Dynamic Automata in LARVA

John Paul Cassar
Dept. of Comp. Science
University of Malta
jcas0021@um.edu.mt

Christian Colombo
Dept. of Comp. Science
University of Malta
christian.colombo@um.edu.mt

Gordon Pace
Dept. of Comp. Science
University of Malta
gordon.pace@um.edu.mt

ABSTRACT

As computer systems become larger and more sophisticated, they bring about an increased level of possible execution paths and environment configurations, which, generally, cannot be reliably catered for by testing due to its inherent lack of coverage. As such, many developers are turning onto runtime software verification to be able to provide higher system quality assurance, intercepting undiscovered bugs as they arise. However, sophisticated systems tend to involve large specification properties and thus pose a considerable overhead when the states of such properties are *fully enumerated* to perform runtime verification. The problem is even more intricate with infinite-state properties where enumeration is not possible. A solution to this issue is through the use of on-the-fly state generation techniques where the next state is dynamically computed at runtime. In this paper, we present dLarva — an extension of the LARVA runtime verification tool supporting on-the-fly state-generating automata. This enables the definition of automata in a symbolic manner while also making it possible to traverse infinite state properties. To demonstrate the possibilities of dLarva, we provide an implementation of dLarva that accepts properties using regular expressions which are dynamically evaluated at runtime using derivatives. This implementation is used as the basis for a simple rule-based intrusion detection system for the AnomicFTPD FTP server.

1. INTRODUCTION

Computer systems are today, more than ever, present in our daily lives. They are increasingly becoming larger and more sophisticated, bringing about an increased level of possible execution paths and environment configurations, ultimately making them more error prone. It is becoming more evident that, due to lack of coverage, testing alone cannot provide the required guarantees. For this reason, developers are turning onto dynamic software verification, where the system's current execution path is monitored to check for any wrong behaviours [1]. Moreover, using *runtime verification*, which applies dynamic software verification while the system is running, offers the possibility to correct wrong behaviour occurring at runtime. This allows software developers to provide a higher system quality assurance.

Runtime verification properties can be encoded using various representations including formal logics and automata. Automata are generally represented by *completely enumerating* all the possible states and the transitions between them. This may pose an overhead when working with sophisticated systems as they tend to involve large specification properties in order to monitor their behaviour.

In this paper, we present an alternative approach where the user can specify states symbolically by using a transition function which, given the current system's configuration and an event, is able to produce the next automaton's state. This eliminates the need for explicit state enumeration since it uses an on-the-fly dynamic state generation approach. This makes it possible to traverse infinite state automata which brings about numerous advantages such as always having a concise representation of the property and its state at hand. Moreover, we posit that in certain cases, it could be easier to specify a dynamic automaton by specifying its transition function rather than specifying an equivalent static automaton which would need to be statically derived from its (possibly natural) dynamic definition.

In what follows, we first provide background knowledge of runtime verification and detail LARVA's current implementation in section 2 and then describe our system, dLarva, in section 3. Finally, before concluding this paper, in section 4 we describe an implementation of dLarva that accepts properties using simple regular expressions, which are then evaluated at runtime using derivatives. We also provide a simple case scenario where these are used to implement a rule-based intrusion detection system for the Anomic FTPd server.

2. BACKGROUND

2.1 Runtime Verification

Dynamic software verification is a branch of software verification where only the system's current execution path is monitored [1]. However, because of this, it can only guarantee correctness for one particular execution trace and not for all possible executions.

A dynamic software verification technique is *runtime software verification*, which apart from analysing the behaviour of the monitored system, is also able to act on the monitored system in order to attempt to correct any wrong behaviour

WOODSTOCK '97 El Paso, Texas USA

[1]. This typically occurs via the use of a monitor, which is able to receive events from the system and act upon them (see Figure 1).

