High Performance Relaying of `C++11` Objects Across Processes and Logic-Labeled Finite-State Machines

Vlad Estivill-Castro, René Hexel, and Carl Lusty

Machine Intelligence and Pattern Analysis Laboratory (MiPal)
Griffith University, Nathan 4111, Qld, Australia

Abstract. We present `gusimplewhiteboard`, a software architecture analogous to `ROS:services` and `ROS:messages`, that enables the construction and extremely efficient inter-process relaying of message-types as `C++11` objects, All `gusimplewhiteboard` objects reside in shared memory. Moreover, our principle is to use idempotent message communication, in direct contrast to previously released platforms for robotic-module communication, that are based on an event-driven subscriber model that queues and multi-threads. We combine this with compiled, time-triggered, logic-labeled finite state machines (*llfsms*) the are executed concurrently, but scheduled sequentially, in an extremely efficient manner, removing all race conditions and requirements for explicit synchronisation. Together, these tools enable effective robotic behaviour design, where arrangements of *llfsms* can be organised as hierarchies of machines and submachines, enabling composition of very complex systems. They have proven to be very powerful for Model-Driven Development, capable of simulation, validation, and formal verification.

1 Introduction

Since its inception, the *blackboard control architecture* [17], has become ubiquitous as a mechanism to integrate cognitive processes, behaviours, and problem-solving. It has also become central to agent architectures and publish/subscribe patterns among the software engineering community. Over and above the publisher/subscriber pattern, a blackboard allows a further level of decoupling by being data-centric (rather than component-centric). The provider may supply information for unknown (possibly inactive) consumers without the need to be aware of a consumer's interface, only the interface to the blackboard is necessary.

From a software architecture perspective, the flexibility of a blackboard is also incorporated into the notion of a broker, enabling a sender to issue what we will refer to as `add_Message`($msg : T$), a non-blocking call that may optionally include additional information, e.g. a sender signature, a timestamp, or an event counter that records the belief the sender has of the currency of the message. There are essentially two modes for retrieving a message.

`subscribe`(T, f): The receiver subscribes to messages of a certain *type* T (of an implied *class*) and essentially goes to sleep. Subscription includes the name f of a function. The blackboard will notify the receiver of the message msg every time someone posts for the given *class* T by invoking $f(msg)$ (usually queued in a *type* T specific thread). This is typically called PUSH technology.

`get_Message`(T)**:** The receiver issues a `get_message` to the blackboard that supplies the latest *msg* received so far for the *type* T. This is usually called PULL.

The *type* identifies the communication channel, and in `ROS`' PUSH technology is named a `ROS::topic`. The modules posting (publishing) or getting (subscribing to) messages are called *nodes*. The data structures available for messages (and described in `msg` files) are restricted to a simplified message description language, because `ROS` aims at supporting cross-programming-language communication. This architecture is common in robotic systems and other robotic projects have produced similar infrastructures: `Carmen` (carmen.sf.net), `Microsoft Robotics Studio` (msdn.microsoft.com/en-us/robotics/), `MIRA` [6], `MOOS` (www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php), `Orca` (orca-robotics.sf.net), `Orocos` (www.orocos.org), `Player` (playerstage.sf.net), and `YARP` (http://eris.liralab.it/yarp/).

Such robotic-system architectures organise many modules under different paradigms. One of them is the sense-decide-act cycle. Here, sensors post information onto the blackboard. This information may be processed by decision makers (even as complicated as planners) that then publish commands to actuator modules. The blackboard enables very flexible information processing about the state of the world that is supplied by sensors. For example, if sensors are noisy, then an intermediate *filter* (such as a Kalman filter) can be placed in between the raw posting of the sensor and the decision maker. This mechanism can be extended to a whole pipeline of publishers and subscribers between the sensor data and the final actuators. The features of `gusimplewhiteboard` include:

1. Completely `C++11` and POSIX compliant; thus, platform independent: used on Mac OS X (Mountain Lion), LINUX 13.10, Aldebaran Nao 1.14.3, `Webots` 7.1, the Raspberry Pi (www.raspberrypi.org), and Lego NXT.
2. Released as a `ROS`:`catkin` package (`mipal.net.au/downloads.php`).
3. Extremely fast performance for `add_Message` and `get_Message`, intra-process as well as inter-process.
4. Completely OO-compliant. The classes that can be used are not restricted, the full data-structure mechanisms of `C++11` are available.
5. Very clear semantics that removes lots of issues of concurrency control.

