Tracing Distributed Component-Based Systems, a Brief Overview

Yliès Falcone[1], Hosein Nazarpour[2], Mohamad Jaber[3], Marius Bozga[2], and Saddek Bensalem[2]

[1] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr
[2] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, Verimag, 38000 Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr
[3] American University of Beirut, Beirut, Lebanon mj54@aub.edu.lb

Abstract. We overview a framework for tracing asynchronous distributed component-based systems with multiparty interactions managed by distributed schedulers. Neither the global state nor the total ordering of the system events is available at runtime. We instrument the system to retrieve local events from the local traces of the schedulers. Local events are sent to a global observer which reconstructs on-the-fly the global traces that are compatible with the local traces, in a concurrency-preserving and communication-delay insensitive fashion. The global traces are represented as an original lattice over partial states, such that any path of the lattice projected on a scheduler represents the corresponding local partial trace according to that scheduler (soundness), and all possible global traces of the system are recorded (completeness).

1 Introduction

Component-based design consists in constructing complex systems using pre-defined components which are atomic entities with some actions and interfaces. The behavior of a component-based system with multiparty interactions (CBS) depends on the behavior of each component as well as the interactions between the components. A multiparty interaction is a set of simultaneously executed actions of components [9]. To allow for the concurrent execution of non-conflicting interactions (with no shared component), interactions are distributed on several schedulers. Schedulers and components interact (by exchanging messages) to ensure the correct execution of multiparty interactions [10].

Problem statement. Our goal is to conduct runtime verification [17,4,5] of a distributed CBS against properties referring to the global states of the system. This implies, in particular, that properties cannot be projected and checked on individual components. We use neither a global clock nor a shared memory. On the one hand, this makes the execution of the system more dynamic and parallel by avoiding synchronization to take global snapshots [11], which would go against the distribution of the system. On the other hand, it complicates the monitoring problem because no component can be aware of the global trace. Since the execution of interactions is based on sending/receiving messages, communication is asynchronous, and delays in the reception of messages are

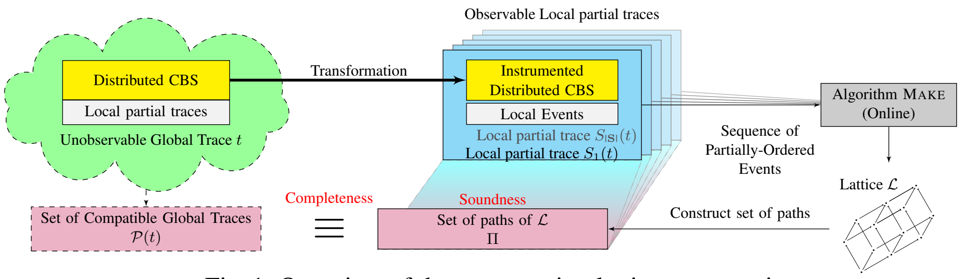
Distributed CBS
Local partial traces
Unobservable Global Trace t
Transformation
Observable Local partial traces
Instrumented Distributed CBS
Local Events
Local partial trace $S_{|S|}(t)$
Local partial trace $S_1(t)$
Algorithm MAKE (Online)
Sequence of Partially-Ordered Events
Lattice $\mathcal{L}$
Set of Compatible Global Traces $\mathcal{P}(t)$
Completeness
$\equiv$
Soundness
Set of paths of $\mathcal{L}$ Π
Construct set of paths

Fig. 1: Overview of the computation lattice construction

inevitable. Moreover, the absence of ordering between the execution of the interactions in different schedulers makes the actual execution trace unobservable. To allow for the RV of distributed CBSs, we instrument them so as to *trace* and *reconstruct* their global behavior in a concurrency-preserving and communication-delay insensitive fashion. We shall leverage the component-based nature of the system under scrutiny and account for the existing causalities in the execution of distributed CBSs.

