HAL
open science

Who is to Blame? Runtime Verification of Distributed Objects with Active Monitors

Wolfgang Ahrendt, Ludovic Henrio, Wytse Oortwijn

To cite this version:

Wolfgang Ahrendt, Ludovic Henrio, Wytse Oortwijn. Who is to Blame? Runtime Verification of Distributed Objects with Active Monitors. Electronic Proceedings in Theoretical Computer Science, 2019, 302, pp.32-46. 10.4204/EPTCS.302.3 . hal-02303148

HAL Id: hal-02303148

https://hal.science/hal-02303148v1

Submitted on 2 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Who is to Blame? – Runtime Verification of Distributed Objects with Active Monitors

Wolfgang Ahrendt
Chalmers University of Technology
Gothenburg, Sweden
ahrendt@chalmers.se

Ludovic Henrio
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP
Lyon, France
ludovic.henrio@ens-lyon.fr

Wytse Oortwijn
University of Twente
Enschede, the Netherlands
w.h.m.oortwijn@utwente.nl

Since distributed software systems are ubiquitous, their correct functioning is crucially important. Static verification is possible in principle, but requires high expertise and effort which is not feasible in many eco-systems. Runtime verification can serve as a lean alternative, where monitoring mechanisms are automatically generated from property specifications, to check compliance at runtime. This paper contributes a practical solution for powerful and flexible runtime verification of distributed, object-oriented applications, via a combination of the runtime verification tool LARVA and the active object framework PROACTIVE. Even if LARVA supports in itself only the generation of local, sequential monitors, we empower LARVA for distributed monitoring by connecting monitors with active objects, turning them into active, communicating monitors. We discuss how this allows for a variety of monitoring architectures. Further, we show how property specifications, and thereby the generated monitors, provide a model that splits the blame between the local object and its environment. While LARVA itself focuses on monitoring of control-oriented properties, we use the LARVA front-end STARVOORS to also capture data-oriented (pre/post) properties in the distributed monitoring. We demonstrate this approach to distributed runtime verification with a case study, a distributed key/value store.

1 Introduction

The days of stand-alone software applications are largely over. *Cloud solutions* and *mobile applications* are prominent instances of a general development towards ever more distributed computing. Distributed software is already ubiquitous, and will only grow from here. At the same time, the overwhelming combinatorial complexity of possible interactions and interleavings makes distributed software systems particularly prone to unforeseen, unintended behaviour of multiple criticality. This makes validation efforts even more important than in the stand-alone case. Distributed computational scenarios pose enormous challenges to analysis and verification, however. There exist many approaches in the literature, partly supported by tools. But in general, sufficiently powerful static verification approaches tend to be very heavy.

There is a recent trend towards more lightweight formal methods, which are easier to exploit but give limited guarantees. One of them is *runtime verification*, which combines full precision of the execution model (even including the real deployment environment) with full automation. On the other hand, it only ever judges the observed runs, and cannot judge alternative and future runs. Another challenge in runtime verification is the computational overhead of monitoring the running system which can be prohibitive in certain settings.

D. Ancona and G. Pace (Eds.): Verification
of Objects at RunTime EXecution 2018 (VORTEX 2018)
EPTCS 302, 2019, pp. 32–46, doi:10.4204/EPTCS.302.3

© W. Ahrendt, L. Henrio & W. Oortwijn
This work is licensed under the
Creative Commons Attribution License.

This paper offers *practical solutions* for flexible runtime verification of distributed, object-oriented applications. Contemporary runtime verification approaches allow users to specify properties on a high level, and hide the details of how the actual monitoring is performed. The active object design pattern allows users to program distributed nodes by writing seemingly sequential code, and hide the details of how the proper communication and coordination between different machines is performed. By combining these two principles, we achieve high-level monitor descriptions and high-level distributed programming at once. Another aim is to allow for a *variety of monitoring architectures* in a natural manner, such that the user can tailor the monitoring architecture to the characteristics of the monitored application and of the underlying network. We achieve this not only by monitoring active object *applications*, but also by using the active object paradigm *in the monitors themselves*. Another contribution is the integration of *blame-shifting* into the monitoring, in the spirit of assume-guarantee reasoning. The specification of a node states for every failure whether it is blamed on the monitored node or on its environment. The implementation of the node has to ensure the absence of any failure that is blamed on the node, under the assumption that no failure occurs which is blamed on the node's environment. This supports the localisation of failure while limiting the communication load in the monitoring.

Concretely, we made a connection between the runtime verification tool LARVA [11] and the active object framework PROACTIVE [1]. LARVA (Logical Automata for Runtime Verification and Analysis) is a tool for monitoring the execution and verifying at runtime the correct behaviour of programs, but it is only adapted to a sequential setting. It could be used to monitor a distributed application but only from a centralised point of view [10], limiting the scalability of the approach. In this work we investigate the usage of LARVA to perform *distributed monitoring* of applications. LARVA generates a set of monitors for several entities of the observed system and we use PROACTIVE to coordinate the different monitors. PROACTIVE is an active object library that integrates well with LARVA because it has no specific syntax for parallelism and distribution: coordination between active objects is performed automatically by the middleware while the programmer only writes standard (sequential) Java code. This way standard LARVA monitors can be generated and PROACTIVE coordinates them in a natural manner. Further, while LARVA itself focuses on monitoring of control-oriented properties, we use the STARVOORS (STAtic and Runtime Verification of Object-ORiented Software) front-end for LARVA [8], to also support the monitoring of data-oriented (pre/post) properties.