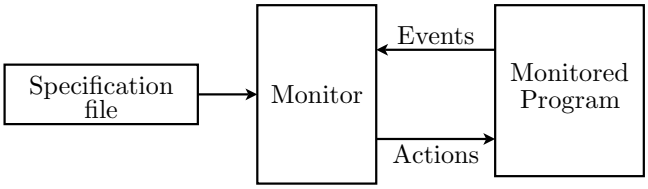


Figure 1: Generalisation of how runtime verification works

2.2 Larva

Larva [2] is a runtime verification tool for Java programs which uses event-triggered automata at the core of its specification language. Properties are defined as a set of automata, events, timers and channels. Events are either method-based events, timer-generated events, or channel-receive events. Method-based events can be set to trigger either on method calls; method termination; or the throwing or handling of an exception within a method. Timers enable Larva to monitor timed properties while channels are used to allow automata to communicate together.

Larva defines three different types of automaton states: accepting states, bad states, and normal states. When an automaton reaches an *accepting* state, this means that the property being monitored has been satisfied. On the other hand, *bad* states generate an alarm in Larva, since the monitored property was not respected and thus the system exhibited an unexpected behaviour. The last state type, *normal* states, are simply those states that are neither property breakers nor satisfiers.

Furthermore, Larva provides support for property contexts which essentially provide per object property definitions, namely, a copy of the monitor is generated for each new encountered object. A simple example of such definitions would be when, in a banking scenario, it is required to individually monitor account instances to verify whether any transactions requested should be marked as suspicious.

2.3 Dynamic Automata with Timers and Events

Dynamic Automata with Timers and Events (DATEs) [2] — the underlying mathematical framework of Larva — are *symbolic automata* with event-based transitions supporting timers and channels. Transitions from one state to another are performed based on the occurrence of specific events given that a condition on the current automaton's state (namely its internal variables and timer states) holds. Moreover, in DATEs, when an automaton performs a state transition, it is able to perform an *action* upon the monitored system (or the monitor itself) which may trigger yet another event. DATEs also provide the concept of *timers* which can be set, paused, and reset. They also provide asynchronously communicating *channels* that can be used by different automata to communicate with each other.

DATEs allow three event types: *system events* which are triggered by a system's action, *timer events* which fire at specific timer intervals, and *channel events* which are triggered upon receiving a message on a particular channel.

Since each property in DATEs is defined as an automaton,

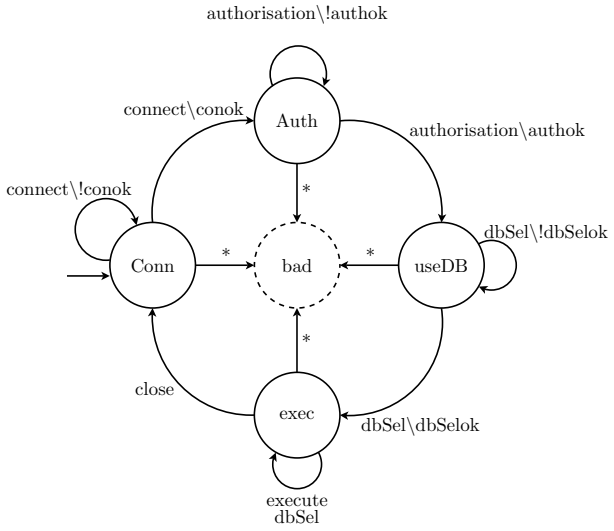


Figure 2: A session sequentiality property for a typical database scenario

DATEs are defined as being composed of a set of initial automata and a set of event-triggered automata (property) constructors. Thus, for example, in a banking scenario, a property can be defined for each instance of the customer class, in which case, the creation of a new customer class instance will also trigger the creation of a new automaton which will be associated to that particular instance.

To further illustrate the expressiveness of DATEs, consider a database monitoring scenario where it is required to monitor that commands to the database are executed in a particular order. One such sequence could be that after opening a connection to the database manager, the client must seek authentication and select the database to work with. The database session ends with the closure of the connection, at which point the client may start the sequence all over again. Assuming that all database interactions are performed within a provided `SQLRunner` class, the DATE property in Figure 2 captures the required property.