2 Challenges of Inter-Module Communication

Control modules in charge of robot behaviour rely heavily upon predictable communication latency. Even very standard algorithms in robotics, like the Kalman filter, would be significantly less effective if the time between the reading of an observation and the execution of the filtering step was randomly perturbed. The motion model would not be able to make sufficiently accurate predictions, and the integration of information provided by the next observation with the prediction would be jeopardised. Similarly, issuing commands to actuators heavily depends on the issuer having reasonably accurate information of the position of the actuator at the time of issuing the next command. If sensor information or control commands are unboundedly delayed, the safety of actions can be seriously compromised. Thus, the emergence of compliant actuators is not enough: the software architecture is equally responsible for safe operation.

Delays not only depend on the type of channel used by the blackboard architecture, but also the determinism (or lack thereof) of the concurrency model used. Our software architecture proposes to schedule publishers and subscribers sequentially (see Section 3). In such a model, the use of `subscribe` can be minimised or avoided. The use of PUSH technology (for example `ROS`) results in the classical *producer-consumer problem* (or *bounded-buffer* problem) of multi-process synchronisation and its associated challenges (e.g. critical sections, and message queues). Throughout, the call-back function f must be fast enough to terminate and be ready to process the next invocation. Deadlocks, live-locks, starvation, or similar concurrency issues can result in catastrophic consequences. But even in the best of scenarios, a traditional, multi-threaded operating system cannot usually guarantee a schedule that will meet deadlines for all tasks; hence, there is simply no guarantee when an specific event will be handled. With this in mind, it is somewhat surprising that the robotics community is adopting this approach by endorsing `ROS` which for inter-process communciation uses the network stack to relay messages (wiki.ros.org/ROS/TCPROS), and even *nodelets* (that only work within the same process) use a `subscribe` and queuing mechanism. In fact, the issues of using networking infrastructure as the transport layer has prompted some developers to state that `ROS` was originally intended for a single host, and not necessarily suitable for distributed communication [14]. However, others (`MOOS`, `Microsoft Robotics Studio`, `naoqi` non-local modules, etc.) have also built on top of TCP/IP, nondeterministic, multi-threaded processing and/or event-handling of the underlying operating systems; and thus face the very same challenges.

3 Arrangements of Logic-Labeled Finite-State Machines

We now present `clfsm`, our compiler and scheduler for arrangements of *llfsms*. Since Harel's seminal work [16], finite-state machines (FSMs) have become ubiquitous models of system behaviour. Finite-state machines guide coding and model-based system development [29,30]. Among the software engineering community, FSMs machines are ubiquitous. Studies have demonstrated that, jointly with class-diagrams, state-charts are the top most used UML artefact [7,27]. FSMs are the best understood tool for model-driven development in software engineering [24]. For modelling the behaviour of robots, variants of FSMs have also become fundamental. *Augmented* FSMs are the basis of the subsumption [4] and reactive software architectures [23]. In RoboCup, several teams and their research groups use FSMs to model and implement behaviours [18,22,25,28].

Tools for deploying systems using FSMs include the robotics simulator `Webots` (offering *BotStudio*) [26], *StateWORKS* [30] and *MathWorks*® *StateFlow*, `ROS` has a tool named `smach` [3] (wiki.ros.org/smach), Qt's *State Machine Framework* (qt-project.org/doc/qt-5/statemachine-api.html) that is based on the W3C's state chart XML (SCXML) (www.w3.org/TR/scxml/), the `rFSM` [18] framework in `Lua`, and the `boost` library `Meta State Machine` and `StateChart` templates.

Key characteristics of `clfsm` include the following.

1. Complete POSIX and `C++11` compliance.
2. Open source `catkin` `ROS` package release (`mipal.net.au/downloads.php`).
3. Transitions are labeled by Boolean expressions (not events), facilitating formal verification and eliminating all need for concerns about event queues.
4. Transition labels are arbitrary `C++11` Boolean expressions, enabling reasoning into what may otherwise seem a purely reactive architecture.
5. Handling of machines constructed with states that have UML 2.0 (or SCXML) **OnEntry**, **OnExit**, and **Internal** sections with clear semantics.
6. Guaranteed sequential ringlet schedule for the concurrent execution of FSMs (removing the need for critical sections and synchronisation points).
7. Efficient execution as the entire arrangement runs as compiled code without thread switching.
8. Being agnostic to communication mechanisms between machines, allowing, for example use with `ROS`:services and `ROS`:messages – however, we recommend the use of our class-oriented `gusimplewhiteboard`.
9. Mechanisms for sub-machine hierarchies and introspection to implement complex behaviours. FSMs can be suspended, resumed, or restarted, as well as queried as to whether they are running or not.
10. Formal semantics that enables simulation, validation, and formal verification.
11. Associated tools such as (`MiEditLLFSM` and `MiCASE`) that enable rapid development of FSM arrangements.