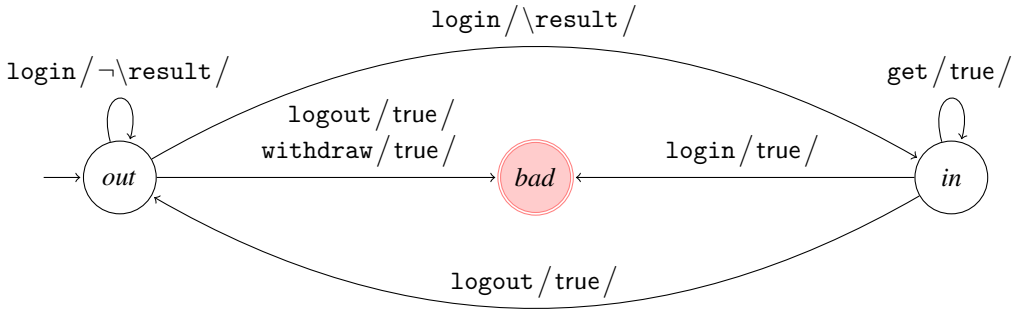
The paper is structured as follows. Section 2 gives prerequisites on LARVA and PROACTIVE. Section 3 discusses the connection of LARVA with PROACTIVE to enable distributed monitoring. In particular, our solution allows LARVA monitors to be active objects and enables a variety of monitoring configurations. The notion of blame-shifting and the monitoring of data-oriented properties are also discussed. Section 4 applies our approach on a case study: a distributed key/value store. Section 5 compares our work to related research and Section 6 concludes.

2 Background

Rather than writing our tooling from scratch, we combined the LARVA runtime verifier with the PROACTIVE programming platform for writing distributed Java applications. This section introduces both LARVA and PROACTIVE and outlines some of their aspects that are relevant for the remainder of the paper.

```
1 public class Bank {
2   public bool login(String code) { ··· }
3   public void logout(String code) { ··· }
4   public void withdraw(String code, double m) { ··· }
5 }
```

(a) The Java implementation of a small banking system.



(b) The *DATE* property that is used for runtime verification.

Figure 1: An example of runtime verification with LARVA.

2.1 Runtime verification with LARVA

LARVA [11] is a runtime verification platform that allows to verify control-flow properties of Java programs, written in an automata-based specification language called *DATE* (*D*ynamic *A*utomata with *T*imers and *E*vents). LARVA generates runtime monitors as Java code out of the automata descriptions of the input properties, and links these monitors to the Java system by using ASPECTJ.

We illustrate the use of LARVA with a small example, a banking system with access control, where users may login, logout, and withdraw money. Figure 1 sketches a Bank class (1a), and the *DATE* property (1b) that users must first login before being able to logout or withdraw, and logged-in users should not log in again.

The *DATE* specification in Figure 1b consists of three states: (i) the initial state *out*, describing the logged-out state of the program; (ii) the normal state *in* that models the logged-in program state; and (iii) the bad state *bad* that models the erroneous state as result of runtime violations. Moreover, *DATE* transitions are labelled by a triple of the form $e \,/\, c \,/\, a$, where: e is an *event* that is connected to method invocations or executions in the program and triggers the transition; c is a *condition* that must hold for the transition to be taken; and a is an *action*, a code snippet that is executed when the transition is taken. In general, the condition- and action components of transitions may access or update global variables. In fact, since the LARVA compiler translates *DATE* specifications to Java, the c and a components may contain arbitrary Java snippets.

The transitions in Figure 1b use three different events, namely `login`, `logout` and `withdraw`, which correspond to invoking the corresponding methods in the implementation. The `\result` placeholder in the login transitions is bound to the return value of `login`, so the self-loop in *out* is taken if the login in the program appeared unsuccessful. The other transitions have true as their condition; these transitions are taken unconditionally when triggered. Furthermore, none of the transitions have an associated action,

```
1 Bank b = (Bank)PAActiveObject.newActive(Bank.class.getName(), null);
2 Account a = b.createAccount(userName);
3 b.deposit(a, 100);
4 Balance m = b.getBalance(a);
5 int i = m.getValue();
```

Figure 2: A simple PROACTIVE example.

so they are left blank.

LARVA automatically generates monitors from *DATE* property specifications (via ASPECTJ). We therefore often identify, in the discussion, a *DATE* property with the monitor generated from it.

2.2 The PROACTIVE programming platform

PROACTIVE [1, 6] is a Java library for building concurrent and distributed software that implements the *active objects* [5] design pattern. The active objects paradigm—largely inspired by Actors [19]—simplifies distributed programming by abstracting concurrency and locality from the programmer. Similar to actors, the primitive unit of computation are objects that have their own thread of control. Active objects may have private state and public methods, and threads may communicate by calling the methods on remote objects. Method invocations are decoupled from method execution, which simplifies the design of distributed systems. By invoking a method, the calling thread pushes a request message into the queue of the callee thread, which will return a *future* and will eventually process the request by executing the method. This asynchronous construction allows threads to continue working while waiting for remote calls to finish.

The PROACTIVE platform implements the active objects paradigm for Java. In particular, PROACTIVE allows to register Java objects as being "active", which exposes the public methods of these objects to other active objects, which may be hosted on different JVMs, possibly located on different machines. Under the hood, when activating a Java object, PROACTIVE spawns a worker thread for the object and constructs a proxy that handles all (network) communication the object gets involved in (method calls on active objects are handled via RMI). However, PROACTIVE hides all these technical details from their users; handling active objects has the same look-and-feel as handling ordinary Java objects.

Figure 2 shows a code excerpt illustrating a simple use of PROACTIVE. The first line creates an active object b, thus all subsequent uses of object b are active object requests that will be handled asynchronously by the bank object. Each of these call returns a future, stored locally in a and m. The code shown in the figure does not use locally the future stored in a, but it transmits the future to the bank. The invoker does not need to wait for the future resolution to send it but the bank will probably synchronise on the account creation. It is worth noting that communication between active objects is FIFO, ensuring that all the operations will be handled by the bank in the order of the program. The balance object m receives a future at line 4, and the method invocation at line 5 will be blocked until the balance is obtained (returned by the bank object).