2.4 Example

Implementing the database scenario mentioned in the previous section can be done by first converting the property into a DATE automaton as shown in Figure 2. From this automaton, it is possible to construct the equivalent Larva specification, a snippet of which is given in Listing 1. The DATE in Figure 2 can be read as: (i) first the user connects to the database manager (*conn*), (ii) performs authentication (*auth*), (iii) selects the database (*useDB*), (iv) then either executes an SQL query or selects another database, and (v) finally the user closes the connection (at which point the user session can start again). If the user does not successfully perform any step, the user is expected to repeat the last step until it succeeds, otherwise the DATE enters the bad state *bad*. As such, all that is required is to enumerate all the possible states, and then describe the possible transitions. There is only one bad state: *bad* and four normal states: *auth*, *useDB*, *exec*, and *conn* (note that this state is marked as a starting state). These represent the normal state flow for our user session example. Moreover,

```
GLOBAL
{
 FOREACH (SQLRunner r)
 {
  EVENTS
  {
   connect(boolean conok) = {
    SQLRunner r1.openConnection()
    uponReturning(conok)
   } where {r = r1;}
   ...
  }
  PROPERTY userSessionProperty
  {
   STATES
   {
    BAD{bad}
    NORMAL{auth useDB exec}
    STARTING{conn}
   }
   TRANSITIONS
   {
    conn -> auth [connect\conok\]
    conn -> conn [connect\!conok\]
    %% this line serves as a catch all
    %% clause (it is the last of the three
    %% transitions and will thus be matched
    %% only if the above ones fail)
    conn -> bad  [allmethods\\]
    ...
   }
  }
 }
}
```

Listing 1: Extract of the LARVA specification for the property shown in Figure 2

using LARVA's contexts, we can monitor each `SQLRunner` class independently of each other.

2.5 LARVA Monitoring Architecture

LARVA's underlying monitoring architecture is quite sophisticated and involves three main phases:

Code generation: where LARVA takes a set of system properties specified as DATEs (similar to the one given in Listing 1) to create various source files including a single Java class per LARVA context. Each property is encoded as a Java method that emulates automata operations using a series of Java `if` statements and is placed into the relevant context class. When compiled, these are semantically equivalent to the intended DATE specification.

Code compilation and instrumentation: LARVA leaves code compilation and instrumentation to the AspectJ compiler which is a generic Java-based instrumenter that supports various instrumentation constructs. AspectJ requires all instrumentation code to be encapsulated within an aspect file which is then automatically compiled and instrumented at the required points in the monitored system.

Monitor running: When the monitor is in the running phase, it is executed as an integral part of the system. In fact, the monitor generated by LARVA is an in-process monitor and so it runs in the same thread as the one executing the main system.

3. DLARVA

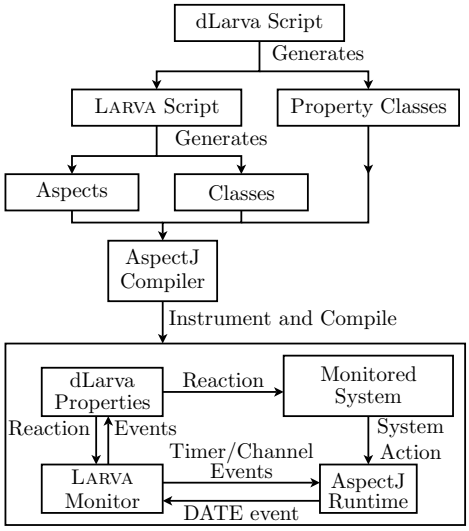


Figure 3: A detailed illustration of the dLarva architecture

3.1 On-the-fly state generation

The current LARVA implementation requires a priori knowledge of all the states and the possible transitions by requiring their *complete enumeration* before it compiles the DATE into a monitor. An alternative approach is to allow the user to specify states symbolically by using a transition function that, given the current system's configuration and an event, is able to produce the next automaton's state. As such, using our on-the-fly approach, dLarva extends LARVA in order to eliminate the need for an explicit enumeration of the automaton's states and transitions. Using this transition function, it is possible to encode all the operations performed within a typical DATE specification, mainly checking for event occurrences and performing any actions associated with that event including transitioning to a newly created state. In fact, the only thing that cannot be performed within such a function is the property initialisation. As such, a dLarva property can be defined to be a function pair consisting of the initialisation and state transition functions.

