Simple, Not Simplistic
The Middleware of Behaviour Models

Vladimir Estivill-Castro and René Hexel
School of Information and Communication Technology, Griffith University, Nathan, QLD, Australia

Keywords: Model-driven Engineering, Software Modelling, Behaviour Models, Middleware.

Abstract: There are many areas where software components must interact witch each other and where middleware provides the appropriate benefits of robustness, decoupling, and modularisation. However, there is a potential performance overhead that, for autonomous robotic and embedded systems, may be critical. Proposals for robotic middleware continue to emerge, but surprisingly, they repeatedly follow the publish-subscriber model. There are several disadvantages to the *push* paradigm of the publisher-subscriber approach; in particular, its implication of a closer coupling where the subscriber must be active and able to keep up with the pace of events. We propose an alternative *pull* model, where consumers of messages handle information at their own time. We show that our proposal aligns with fundamental, time-triggered design principles, and produces simple module communication that reduces thread management and can enable rapid prototyping, validation, and formal verification.

1 INTRODUCTION

Nowhere is the need for middleware more apparent than in the world of mobile robotics (Brugali, 2007). The system on board a mobile robot consists of the integration of drivers of a multitude of simple and complex sensors that deliver data at different rates and in different volumes, with varying degrees of accuracy. Filtering, sensor-fusion, information processing, and analysis modules compute and construct the picture that reflects the state of the robot and the world. Based on this, locomotion, localisation, planning, reasoning, and behaviour modules can collaborate in generating the corresponding commands to output modules that direct actuators and effectors to achieve complex missions and tasks. Thus, it is not surprising that in recent years, several major projects have resulted in the production of such robotic middleware. Currently, one of the most popular suites is `ROS` (Quigley et al., 2009) and some very recent analyses (Chitic et al., 2014) place `ROS` somewhat ahead of the others when considered under the criteria for middleware. Nevertheless, there continue to be emerging proposals (Anzalone et al., 2014; Huang et al., 2010).

What motivates our argument here is that, from the perspective of software engineering, these frameworks have some important, common characteristics. While they all use a *blackboard control architecture* as the paradigm central to integrate the modules (agents) performing cognitive processes, handling behaviours and managing adaption and problem solving tasks, they largely re-incarnate the publisher/subscriber pattern. While this pattern eliminates the need to create complex communication mechanisms, the blackboard control architecture allows a further level of decoupling by being data-centric (vs. component-centric) not only in the value domain, but also the time domain, which is important for timing-critical systems such as robots, but regrettably, often overlooked. The provider may supply information for unknown consumers who may not even be active at the time the information is provided. There is no need to be aware of a consumer's interface, only the interface to the blackboard is necessary. The blackboard also can be considered a *repository architecture* (Sommerville, 2010, Page 159) as well as the knowledge base of an artificial agent.

From the perspective of software architectures, the flexibility of a blackboard is also incorporated into the notion of a broker. Thus, it is not surprising that this pattern has also emerged as the CORBA standard (of the Object Management Group, OMG) with the aim of facilitating communication on systems that are deployed on diverse platforms. In simple terms, these types of infrastructures enable a sender to issue what we will refer to as an `add_message`$(msg : T)$ which is

Estivill-Castro V. and Hexel R..
Simple, Not Simplistic - The Middleware of Behaviour Models.
DOI: 10.5220/0005371101890196
In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2015), pages 189-196
ISBN: 978-989-758-100-7

a non-blocking call. In a sense, posting *msg* to the blackboard is simple. Such a posting may or may not 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. But when it comes to retrieving the message, there are essentially two modes.

`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.

For example, ROS' PUSH technology names a communication channel, a `ROS::topic` (corresponding to what we call a *type*). The modules posting or getting messages are called *nodes*. Posting a message in ROS is also called publishing[1].

Development of robot-controlling software modules under the PUSH approach assumes predictable communication latency. For instance, standard algorithms in robotics, like the Kalman filter, would be significantly less effective if the time between the sensor reading 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 is seriously compromised, even with mobile robots that are not considered hard real-time systems.