The support for runtime monitoring is very limited in PROACTIVE. Entry points are provided to intercept inter-object communications. They are used for example in an execution visualizer, but any precise monitoring of runtime properties has to be done by hand in PROACTIVE (prior to this work).

```
1  public class Server {
2    public static void main(String[] args) {
3      Bank b = (Bank)PAActiveObject.newActive(Bank.class.getName(), null);
4      PAActiveObject.registerByName(b, args[0]);
5    }
6  }
7
8  public class Client {
9    private static void startInterface(Bank b) { ··· }
10   public static void main(String[] args) {
11     Bank b = (Bank)PAActiveObject.lookupActive(
12                    Bank.class.getName(), args[0]);
13     Client.startInterface(b);
14   }
15 }
```

Figure 3: A distributed implementation of the Banking example, using PROACTIVE.

3 Monitoring Distributed Objects

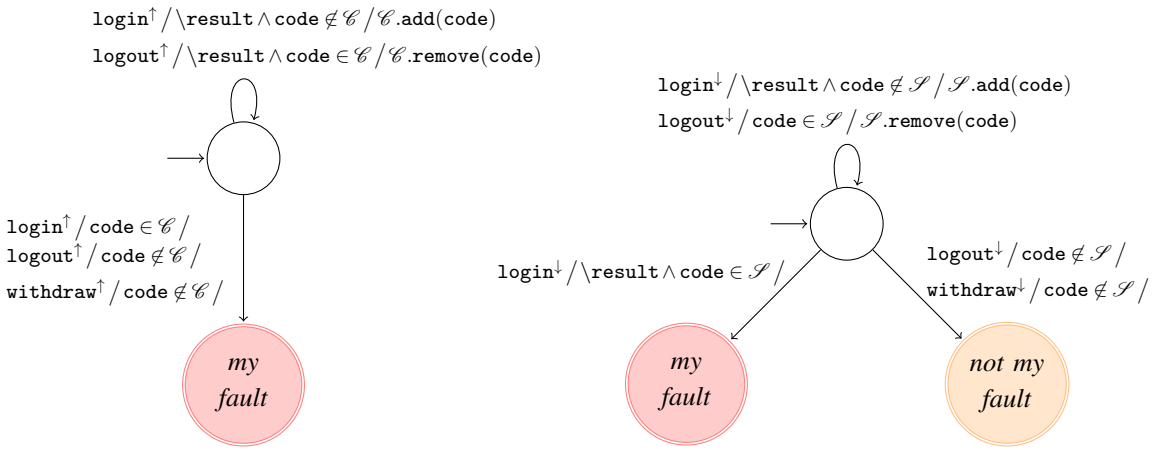
In the following, we describe how we employ a runtime verification framework for *sequential* applications (LARVA) to generate *distributed* monitors for *distributed* (PROACTIVE) objects. We discuss how distributed monitoring is achieved by letting LARVA distinguish between method *invocations* and method *executions* on active objects. This section also explains how different monitoring configurations are realised, including orchestrated, centralised and choreography based monitoring, by letting distributed LARVA monitors communicate using active objects. Finally, a notion of *blame-shifting* is discussed that is inspired by static assume-guarantee (AG) style reasoning.

Running example.

Throughout this section, we explain and discuss the combination of LARVA and PROACTIVE with a distributed version of the banking example discussed in Section 2.1, implemented using PROACTIVE. An excerpt of the implementation is given in Figure 3, where we reuse the Bank class from Figure 1. In fact, this implementation consists of two separate programs, namely: (i) a server that creates and hosts an active object for the bank, and (ii) a client that connects to this active object in order to login, logout, or withdraw money. The client and the server are intended to be instantiated on different JVMs. The only active object, i.e. the only remotely accessible object, is the bank.

By running the server program, PROACTIVE constructs a proxy for the active object in the API-call in line 3, and assigns a worker thread to this active object. Any call to methods on b are actually translated to messages sent to this worker, but these details are neatly hidden by PROACTIVE. The API call at line 4 exposes the new active object b to the outside world, by assigning a network name to it, specified as the argument $args[0]$ in the console. Note that, after executing line 4, the thread associated to the active object b keeps on running and is ready to process incoming method invocations.

Another machine might run the Client program and connect to the active object hosted by the server (identified by the network name given as an input argument, $args[0]$). The client program will connect



(a) Runtime monitor for the client

(b) Runtime monitor for the server

Figure 4: Runtime monitors for the distributed banking example. Edges with multiple labels abbreviate multiple edges, each with a single label.

to the specified active object by the API-call in line 11 and line 12. Observe that the objects *b* returned in line 3 and line 11 by PROACTIVE are typed as Bank objects: one may use these as ordinary Java objects, even if the object *b* returned in line 11 is actually a *stub*—a proxy generated by PROACTIVE that translates all method invocations on *b* to network messages (actually to RMI communication) to the JVM that physically hosts the bank active object (here the JVM that runs the Server program). After connecting to the Bank active object, the client may display some user-interface via `startInterface` on line 13, to allow user interaction with the bank.

3.1 Monitoring distributed objects with LARVA

Even though LARVA is not designed for runtime verification of distributed programs, PROACTIVE and LARVA make a good match nonetheless, because:

1. Although each JVM may host several active objects, each active object only has a single worker thread. Therefore, operations on active objects are essentially resolved sequentially. Note that asynchronous communication with futures does not break support for runtime monitoring with LARVA.
2. The PROACTIVE layer hides all details regarding (network) communication between active objects. These details are captured in stub and proxy classes, and those do not break the ASPECTJ bindings of LARVA.