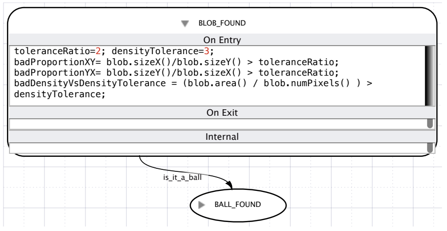
Some details of these characteristics as well as examples of their use will follow. The corresponding download includes full documentation and examples. Videos illustrating the tools (`youtu.be/gN6rIveCWNk`) and using `ROS` (`youtu.be/AJYA2hB4i9U`) are available online.

4 The Logic-Labeled Finite-State Machine Model

Each *llfsm* consists of a set S of **states**, and a transition table $T : S \times E \to S$. There is a distinguished state $s_0 \in S$, named the initial state. Our *llfsm*s are of the synchronous type [16]. The set E are expressions. This is a very important distinction from all other approaches where *events* label transitions (the dominant UML approach). The use of Boolean expression to label transitions has a series of advantages: it simplifies semantics, facilitates scheduling and handling of concurrency [9,10,12] and enables validation and formal verification [11]. This produces rapid development and simulation of robot behaviours [5].

Since any `C++11` expression can label a transition, we can incorporate reasoning and deliberative architectures in what otherwise would be a reactive architecture. Thus, it is possible to include an entire reasoning system, for example using `Prolog` and invoke it from `C++11` using standard APIs. This approach was used for poker hands [2] and to build a poker playing robot. We have found `DPL`, a common sense non-monotonic logic, very useful for declarative aspects.

For example, DPL can be used for expressing the soccer off-side rule in a way similar to the original FIFA specification [1], or describing when it is dangerous for an elderly lady to face a stranger [1]. The C++11 *llfsm* mechanism has been shown to suitably integrate planning incorporating *Planning Domain Definition Language* (PDDL) planners [8]. Fig. 1 illustrates the power of merging reasoning with the C++11 statements in state activities using a simplified example of a vision pipeline. Here, a blob (of orange pixels) reported by the vision module is



```
toleranceRatio=2; densityTolerance=3;
badProportionXY= blob.sizeX()/blob.sizeY() > toleranceRatio;
badProportionYX= blob.sizeY()/blob.sizeX() > toleranceRatio;
badDensityVsDensityTolerance = (blob.area() / blob.numPixels() ) >
densityTolerance;
```

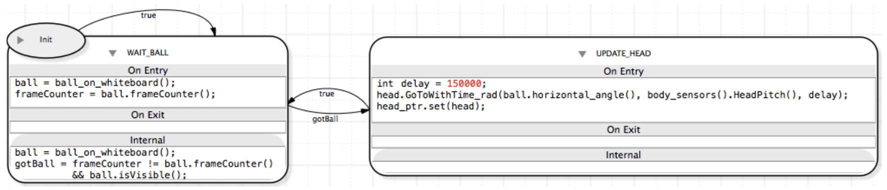
```
%%%%%%%%%%% BallConditions.d
name{BALLCONDITIONS}.
input{badProportionXY}.  input{badProportionYX}.
input{badDensityVsDensityTolerance}.

BC0: {}                  => is_it_a_ball.
BC1: badProportionXY  =>  ~is_it_a_ball. BC1 > BC0.
BC2: badProportionYX  =>  ~is_it_a_ball. BC2 > BC0.
BC3: badDensityVsDensityTolerance  =>  ~is_it_a_ball. BC3 > BC0.

output{b is_it_a_ball, "is_it_a_ball"}.
```

Fig. 1. A state with calculation and a Decisive Plausible Logic (DPL) theory that defines when a blob is a ball.

analysed for its fitness to correspond to a ball. The C++11 arithmetic enables the calculation of values such as the ratio of orange pixels to other pixels in the blob. The DPL rules determine that blobs whose orange-colour density is not higher than a threshold are not to be considered balls. Other conditions include how close the blob matches a square as opposed to an elongated rectangle[1]. Figure 2 shows a feedback loop control for keeping the ball in sight.



```
ball = ball_on_whiteboard();
frameCounter = ball.frameCounter();
```

```
ball = ball_on_whiteboard();
gotBall = frameCounter != ball.frameCounter()
          && ball.isVisible();
```

```
int delay = 150000;
head.GoToWithTime_rad(ball.horizontal_angle(), body_sensors().HeadPitch(), delay);
head_ptr.set(head);
```

Fig. 2. A simple ball tracker.

