Considerations for Monitoring Highly Concurrent Systems

Ruth Mizzi Christian Colombo Adrian Francalanza Gordon Pace
Department of Computer Science
University of Malta
{rmiz0015 | christian.colombo | adrian.francalanza | gordon.pace}@um.edu.mt

ABSTRACT

Sequential monitoring tools such as LARVA are impractical for monitoring highly concurrent systems such as online establishments handling hundreds of transactions a second — they lock valuable resources which may otherwise be used to serve valid user requests. In the context of an open-source e-commerce system, we discuss design issues involved in allowing monitors to run concurrently while at the same time ensuring that they remain correct: free from race conditions and faithful to the properties they embody.

1. INTRODUCTION

The extensive use of software systems in everyday operations requires the guarantee that they do not fail or act in unexpected ways once deployed. Runtime monitoring [2, 5, 1] can play an important part in ensuring the quality and correctness of software systems. A runtime monitor can be viewed as an external entity that oversees the behaviour of a system's execution and gives feedback to an observer if and when this behaviour goes against a pre-defined specification.

LARVA [4] is a synchronous runtime verification architecture with a focus on the expressiveness of a specification language for defining system properties and contextual monitoring. With their rigid locking mechanism, LARVA monitors process system events sequentially. Sequential monitoring is impractical for highly concurrent systems such as online establishments processing hundreds of transactions simultaneously — valuable parallel resources would lie idle waiting for sequential monitor feedback. To address this (and other) issues we are currently building polyLARVA, a second generation of the tool LARVA. The focus of this paper is to discuss the design decisions taken for the polyLARVA runtime monitoring framework in order to support highly concurrent systems, taking into consideration performance overheads.

The paper is structured as follows: In the next section we give an introduction to the runtime monitoring architecture implemented in polyLARVA and highlight the issues that might occur during monitoring of concurrent systems. In Section 3 the various alternatives considered for solving these issues are highlighted with details given as to why each alternative is appropriate or what disadvantages are associated with it. We conclude in Section 4 by giving an indication of the future work that can be built on our conclusions.

2. CONCURRENCY AND RUNTIME MONITORING

While LARVA has been applied to runtime monitor real-life systems [3], monitoring properties over parts of the system which run concurrently can be rather challenging: Property satisfaction/violation frequently depends on inputs from several concurrent execution threads. The complexity increases further if the properties depend on the system state, which may change while monitoring. Since nowadays many applications support thousands of parallel threads (making global locking unfeasible), this is an important challenge to be addressed. For runtime monitoring to be useful, we need to ensure that the original system does not suffer large performance overheads. In addition, the monitoring logic must not be adversely affected by different processes running in parallel on the system and interacting with it.

In order to illustrate the issues that concurrency imposes on the design of polyLARVA, we first explain the architecture of a polyLARVA runtime monitor in the context of monitoring an open-source e-commerce system[1]. The choice of an e-commerce system, designed to support multiple user sessions running in parallel, emphasises the issues that arise in a multi-transactional, concurrent system.

A traditional e-commerce system offers features such as (i) user registration, where new users can open an account with the online shop; (ii) user account log-in, which allows a user to log into his account and start an online shopping session; (iii) shopping cart, which is used during a session to accumulate a list of items for purchase; and (iv) payment, which normally involves communication with a payment gateway that can authorize or refuse credit card transactions. A runtime monitor observing such a system would be made aware of certain system events occurring, among which will be events associated with each of the stated functions.

A simple property that can be monitored on an e-commerce system is that of keeping a count of failed transactions (where failed transactions are ones refused by the payment gateway) and alerting an administrator if it grows uncommonly large or fast. Figure 1 shows that the interaction between system and monitor occurs everytime a payment fails on the system. The monitor acts on that notification to carry out the required monitoring logic, in this case a simple increment of its state variable.

Such monitoring is straightforward, but it quickly becomes less so when, for example, we attempt to monitor the count of failed payments *per user account* (thus giving an administrator a clear

[1]JadaSite e-commerce solution — `http://www.jadasite.com`

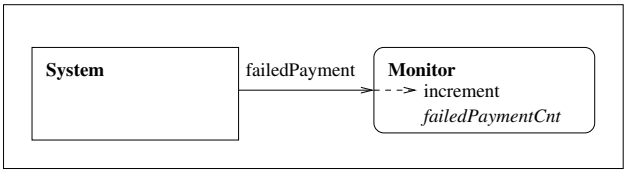

Figure 1: Monitoring of failed payments

idea of whether there are particular user accounts which regularly encounter problems). Such a property would be more naturally monitored through a modular approach where, instead of regarding the monitor as one global entity, we set up a monitor *per user account*. While decomposing monitors facilitates their specification, it introduces several concurrency issues which are further discussed in what follows.