Additionally, many modern distributed programming architectures and protocols can naturally be modelled and constructed with the active objects design pattern, including web and cloud services.

3.1.1 Distributed banking example.

To monitor the distributed banking implementation in Figure 3 with LARVA, separate runtime monitors should be assigned to the client and server program, as they are run on different JVMs. Figure 4 shows the automata-representations of the two LARVA monitors. Both LARVA monitors check interaction with

the Bank active object, but from different perspectives: Figure 4a monitors from the client's perspective, whereas Figure 4b monitors from the server's perspective. In both cases, the triggers `login`, `logout` and `withdraw` correspond to their eponymous methods in the Bank class.

These runtime monitors again express the property that clients must first login before being able to logout or perform a withdrawal. However, since there are now two different parties involved—a client and a server—we choose, in this example, to monitor the property from two different perspectives. In the remainder of this section, we discuss the different states and transitions used in Figure 4, together with their underlying principles.

Call- and execution triggers. The *DATE* specification format allows the distinction between *method calls* and *method executions* when specifying triggers. In Figure 4, the superscript $\uparrow$ indicates that the transition should be taken upon *calling* the corresponding method in the program, whereas $\downarrow$ indicates an transition triggering upon starting the method *execution*. This distinction between call and execution events is supported by LARVA in the sequential setting, but becomes particularly useful when monitoring distributed objects.

To give an example, when a client invokes a `login` method on a server active object, the client itself does not execute the method `login`. Instead, the invocation becomes a network call to the server. A $\texttt{login}^{\downarrow}$ trigger would therefore not be meaningful in the monitor of the client. Moreover, within the server, the `login` method is executed but not invoked. Therefore, a $\texttt{login}^{\uparrow}$ trigger is meaningless in the monitor of the server. Generally speaking, a remote call to a method `m` causes two events in different contexts, $\texttt{m}^{\uparrow}$ in the context of the caller, and $\texttt{m}^{\downarrow}$ in the context of the callee.

To conclude, the call and execution triggers of *DATE*/LARVA match well the events in distributed caller-callee scenarios.

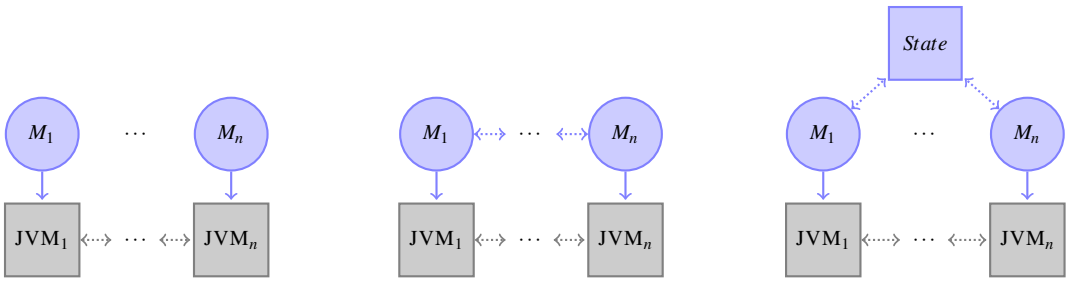
Monitor variables. Both runtime monitors in Figure 4 maintain a monitor-local variable, namely a list $\mathscr{C}$ or $\mathscr{S}$ of the codes of logged-in users. Initially these lists are empty. Some transitions update $\mathscr{C}$ or $\mathscr{S}$ via their action component by adding or removing codes. By using state in this way, runtime monitoring can be performed in scenarios where multiple clients log-in multiple users. Moreover, the formal parameter `code` is bound to the matching actual parameter given to the methods of Bank, and the placeholder `\result` captures the return value of the method corresponding to the transition's trigger.

Clarifying the transitions, client monitors go to a bad state when they: *(i)* perform a login attempt with a code that is already logged-in by the same client, or *(ii)* perform a logout with a code that is not logged-in, or *(iii)* attempt to withdraw money on an account that is not logged in.

The server distinguishes between two kinds of 'end' states. The server's monitor goes to the '*my fault*' state when it performs an invalid operation, i.e., upon a successful login for a user if the monitor knows from its state that the user is already logged in (i.e., its code is already in $\mathscr{C}$ or $\mathscr{S}$). On the other hand, the monitor goes to the '*not my fault*' state if it detects that a client violates its intended protocol. We use these two different end states for assume-guarantee inspired blame-shifting, see Section 3.3.

3.2 Distributed monitoring configurations

The runtime monitoring setup that we considered so far is purely distributed, as illustrated in Figure 5a. Each participating JVM (JVM_i) has its own LARVA monitor (M_i), and these monitors do not communicate with each other. However, there are many more monitoring configurations proposed in the literature [14, 9], some of which rely on monitor communication. Indeed, if independent monitors are



(a) Independent (b) Communicating (c) Orchestrated

Figure 5: Different monitoring configurations: 5a depicts purely independent monitors, 5b shows monitoring with communication, and 5c depicts orchestrated monitoring, where some centralised component *State* maintains the global state.

(a) Client monitor snippet (b) Server monitor snippet

Figure 6: Synchronisation of monitors using active objects. Here r is an active object with a public method `notify`(), that is instantiated by the server monitor and called by the client monitor (for notifying the server monitor).

by nature extremely efficient, inter-monitor communication is sometimes necessary to establish correctness. We explain below how to enable monitor interactions in our context and what interaction patterns we envision for monitors.

