. To verify the trace, we utilize a set of automata, where each group has their own specification. Peers in multiple groups need to combine the specifications to perform additional checking that overlaps the multiple groups. As such we have the notion of *border actions*, which serve a similar purpose to references in decentralized specifications, and are used to synchronize the automata of the various groups.

Lists

List of Definitions, Propositions, Theorems, Corollaries, Lemmas, and Examples

List of Figures

List of Tables

Runtime Verification of Hierarchical Decentralized Specifications

Runtime Verification (RV) is a lightweight formal method which consists in verifying that a run of a system is correct with respect to a specification. The specification formalizes the behavior of the system typically using logics or finite-state machines. While RV comprehensively deals with monolithic systems, multiple challenges are presented when scaling existing approaches to decentralized systems, that is, systems with multiple components with no central observation point. We focus particularly on three challenges: managing partial information, separating monitor deployment from the monitoring process itself, and reasoning about decentralization in a modular and hierarchical way. We present the notion of a decentralized specification wherein multiple specifications are provided for separate parts of the system. Decentralized specifications provide various advantages such as modularity, and allowing for realistic monitor synthesis of the specifications. We also present a general monitoring algorithm for decentralized specifications, and a general datastructure to encode automata execution with partial observations. We develop the THEMIS tool, which provides a platform for designing decentralized monitoring algorithms, metrics for algorithms, and simulation to better understand the algorithms, and design reproducible experiments.

We illustrate the approach with two applications. First, we use decentralized specifications to perform a worst-case analysis, adapt, compare, and simulate three existing decentralized monitoring algorithms on both a real example of a user interface, and randomly generated traces and specifications. Second, we use decentralized specifications to check various specifications in a smart apartment: behavioral correctness of the apartment sensors, detection of specific user activities (known as activities of daily living), and composition of properties of the previous types.

Furthermore, we elaborate on utilizing decentralized specifications for the decentralized online monitoring of multithreaded programs. We first expand on the limitations of existing tools and approaches when meeting the challenges introduced by concurrency and ensure that concurrency needs to be taken into account by considering partial orders in traces.We detail the description of such concurrency areas in a single program execution, and provide a general approach which allows re-using existing RV techniques. In our setting, monitors are deployed within specific threads, and only exchange information upon reaching synchronization regions defined by the program itself. By using the existing synchronization, we reduce additional overhead and interference to synchronize at the cost of adding a delay to determine the verdict.

Abstract

Vérification à l'Exécution de Spécifications Décentralisées Hiérarchiques

La vérification à l'exécution est une méthode formelle légère qui consiste à vérifier qu'une exécution d'un système est correcte par rapport à une spécification. La spécification exprime de manière rigoureuse le comportement attendu du système, en utilisant généralement des formalismes basés sur la logique ou les machines à états finies. Alors que la verification a l'éxecution traite les systèmes monolithiques de manière exhaustive, plusieurs difficultés se présentent lors de l'application des techniques existantes à des systèmes décentralisés, c-à-d. des systèmes avec plusieurs composants sans point d'observation central. Dans cette thèse, nous nous concentrons particulièrement sur trois problèmes : la gestion de l'information partielle, la séparation du déploiement des moniteurs du processus de vérification lui-même et le raisonnement sur la décentralisation de manière modulaire et hiérarchique. Nous nous concentrons sur la notion de spécification décentralisée dans laquelle plusieurs spécifications sont fournies pour des parties distinctes du système. Utiliser une spécification décentralisée a divers avantages tels que permettre une synthèse de moniteurs à partir des spécifications complexes et la possibilité de modulariser les spécifications. Nous présentons également un algorithme de vérification général pour les spécifications décentralisées et une structure de données pour représenter l'exécution d'un automate avec observations partielles. Nous développons l'outil THEMIS, qui fournit une plateforme pour concevoir des algorithmes de vérification décentralisée, des mesures pour les algorithmes, une simulation et des expérimentations reproductibles pour mieux comprendre les algorithmes.

Nous illustrons notre approche avec diverses applications. Premièrement, nous utilisons des spécifications décentralisées pour munir une analyse de pire cas, adapter, comparer et simuler trois algorithmes de vérification décentralisée existants dans deux scénarios: l'interface graphique Chiron, et des traces et spécifications générées aléatoirement. Deuxièmement, nous utilisons des spécifications décentralisées pour vérifier diverses propriétés dans un appartement intelligent: correction du comportement des capteurs de l'appartement, détection d'activité spécifiques de l'utilisateur (Activities of Daily Living, ADL) et composition de spécifications des deux catégories précédentes.

En outre, nous élaborons sur l'utilisation de spécifications décentralisées pour la vérification décentralisée pendant l'exécution de programmes parallélisés. Nous commençons par discuter les limitations des approches et des outils existants lorsque les difficultés introduites par le parallélisme sont rencontrées. Nous détaillons la description de zones de parallélisme d'une unique exécution d'un programme et décrivons une approche générale qui permet de réutiliser des techniques de verification à l'éxécution existantes. Dans notre configuration, les moniteurs sont déployés dans des fils d'exécution spécifiques et échangent de l'information uniquement lorsque des points de synchronisation définis par le programme lui-même sont atteints. En utilisant les points de synchronisation existants, notre approche réduit les interférences et surcoûts résultant de la synchronisation, au prix d'un retard pour déterminer le verdict.

Résumé