Approach overview (Fig. 1). We define a *monitoring hypothesis* by defining an abstract semantic model that encompasses a variety of distributed CBSs. Our model relies only on partial-state semantics of CBSs, in terms of (1) Labeled Transition Systems with unobservable internal actions and observable actions and (2) a set of schedulers defining multiparty interactions (i.e, barriers) on sets of observable actions from different components. Our model is, however, not bound to any CBS framework. Due to the parallel executions in schedulers (i) events (i.e., actions changing the state of the system) are not totally ordered, and (ii) the actual global trace is unobservable. Although each scheduler is aware of its local behavior (local partial trace) and its local events, to evaluate the global behavior, we need the set of possible orderings of the events of all schedulers, that is, the set of compatible global traces. In our setting, schedulers do not communicate together and only communicate with their associated components. Indeed, only the shared components involved in several multiparty interactions managed by different schedulers make the actions of different schedulers causally related. In other words, the executions of two actions managed by two schedulers and involving a shared component are causally related, because each execution requires the termination of the other execution to release the shared component. To account for existing causality, we (i) employ vector clocks to define the ordering of events (ii) instrument the system to compose each scheduler with a controller to compute the correct vector clock of each generated event (iii) compose each shared component with a controller to resolve the causality, and (iv) introduce a procedure to reconstruct a set of compatible global traces that could possibly have happened with the received events. We represent the set of compatible global traces using a computation lattice tailored for CBSs. Such a computation lattice consists of a set of partially connected nodes. Created nodes are partial states and become global states during monitoring. Any path of the lattice projected on a scheduler represents the corresponding local partial trace according to that scheduler (soundness). All possible global traces are exactly recorded (completeness).

An extended version of this paper with more details and proofs is available in [28].

2 Semantics of Distributed CBSs with Multiparty Interactions

We describe a general semantics of CBSs, where neither the exact model nor the behavior of the system are known. How the behaviors of the components and the schedulers are obtained is irrelevant for monitoring. Inspiring from conformance-testing theory [30], we refer to this as the *monitoring hypothesis*. *Components* are in the set $\mathbf{B} = \{B_1, \ldots, B_{|\mathbf{B}|}\}$ and *schedulers* in $\mathbf{S} = \{S_1, \ldots, S_{|\mathbf{S}|}\}$. Each component B_i is endowed with a set of actions Act_i. Joint actions of components, aka multiparty interactions, involve several components. An interaction is a non-empty subset of $\cup_{i=1}^{|\mathbf{B}|} Act_i$; Int denotes the set of interactions. At most one action of each component is involved in an interaction: $\forall a \in Int\,.|a \cap Act_i| \leq 1$. Moreover, each component B_i has internal actions modeled as a unique action β_i. Schedulers coordinate the execution of interactions and ensure that each multiparty interaction is jointly executed. We describe the behavior of components, schedulers, and their composition. Component B (i) has actions in set Act_B which are possibly shared with some of the other components, (ii) has an internal action $\beta_B \notin Act_B$ which models internal computations of component B, (iii) the state of B is busy (unknown) while it is performing its internal action, and (iv) alternates moving from a ready state to a busy state (after executing an action), and vice-versa (after executing an internal action). Intuitively, when a scheduler executes an interaction, it triggers the execution of the associated actions on the involved components, and updates its internal vector clock. Moreover, when a component executes an internal action, it triggers the execution of the corresponding action on the associated schedulers and also sends the updated state of the component to the associated schedulers, the component sends a message including its current state to the schedulers. Note, by construction, schedulers are always ready to receive such a state update.

Global behavior. The global execution of the system can be described as the parallel execution of interactions managed by the schedulers. Components execute independently according to the decisions of schedulers. Any executed global action contains at most one interaction involving each component. Whenever an interaction managed by a scheduler is executed, this scheduler and all components involved in this interaction must be ready to execute it. Internal actions are executed whenever the corresponding components are ready to execute them. Moreover, schedulers are aware of internal actions of components in their scope. The awareness of internal actions of a component results in transferring the updated state of the component to the schedulers. The components and schedulers not involved in an interaction remain in the same state.