The semantics for $T(s_i, e_t) = s_j$ is that when the machine is in state $s_i \in S$ and e_t evaluates to true, the machine will move to the state s_j. Without loss of expressivity, the transitions are considered in sequence (as in *MathWorks*® and StateFlow with SymLink), that is, $T(s_i, e_t) = s_j$ will transition to state s_j if e_t evaluates to true and no prior e_s $(\forall s < t)$ evaluates to true in $T(s_i, \cdot)$.

The **OnEntry** section is executed upon arrival from a different state, while actions in the **OnExit** section are executed iff a transition fires. Thus, the actions in these two sections are executed once and only once. The third section (like UML's do) is executed only if none of the transitions fires. When the internal actions are completed, the cycle repeats with evaluating the sequence of expressions of the transitions out of the state. One pass over the cycle is a *ringlet*.

[1] We are thankful to Francisco Martín-Rico from the SPiTeam for this example.

Variables and machine communication. The compiler is completely agnostic of custom libraries or communication mechanisms. Variables can be created in any `C++11` context. Importantly, variables can be created at the state level (*intra-state-variables*), or at the machine level, exclusive to one and only one *llfsm* instance and its states.

Concurrency model. In one ringlet execution there is only one **read** operation at the beginning, by which a local copy of each external variable is made before the execution of any section or the evaluation of any expression labelling any transition. This read operation is a snapshot phase (similar to `rFSM` evaluation contexts, in order to avoid *open environment* [18] inconsistencies). That is, the execution context of a ringlet for external variables remains the same throughout. At the end of the ringlet, a **write** reflects external variables. While the ringlet dispatcher will ensure ringlet atomicity for concurrent *llfsm*, the snapshot phase will also ensure a consistent view of the world outside the FSM arrangement, such as external events (e.g. new sensor readings). If no transition fires and the internal actions complete, then a new ringlet commences.

An ensemble of *llfsms* is executed in a round-robin fashion from one ringlet of one *llfsm* to the next. Thus, the *llfsm* arrangement is a single sequential execution, executed by one thread. While event-driven execution of ringlets is possible, the evaluation of logic expressions is predominantly state-based. This is also reflected in the convention to use idempotent whiteboard messages for communication. Moreover, time-driven guards such as `after()` and `after_ms()` allow the designation of precise state times and an execution style that follows the principles of the time-triggered architecture [20].

The use of a deterministic schedule for the arrangement brings several advantages over a nondeterministic, multi-threaded approach. From the design point of view, open concurrency (where the management of switches between threads is left to the system) puts an unnecessary cognitive load on the behaviour designer as it opens all sorts of needs for synchronisation, and vigilance of nondeterministic communication delays. Burdens include managing critical sections and fairness as well as avoiding deadlocks, live-locks, and starvation (not to mention the associated complexity of CPU context switch overhead, and other system overhead). Our model enables formal correctness verification of models. Model-checking on concurrent threads, by contrast, quickly becomes infeasible for all but the most trivial tasks, as it must consider the combinatorial explosion of all possible state combinations in the system. For robotic systems and embedded systems where there are strict timing requirements, sequential execution is superior to the multiplication of threads [25]. The models produced with *llfsms* can be verified using public domain model-checking technology (`NuSMV`) within a matter of seconds [13,5], but using behaviour trees – which spawn parallel threads – requires several days of CPU time [15].

Scalability. Composition of machines is not only essential for abstraction, but it is a very powerful encapsulation mechanism when building complex (deeply nested) state machines. Complex models can be created by composing simple *llfsms* into more complex behaviours, and those in turn, into still more complex behaviours; `clfsm` supports Brooks' famous subsumption architectures [4], without prescribing strict, hierarchical dependencies. In fact, an entire multi-agent system can be built this way. Here is how the `clfsm` tool supports subsumption architectures or similar organisations.