Distributed monitor communication. Recall that LARVA monitors are translated to Java code (to be bound to the monitored implementation using ASPECTJ), and that the transition in the automata may contain arbitrary snippets of Java. We exploit this to realise monitor communication, as shown in Figure 5b, by instantiating active objects *inside the* LARVA *monitors* and have them connect to each other. Runtime monitors may call the public methods on the active objects of remote monitors and thereby influence their state. Moreover, since active objects are used as ordinary Java objects, the LARVA monitors may define triggers on the public methods of their active objects. By doing this, a transition can be taken when a remote monitor invokes a public method on a local active object, thus implementing monitor synchronisation.

Figure 6 illustrates the integration of synchronisation between independent distributed LARVA monitors using PROACTIVE (this illustration might for example extend the setup from Figure 4). The server monitor may *itself* instantiate and host a new active object r (with a public method `notify`), and the client monitor may connect to this active object, like illustrated in Figure 3. Then when the client monitor transitions to some bad state for example, it may call r.`notify`() in the action component of the transition (6a). The object r in the client monitor is only a proxy object generated by PROACTIVE, so this method invocation targets the monitor of the server. Since `notify` is an ordinary Java method, the server's monitor may have a trigger for its execution (i.e. `notify`$^{\downarrow}$), so that the monitor can take a transition when the client invokes `notify` (6b).

Orchestrated monitoring. By allowing monitors to communicate with each other, we can also realise elaborate monitoring configurations like *orchestrated monitoring*, illustrated in Figure 5c. Instead of having the monitors communicate with each other directly, they communicate with a *centralised* active object (called *State* in the figure). This schema is the best way to implement a globally shared memory. Indeed active objects do not share data, this organisation can allow us to an active object to store a global state, together with getters and setters invokable by the distributed monitors. In addition, since the centralised component is an active object, and therefore plain Java, it may have its own LARVA monitor and verify invariants on the global state.

These different configurations open-up a design space of distributed monitoring, as they may be combined in many ways. For example, one may consider orchestrated monitoring as in 5c, but still allow LARVA monitors to communicate with each other, as in 5b. Moreover, if a single orchestrator (like shown in 5c) does not provide enough scalability, one may consider a hierarchical structure of orchestrators, each coordinating a subset of the distributed monitors.

3.3 Assumption-guarantee style blame-shifting

In some scenarios, independent distributed monitoring is insufficient to establish global correctness, while monitor communication is performance-wise too heavyweight. The distributed hash table case study in Section 4 falls into this category, since communication-based monitoring would break its scalability, while independent monitoring alone is not enough to establish correctness globally.

For these scenarios we propose a notion of *blame-shifting* that is inspired by assume-guarantee (AG) reasoning [12]. The idea is that, instead of having runtime monitors rely on network communication, they could instead rely on *assumptions* from the environment, while giving certain *commitments* to the environment in return. This system of assumptions and commitments must be consistent: a monitor may only assume properties from the environment that are guaranteed/committed by the monitors of the environment. (Otherwise, the insights from the runtime verification would be limited.)

This implementation of AG reasoning constitutes a 'my fault/not my fault' blame system. A 'not my fault' occurs when a monitor observes interaction with remote objects that violates its assumptions on the environment. A 'my fault' occurs when a monitor either observes interaction with a remote object that violate its commitment to the environment, or if it violates a local sub-property (possibly leading to interaction that violates its commitment to the environment). Moreover, if the assumptions and commitments of interacting distributed objects are in sync, then a 'not my fault' in some objects will be mirrored by a 'my fault' in one or several other objects. In conclusion, AG style monitoring can be used to detect and locate distributed runtime violations, without having to resort to centralised monitoring or other configurations that require communication.

Blame-shifting by example. An example of blame-shifting was already presented in Figure 4. Here, the server monitor distinguishes between violations from the environment and local violations via two different end states, named '*not my fault*' and '*my fault*'. The client monitor does not have a '*not my fault*' state in this example, as the intended protocol only really restricts the client (e.g. the *client* must not withdraw before logging in). Also notice that the assumptions of the server are consistent with the commitments of the client; if the server's monitor goes to the '*not my fault*' state, it means that a client performed a logout or a withdrawal without being logged-in, and therefore goes to its '*my fault*' state.

Although the two runtime monitors described in Figure 4 do not communicate, the server can still determine whether clients violated their protocol. This concept of blame-shifting is demonstrated further in the case study (Section 4) in particular in a system consisting of more than two objects.

Global consistency. At the moment we do not have a mechanised way of checking whether the assumptions made by LARVA monitors are consistent with the commitments of the monitors of the environment. This currently has to be established manually. We are investigating ways to make this check mechanical, as a static verification task. This would result a combination of *static global* consistency checking and *local runtime* verification.

3.4 Monitoring data-oriented aspects

Distributed systems typically consist of nodes that perform local computations and distribute the computed results over the network, via protocols. Runtime verification of distributed systems should therefore cover both the computation and the distribution of data. Even though LARVA allows *DATE* specifications to use data to a limited extend, it is mostly focused on control-oriented properties.

In order to extend our approach to the monitoring of data-oriented properties, we define *ppDATE*, an extension of *DATE* with *Hoare triples* (*pp* stands for *p*re/*p*ost-condition), as supported by the STARVOORS [2, 8] tool (STAtic and Runtime Verification of Object-ORiented Software). Hoare triples are of the standard form $\{P\}\,\mathtt{m}\,\{Q\}$, with P and Q assertions expressed in first-order logic and `m` the header of a Java method. In *ppDATE*, Hoare-triples are *state dependent*: each state in the runtime monitor automaton contains its own set of Hoare triples that have to hold in that state. STARVOORS translates *ppDATE* to *DATE* and uses LARVA thereafter.[1] Section 4 illustrates how Hoare triples can make the specification and monitoring of distributed applications easier.