Traces of distributed CBSs with multiparty interactions. A trace is a sequence of states traversed by the system at runtime, from some initial state and following the transition relation of the LTS. For clarity and our monitoring purposes, the states of schedulers are irrelevant in the trace, and thus we restrict the system states to those of the components. A *partial trace* has partial states where at least one component is busy (with internal computation). Although the partial trace of the system exists, it is not observable because it would require a perfect observer having simultaneous access to the states of components. Introducing such an observer requires to synchronize all components and defeat the purpose of building a distributed system. Instead, we shall instrument the system to observe the sequence of states through schedulers.

3 Computation Lattice of Distributed Component-Based Systems

We briefly overview the on-the-fly construction of a computation lattice representing the possible global traces compatible with the local partial traces (Algorithm MAKE). Since schedulers do not interact directly, the execution of an interaction by one scheduler is concurrent with the execution of all interactions by other schedulers.

System instrumentation. To retrieve the actual ordering and obtain the local partial traces, one needs to instrument the system by adding controllers to the schedulers and to the shared components. Each time a scheduler executes an interaction, the involved components are notified by the scheduler to execute their corresponding actions. Moreover, the controller of the scheduler updates its local clock and notifies the controller of the shared components involved in the interaction by sending its vector clock (be stored in the controller). Whenever a shared component executes its internal action β, schedulers that have the shared component in their scope are notified by receiving the updated state. Moreover, the vector clock stored in the controller of the shared component is sent to the controller of the associated schedulers. Consequently, schedulers having a shared component in their scope exchange their vector clocks through the shared component. Intuitively, for scheduler S_j, the execution of an interaction (labeled by a vector clock), or notification by the internal action of a component which the execution of its latest action has been managed by scheduler S_j, is defined as an *event* of scheduler S_j.

Extended computation lattice (overview). Intuitively, an extended computation lattice (lattice for short) consists of a set of partially connected nodes, where each node is a pair, made of a state of the system and a vector clock. A system state consists in the states of all components. The computation lattice is represented implicitly using vector clocks. The construction mainly performs the two following operations: (i) *creations of new nodes* and (ii) *updates* of existing nodes in the lattice. The observer, which is charge of building the lattice, receives two sorts of events: (1) events related to the execution of an interaction in Int, referred to as *action events*, and (2) events related to internal actions referred to as *update events*. (Recall that internal actions carry the state of the component which has performed the action – the state is sent to the observer by the controller that is notified of this action). Action events lead to the creation of new nodes, while update events complete the information in the nodes of the lattice related to the state of the component related to the event. Since the received events are not totally ordered (because of communication delay), we construct the computation lattice based on the vector clocks attached to the received events. Note, we assume that the events received from a scheduler are totally ordered.

Intermediate operations. We consider a lattice $\mathcal{L}$. A newly received event either modifies $\mathcal{L}$ or is kept in a queue for later. Action events extend $\mathcal{L}$ and update events update the existing nodes of $\mathcal{L}$ by adding the missing state information into them. By extending the lattice with new nodes, one needs to further complete the lattice by computing joints of created nodes with existing ones so as to complete the set of possible global traces.

Receiving an action or update event might not always lead to extending or updating the current computation lattice. Due to communication delay, an event that happened before another event might be received later by the observer. It is necessary for the construction of the lattice to use events in a specific order. Such events must be kept in a waiting queue to be used later.

Properties guaranteed by lattice construction The first property states that the ordering of the events does not affect the lattice. The second property is *correctness* (soundness), meaning that the resulting computation lattice encodes a set of the sequences of global states, s.t. each sequence represents a global trace of the system. The third property is *completeness*, meaning that for any sequence of events, we construct a lattice whose set of paths consists of all the compatible global traces.

Remark 1 (Garbage collection). For performance reasons, a garbage collector regularly removes non-frontier nodes from the lattice and checks for the existence of events that can be treated. This ensures that the lattice size remains almost constant at runtime, while maintaining soundness and completeness.

