Proceedings of the 2016 IEEE International Conference on
Simulation, Modeling, and Programming for Autonomous Robots
San Francisco, USA, Dec 13-16, 2016

Run-time Verification of Regularly Expressed Behavioral Properties in Robotic Systems with Logic-Labeled Finite State Machines*

Vladimir Estivill-Castro[1] and René Hexel[1]

Abstract— **More and more, pre-runtime testing or verification of software in robotic systems is not feasible. For example, autonomous robots using learning behaviors to handle new environments. What sort of software architecture would then enable runtime-verification? We propose that logic-labeled finite-state machines (LLFSMs), jointly with regular expressions are a potent tool to describe acceptable and unacceptable system behaviors. Our architecture has the advantage of expressive power, since sometimes regular expressions are more succinct and comprehensible, while other times, the visual display of the LLFSM would be more accessible to requirements engineers. Moreover, LLFSMs have a prescribed schedule that enables their comprehensive testing and formal verification, and in similar ways, allows monitoring at run-time. The architecture facilitates verification of monitor LLFSMs and higher safety than alternatives such as `ROSRV`. Moreover, for a subset of *LTL* (co-safe *LTL*), LLFSMs can be synthesised from *LTL* constraints.**

I. INTRODUCTION AND CONTEXT

The fridge talks to the car that talks to the air-conditioning, which regulates the water heater that, in turn, negotiates with the pool-filter pump. These and many more distributed devices with their sensors and actuators make up the vision of the Internet of Things (IoT). Perhaps this vision is far-fetched for the everyday, individual home, but with autonomous robotics, it may become closer to reality soon for many commercial, manufacturing, or office environments, as well as urban services and infrastructure, and transport systems.

Many embedded and robotic systems are on the verge of revolutionising transport and communication industries. For example, in early 2016, Australia Post completed its first trials of drone-delivered parcels. Research by Ovum found that technology and telecommunication organisations globally spent over 13 billion USD between 2011 and 2015 in IoT-related purchases and investments. Gartner estimates there are 7.5 million smart devices in the IoT now across manufacturing, utilities, and transportation. Such a vast amount of connected things brings with it radical changes in society, creating new services and usage scenarios. However, as a result of insufficiencies in software quality many are alert to the dangers of malfunction [1], [2], [3].

Ensuring software quality is hard, not only because of the sheer size of software systems, but also because these systems are acquiring non-deterministic or evolving behaviors. A simple illustration comes from the context of autonomous mobile robots. Most software development depends on far too many assumptions about the environment that the robot will operate in. Some argue that all robotic behaviors are nothing more than emergent behaviors: the observable behavior is found in the intersection of the configuration of the body of the robot, the environment, and the software executing on the robot. Any modification of the robot's sensors, effectors/actuators, software/hardware, or the environment results in something completely different from the desired behavior. Moreover, the field of artificial intelligence (AI) is demonstrating that, although far from past ambitious goals of reproducing human intelligence, smart software can reason, learn, plan, and adapt behaviors, even with software that modifies its capabilities as it runs. Thus, how can one ensure proper operation, if the robot can travel and visit new environments, and/or manipulate and modify the environment, and even itself? Assuming there is no other external factor, such as unpredictable behavior of nearby humans or other robots, what about the latent failure of hardware components or software subsystems?

The challenge with software quality is that the rules regulating software are different than those of chemistry and physics that apply to other engineering disciplines. A piece of software of moderate size can be in millions of states, and can have millions of transitions to other states; so simply exploring all possible paths (or traces) of behavior that can be achieved by a system governed by software while in operation is, in most cases, completely infeasible.

Nevertheless, high-level software modelling has enabled constructed systems that are a magnificent feast of engineering, ensuring reliability and maintainability as well as safety and security. However, with robotic or embedded systems interacting with the world in real-time, response time becomes critical. Particularly when distributed, guaranteeing delivery of results or messages in bounded time poses big challenges to such systems, and temporal non-determinism can even have catastrophic consequences.

Hence the need for reliable infrastructure that facilitates timely communication between the modules making up the system. Enabling communication is predominantly the role of middleware, which unfortunately, often is governed by an event-driven approach that offers little or no timing guarantees. In such systems, modules typically subscribe by providing a call-back function that gets enacted in preemptive fashion when the publisher posts a message through the middleware. These approaches to module communication have been shown formally and in practice to be completely unsuitable to deliver information within a bounded time, thus are inadequate for real-time systems. Nevertheless, at the moment, the ROS (Robotic Operating System) middleware has

*This work was not supported by any organization

[1]School of ICT, Griffith University, Nathan, QLD 4111, Australia
{v.estivill-castro,r.hexel}@griffith.edu.au

become the bandwagon used by almost every robotic system and some embedded systems for inter-node communication, despite this fundamental drawback.

The discipline of software verification aims at producing absolute guarantees of correctness for all possible inputs and conditions at run time. Because of the state explosion we alluded before, this can usually only be achieved in the simplest of cases, but certainly is infeasible in environments where uncontrolled concurrency is the modus operandi.

Experts suggest that the software models for IoT protocols as well as for the behavior of the smart things are likely to be based on some sort of state machine [4]. This makes software development faster, as the behavior of the embedded device is specified on a higher level of abstraction than traditional programming languages. Logic-labeled finite-state machines (LLFSMs) allow executable models to be developed at such a high level of abstraction, while guaranteeing a deterministic, low-level execution schedule of an arrangement of multiple such machines, making formal verification computationally feasible [5]. Moreover, LLFSMs allow describing behavior more generically than the timed state-machines used by Rodney Brooks in his well-known subsumption architecture [6], and also Nilsson's famous teleo-reactive systems [7]. LLFSMs have further advantages in that they can be nested, but do not need to strictly follow the subsumption architecture, allowing suspension, resumption, restarting, and loading/unloading with clear semantics, unlike, for example, teleo-reactive systems.