Each *llfsm* has a single designated state, the SUSPEND state, that has an (implicit) transition to this state from each of the machine's states. This transition is the first one evaluated in every sequence of outgoing transitions and checks whether the machine shall be suspended. The `libclfsm` run-time library provides the `suspend()` function which allows a machine to suspend another. When the token of execution arrives at the machine named in the suspend, the **OnEntry** section of SUSPEND gets executed. Implicit transitions from the SUSPEND state back to the each state also exist that are labelled with the destination state's name. A transition to the previously running state, the *resume state*, gets triggered by the `resume()` `libclfsm` call. Alternatively, `restart()` can be used to unsuspend a machine and restart it from its initial state. Thus, SUSPEND acts like any other state, with exactly the same semantics (e.g. the machine will execute its **Internal** section while suspended). Any state can be designated the SUSPEND state (an empty one is create by `clfsm` if none exists). Based on this, hierarchical control of machines that, in turn, start other machines, can be achieved by explicitly suspending sub-machines in the **OnEntry** section of SUSPEND (by issuing `suspend()` calls to all sub-machines).

In addition to controlling the suspension of *llfsms*, `libclfsm` provides an `is_suspended()` introspection predicate that can be used as a transition label (or as part of any other Boolean expression in the `C++11` code) to detect whether a given machine is suspended or not.

5 The `gusimplewhiteboard` Implementation

`gusimplewhiteboard` is a library that implements a decentralised, distributed access pattern without the need to initiate a broker (in `ROS`, `ROS:roscore` is a pre-requisite and must be running before any nodes can communicate). To use `gusimplewhiteboard`, a module simply needs to include the corresponding headers and link against the library. The first module to execute on a host creates the corresponding data structures for the blackboard in shared memory.

For example, to issue a message for debugging purposes, one can use a predefined message type `Print`. The module must then includes two public files.

```
#include "gugenericwhiteboardobject.h"
#include "guwhiteboardtypelist_generated.h"
```

Then, to `add_Message` to the blackboard, one declares (in the `guWhiteboard` name space), a blackboard singleton instance for the object (using a known blackboard type) by appending `_t` to the type name:

```
Print_t print;
```

Now, we can use a setter (or, for convenience, the overloaded function call operator `()`) to actually post a message (a `std::string` for `Print`):

```
print("Hello, blackboard");
```

To observe the effects of the module we provide a tool `guWhiteboardMonitor` to inspect the messages as they are posted to the blackboard[2]. The monitor makes visible the effect of the print by displaying the following output.

```
Type: Print     Value: Hello, blackboard
```

To construct the classes for objects to become known to the blackboard, a default constructor, the assignment operator, and a `description()` serialisation method that returns a `std::string`, are required. The header file of newly defined classes must be placed in a well-known directory with some pre-processor directives and the class name must be associated with its type(s) in a well-known file[3]. With this, any module can construct blackboard objects of that class. E.g., a class `Ball_Belief` could describe the coordinates of the centre of an orange blob (likely to be the ball) in the reference framework of the camera image. The following `C++11` code constructs an object of such a class.

```
Ball_Belief a_ball(50,30);
```

The `gusimplewhiteboard` approach to `add_Message` comprises two statements, first declaring the handler, then adding the object to the blackboard.

```
Ball_Belief_t wb_ball;     wb_ball.set(a_ball);
```

This is much simpler than the analogous construction of a publisher in `ROS` (there is no need to explicitly register as a node and obtain a `NodeHandle` as well as requesting to obtain a `ROS:Publisher` object).

Introspection. We already mentioned `guWhiteboardMonitor`, a tool that makes use of `description()`. Readers familiar with `ROS` may also be aware of the versatility provided by being able to publish messages of a certain topic onto the communication mechanism through `rostopic pub`. Our corresponding tool is `gusimplewhiteboardposter`. This tool is based on the requirement that pre-existing classes, as well as new user-defined classes that want to support introspection, need to implement a method called `from_string()`. This method, at a minimum, deserialises an instance of the class previously serialised by the `description()` method (but may include arbitrarily versatile parsing of more user-friendly input strings). It should be noted that this is optional and its implementation only impacts on `gusimplewhiteboardposter`.

Getting messages from the blackboard. As discussed before, the preferred approach of our software architecture is a synchronous type of concurrency, analogous to a time-triggered architecture (as opposed to an event-driven architecture). It is well documented in the literature [19,21] that the reliability of time-triggered systems is significantly easier to determine than event-driven systems. Time-triggered systems handle peak-load situations by design. The bandwidth of communications and message rate is constant across low, regular, and peak load situations. Event-driven systems are inherently unpredictable, they can collapse

[2] This is analogous to `ROS`'s `rostopic echo`. By default, the `guWhiteboardMonitor` displays every object posted, but it is possible to specify a type and the effect is analogous to `rostopic echo` displaying the data published on a `ROS:topic`.

[3] Again, this is analogous to the `ROS` construction of a `msg` description.

during peak loads or event showers, and no analytical guarantees can be given for their performance in worst-case scenarios.