2.1 Contexts

Just as in LARVA, polyLARVA facilitates the observation of properties that are specific to particular system entities during their lifetime by introducing the notion of *contexts*.

A context can be viewed as a monitoring entity which is created dynamically during runtime monitoring, when a new specific instance of a concept appears. Typically, in a language such as Java, these concepts correspond to instances of classes, but in general, they can have a different interpretation (e.g. each new thread, or new file opened). Furthermore, one can have nested contexts, with each context being aware of its container's data. For instance, every user can have a separate context, each of which may contain a separate context for each transaction carried out by that user. Concretely, a context is described by a trigger which will create an instance of that context, and a description of the content of each such instance, consisting of:

- Data structures storing the local monitoring state of the instance of that context.
- A number of properties which will be monitored on the instance.
- A number (possibly zero) of sub-contexts, instances of which would be children of this particular instance.

The key to state partitioning is that contexts induce a tree of context instances, with the properties at a particular node in the tree being able to access (read and write) the local monitoring state within that node and the nodes of its ancestors. Access to other nodes' local state (including siblings and children) is not possible.

In the e-commerce scenario, each user will have a separate context, responsible for monitoring properties which are related to it. Such a model would ensure that every user account context keeps its own record of failed payments. Another important context is that of a session, used to check properties such as ensuring that a session is not inactive for more than ten minutes.

One may model both types of contexts at the same level of abstraction (see Figure 2), creating contexts for each new user encountered and session opened. However, each session is associated to a particular user, and one may want to monitor, for instance, a property stating that:

> *A user account may not have more than 3 active sessions.*

This would require the different contexts to have access to shared state (at the global level) to store the number of active sessions per user, which can lead to intricate race conditions, which would be undesirable. In our case, all sessions owned by the same user would need to access a variable storing the number of open sessions that each user has. For this reason, an alternative and more appropriate context model is to have the session contexts nested inside the context of the user (their owner). In this manner, the shared state is local to the user context and can be easier to handle correctly. Figure 3 shows a system with three user accounts being monitored, each of which may have a number of active sessions. In this example there are two concurrent active sessions for the user holding account *C*. The local state storing the sessions count is kept separate at each user's context node. As highlighted in this figure, the contexts are now related to each other. Since we know that every user session is associated with one user account, we can adopt a hierarchical model where the user session context started for a new active session is immediately linked to its associated user account context. This simplifies monitoring of related properties. When a user logs into the system, the monitor is notified so that it can launch a new user session context for a particular user account context. There is therefore a direct link to the state of the user account context which includes the count of currently active sessions.

The context-based model permits a structured decomposition of a monitor into separate submonitors which observe the behaviour of different parts of the system. In our example, the monitors can work independently since the properties they monitor have no overlap. For instance, the user account monitor is taking care of observing failed payments while the user session monitor is concerned with the length of the said session.

Program 2.1 polyLARVA specification

```
foreach useraccount:UserAccount {
    local failedPaymentLimit = 0

    on (useraccount a.setFailedPaymentLimit()) do
      update local (failedPaymentLimit)

    foreach session:UserSession {
        local failedPaymentCount = 0

        on (session s.paymentFail()) do
          update local (failedPaymentCount)


        on (session s.paymentFail()) do
          compare local(failedPaymentCount)
          to shared (failedPaymentLimit)
    }
}
```

The setting becomes more intricate when there are dependencies between context levels. For instance, consider the property:

> *Every user account has its own fixed threshold of allowable failed payments per session and an error is to be logged if a session exceeds this limit.*

If the system gives facilities to an administrator to modify the failed payment limit associated with an account at any point in time, both a user transaction and such an action may trigger the monitor reparation. An extract of the polyLARVA specification for this behavior is shown in the code extract given in Program 2.1. A hierarchical structure of contexts is adopted in order to maintain a link between a user account context and its sessions. The specification shows that a user account context is made up of:

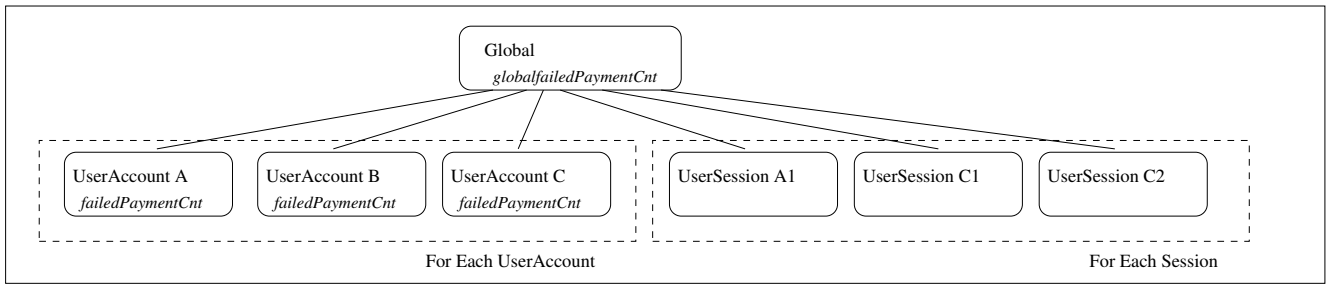

Figure 2: Contexts in monitor model

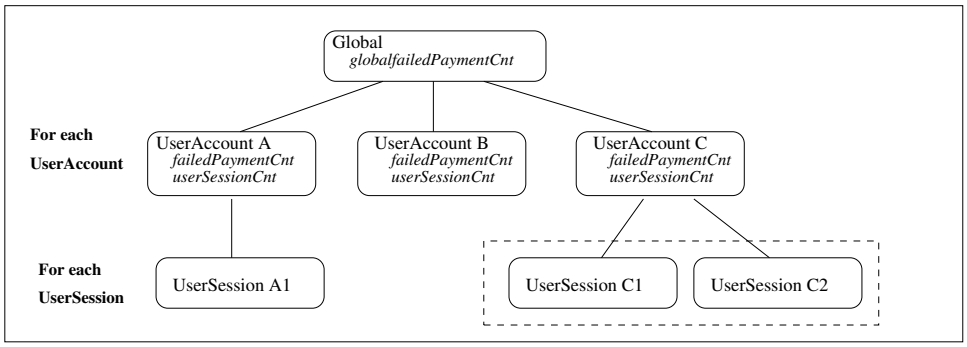

Figure 3: Hierarchy of contexts in monitor model

- a local variable *failedPaymentLimit* maintaining the limit of failed payments allowed on that particular account.
- a property that specifies the action that is to occur whenever the system's failed payment limit for the associated account is modified. In this case, the local variable is updated to reflect the new system limit.
- a sub-context, session, instances of which will be related to this particular instance of user account.

The behaviour of the session context is defined in terms of two actions that occur sequentially whenever a session payment transaction fails.

Figure 4 illustrates the polyLarva monitor model resulting from the specification given in Program 2.1. The user session context needs to communicate at each failed payment event with its corresponding user account to access the threshold value and compare with its current count. An administrator user might be modifying the user account settings on the system to change the threshold of accepted failed payments for a particular account while (concurrently) the customer is making a payment. The concurrent access on the value of *failedPaymentLimit* leads to potential race conditions. Such an example highlights the fact that to observe more complex and interesting properties, some form of communication between the different contexts is necessary. This concurrent inter-context communication requirement in the monitored systems may lead to problems in the monitoring logic, unless these resources are handled properly. Locking down all such resources solves these problem, but at too high a cost on performance. We will look into ways of addressing this issue in a more fine-grained manner.

3. CONCURRENCY IN MONITORING

In the previous section we highlighted how concurrency may undesirably lead to race-conditions in the monitoring logic. Two pos-