Nevertheless, pre-runtime verification might not always be feasible. For example, in the context of autonomous robots, a humanoid on legs, may use AI techniques such as genetic algorithms to optimize its gait, or learn to move its head to track an object and keep it within its camera's field of vision. Many such emergent behaviors would no longer be considered correct or safe. Movements of the head can, for example, distort the center of gravity, making the robot prone to falling over, in which case it may be necessary to revert to a safer, simpler, default gait. Such a use case was presented in studies [8] using `ROSRV` [9], using a simulator of the LandShark UGV robot (an unmanned ground vehicle), where a movable turret can be put in such a position that certain accelerations would cause the robot to tip over.

We will demonstrate a software architecture based on LLFSMs, that not only recreates two of these case studies, but also shows how run-time verification of robotic and embedded systems can be modelled and implemented such that system safety can be ensured using an independent monitor machine that is able to replace, in real time, a faulty, learnt behavior with a previous behavior that is known to be safe. To this end we recreate analogous case studies using the ROS and Gazebo [10] version for the Komodo robot.

II. Logic-labeled Finite-State Machines

Most software engineers are familiar with the initial, event-driven postulation by Harel and Politi [11], where computation resides in states, but is paused until an event activates a transition. This was adopted by OMT [12] and later by UML [13], [14] despite that even from its origins, timing, concurrency and response time guarantees are essentially impossible [15], [16]. By contrast, in LLFSMs [17], nodes and modules are not idle waiting for events, but each node has a turn under a deterministic schedule [18]. Such an approach reduces concurrency to a sequential execution with predictable timing, where formal verification is feasible [5].

Figure 1 shows a simple LLFSM that creates a random, backwards and forwards stroll for the Komodo robot under Gazebo. The first point to note is that LLFSMs are executable models as in the strict sense of Model-Driven Software Development (MDSD). The LLFSMs are directly compiled into loadable libraries. States have ONENTRY, ONEXIT, and INTERNAL sections as in OMT or UML, but with very specific semantics. ONENTRY is executed once and only once upon arrival into the corresponding state. Similarly, ONEXIT is executed once, when a transition fires and the thread of control departs from that state.

In contrast to event-driven approaches, transition labels are Boolean expressions (for example, in Figure 1, `1==dice_roll` labels the transition from STROLL to MOVE_FORWARD). The model has a predicate analogous to the *timed* state-machines of the subsumption architecture: `after_ms(1000)` evaluates to true only if 1000ms have gone by (equivalent to `after(1)`, if 1s has passed).

After the execution of the ONENTRY, the transitions are evaluated in an order specified by the designer (in our example, the transition from STROLL to STOP is labeled `true`, so it would always fire, but has the lowest-priority – the ticks across define the sequence order). In fact, since the disjunction of the other two transitions, `1==dice_roll` and `1!=dice_roll` is always true, one of the two will always fire, never reaching the transition in question. The aim is not to build the most succinct machine that makes Komodo randomly walk in its environment, but to also illustrate the fundamentals of the modelling tools and execution model and the software architecture. Only when none of the transitions fire will the INTERNAL section execute.

Machines are arranged in sequence, and after one round of checking transitions and executing ONEXIT or INTERNAL, the execution token passes to the next LLFSM in the sequence. Thus, such machines execute concurrently, but within a predefined schedule. This makes their formal verification [5] and their testing [19] significantly simpler.

Since any `C/C++` code[1] can be inside a state, LLFSMs are Turing complete; thus any software could potentially be developed using this MDSD approach. We do not model control loops inside a single state, as that would starve other LLFSMs in the arrangement. The logic-expressions that label the transitions can correspond to queries to quite complex artificial reasoning and planning components. This capacity makes LLFSM transcend from reactive systems to deliberative systems. In the past, we have incorporated plausible logic, which can be compiled to `C` expressions, but recently we have incorporated Prolog programs [20], where

[1]We also have a Simple-C interpreter and compilers for Java and Swift.

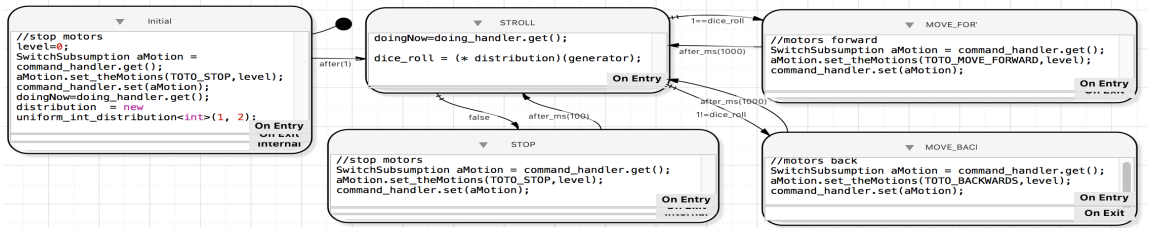


Fig. 1. A simple LLFSM that randomly swings the robot back and forward.

more care must be taken to maintain temporal consistency and execution determinism due to communication with a potentially long-running inference engine.