In a round-robin scheduling of modules interested in a message, a module that has the execution token (the module that has the CPU) can request information from the blackboard. Besides including the corresponding header files for the user-defined class, the actual code to achieve this is also very simple.

```
Ball_Belief_t wb_ball;
Ball_Belief ball = wb_ball.get();   // or alternatively: ball = wb_ball();
```

This PULL approach always retrieves the most recent information, i.e. the last information that was published. For ease of implementation of event-triggered subscribers, `gusimplewhiteboard` provides a class analogous to the PUSH approach, the `whiteboard_watcher`. A module that wishes to become a subscriber carries out a `subscribe`(T, f) operation as follows.

```
whiteboard_watcher *watcher;
watcher = new whiteboard_watcher();
SUBSCRIBE(watcher, class T, subscriber_class, subscriber_class::f);
```

The semantics of such a subscription (e.g., the semantics of `ROS:subscriber`) is fixed size queuing followed by the invocation of the callback function *subscriber_class*::*f* (using `libdispatch`, `https://libdispatch.macosforge.org`). There is limited queuing of messages and, for the reasons outlined earlier, in our own code we deprecate the use of event-triggered queuing in favour of time-triggered handling of idempotent messages.

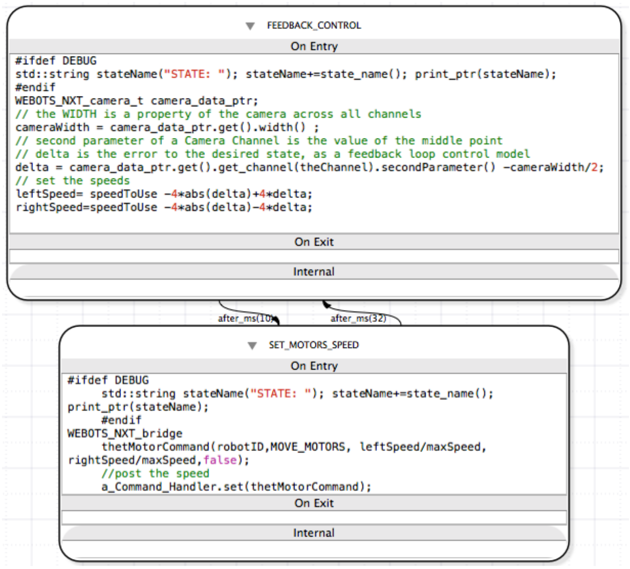
6 Putting `gusimplewhiteboard` into Practice

Our `gusimplewhiteboard` has proven a very efficient and effective communication infrastructure of objects defined by fully fleshed `C++11` classes. In combination with `clfsm`, they provide a very flexible control architecture [2] that minimises concurrency concerns and has facilitated the rapid development of complex, high-level behaviours through composition of modules and *llfsms*. Moreover, `C++11`'s static type system enables far more secure software development. Libraries and modules have been developed for image processing, sensor noise filtering, localisation and navigation, object tracking, and motor control, for Naos, as well as for the ePuck, and simulators such as `Webots`. Fig. 3 on the next page shows the power of fully `C++11` compliant messages with `clfsm`. The two states implement a feedback loop for an ePuck to follow a line. The code in between `#ifdef DEBUG` and `#endif` demonstrates that even pre-processing directives are handled and thus, debugging and monitoring information can be relayed to the blackboard, and reviewed by `guWhiteboardMonitor`. As per control theory, the sensing and estimation of the discrepancy between the desired system output and the sensor reading is encapsulated in the FEEDBACK state. The statements

```
WEBOTS_NXT_camera_t camera_data_ptr;
cameraWidth = camera_data_ptr.get().width();
```

retrieve the camera object from the blackboard using the `camera_data_ptr` handler for a known message of the `WEBOTS_NXT_camera` class. We can PULL the object and obtain an attribute in one go, e.g. `camera_data_ptr.get().width()`.

```
delta = camera_data_ptr.get().get_channel(theChannel).secondParameter()
      - cameraWidth/2;
```



```
#ifdef DEBUG
std::string stateName("STATE: "); stateName+=state_name(); print_ptr(stateName);
#endif
WEBOTS_NXT_camera_t camera_data_ptr;
// the WIDTH is a property of the camera across all channels
cameraWidth = camera_data_ptr.get().width() ;
// second parameter of a Camera Channel is the value of the middle point
// delta is the error to the desired state, as a feedback loop control model
delta = camera_data_ptr.get().get_channel(theChannel).secondParameter() -cameraWidth/2;
// set the speeds
leftSpeed= speedToUse -4*abs(delta)+4*delta;
rightSpeed=speedToUse -4*abs(delta)-4*delta;
```

```
#ifdef DEBUG
     std::string stateName("STATE: "); stateName+=state_name();
print_ptr(stateName);
     #endif
WEBOTS_NXT_bridge
     thetMotorCommand(robotID,MOVE_MOTORS, leftSpeed/maxSpeed,
rightSpeed/maxSpeed,false);
     //post the speed
     a_Command_Handler.set(thetMotorCommand);
```