4 Case Study: ActiveCAN

This section demonstrates our verification approach on a case study: verifying the correctness of a CAN [21]—a Content-Addressable Network—implemented with active objects in the PROACTIVE platform [20]. In the sequel we refer to this case study as ActiveCAN. The main motivations for considering the ActiveCAN case study are: (i) the case study is external, rather than a toy example that we constructed ourselves (an evolved version has also been used in a large-scale application)[2]; (ii) the property to be runtime verified is a combination of both data- and control-oriented properties; and (iii) by nature the example consists in many identical peers, which highlights the meaning of the distinction 'my fault' vs. 'not my fault', where 'not my fault' means either another peer or the rest of the system.

Application description. A CAN is a distributed infrastructure that provides hash table-like functionality. The basic operations of a CAN are: $\mathtt{insert}(k,v)$ for inserting a value v at key k, and $\mathtt{lookup}(k)$ for looking-up the possible value stored at key k. The originality is that keys are tuples. CANs consist of many peers that are connected via a network, and each peer owns a fragment of the entire key space, i.e. a hyperrectangle. When a new peer joins the network, the key space in the CAN is split, allowing the joined peer to receive ownership of the new fragment. Moreover, when performing an `insert` or `lookup` on a peer, the operation may either be handled locally if the key is owned by the local peer, or otherwise be relayed to a remote peer. For relaying operations to remote peers the CAN infrastructure contains a routing protocol that is scalable.

[1] STARVOORS also supports monitor optimisation through (partial) static verification results, a feature which we do not use yet in the distributed setting.

[2] See `http://play.ow2.org`.

The ActiveCAN consists of a set of active objects, each responsible for the storage of data indexed in one hyperrectangle. Each peer knows the zone it manages, but also the zones of the neighbouring peers to route messages to the right target. The ActiveCAN peers provide three operations:

- `join`(p), to add one new peer p in the network. The joined peer will split its hyperrectangle into two parts and delegate one to the new peer.
- `insert`(k,v), to insert a new value v at a given key k where k is a tuple.
- `lookup`(k), to fetch the value previously stored at a given tuple k.

Communication between neighbouring peers is performed by method calls on active objects, and each request is transmitted this way to the adequate target. Finally, lookup relies on futures to return the fetched value.

Verified properties. Essentially, the following high-level properties are addressed:

1. The behaviour of the hash table is *consistent*, meaning that: *(i)* after calling `insert`(k,v) the data element v is stored somewhere in the network at key k, *(ii)* if `lookup`(k) returns a positive result, then k is mapped somewhere in the network, *(iii)* if `lookup`(k) returns a negative result, then k is not mapped in the network.
2. There are no cycles in the routing protocol. In particular, we verify that: *(i)* any `lookup` and `insert` operation is either handled locally by a peer, or the peer has at least one neighbour to defer the operation to, and *(ii)* if a `lookup` or `insert` needs to be remotely resolved, the operation is deferred to a remote peer that is *closer* to the target peer (hence there are no cycles in the routing protocol).

Property (1) in the above is data-oriented, whereas (2) is control-oriented, and both are necessary to establish correctness of the overall infrastructure. The case study thereby highlights the necessity for verification techniques to include data and control aspects and motivates the usage of the STARVOORS front-end to LARVA. The distributed setting is particularly interesting here because peers typically perform some computation over data and distribute requests and results via some routing protocol. LARVA generates monitors checking these properties, reporting when any of them is violated while using the hash table.

To verify property (1), for each peer, the runtime monitor maintains a list $\mathscr{K}$ of keys to be mapped in the key/value store. For each key, it also maintains a boolean flag that indicates whether the key is stored locally or remotely. Therefore, $\mathscr{K}$ contains all keys stored locally by the peer, but also records the keys of remote `lookup` and `insert` operations that routed through the peer.

The runtime monitors use blame-shifting in the style of AG reasoning to determine whether a lookup or insert is handled correctly. In more detail, when a locally resolved `lookup` behaves incorrectly, the runtime monitor of the local peer is able to detect this: it is classified as 'my fault'. When a remotely resolved `lookup` behaves incorrectly (e.g. stating the key is unmapped while the runtime monitor knows from $\mathscr{K}$ that the key has already been stored), then the local peer can determine that another peer has violated the property: it is classified as 'not my fault'. Since the remotely resolved `lookup` has been deferred to a neighbouring peer, that peer is closer to figuring out which peer actually violated the property. In particular, due to the consistency of the AG-style assumptions and commitments, the neighbouring peer's monitor must also be in a *not my fault* or *my fault* state. This constitutes a chain of blames, ending-up in the misbehaving peer (i.e. the peer that is in the *my fault* state).

Property (2) is verified using the Hoare triple extensions that are discussed in Section 3.4. Every time a `lookup` or `insert` operation needs to be deferred to a remote node, the implementation calls the helper method `getZoneClosestTo`(k), which returns the neighbouring zone that is closest to the destination peer. The property of loop freedom in the routing protocol is captured by the following Hoare triple.

$$\{\mathsf{true}\}\ \texttt{getZoneClosestTo}(k) \left\{ \begin{array}{c} \backslash result \neq \mathsf{null} \wedge \\ \backslash result.\texttt{distance}(k) < localzone.\texttt{distance}(k) \end{array} \right\}$$

Every zone z has a distance function $z.\texttt{distance}(k)$ that calculates the Euclidean distance between the center of z and the specified key k. The variable *localzone* refers to the zone of the callee peer. This Hoare triple is checked every time the `getZoneClosestTo` method is called in the implementation. This method returns an object of type Zone—the zone of the neighbour closest to the destination peer.

Availability. The source code of the ActiveCAN case study, together with verification instructions, are available online[3]. Our experiments did not reveal runtime violation of properties (1) and (2).