III. Communication Middleware

While an arrangement of LLFSMs executes in a deterministic sequence without preemption, allowing inherently atomic communication, we need to have the capacity to communicate with modules outside the arrangement. Importantly, we need to achieve this without creating extra complexity by putting independent modules into each other's sphere of control. To this end, we have proposed an object-oriented, shared-memory communication middleware named the `gusimplewhiteboard` [21] that allows low-overhead communication between modules (with an access performance similar to accessing a global variable) and implements the blackboard communication architecture [22].

We therefore communicate with other machines in the same arrangement or other processes (such as an inference engine), using the readers/writers interface provided by `gusimplewhiteboard`. In our example, `command_handler` is a command handler of a slot of class `SwitchSubsumption`. Reading the corresponding object from the status area of the whiteboard is as simple as `command_handler.get()` (copying object to the local context allowing setters and getters to be used on individual attributes). In particular, a certain motion command can be placed with a certain subsumption priority. This is what `aMotion.set_theMotions(TOT_STOP,level)` would do if state STOP were ever to execute. The object can then be posted back to the control area of the whiteboard, using `command_handler.set(aMotion)`.

Unlike most robotic middleware that uses a publisher/subscriber Push paradigm, we use a Pull-based approach of idempotent messages with the whiteboard [18], that allows the recipient to read messages at its own pace. In both approaches, the sender issues a non-blocking `add_message`$(msg : T)$. ROS and many others strictly work under a schema where the receiver subscribes to messages with a callback function f that gets called every time someone posts a corresponding message. This creates all sorts of coupling issues by placing the receiver sphere of control on the publisher: the subscriber must (at least on average) finish running f before the next post. By contrast, our whiteboard lets the receiver performs a `get_message`() returning the most recently posted $msg : T$.

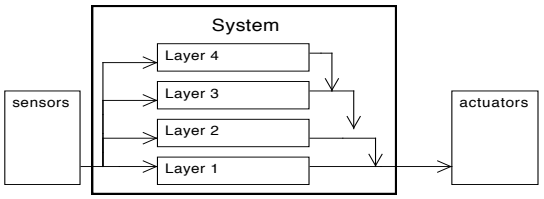
Furthermore, to profit fully from this decoupling, we promote that our message types (classes) are structured into two slots, *control* for messages sent, e.g., by the controller, and *status*, for messages reporting back the current status. Moreover, the whiteboard provides further decoupling by being data-centric, reducing the design complexity from $O(n^2)$ to $O(n)$ (where n is the number of components utilizing the middleware). Our `gusimplewhiteboard` implementation offers fast, lock-free atomic reader/writer semantics for a single writer and multiple readers, and, in conjunction with our LLFSM execution semantics, allows even multiple LLFSM writers to use lock-free synchronisation.

IV. Run-Time Verification using LLFSMs

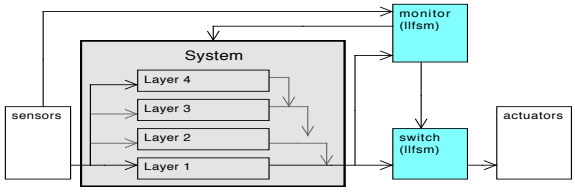
With our infrastructure being a superset of the subsumption architecture, we now proceed to show how to run-time verification can be implemented as a direct consequence. In its time, the subsumption architecture postulated a general framework to solve scalability issues given the complexity of robotic systems that seemed too brittle because of the slow sensor-plan-act paradigm. It was based on augmented finite-state machines: a machine is named a module, and, in fact, such machines were logic-labeled when presented with LISP-syntax [23]. The architecture provided three mechanisms: one to *suppress* an input to a module, another to *inhibit* the output to a module, and a third mechanism to *suspend/restart* a module. Nodes were organized in architectural layers, where upper layers worked on the basis that lower layers were correct and completely fulfilled their responsibilities. Figure 1 was written so it interacts with a subsumption `switch` [24]. In that example, this is absolutely not necessary, the strolling behavior could have directly sent commands to the motor. However, the existence of the switch immediately suggests how to position a monitor for run-time verification. We illustrate our idea with Figure 2. A complex system that is built from many modules and nodes can be treated as a black box that produces command configurations to actuators/effectors. In fact, such a system may not even have been built as a layered subsumption architecture (as LLFSMs allow more elaborate alternatives) [25].

We build the run-time verification system by adding two extra LLFSMs, a monitor that observes sensor inputs, but more importantly, the messages and the execution traces (paths of states and the transitions of the LLFSMs of the system) to detect undesired conditions. The other module is the switch we have alluded to before.

While the original subsumption architecture integrates layers to increase the competencies of the overall system, it always assumes the correctness of lower layers. We offer a radical new view on this by designing the nodes acting as



(a) Illustration of the layered subsumption architecture.



(b) With a monitor-LLFSM (and, potentially, another switch) system properties can be monitored and actions can be taken at run-time.

Fig. 2. Generic architecture of the safety monitor.

monitors to discover malfunctions of lower layers. Monitors perform one of the following alternatives if lower layers violate some requirement.

1) Inhibiting the output of lower layers and replacing it with newer, safer output.
2) Sending input to lower-level machines to steer them away, suspend them, or restart them.
3) Completely reconfigure the arrangement by unloading some of its LLFSMs and loading replacements.

That is, run-time verification LLFSMs can rebuild the lower layers which are not considered safe. With our current running example, a monitor machine that overlooks the earlier Komodo behavior (Figure 1) does not allow to go beyond a certain safety zone as shown in Figure 3.