Consider, for example, a routing system where the network usage (packets sent and received per second per user) is monitored in order to prohibit users from hogging the network. If we were to take the packet status combinations as different states, the property would require a large state space, making it impractical to capture such a property. However, without having to resort to another state encoding, one can generate the states dynamically (by the transition function) as they occur, keeping only the latest one in memory. Since DATEs provide variable definitions, an alternative solution would be to monitor the packets by using counters thus reducing the naïve automaton's size. However, with on-the-fly state generation, the property can

be encoded in a more intuitive and explicit manner while also providing the possibility to define infinite state automata.

3.2 Property definition language

In order to define such a property function pair, a structured *language* with which the user can *program* the required behaviour is required. However since the user should already be familiar with Java (as the monitored system must be in Java) and such a structured language would inevitably provide capabilities comparable to the Java programming language itself, we opt to use Java itself as the function definition language.

3.3 dLarva specification format

Although Java is a rather logical choice for dLarva property specification, the full LARVA specification provides various abstractions which can be used to automatically generate various code segments related to generic monitor operations. As such, dLarva accepts a LARVA-like specification language where properties are specified using the aforementioned property function pair as shown below:

```
property {
  init(){ /*initialisation*/ }
  next(){ /*transition function*/ }
  {/*property-related helper definitions*/}
}
```

3.4 Access to dLarva functionality

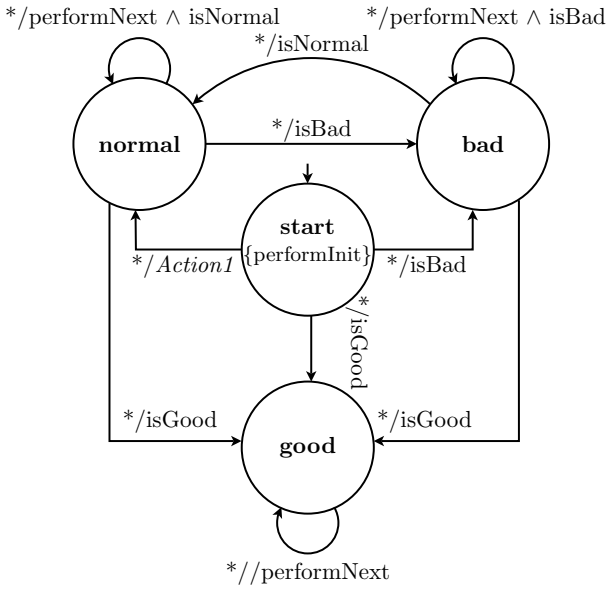
In order to adequately complement the chosen specification language, access to the dLarva functionality is provided via Java classes and interfaces. More specifically, the required property methods are mandated by a property interface while basic state functionality is provided as a generic Java class. Thus, creating a new bad state with a `String` identifier can be done using

```
new dLarva<String>("bad", StateType.BAD)
```

Access to the available events is provided as part of an automatically generated event interface which all properties inherit from (according to the context they are defined in). Moreover, dLarva also provides state history functionality which can be used to recall an arbitrary number of previously traversed states. This can be used to avoid unnecessary state generation by re-using previously generated states.

3.5 Integration with LARVA

For the scope of our implementation, dLarva integrates with LARVA by converting all dLarva specifications into LARVA specifications (the full architecture including the LARVA subset is shown in Figure 3). This is done by placing each property in a separate class, which is placed within the dLarva directory hierarchy in such a way that LARVA will be oblivious to the dLarva properties when running them. This is achieved by letting LARVA operate the property classes using the automaton described in Figure 4. Moreover, for logging purposes, access to the current dLarva state is provided via aspect injection into the relevant LARVA logging segments. This automaton uses two main helper functions, the `performInit` function which performs property initialisation, and the `performNext` function which controls state transitioning. Conceptually, if the `performNext` function returns true and LARVA is in the correct state *type*, then a transition onto the current state itself is made. Otherwise, if the state changes after calling the `performNext` function, the remaining transitions move LARVA into the correct state. The transition function defined by the user is used by the `performNext` function to identify whether a new state was generated. If the user-defined transition function returns the special value `null`, then no transition is made.