Surprisingly, the PUSH (subscriber with call-back) approach is ubiquitous among all these proposals of robotic middleware. Its choice implies several consequences. First, the overall approach to behaviour

[1]In fact, there is another mechanism for communication, called ROS-services, which is essentially a remote-procedure call, the requester/client (an example appears later in Figure 3) invokes though the middleware a function (`client.call` which is blocking) and obtains a data structure as a response (or a failure signal) from a call-back in a server (an example appears alter in Figure 4). This also implies significant handshaking, and sadly, the responses cannot be objects; that is, not members of a class with methods.

```c
#include <sys/types.h>

#ifndef temperature_msg_h
#define temperature_msg_h

struct wb_temp
{
    int8_t  degC;   ///< temperature in Centigrade

# ifdef __cplusplus
    /** constructor */
    wb_temp(int8_t temp = 0): degC(temp) {}
# endif
};

#endif  // temperature_msg_h
```

Figure 1: a simple blackboard message in C, containing a single temperature value (in Centigrade).

control becomes event-driven. That is, the treatment is that events are deposited into the middleware which relays them to call-backs from the subscribers or service providers. This also implies an optimistic approach; namely, there is the assumption that there will be sufficient computational resources to enact all the threads generated and to execute the subscribers' call-backs. Moreover, events would occur with enough sparsity that call-backs would be completed by the time they are executed again, or alternatively require to handle concurrency issues. Both of these aspects actually are not uncommon, e.g., middleware providing facilities for queuing events for call-backs, or alternatively, developers performing real-time analyses to establish confidence in the correctness of the system by evaluating the duration of a call-back. Perhaps more dramatic are the coupling and timing consequences: ultimately this approach assumes that subscriber modules react or act on the data immediately, thus data is presumed to be as fresh and as recent as possible. Not only is timeliness taken for granted, but so are sequencing and reliability of data. Such an optimistic approach works in best-case and, perhaps, average-case scenarios but it is well known to suffer dramatically in the worst-case. It leads to more coupling as senders must eventually be slowed down or complex handshaking protocols have to be deployed in order to raise tolerance to message loss.

2 SPHERE OF CONTROL

We look first at a typical scenario of data-driven information exchange, and we contrast using the PUSH versus the PULL model. Later, we look at more complex scenarios, but for simplicity, we start with a unidirectional message. Figure 1 shows a simple C data structure that contains a single value of a temperature sensor. Superficially, following the PUSH model seems intuitive. A recipient might simply subscribe

to the temperature message in order to receive updates whenever the temperature changes. However, when analysing this further, a lot of questions arise. First, who is to say when messages are posted? Is this defined by the subscriber, by the publisher, or by the API? The convention to only post changes appears obvious at first glance, but what is it that constitutes a change? Does the sensor module need to poll the hardware or is there an interrupt that gets generated whenever the temperature changes? Can different hardware (e.g. with different temperature resolution) cause different message update rates? Will the subscriber be able to keep pace with the publisher under all circumstances? The answer to these questions depends on the sphere of control (Kopetz, 2011) for the relevant message. The PUSH paradigm of the publisher/subscriber model puts messages in the sphere of control of the publisher. The subscriber will need to keep up in order to not lose any messages, no matter whether that specific update rate is necessary or convenient for the receiver. In other words, the timing of the subscriber is determined by (or tightly coupled with) the publisher. While it would be possible for the interface to not only specify the value domain (e.g. `int8_t`, *degrees Celsius*), but also the time domain (e.g. an update rate of once per second), this only increases the complexity of the interface (raising the question how the update rate should be verified or enforced). Furthermore, this approach does not loosen the level of coupling. Quite the opposite, while the pace of the subscriber is still dictated by the rate with which messages are posted, now the pace of the publisher is also mandated (by the temporal interface specification). This does little to reduce the complexity of the recipient, but may now add complexity to the sender (e.g. hardware-driven transmitter modules may now need to throw away messages that do not comply with the interface's timing specification).