This example is analogous to the `ROSRV` case study [26] of a robot staying between boundaries, but scaled down for illustration purposes [2]. The LLFSM from Figure 1 is just one instance of system shown in Figure 2a. We add the monitor (Figure 3) as the upper, blue box in Figure 2b, and place the subsumption switch in the lower box. Thus, our contribution here is that such a transformation is generic. It transforms any such system (receiving input from sensors drawn on the left and delivering outputs to actuators) to an expanded and safer system where a monitor software (and a subsumption switch) ensures fundamental safety properties at runtime.

The first observation is that the original system can now be abstracted and considered a black box from the perspective of the two blue modules added. Thus, these new systems have well defined inputs and outputs and are significantly simpler. They are developed under MDSD using LLFSMs, forming executable models that can be formally verified.

While this is analogous to the `RVMaster` in `ROSRV` [26], the `ROSRV` architecture requires to place a monitor for each publisher/subscriber pair in the ROS environment with slow message passing and high overhead. Moreover, the number of their monitors is $O(n^2)$ in the number of components (n).

[2]A video demonstrating this first case study is available at youtu.be/MVlghB0JZ1g.

By contrast, we place the commands from the subsumption switch to effectors and actuators as OO-messages on the `gu-simplewhiteboard`, providing compile-time type safety. The only LLFSM that has access to the class definitions of the final actuator messages is the subsumption switch. All others only have access to the abstraction and interface the subsumption switch offers, and therefore cannot access effectors and actuators directly. The subsumption switch only forwards commands that are deemed safe by the monitor. This not only adds safety, but also security that is completely impossible with ROS, as `roscore` globally advertises `ROS-topics` and `ROS-services`, causing safety-critical callbacks to also be exposed globally. This has been recognized as an important ROS runtime security loophole [26]. Our approach is therefore significantly more secure.

Naturally, LLFSMs in an arrangement performing *load*, *unload*, *suspend*, and *resume* operations are privileged. For our first prototype, we have chosen a mandatory access control approach as opposed to only discretionary control. Each operation affecting the execution of some LLFSM is an object with a security class. Each LLFSM is a subject, which is assigned a clearance. A typical mandatory rule is that system LLFSM do not have clearance for privileged operations, while monitoring LLFSMs do. The rules are established in a configuration file read when the arrangement is configured. While this would need to be complemented with other security mechanisms (e.g., to rule out a Trojan horse LLFSM), we want to focus here on the fact that no matter how complex the behavior exhibited by the underlying machine, its unprivileged nature makes it impossible to make the system behave unsafe or maliciously without acceptance by the monitor. Importantly, this facilitates protection from faults of well-intentioned components that have evolved though potentially independent constraints and objectives, and whose synergies could cause system malfunction.

Multilevel safety and security can be achieved by adding an additional layer of LLFSMs that have the privilege to control or replace monitors, also facilitating testing with different monitors/configurations [26]. That is, the transformation from Figure 2a to Figure 2b can be re-iterated as the designer sees fit. We illustrate this by extending our previous example. Consider that we move the arm at the top of the Komodo with several degrees of freedom. This can be archived by inverse kinematics or by computing plans to perform a certain trajectory for the gripper to grab or move an object. In fact, moving the arm requries learning, as movement up requires more power than moving down even by the same angle. Thus, we place a learning behavior to control the arm. But the arm motions may place the arm in undesirable positions relative to the body, as the position of the arm affect the centre of gravity of the entire robot, and thus the vehicle velocity may have to be reduced. Also undesirable is the arm impinging on the vehicle itself similar to the issues in the LandShark UGV `ROSRV` simulation [26].

To solve this problem, in addition to the machine that controls the motions of the robot around its environment we add a second machine to the system that moves the arm to

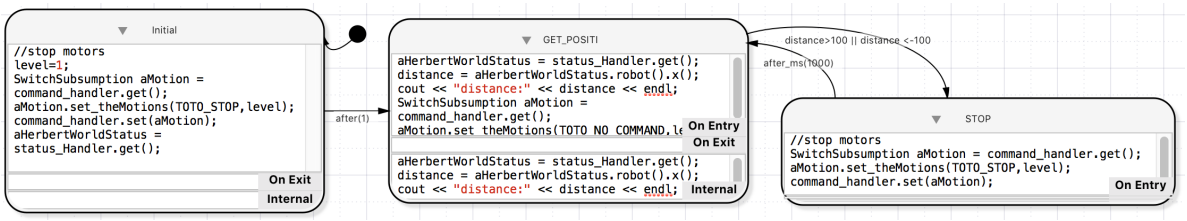


Fig. 3. A simple LLFSM that monitors the Komodo not to escape a region.

a different position. We only need to add a second monitor that observes the location of the end effector of the arm and restricts the speeds of the robot wheels to smaller ranges in proportion to arm extension[3]. This illustrates that our proposed architecture compares favorably with ROSRV, where security, scalability, and formal verification were identified as issues for further work [26]. Let's focus on the scalability issue. It would be beneficial that the switch in the architecture be re-factored such that not every new property or monitor adds a new switch module. Similarly, the monitor module could potentially be the merge of several monitors. However, there is an advantage of small switch or monitor modules. The LLFSMs that act as switch and monitor can be small enough that they can be formally verified in our architecture.

There are further advantages of our proposal, namely the specification of conditions to monitor can naturally and automatically be derived and expressed from the models of the LLFSMs in model-driven development style.