4 Evaluation

We implemented the RVDIST tool [1] to evaluate our approach on a robot navigation system and the two-phase commit protocol. We consider metrics related to lattice construction. Our experiment show that the size of the constructed lattice remain constant at runtime when executed on two systems generating a few thousands of events. Moreover, the size of the constructed lattice and the number of paths of the lattices is *inversely proportional* to the number of shared components.

5 Related Work

This paper extends our runtime verification frameworks for component-based systems, in the sequential [19] and multi-threaded [27] settings. Regarding distributed systems, a lot of tracing and debugging frameworks have been defined for instance in the system community, however, generally with an offline, less general and less formal approach (e.g., no correctness guarantees). Henceforth, we rather focus in this section on the formal approaches to monitoring distributed systems (see [21] for an overview). The approach in [8] presents an algorithm for decentralized monitoring LTL formulas for synchronous distributed systems. We rather target asynchronous distributed CBSs with a partial-state semantics, where the global state of the system is unavailable at runtime. Hence, instead of having a global trace at runtime, we deal with the compatible partial traces which could have happened at runtime. The approach in [7] presents a framework for detecting and analyzing synchronous distributed systems faults in a centralized manner using local LTL properties that require only the local traces. In our setting, the global trace allows monitoring global properties that cannot be projected and checked on individual components/schedulers. Thus, local traces cannot be directly used for verifying properties. In [29], the authors design a method for monitoring safety properties in distributed systems relying on existing process communication. Compared to [29], our algorithm is sound as we reconstruct the behavior of the distributed system based on all possible partial traces. In our work, each trace could have happened as the actual trace of the system, and could have generated the same events. The approach in [23] shows that the trace monitoring problem with automata is NP-complete in the number of concurrent processes. The approaches in [12,22,2], generalized in [26], monitor temporal requirements over distributed processes where local monitors are attached to processes and circulate tokens. Interestingly, the approach [26] is decentralized (as is [8]). Compared to [23,26] which use simpler computational models, our approach is tailored to and leverages CBS where traces are defined over partial states. Also close to our work is [24] for the monitoring of violations of invariants using knowledge. Model-checking the system allows to pre-calculate the states where a violation can be reported by a process alone. When communication (i.e., more knowledge) is needed between processes, synchronizations are added. The focus of [24] is to minimize communication induced by synchronization while our approach does not impose synchronization to the system. The approach in [3] introduces a component-based model of Apache ZooKeeper for testing using a model-checker. It describes code that maintains an event graph similar to our lattice construction. However, [3] is specific to Zookeeper, whereas our method can be applied to any distributed system.

6 Conclusions and Perspectives

We efficiently trace distributed CBSs where interactions are partitioned over distributed schedulers. Our technique (i) transforms the system to generate events associated with the partial traces of schedulers, (ii) synthesizes a centralized observer which collects the local events of all schedulers (iii) reconstructs on-the-fly the possible orderings of the received events which form a computation lattice. We showed that the constructed lattice encodes exactly the set of compatible global traces: each could have occurred as the actual execution trace of the system. The experimental results show that, even for long execution traces, the size of the constructed lattice is constant.

Tracing distributed CBSs in a sound and complete allows us to tackle the problem of the runtime verification of distributed CBSs. We plan to address this problem in the future as well as the following ones:

- (i) define specification formalisms tailored to our model of CBSs and study their monitorability [16];
- (ii) decentralize observers/monitors according to the system architecture by using decentralized runtime verification frameworks [6,15,13];
- (iii) adapt techniques for runtime enforcement [20] of sequential CBSs [18] to the distributed setting;
- (iv) use heteregoneous communication primitives (synchronous and asynchronous) [25] for facilitating the implementation of optimized monitors;
- (v) leverage aspect-oriented programming on CBSs [14] to define source-to-source transformations to inject runtime verification monitors.

Acknowledgement. This article is based upon work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology). The authors acknowledge the support of the Brain-IoT project (`www.brain-iot.eu`). This work was supported by the French national program “Programme Investissements d’Avenir IRT Nanoelec" (ANR-10-AIRT-05).