Action1 = (performNext ∨ true) ∧ isNormal

Figure 4: The DATE used to run dLarva on LARVA

4. CASE STUDY

4.1 Regular Expressions as Properties

A *regular expression* (RE) can be informally defined as a string that is used to match a given input string. In our case, we will be using regular expressions where the basic element is an *event*, and hence all input strings are event strings. An *empty event string* is a string that contains no events, and can thus be matched instantly to a given input string without consuming any part of it. Regular expressions are defines as having three operations: *concatenation* ($\cdot$) which allows to specify event sequentiality (their matching order), *alternation* ($+$) which provides the possibility to match either one of two different event sequences, and *Kleene-closure* (*) which allows to match zero or more occurrences of a particular event. For example, matching the regular expression $(a \cdot b) + c$, which is interpreted as "either wait for an a then for a b, or wait only for a c", with the input string `ab` will be successful but it will fail for the input string `bca`.

In our implementation, regular expression properties utilise the following syntax:

r,s	::=	`#{k}`	empty event string
	|	`![c]{k}`	error state
	|	`e[c]{k}`	e is an event
	|	`r ; s`	sequentiality
	|	`r + s`	alternation
	|	`r*`	Kleene-closure

where `c` is any valid Java boolean expression that must be satisfied when an event occurs for the RE event to match and `k` is any valid Java code that will be executed after a valid RE event match. Note that both `c` and `k` are optional.

Regular expression matching is performed using the derivatives approach [3] which can be informally described as being the residual of partially matching an RE to a single event. Thus, for example, the derivative of `a;b` with respect to a single event input `a` is `b` while the derivative for the same RE with respect to `b` would be $\emptyset$ which denotes the empty set.

Regular expression matching is extended to reflect the tri-state implementation provided by DATEs. All RE properties start in a *normal* state. When an expression is reduced to a single `#`, then the property is moved into a *good* state. On the other hand, if while evaluating the property, a `!` is encountered, the property is moved into a bad state.

Consider for example the following rule prohibiting multiple simultaneous downloads from a file server:

```
(startDl ; ( startDl ; ! + endDl))*
```

This can be loosely read as "trigger a bad state transition if after a `startDl` (start download) event, an `endDl` (end download) event is not received before getting another `startDl` event".

4.2 Intrusion Detection for an FTP server

As a practical application for the previously defined RE-based monitor, we have implemented a simple intrusion detection system (IDS) for the AnomicFTPD FTP server. The provided IDS caters, among other issues, for command sequentiality, brute force login attacks, and illegal file system reads and writes. Using regular expressions proved to be quite useful in defining the required rules. For example, in an FTP server, file renaming mandates calling `RNTO` (rename to) after issuing the `RNFR` (rename from) command which can be expressed using the property below.

```
regex {
 ((rnfr ; rnto) + (rnto ; ! {warnUser();})*
}
```

Slightly more complex rules can be achieved by defining Java helper methods that perform the required checks. For example, in the listing below, we specify that if an operation that can modify a system's file (such as uploading a file onto the server) is performed outside the users *home* path or in an *inaccessible* path, the session is forcefully terminated:

```
listen_to = {fileModifyOp}
regex{(
 fileModifyOp[isInvalidPutDir(path)
   || notInRoot(path)]
 ; ! {endSession();}
)*}
```

Note that we utilise a `listen_to` statement which instructs the monitor to discard events that are not `fileModifyOp`s. This allows properties to be defined in isolation of other system events.

5. CONCLUSIONS

Computer systems are increasing becoming larger and more sophisticated forcing software developers to turn onto runtime verification due to the inherent lack of coverage involved with testing. However, such systems tend to involve large specification properties which pose considerable overheads if the property's state space and the transitions between them are completely enumerated. For this reason, in this paper we have presented dLarva — an extension of the runtime verification tool Larva — which allows to define properties in a symbolical manner by using a transition function to generate the automata states on-the-fly. Since the states are generated on-the-fly, it makes it possible to traverse infinite state properties (automata). Moreover, dynamically generating the states brings about numerous advantages such as always having a concise representation of a property and its state while also reducing the overheads for complete enumeration down to a single state, since only the current state needs to be kept in memory. Furthermore, dLarva provides a more natural setting for defining properties that have natural dynamic definitions as was the case with our regular expression case study, where state evaluation is provided via derivatives.

6. REFERENCES

[1] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[2] Christian Colombo. Practical runtime monitoring with impact guarantees of java programs with real-time constraints. Master's thesis, University of Malta, 2008.

[3] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, 2009.