V. Deliberative Embedded System

Our next example derives from a Prolog textbook [27] and is a simple illustration that reasoning can be integrated with LLFSMs [20]. The scenario is an intersection with 6 different actuator lights, organised into 2 sets named East-West (EW) and North-South (NS) of 3 lights each: red, amber and green. We look into the advanced version where one sensor in the EW-direction enables providing priority to the NS-direction in the absence of a car waiting in the EW-direction. The complete model of this case study appears in Figure 4. Additionally, 4 LLFSM are derived directly from the Prolog program in Figure 4a and constitute wrappers for invoking Prolog queries. They run asynchronously in their own `clfsm` process, separate from the arrangement that executes the other three machines (the timer in Figure 4b, the NS light in Figure 4c, and the EW light in Figure 4d). We can perform run-time verification of fundamental properties such as "both directions do not show green simultaneously" with the method proposed earlier. The corresponding subsumption switch is shown in Figure 5. The monitoring LLFSM has a well-defined pattern, ensuring a valid state combination of the two controller machines. Our infrastructure allows the monitors to capture state changes testing for a validity condition. The monitors will unload and load new behaviors if there is a violation of the condition. Importantly, the LLFSMs can be constructed (that is, generated) at a high level, using a graphical editor. Figure 6 presents a simple illustration of a section of the monitoring LLFSM that observes the condition that the machine controlling the NS-direction is not displaying a green light (GREEN_ON_NS) at the same time as the machine controlling the EW-direction (GREEN_ON_EW). The state MONITOR_STATE collects the current states of the two named machines (the NS-controller and the EW-controller). If any of these machines have changed state from their previous state, the monitor moves to its TEST state, where the names of the states are obtained and compared to evaluate the condition in the Boolean variable `both_r_green`. If the value of this variable is `false`, then it just loops back to state MONITOR_STATE. Otherwise, the LOAD_BLINKING initiates a safe behavior, setting both lights to flashing amber (Figure 7). Our case study is more sophisticated, as after 20 seconds of flashing, the monitor LLFSM replaces the main arrangement with default, safe controllers for both directions, replacing the two faulty controllers and the flashing behavior with the default controller that is known to be safe. This illustrates that a new behavior can be incorporated as well as removed at runtime[4].

VI. Regular Expressions and co-safe *LTL*

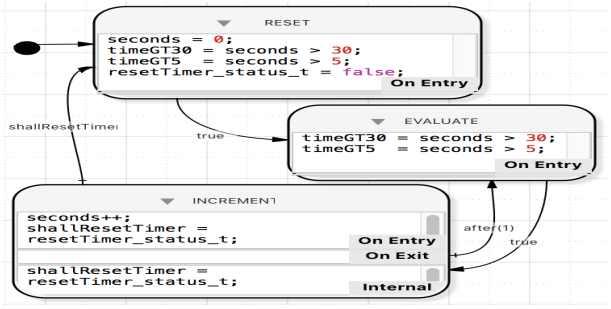
Regular expressions correspond directly with finite-state machines (FSM) (and deterministic FSMs are equivalent to non-deterministic FSMs). Thus, it should be clear that we can build an executable LLFSM that monitors a condition expressed by a regular expression. That is, the monitor LLFSM can be an observer that monitors whether a sequence of states and/or values in the whiteboard belong to a regular language (and since the complement of a regular language is regular, we can monitor positive or negative conditions of execution traces). Moreover, we argue here that LLFSMs provide a very suitable equivalent as visual models of such conditions. We can profit from the equivalence between regular languages and finite-state machines. In particular, 1) the deterministic schedule of the `clfsm` tool [21] and 2) the fact that the monitor LLFSM can inspect the states of system LLFSMs enables us to define precisely what the alphabet is to build such regular expressions. The basic elements of the logic expressions we will be using are predicates regarding states of the LLFSMs that are in the *System* box of Figure 2b. Moreover, the monitor LLFSM in Figure 2b is a model completely constructed from such logic expressions, significantly

[3]We also have a video (youtu.be/_3VylSPQoEE) illustrating this scenario where two learning behaviors are operating simultaneously and independently, and thus a monitor is required to intervene where their synergy results in dangerous positions of the arm.

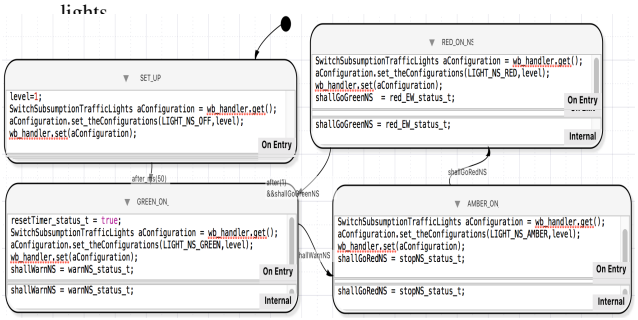
[4]A video for this case study is also avilable (youtu.be/HFm6fbZ6lkg).

```
causeew(warnew,greenew,timerGT30).
causeew(warnew,greenew,not_detectEW)
causens(warnns,greenns,timerGT30,detectEW)
causeew(stopew,amberew,timerGT5).
causens(stopns,amberns,timerGT5).
```

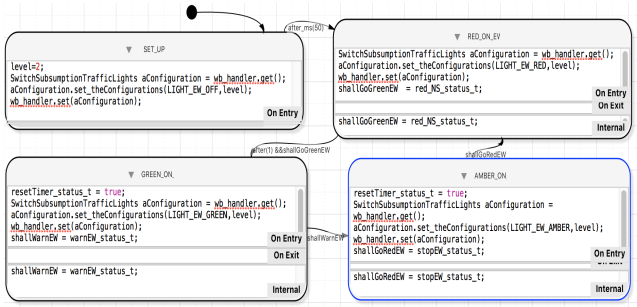
(a) Logic-programming description of requirements to swap lights



(b) The Timer.