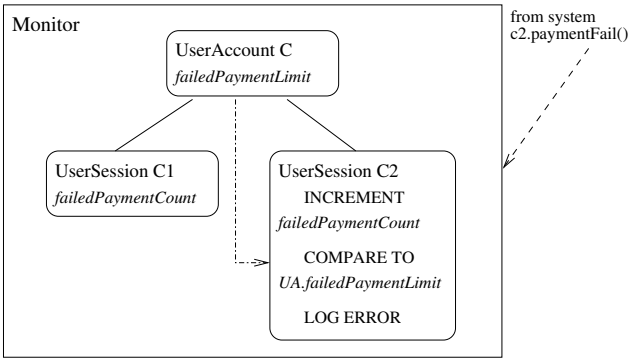

Figure 4: Monitors require inter-communication

sible extreme solutions are to either lock the whole system while monitoring, or asking the specification writer to manually handle possible race-conditions. The former is unreasonable since concurrency is crucial in transaction handling, while the latter means that monitoring becomes more error-prone, which is clearly undesirable. The use of contexts is key to the solution, since it partitions data access across different parts of the monitor. There are two issues arising from concurrent data access that we need to address: (i) since every monitoring context maintains a local state, we need to ensure that no race conditions occur due to concurrent processing of system events; and (ii) access to shared variables: different monitoring contexts might concurrently access and modify shared variables belonging to their ancestor(s). In this section we will look into different ways of handling these issues from global to selective local locking of resources to make monitor design with shared state more tractable.

3.1 Full Locking

The simplest approach to monitoring concurrent processes is to lock all resources of the runtime monitor every time an event occurs on the system which is relevant for any of the contexts in the polyLarva monitors. This ensures that the monitor only processes one event at a time, providing a solution to both issues identified — a monitor's local state can only be accessed by one event at one time, and concurrent access to shared variables may not occur. However, the approach induces high overhead on the system. For instance, in the e-commerce scenario, this approach will lead to many requests being queued waiting for the release of the monitor lock which would thus result to substantial system slow-down.

3.2 Locking on a context

Contexts provide a natural partitioning of the monitor state. One can thus use this structure to identify boundaries at which parts of the shared state need to be locked before proceeding. Figure 5 shows how the monitor acquires a lock on the session context to act on the event when it is notified of a failed payment received for a particular user session. This ensures that a monitor's local state is never accessed concurrently by other monitors in that context. Although the lock queues failed payments on other sessions of the user until the lock is released, other monitorable events occurring on the system effecting other users are still allowed to proceed. For instance, administrator actions that affect other user account contexts by modifying *failedpaymentLimit* is still processed.

This solution, however, still does not offer a solution for the problem of concurrent access on states by different contexts. A lock on the user session monitor would be acquired to process a failed payment on that session. The processing of this event would require access to the associated user account's *failedpaymentlimit*. In this setup there is no guarantee that this variable is not concurrently undergoing a change in value by the user account monitor. This can occur since the user session context and user account context are not controlled by the same lock.

3.3 Locking on Selected Actions

Program 3.1 Specification with explicit locking

```
foreach useraccount:UserAccount {
    local failedPaymentLimit = 0

    with lock (useraccount.failedPaymentLimit)
    do {
        on (useraccount a.setFailedPaymentLimit())
        do
          update local (failedPaymentLimit)
    }

    foreach session:UserSession {
        local failedPaymentCount = 0

        with lock (useraccount.failedPaymentLimit)
        do {
          on (session s.paymentFail()) do
            update local (failedPaymentCount)


          on (session s.paymentFail()) do
            compare local(failedPaymentCount)
            to shared (failedPaymentLimit)
        }
    }
}
```

Since contexts share state with their descendants (as highlighted in the script given in Program 2.1), a user session monitor may access the state of the associated user account monitor. Assuming such a shared state means that we have to control access to the context's state variables to avoid problems occuring with concurrent access. This can be done through explicit specification on what monitor variables need to be locked during the processing of a particular event. The specification of the monitor contexts given in Program 2.1 could therefore be extended to state that the events *setFailedPaymentLimit()* and *paymentFail()* should not be handled before a lock on the commonly accessed variable *failedPaymentCount* is acquired, which would require a specification as shown in Program 3.1.

Figure 6 depicts how this approach works: All events that need access to the user account monitor state variable *failedPaymentLimit* must first acquire a lock on that variable before being processed. Since each user session is associated with one user account then the lock is acquired on the variable associated with that particular account. Figure 6 shows how such a locking mechanism would ensure that concurrent access on the same instance variable is controlled while still allowing events on other user session and account entities to be processed. In the example shown, the *paymentFail()* notification occurring on session $A1$ can be processed while concurrent events on account C and session $C2$ are being controlled by the locking process. This setup solves both issues identified at the start of Section 3: Locks on specific variables ensure that concurrent events do not attempt to access the same variables, both when considering one particular context as well as in the case of different contexts accessing common variables.

3.4 Message Passing

While locking on selected variables is a valid approach towards controlling concurrent data access, identifying which locks need to be acquired requires considerable work and can be error-prone. An alternative approach is to allow contexts to share (state) information through message passing. Thus, access to the global and ancestors' states is disallowed except through explicit channel communication, which ensures that all shared data access is safely done at the cost of more verbose specifications. Further details on this approach are given in the next subsection.