Fig. 3. Two states that use `gusimplewhiteboard` services to perform a feedback loop to control an ePuck that follows coloured lines.

obtains a sensor message of the `WEBOTS_NXT_bridge` class. This demonstrates the use of sophisticated class composition allowed by `C++11` (as opposed to the `ROS`:`msg` restrictions). Finally, `delta` stores the measured error, followed by the computation of the desired motor target speeds. The SET_MOTORS_SPEED state simply constructs a local `WEBOTS_NXT_bridge` object and posts it to the whiteboard. You can see a video of this state machine in action at `http://youtu.be/F8K4V78vUbk`

Of course `gusimplewhiteboard` can be considered an alternative to the event-driven blackboard control architectures developed by the robotics community. Our approach aims at establishing components for which formal verification is possible. However, its presentation here is not meant to be a replacement for `ROS`:`roscore`, `ROS`:`services`, or `ROS`:`nodelets`, but a complement. In fact, we have a `gusimplewhiteboard-ROS` bridge that enables relaying of data across ROS:`core` and `gusimplewhiteboard`: since one of the basic `ROS`:`msg` types is `String.msg` and classes known to `gusimplewhiteboard` implement the `description()` method as a serialisation to string and the `from_string()` methods as a materialisation from a string, the bridge is a publisher/subscriber across `ROS`:`roscore` and `gusimplewhiteboard` that relays messages.

The capacity of `clfsm` and `gusimplewhiteboard` in day-to-day use is remarkable. We currently run 27 *llfsms* on board Alderbaran's Nao robot to implement the behaviour of *MiPal*'s soccer player. We cross-compile the machines to native code before they are set-up for execution on the robot. Among those 27, there is one machine that follows the states of the SPL-league game controller, providing an unambiguous, formal interpretation of the standard platform league rules.

Performance. We have implemented a comparison catkin package, benchmarking `gusimplewhiteboard` posting and `ROS` topic publishing. Thus, the compiler used is the same and so are the optimisation flags. The benchmark has been tested with several computers, but the data shown is from a late 2013 Mac Pro, 3 GHz 8-Core Intel Xeon E5, 32 GB memory 1867 MHz DDR3 ECC RAM. The data type is very simple, it is a Boolean value using `std_msgs::Bool` in the case of `ROS` and the boolean type from `C++11` for the `gusimplewhiteboard`. Larger, more complicated types make `ROS` even slower. For example, for an `add_Message`, the `gusimplewhiteboard` delivers 411,895,543 messages per second while `ROS` only manages 47,925. Moreover, `ROS` seems to be affected by kernel and networking constraints, e.g., a bottleneck in the number of messages per second the kernel can push through locally. The delays in `ROS` have been documented before [6], but our `catkin` benchmark here shows at least 50 times faster performance (and in the CPUs on board of robots, the gap would be larger).

gusimplewhiteboard		ROSmacports *Hydro*	
get_Message	0.0024 μs	ROS:subscribe()	20.14 μs
add_Message	0.0120 μs	ROS:publish()	20.87 μs

7 Conclusion

The released `clfsm` `ROS` package contains simple examples that demonstrate the construction and execution of *llfsms*. Our videos demonstrate an arrangement of 6 *llfsms*that make a Nao avoid obstacles. The tools for building this behaviour were used in a third year undergraduate course and students could construct this behaviour within a single, two hour laboratory session. This provides evidence of the flexibility and rapid prototyping and development that can be achieved with `clfsm`. The full construction of this behaviour also appears in the download and documentation of the MiEditLLFSM state machine editor. Compositions of machines have also been used to create higher levels of navigation and planning for the ePuck in the `Webots` simulator (also a student lab session), using a feedback loop control approach to construct a coloured-line follower.