(c) North-South (NS) controlling LLFSM.



(d) East-West (EW) controlling LLFSM.

Fig. 4. Complete software of the traffic lights with sensor.

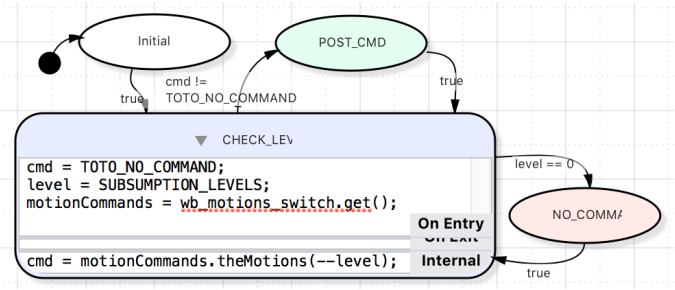


Fig. 5. LLFSM for the subsumption switch of the traffic lights.

simplifying the implementation of such monitoring LLFSMs. The following grammar describes the basic, logic constructs to express forbidden or compulsory conditions by monitoring LLFSMs. The first building blocks are formulas that work as atomic propositions.

<formula> → <term> | (<formula> <connective> <formula>) | `not` (<formula>)
<term> → <state_formula> | <wb_variable_formula>
<state_formula> → <machine_name> @ <state_name>
<wb_variable_formula> → <value> == <wb_variable_name>
<connective> → ∧ | ∨

An example of the term that expresses that in the LLFSM arrangement of the traffic lights the two controlling machines cannot both be in their respective states where they set their respective lights to green is the following formula.

```
!(light_ns_subsumption @ GREEN_ON_NS ∧
light_ns_subsumption @ GREEN_ON_EW)
```

For many embedded or robotic system, the safety requirements can be expressed using these fundamental terms. The designer of runtime verification LLFSMs selects states and whiteboard variables in order to build corresponding formulas (e.g. using a GUI tool to compose these).

It should be clear at this stage that our logic for forbidden/enforceable formulas is structurally and semantically equivalent to propositional logic. As we already mentioned, an LLFSM that checks such a formula is basically including the corresponding formula in a transition of a state that has collected the necessary information. This was one of the points of showing the pattern in Figure 6. Such monitoring LLFSMs, although synthesized automatically, could become models reviewed by human designers. Most of the conditions or safety rules we have found in case studies on system safety seem to be of this form. However, we note that in some situations the forbidden/enforceable scenario more closely corresponds to a sequence, namely the a trace of a behavior.

Those scenarios correspond to sequences of formulas. For example, with the traffic lights, the controller for one direction must cycle from green to amber, then to red, and back to green. For such a loop, the enforceable behavior can be specified by the regular expression `(red green amber)`*. Moreover, the equivalence of regular expressions and non-deterministic automata (and thus, deterministic automata) shows that we can construct monitoring LLFSMs automatically that verify the a trace of basic formulas (about states and whiteboard variables) against a regular language where the alphabet are basic formulas.

While we currently can directly move from LLFSMs to regular expressions about compulsory behaviors or forbidden behaviors, we are also incorporating potentially more sophisticated mechanisms to express component and system safety

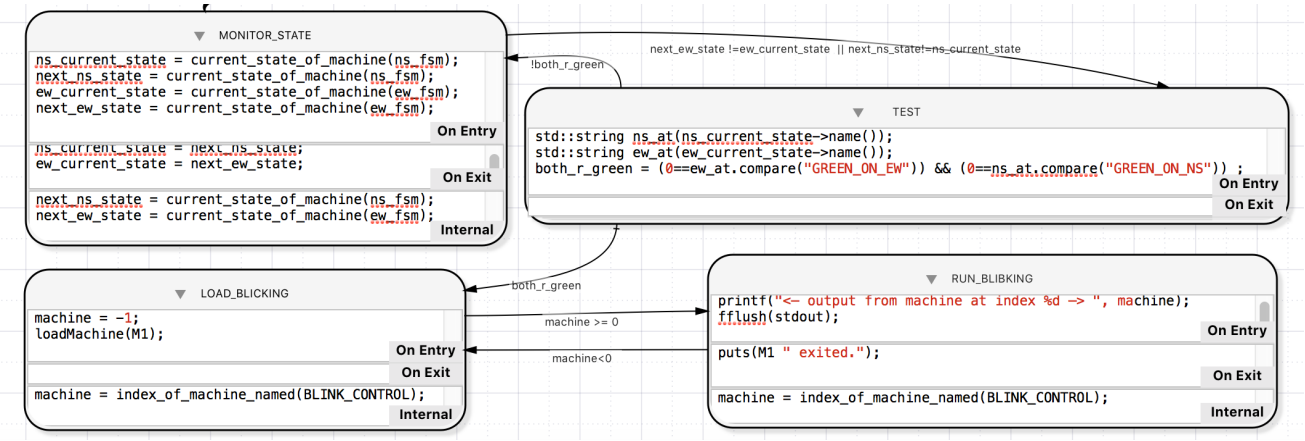


Fig. 6. Fragment of the monitoring LLFSM for rearranging the controllers of the traffic lights if it is detected controllers in both directions are in the light green state. If the condition holds, a blinking behavior is loaded.