5 Related Work

Even if the area of runtime verification is much more elaborate in the stand-alone setting[4], runtime verification of distributed systems is a very active area of research. The recent article [16] provides an excellent overview over scenarios and characteristics of this sub-field, and discusses the existing approaches on that background. Here, we only include works with sufficient overlap in aim, method, or application area. Generally, our discussion of configurations in Section 3.2 is consistent with the monitor organisation categories in [16]. However, compared to the analysis there the LARVA methodology for specification by nature splits the monitoring into properties specified and verified for each object, which naturally extends to a truly distributed monitoring approach. A recent work use distributed monitors to control the executions of Erlang actors [15], the behaviour of the program is expressed as a choreography and the monitors control the execution so that, in case of failure, the application can rollback to a safe state and then run along a different path. Compared to these choreography-based approaches, we provide the monitoring of a reactive system that does not have a service choreography specification. For example we are able to monitor a peer-to-peer system that has a more concurrent behaviour than choreographies. In such a system it makes more sense to specify the correctness of each entity based on assumptions on the environment than from a global perspective, and this naturally provides distributed monitors. However, it is more difficult to state properties of the global system in our approach than from a choreographed perspective, but to the best of our knowledge, no other runtime-verification approach is able to verify at runtime a reactive system made of distributed objects.

The ELARVA runtime verifier [10] is an adaption of LARVA for monitoring concurrent Erlang programs. In particular, ELARVA adapts LARVA by translating the object-oriented monitoring constructs to a process-oriented setting, and gives an asynchronous interpretation to LARVA's monitoring semantics. However, ELARVA only supports centralised monitoring and is unable to monitor across multiple machines. Our implementation allows both centralised and decentralised monitoring, and allows distributed monitors to communicate and coordinate using active objects, as described in Section 3. Another notable difference is that we did not change the synchronous LARVA's monitoring semantics, and instead leave

[3] `https://github.com/utwente-fmt/RV2018-ActiveCAN`.

[4] This includes the analysis of single nodes, only, running in a distributed environment.

it to PROACTIVE to 'hide' the asynchronous nature of the underlying communication (not only from the programmer but also from LARVA).

There are several tools for static verification of active objects, relying on model-checking [4, 22], static analysis [3], behavioural types [17], or deductive techniques [13], but the support for monitoring and runtime verification of active object systems is quite weak. Basic monitoring tools exist for actors and active object systems. For example Akka [23] provides an interesting hierarchy of actors for monitoring failures, both ABS and PROACTIVE feature a tool for viewing or debugging active object execution (see e.g. Section 3.3 of [18]), and a monitoring framework has been developed for Erlang [7]. In existing platforms runtime verification of functional correctness for active object applications must be done by hand. We believe that we can fit our approach to the infrastructures proposed in actor monitoring systems by generating monitors specific to an actor monitoring framework, like [7]; however our current approach has the crucial advantage to minimise the changes to the monitor generation of LARVA.

The choice of locations of the monitors is quite an important issue because communication across locations is usually expensive and information-sensitive. A good discussion about this choice is presented in [14], where a theoretical framework is presented for comparing those choices. Such a discussion is complementary to the solution presented in this system paper, which shows how different monitoring architectures can be naturally realised for a distributed variant of Java.

We are not aware of other work which adapts the assumption-guarantee paradigm, known from compositional static verification, to the runtime verification setting.

6 Conclusion

This article shows the importance of two key challenges in monitoring of distributed applications: distributed monitoring by independent active monitors, and verification of properties mixing data-oriented and control-oriented aspects.

From a technical point of view, this article provides two contributions addressing these two challenges. First it presents an effective distributed monitoring mechanism. This is realised thanks to active monitors that are generated by our framework. Our runtime verification environment combines monitor generation of the LARVA framework with an active object middleware, PROACTIVE. This way, the standard Java code generated by LARVA generates distributed active monitors at runtime to detect the violation of safety properties in a distributed monitoring fashion. The verification setup integrates assumption-guarantee-style blame shifting to efficiently localise runtime failures while limiting communication between monitors. Moreover, the Hoare triple extension to LARVA provides to the programmer a better abstraction for specifying properties mixing data-oriented and control-oriented aspects. This is particularly relevant in connection with the active object paradigm, where a task is the execution of a method.

The approach has been illustrated by monitoring a distributed peer-to-peer system implementing a key-value store. This example and the properties we were able to monitor illustrate well the contributions of this work: it is by nature distributed, and needs a distributed monitoring infrastructure; and the application mixes control-oriented and data-oriented properties both in the routing of messages and in the storage/retrieval of data; and the blame-shifting allows to trace the cause of an error along the routing path (even if this is error tracing is currently done on the meta-level and not yet automated).

Our approach is by nature distributed. In particular, we do not favour verification of global properties and prefer to focus on a local rely-guarantee approach. The counterpart is that our framework is not particularly adapted to reason at a global level, and in particular an object is unable to distinguish if an

external error is due to another object or to a communication error. However, as monitors can communicate and a global state can be stored, we should be able in the future to extend our framework with a better support for global properties and better classification of external errors.

In the future, we want to investigate the runtime overhead of our distributed monitoring methodology and its variants. Also, we want to explore how easy/difficult it is for users to specify the properties of interest in the suggested style. Another line of future work is to migrate the StaRVOOrS approach [2] to the distributed setting, to combine static verification with our distributed runtime verification method. Just like in StaRVOOrS, this has great potential to decrease the runtime overhead and increase the scalability.

Acknowledgements.

The authors would like to thank Gordon Pace and Gerardo Schneider for fruitful discussions in the course of this work, and Mauricio Chimento for implementing some adaptions in the StaRVOOrS tool. This work is partially supported by the NWO TOP 612.001.403 project VerDi, and by the COST Action IC1402 Runtime Verification beyond Monitoring.

References

[1] *ProActive Middleware*. Available at `https://github.com/scale-proactive`.