References

1. RVDist: Runtime Verification for Distributed Component-Based Systems. Available at `https://gitlab.inria.fr/monitoring/rv-dist-pub`
2. Agarwal, A., Garg, V.K., Ogale, V.A.: Modeling and analyzing periodic distributed computations. In: Int. Symp. on Stabilization, Safety, and Security of Distributed Systems. pp. 191–205 (2010)
3. Artho, C., Gros, Q., Rousset, G., Banzai, K., Ma, L., Kitamura, T., Hagiya, M., Tanabe, Y., Yamamoto, M.: Model-based API testing of apache zookeeper. In: 2017 IEEE Int. Conf. on Software Testing, Verification and Validation. pp. 288–298 (2017)
4. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: 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)
5. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification - Introductory and Advanced Topics, Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer (2018)
6. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods in System Design 48(1-2), 46–93 (2016)
7. Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: ASWEC'06, Australian Software Engineering Conf. p. 243–252. IEEE (2006)
8. Bauer, A.K., Falcone, Y.: Decentralised LTL monitoring. In: 18th Int. Symp. on Formal Methods. pp. 85–100 (2012)
9. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: Int. Conf. on Concurrency Theory. pp. 508–522. Springer (2008)
10. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distributed Computing 25(5), 383–409 (2012)
11. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems (TOCS) 3(1), 63–75 (1985)
12. Cooper, R., Marzullo, K.: Consistent detection of global predicates. In: Workshop on Parallel and Distributed Debugging, Santa Cruz, California. pp. 167–174 (1991)
13. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Bultan, T., Sen, K. (eds.) Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017. pp. 125–135. ACM (2017)
14. El-Hokayem, A., Falcone, Y., Jaber, M.: 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)
15. Falcone, Y., Cornebize, T., Fernandez, J.: Efficient and generalized decentralized monitoring of regular languages. In: Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 Int. Conf., FORTE 2014, Held as Part of the 9th Int. Federated Conf. on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings. pp. 66–83 (2014)
16. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? STTT 14(3), 349–382 (2012), `https://doi.org/10.1007/s10009-011-0196-8`
17. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013)

18. Falcone, Y., Jaber, M.: Fully automated runtime enforcement of component-based systems with formal and sound recovery. STTT 19(3), 341–365 (2017)
19. Falcone, Y., Jaber, M., Nguyen, T., Bozga, M., Bensalem, S.: 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)
20. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design 38(3), 223–262 (2011)
21. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Lectures on Runtime Verification, pp. 176–210 (2018)
22. Fromentin, E., Jard, C., Jourdan, G., Raynal, M.: On-the-fly analysis of distributed computations. Inf. Process. Lett. 54(5), 267–274 (1995)
23. Genon, A., Massart, T., Meuter, C.: Monitoring distributed controllers. In: 14th Int. Symp. on Formal Methods. pp. 557–572 (2006)
24. Graf, S., Peled, D.A., Quinton, S.: Monitoring distributed systems using knowledge. In: Formal Techniques for Distributed Systems. pp. 183–197 (2011)
25. Kobeissi, S., Utayim, A., Jaber, M., Falcone, Y.: Facilitating the implementation of distributed systems with heterogeneous interactions. In: Furia, C.A., Winter, K. (eds.) Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11023, pp. 255–274. Springer (2018), https://doi.org/10.1007/978-3-319-98938-9_15
26. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: Int. Parallel and Distributed Processing Symp.,. pp. 494–503 (2015)
27. Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M.: Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation. Formal Asp. Comput. 29(6), 951–986 (2017), https://doi.org/10.1007/s00165-017-0422-6
28. Nazarpour, H., Falcone, Y., Jaber, M., Bensalem, S., Bozga, M.: Monitoring distributed component-based systems. CoRR abs/1705.05242 (2017)
29. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: 26th Int. Conf. on Software Engineering. pp. 418–427. IEEE (2004)
30. Tretmans, J.: A formal approach to conformance testing. In: Sixth Int. Workshop on Protocol Test systems, IFIP TC6/WG6.1. pp. 257–276 (1993)