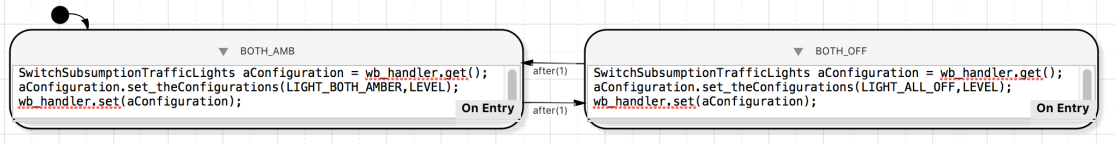


Fig. 7. Blinking LLFSM flashes amber in both directions (1 second on and off).

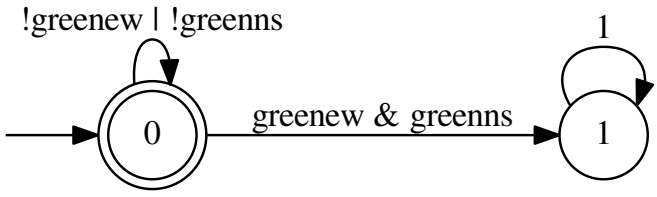


Fig. 8. A Büchi automaton derived using `ltl2tgba` that is directly implementable as an LLFSM.

conditions. Linear temporal logic or linear-time temporal logic (*LTL*) is a modal, temporal logic with modalities referring to time commonly used in formal verification. We are experimenting with the package `spot` and in particular with its tools and capacity to transform *LTL* into Büchi automata. For example, in our case study of the traffic light control, using the command-line tool `ltl2tgba` [28] we can produce a deterministic and complete automaton for the *LTL* formula that verifies that is is always the case that when one LLFSM is in a green-light state, the other one is not.

```
ltl2tgba -f 'G(greenew->!greenns) &
G(greenns->!greenew)' --complete -d
```

This Büchi automaton is deterministic and also, by requesting to be complete, we obtain a rejection state labeled state 1 (Figure 8). Thus, the monitor LLFSMs for this case can be also automatically composed using the `spot` set of libraries.

Currently we favor the use of regular expressions or their equivalent LLFSM model. It is quite delicate to construct expressions in *LTL* for some rather intuitive behaviors whose visual presentation as LLFSM is, at least, very transparent. Consider the example above that the controller for the EW set of lights must always cycle trough green followed by amber, and red, and then back to green. The *LTL* formula for `spot` to produce the deterministic Büchi automaton that displays this cycle of 3 states (while also creating non-accepting states if the constraint in the behavior is violated) is the following.

```
G( (greenew & !amberew & !redew) | (amberew
& !greenew & !redew) | (redew & !greenew &
!amberew))
& G((greenew & !amberew & !redew)->X ( amberew &
!greenew & !redew))
& G((amberew & !greenew & !redew)->X ( redew &
!greenew & !amberew))
& G((redew & !greenew & !amberew)->X ( greenew &
!redew & !amberew))
```

The `spot` package allows the derivation of monitors (option `-M` for `ltl2tgba`); and we could use the `spot` libraries to automatically synthesize the monitor for our architecture directly as an LLFSM. In several robotic systems with planning and manipulation tasks, the *LTL* subset named *co-safe LTL* has been used [29] because it produces deterministic finite-automate [30]. Here again, Büchi automata can be directly modeled by LLFSMs. Our architecture can confirm co-safe *LTL* formulae, but if the formula has the modal operator for "eventually", the monitoring LLFSM can not warranty when such condition is met (in the case of task planning it enables to recognize a plan has found a goal meeting the co-safe *LTL* condition). Nevertheless, we expect to experiment with the fact that formal verification can be invoked from an LLFSM using the `spot` libraries. That is, the architecture enables the possibility of running, on a different thread, a model checking exercise on a subset of LLFSMs in the system and a *LTL* formula, before a particular subset of LLFSMs is actually loaded and run.

VII. CONCLUSIONS

This paper has shown that logic-labeled finite-state machines, which have been very fruitful for model-driven software development as they constitute executable models,

can be used to enhance a subsumption-style architecture to perform runtime monitoring and verification of behavior. Moreover, preventive actions can be taken by such monitors. In particular we have illustrated that LLFSMs can be loaded and unloaded for re-building the system. This is achieved in conjunction with a shared-memory middleware that is simple, real-time, and object-oriented. The deterministic concurrent scheduling of an arrangement of LLFSMs (that can be designed using a GUI) makes execution traces predictable and feasible to be tested against regular expressions, or directly by monitoring LLFSMs inspecting the underlying system LLFSMs. LLFSMs are compiled providing static type safety which is far more secure than the alterantives based on ROS, like `ROSRV`.

We already can incorporate into an LLFSM system other processes whose duration is unpredictable, like reasoning (with examples such as Prolog) or planning (with libraries for task planning). Therefore, our further work is to use formal verification libraries invoked from monitoring LLFSMs into subsets of LLFSMs and *LTL* formulas to actually perform full verification as the embedded or robotic system is running on a different thread, before actually loading and running LLFSMs under scrutiny. The expectation is that, the reduction of complexity and concurrency provided by arrangements of LLFSMs shall result in feasible runtime verification using *LTL*.