[2] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace & Gerardo Schneider (2017): *Verifying data- and control-oriented properties combining static and runtime verification: theory and tools*. Formal Methods in System Design 51(1), pp. 200–265, doi:10.1007/s10703-017-0274-y.

[3] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla & Guillermo Román-Díez (2014): *SACO: Static Analyzer for Concurrent Objects*. In Erika Ábrahám & Klaus Havelund, editors: Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS 8413, Springer, pp. 562–567, doi:10.1007/978-3-642-54862-8˙46.

[4] R. Ameur-Boulifa, L. Henrio, O. Kulankhina, E. Madelaine & A. Savu (2017): *Behavioural semantics for asynchronous components*. Journal of Logical and Algebraic Methods in Programming 89, pp. 1–40, doi:10.1016/j.jlamp.2017.02.003. Available at `http://www.sciencedirect.com/science/article/pii/S2352220817300287`.

[5] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes & Albert Mingkun Yang (2017): *A Survey of Active Object Languages*. ACM Computing Surveys 50(5), pp. 1–39, doi:10.1145/3122848.

[6] Denis Caromel & Ludovic Henrio (2004): *A Theory of Distributed Objects*. Springer, doi:10.1007/b138812.

[7] Ian Cassar & Adrian Francalanza (2016): *On Implementing a Monitor-Oriented Programming Framework for Actor Systems*. In Erika Ábrahám & Marieke Huisman, editors: Integrated Formal Methods, Springer, pp. 176–192, doi:10.1007/978-3-319-33693-0˙12.

[8] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace & Gerardo Schneider (2015): *StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java*. In Ezio Bartocci & Rupak Majumdar, editors: Runtime Verification, Lecture Notes in Computer Science 9333, Springer, pp. 297–305, doi:10.1007/978-3-319-23820-3˙21.

[9] Christian Colombo & Yliès Falcone (2016): *Organising LTL monitors over distributed systems with a global clock*. Formal Methods in System Design 49(1-2), pp. 109–158, doi:10.1007/s10703-016-0251-x.

[10] Christian Colombo, Adrian Francalanza & Rudolph Gatt (2012): *Elarva: A Monitoring Tool for Erlang*. In Sarfraz Khurshid & Koushik Sen, editors: *Runtime Verification*, Springer, pp. 370–374, doi:10.1007/BFb0053381.

[11] Christian Colombo, Gordon J. Pace & Gerardo Schneider (2009): *LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)*. In: *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, IEEE Computer Society, pp. 33–37, doi:10.1109/SEFM.2009.13.

[12] W.P. de Roever, U. Hanneman, J. Hooiman, Y. Lakhneche, Mannes Poel, Jakob Zwiers & F. de Boer (2001): *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press. Imported from HMI.

[13] Crystal Chang Din, S. Lizeth Tapia Tarifa, Reiner Hähnle & Einar Broch Johnsen (2015): *History-Based Specification and Verification of Scalable Concurrent and Distributed Systems*. In Michael Butler, Sylvain Conchon & Fatiha Zaïdi, editors: *International Conference on Formal Engineering Methods (ICFEM)*, *LNCS* 9407, Springer, pp. 217–233, doi:10.1007/978-3-319-25423-4˙14.

[14] Adrian Francalanza, Andrew Gauci & Gordon J. Pace (2013): *Distributed System Contract Monitoring*. *The Journal of Logic and Algebraic Programming* 82(5), pp. 186–215, doi:10.1016/j.jlap.2013.04.001. Available at `http://www.sciencedirect.com/science/article/pii/S1567832613000234`. Formal Languages and Analysis of Contract-Oriented Software (FLACOS'11).

[15] Adrian Francalanza, Claudio Antares Mezzina & Emilio Tuosto (2018): *Reversible Choreographies via Monitoring in Erlang*. In Silvia Bonomi & Etienne Rivière, editors: *Distributed Applications and Interoperable Systems*, Springer, pp. 75–92, doi:10.1016/j.jlamp.2017.11.002.

[16] Adrian Francalanza, Jorge A. Pérez & César Sánchez (2018): *Runtime Verification for Decentralised and Distributed Systems*, pp. 176–210. Springer, doi:10.1007/978-3-319-75632-5˙6.

[17] Ludovic Henrio, Cosimo Laneve & Vincenzo Mastandrea (2017): *Analysis of Synchronisations in Stateful Active Objects*, pp. 195–210. Springer, doi:10.1007/978-3-319-66845-1_13.

[18] Ludovic Henrio & Justine Rochas (2017): *Multiactive objects and their applications*. *Logical Methods in Computer Science* Volume 13, Issue 4, doi:10.23638/LMCS-13(4:12)2017. Available at `http://lmcs.episciences.org/4079`.

[19] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A Universal Modular ACTOR Formalism for Artificial Intelligence*. In Nils J. Nilsson, editor: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, W. Kaufmann, pp. 235–245.

[20] Laurent Pellegrino, Fabrice Huet, Françoise Baude & Amjad Alshabani (2013): *A Distributed Publish/Subscribe System for RDF Data*. In Abdelkader Hameurlain, Wenny Rahayu & David Taniar, editors: *Data Management in Cloud, Grid and P2P Systems*, Springer, pp. 39–50, doi:10.1145/964723.383071.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp & S. Shenker (2001): *A Scalable Content-Addressable Network*. In: *SIGCOMM*, ACM, pp. 161–172, doi:10.1145/383059.383072.

[22] Marjan Sirjani, Ali Movaghar, Amin Shali & Frank S. de Boer (2004): *Modeling and Verification of Reactive Systems using Rebeca*. *Fundamenta Informaticae* 63(4), pp. 385–410.

[23] Derek Wyatt (2013): *Akka Concurrency*. Artima.