3.5 Approach chosen

The approach adopted in polyLarva is a hybrid of message passing and context locking. It uses message passing to enable communication and state sharing between the contexts, but controls concurrent events on individual contexts through context locking. Figure 7 shows the setup as adopted by polyLarva in the context of the *failedPaymentLimit* example. Locking occurs by default at the context instance level — if a context is loaded for user account A, then a lock is associated with this monitor context and must be acquired every time this context is called to carry out any evaluation. In our example, a user session context will be started every time a login under user account A occurs on the system. A separate lock is associated with each session context. In the figure these locks are highlighted as shaded boxes.

Figure 7 also depicts an application of the message passing mechanism: A *failedPayment* occurs on a session and, concurrently, the associated account's *failedPaymentLimit* variable is updated. Both of these events are related to user account C. Since polyLarva monitors do not support shared state then, when the *failedPayment()* is processed in session $C1$ monitor, the comparison to the account's *failedpaymentLimit* variable cannot occur directly. Instead, the context associated with session $C1$ will send an internal message to user account C. Internal messages are parameterised

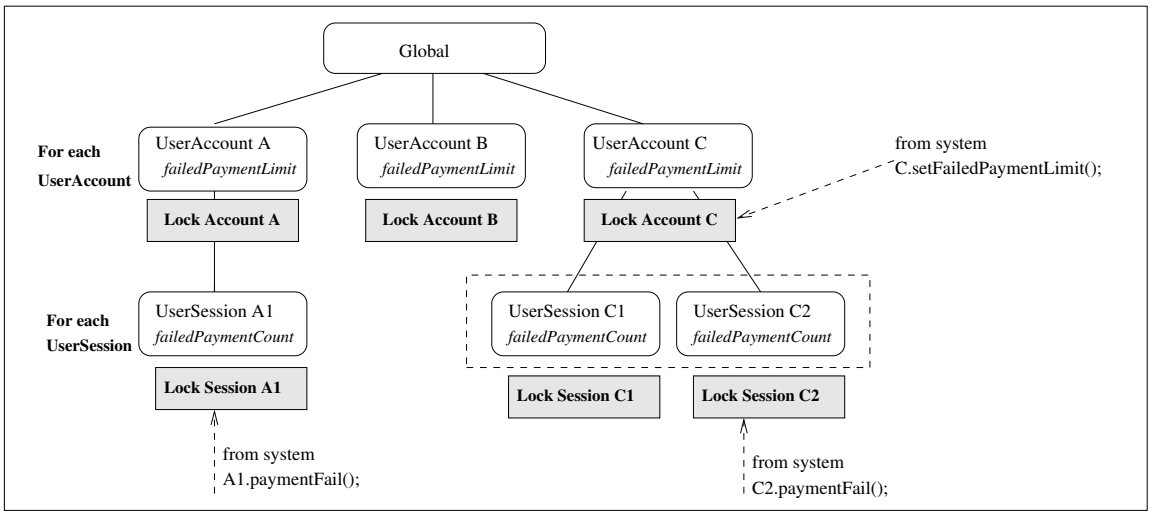

Figure 5: Locking on an object approach

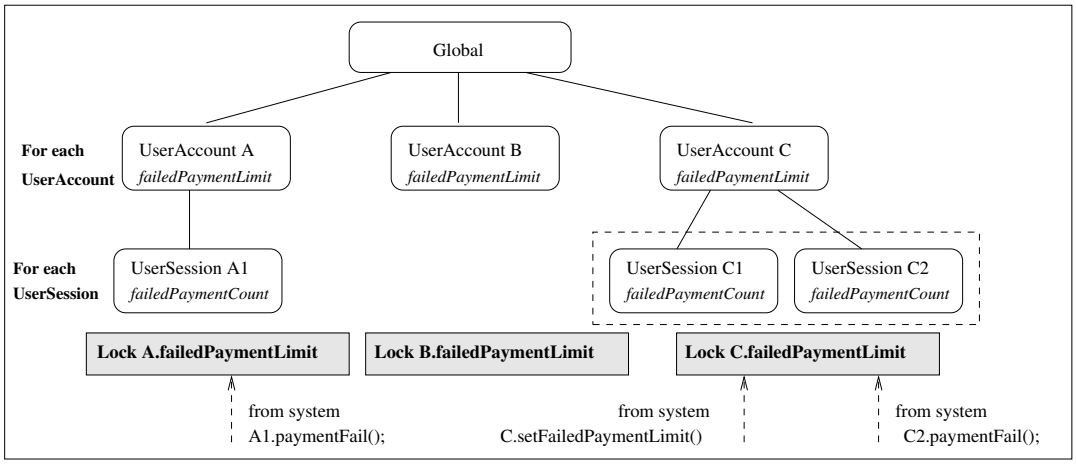

Figure 6: Locking on an action

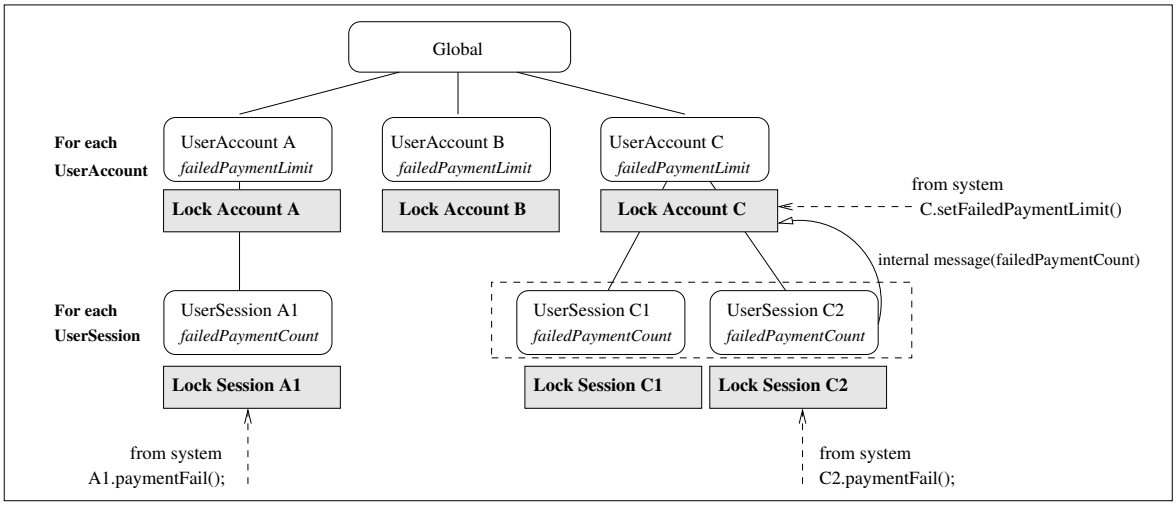

Figure 7: Internal Message Passing for Variable Updates in polyLarva

meaning that the current *failedpaymentCount* for session $C1$ will be passed to user account C which can then evaluate the comparison. The advantage obtained through internal message passing is that the message sent from other contexts is handled as an event and won't be processed until the lock on the user account monitor can be acquired. This prevents concurrent access on any of the state variables in the contexts. With message passing, the specification needs to be updated from that shown in Program 2.1. The user account context specification is extended to include support for the receipt of the internal message as shown in Program 3.2.