REFERENCES

[1] M. Weiss, J. Eidson, C. Barry, D. Broman, L. Goldin, B. Iannucci, E. A. Lee, and K. Stanton, "Time-aware applications, computers, and communication systems (TAACCS)," The National Institute of Standards and Technology (NIST), U.S. Department of Commerce, Tech. Rep. Technical Note 1867, February 2015.

[2] J. Sametinger, J. Rozenblit, R. Lysecky, and P. Ott, "Security challenges for medical devices," *Commun. ACM*, vol. 58, no. 4, pp. 74–82, Mar. 2015.

[3] A. N. Srivastava and J. Schumann, "Software health management: A necessity for safety critical systems," *Innov. Syst. Softw. Eng.*, vol. 9, no. 4, pp. 219–233, Dec. 2013.

[4] R. Bryce and R. Kuhn, "Software testing [guest editors' introduction]," *Computer*, vol. 47, no. 2, pp. 21–22, Feb 2014.

[5] V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth, "Efficient modelling of embedded software systems and their verification," in *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, K. R. Leung and P. Muenchaisri, Eds. Hong Kong: IEEE Computer Society, Conference Publishing Services, December 2012, pp. 428–433.

[6] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, Mar 1986.

[7] N. J. Nilsson, "Teleo-reactive programs and the triple-tower architecture," *Electron. Trans. Artif. Intell.*, vol. 5, no. B, pp. 99–110, 2001.

[8] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd ed. Cambridge, MA: MIT Press, 2011.

[9] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[10] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, vol. 3. IEEE, 2004, pp. 2149–2154.

[11] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[12] J. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

[13] S. J. Mellor and M. Balcer, *Executable UML: A foundation for model-driven architecture*. Reading, MA: Addison-Wesley Publishing Co., 2002.

[14] M. Samek, *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newton, MA, USA: Newnes, 2008.

[15] H. Kopetz, M. Braun, C. Ebner, D. Millinger, R. Nossal, A. Schedl *et al.*, "The design of large real-time systems: the time-triggered approach," in *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*. IEEE, 1995, pp. 182–187.

[16] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," vol. 6, pp. 254–280, 1984.

[17] V. Estivill-Castro and R. Hexel, "Correctness by construction with logic-labeled finite-state machines – comparison with Event-B," in *Proc. 23rd Australasian Software Engineering Conference (ASWEC)*, L. Zhu and J. Steel, Eds. Milsons Point, Sydney, NSW, Australia: IEEE Computer Society Conference Publishing Services (CPS), April 7th-10th 2014, pp. 38–47.

[18] ——, "Logic labelled finite-state machines and control/status pull technology for model-driven engineering of robotic behaviours," in *Proc. 26th International Conference on Software & Systems Engineering and their Applications*, May 2015.

[19] V. Estivill-Castro, R. Hexel, and J. Stover, "Modeling, validation, and continuous integration of software behaviours for embedded systems," in *9th IEEE European Modelling Symposium*, D. Al-Dabass, G. Romero, A. Orsoni, and A. Pantelous, Eds., Madrid, Spain, October 6th-8th 2015, pp. 89–95.

[20] V. Estivill-Castro and R. Hexel, "Architecture for logic programing with arrangements of finite-state machines," in *First Workshop on Declarative Cyber-Physical Systems (DCPS) at Cyber-Physical Systems*, Vienna, Austria, April 12th 2016, to appear.

[21] V. Estivill-Castro, R. Hexel, and C. Lusty, "High performance relaying of C++11 objects across processes and logic-labeled finite-state machines," in *Simulation, Modeling, and Programming for Autonomous Robots - 4th International Conference, SIMPAR 2014*, ser. Lecture Notes in Computer Science, D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, Eds., vol. 8810. Bergamo, Italy: Springer, October 20th-23rd 2014, pp. 182–194.

[22] B. Hayes-Roth, "A blackboard architecture for control," in *Distributed Artificial Intelligence*, A. H. Bond and L. Gasser, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, pp. 505–540.

[23] M. Mataric, "Integration of representation into goal-driven behavior-based robots," *Robotics and Automation, IEEE Transactions on*, vol. 8, no. 3, pp. 304 –312, jun 1992.

[24] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud, "Robotic software integration using MARIE," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 055–060, March 2006.

[25] D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock, "Requirements engineering via non-monotonic logics and state diagrams," in *Evaluation of Novel Approaches to Software Engineering*, vol. 230. Berlin: Springer Verlag, 2011, pp. 121–135.

[26] J. Huang, C. Erdogan, Y. Zhang, B. M. Moore, Q. Luo, A. Sundaresan, and G. Roşu, "ROSRV: runtime verification for robots," in *Runtime Verification - 5th International Conference, RV*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. A. Smolka, Eds., vol. 8734. Springer, September 22nd-25th 2014, pp. 247–254.

[27] D. Maier and D. S. Warren, *Computing with Logic: Logic Programming with Prolog*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1988.

[28] A. Duret-Lutz, "LTL translation improvements in Spot 1.0," *International Journal on Critical Computer-Based Systems*, vol. 5, no. 1/2, pp. 31–54, Mar. 2014.

[29] K. He, M. Lahijanian, L. E. Kavraki, and M. Y. Vardi, "Towards manipulation planning with temporal logic specifications," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 346–352.

[30] O. Kupferman and Y. M. Vardi, "Model checking of safety properties," *Form. Methods Syst. Des.*, vol. 19, no. 3, pp. 291–314, Oct. 2001.