References

1. D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock. Chapter 3: Non-monotonic reasoning on board a sony AIBO. P Lima, ed., *Robotic Soccer*, 45–70, Austria, 2007. I-Tech Education and Publishing.
2. D. Billington, V. Estivill-Castro, R. Hexel, and A. Rock. Architecture for hybrid robotic behavior. *4th HAIS*, v. 5572, 145–156. Springer LNCS, 2009.
3. J. Bohren and S. Cousins. The SMACH high-level executive [ROS News]. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.
4. R.A. Brooks. Intelligence without reason. *12th ICJAI-91.* , 569–595, Sydney
5. R. Coleman, V. Estivill-Castro, R. Hexel, and C. Lusty. Visual-trace simulation of concurrent finite-state machines for validation and model-checking of complex behavior. *3rd SIMPAR*, v. 7628, 52–64, 2012. Springer LNCS.
6. E. Einhorn, T. Langner, R. Stricker, C. Martin, and H.-M. Gross. MIRA - middleware for robotic applications. *2012 IEEE/RSJ IROS*, p. 2591–2598, Portugal.
7. J. Erickson and K. Siau. Can UML be simplified? practitioner use of UML in separate domains. *12th EMMSAD'07*, v. 365, 87–96, CEUR.
8. V. Estivill-Castro, J. Ferrer-Mesters. Path-finding in dynamic environemnts with PDDL-planners. *16th Int. Conf. Advanced Robotics (ICAR)*, Montevideo, 2013.

9. V. Estivill-Castro and R. Hexel. Arrangements of finite-state machines semantics, simulation, and model checking. *Int. Conf. on Model-Driven Engineering and Software Development MODELSWARD*, 182–189, Barceloan, 2013. SCITEPRESS.
10. V. Estivill-Castro and R. Hexel. Module isolation for efficient model checking and its application to FMEA in model-driven engineering. *8th ENASE Evaluation of Novel Approaches to Software Engineering*, 218–225, Angers, France, 2013. INSTCC.
11. V. Estivill-Castro and R. Hexel. Correctness by construction with logic-labeled finite-state machines – comparison with Event-B. *23rd Australasian Software Engineering Conf.*, Sydney, 2014. IEEE Computer Soc. CPS.
12. V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth. Efficient modelling of embedded software systems and their formal verification. *19th Asia-Pacific Software Engineering Conf. (APSEC 2012)*, 428–433, 2012. IEEE Computer Soc., CPS.
13. V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth. Failure mode and effects analysis (FMEA) and model-checking of software for embedded systems by sequential scheduling of vectors of logic-labelled finite-state machines. *7th Int. IET System Safety Conf.*, Edinburgh, UK, 2012. Paper 3.a.1.
14. L. Garber. Robot OS: A new day for robot design. *Computer*, 46(12):16–20, 2013.
15. L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay. Experience with fault injection experiments for FMEA. *Software, Practice and Experience*, 41(11):1233–1258, 2011
16. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM T. on Software Engineering Methodology*, 5(4):293–333, 1996.
17. B. Hayes-Roth. A blackboard architecture for control. *Distributed Artificial Intelligence*, 505–540, San Francisco, 1988.
18. M. Klotzbuecher. rFSM v1.0-beta6. www.orocos.org/rfsm,
19. H. Kopetz. Should responsive systems be event-triggered or time-triggered? *IEICE Transactions on Information and Systems*, 76(11):1325, November 1993.
20. H. Kopetz and G. Bauer. The time-triggered architecture. *Procôf the IEEE*, 91(1):112–126, 2003.
21. L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6:254–280, 1984.
22. M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. *7th Int. Workshop on RoboCup 2003*, v. 3020 of *LNAI*, 114–124. Springer, 2004.
23. M. J. Mataric. *The Robotics Primer*. MIT Press, 2007.
24. S. J. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture*. Addison-Wesley, Reading, 2002.
25. T. Merz, P. Rudol, and M.; Wzorek. Control system framework for autonomous robots based on extended state machines. *ICAS '06*, 14, Silicon Valley, 2006.
26. O. Michel. Webots: Professional mobile robot simulation. *J. Advanced Robotics Systems*, 1(1):39–42, 2004.
27. G. Reggio, M. Leotta, F. Ricca, and D. Clerissi. What are the used UML diagrams? a preliminary survey Technical report, Universitá di Genova, Italy, (DIBRIS), 1998.
28. M. Risler and O. von Stryk. Formal behavior specification of multi-robot systems using hierarchical state machines in XABSL. *AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems*, Estoril, 2008.
29. M. Samek. *Practical UML Statecharts in C/C++, 2nd Edition: Event-Driven Programming for Embedded Systems*. Newnes, 2008.
30. F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, NY, 2006.