Program 3.2 Internal message passing specification

```
foreach useraccount:UserAccount {
    local failedPaymentLimit = 0

    on (useraccount a.setFailedPaymentLimit()) do
      update local (failedPaymentLimit)

    on (compareCountToLimit(sessionCount)) do
      compare local(failedPaymentLimit)
      to (sessionCount)

    foreach session:UserSession {
        local failedPaymentCount = 0


        on (session s.paymentFail()) do
          update local (failedPaymentCount)


        on (session s.paymentFail()) do
          send internal message to
          useraccount(
            compareCountToLimit(failedPaymentCount))
    }
}
```

4. CONCLUSIONS AND FUTURE WORK

We have presented the various alternatives considered when implementing support for the monitoring of concurrent systems in polyLarva. The various issues which each alternative brings to the runtime monitoring model were highlighted and communication via message passing was identified as a possible solution to these issues. With message passing, the different contexts in a polyLarva runtime monitor can transfer knowledge regarding the occurrence of particular system events. A parametrised approach to message passing also allows state variables to be transferred from one context to another. Future work includes the implementation of the suggested solution and the evaluation of its performance on a live case study.

5. REFERENCES

[1] H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. 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*. Springer, 2010.

[2] S. Colin and L. Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 525–555. Springer, 2004.

[3] C. Colombo, G. J. Pace, and P. Abela. Compensation-aware runtime monitoring. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 214–228. Springer, 2010.

[4] C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE Computer Society, 2009.

[5] M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.