We argue here that the sphere of control should remain within the respective component. This means that a publisher should be able to post a message whenever practicable for the publisher (e.g. when the hardware reports a new temperature value, or periodically, whenever the the software reads the temperature value) while the recipient should be able to query the current temperature whenever needed. This kind of decoupling allows for an independent development of components without the requirement for a full, temporal specification of all interfaces upfront. While this approach does not necessarily prevent an impedance mismatch between the sender and the recipient, the remedy is fairly simple and straightforward. If, for example, the temperature sensor module does not provide sensor data fast enough, it can simply be replaced by a faster module that plugs into the same interface. On the other hand, if the recipient cannot keep up with the sender, it can simply read the temperature values at a lower pace that is more suited to the processing that is done by that module.

3 SEPARATION OF CONCERNS

While in the previous section we looked at a model where no synchronisation is required between two components, the question arises whether the decoupled model is also suitable in situations where actions need to be synchronised. A typical example is a controller that sends a message for an action to be initiated and then waits for that action to have started (or completed) before continuing. This example is akin to the *rendezvous* model in the message passing world, or a synchronous remote procedure call (RPC).

```
int32 desiredSpeed
---
int32 reportedSpeed
```

Figure 2: Speed controller `ROS`-service definition.

We now contrast an example that illustrates how this can be achieved using the PULL paradigm without having to resort to the tight coupling caused by the PUSH paradigm. On the PUSH side (server with callback), we preset in Figure 2 a `ROS`-service definition for a speed controller that takes a speed value (e.g. in mm/s) and returns the current speed. The relevant `C++` code for a client for this service can be found in Figure 3. The important bit is `client.call(srv)`, which is a synchronous RPC call that returns `true` if the service call was successful. The actual response from the service can be found in the `response` data structure (i.e., `reportedSpeed` as per the definition in Figure 2). Again, from a simplistic client's perspective, this code structure can be viewed as ideal; if, for example, the RPC call does not return until the actuator has reached the desired speed, the example code might be sufficient for the client to just check the return value of the RPC call and then continue accordingly. In practice, this paradigm has funda-

```
ros::NodeHandle n;
ros::ServiceClient client = n.serviceClient
                        <speed_controller::Speed>("speed");
speed_controller::Speed srv;
srv.request.desiredSpeed = atoi(argv[1]); // set speed
if (client.call(srv))
  ROS_INFO("Speed: %d", (int)srv.response.reportedSpeed);
else
{
  ROS_ERROR("Failed to call service speed_controller");
  return EXIT_FAILURE;
}
```

Figure 3: `ROS speed` controller client.

```
bool set(speed_controller::Speed::Request  &req,
         speed_controller::Speed::Response &res)
{
  res.reportedSpeed = req.desiredSpeed;
    // <set actuator to desiredSpeed> //
  return true;
}
```

Figure 4: ROS speed controller server.

mental problems when it comes to translating from a pure software perspective (e.g. in a distributed object middleware system) to sensors and actuators in the physical world. A key structural issue becomes apparent when we look at a typical, simple server implementation for this ROS speed controller (Figure 4). Here we can see that the server does not even report the actual speed (but simply copies the desired speed back, as an acknowledgement of the speed that it is aiming at). The reason for this is that in reality, actuators and sensors are often separate. It therefore makes sense that, for simplicity, sensor and actuator software modules are also kept separate. Adding sensor functionality to an actuator module violates a very important system development principle of *separation of concerns*. Even if we assume that the actuator module can easily access sensor data and therefore replace the response code with `res.reportedSpeed = sensor.measuredSpeed;` this would not help the client, as the reported speed would represent the speed when the command was issued, not when the actual speed was reached. Such semantics neither justifies the added complexity imposed on the actuator module (server), nor does it justify the overhead of a synchronous RPC. It would, of course, be possible to add further complexity to the actuator module by making it wait until the desired speed is reached. However, this not only makes the actuator module more complex as now sensor accuracy and the possibility of a timeout have to be taken into account (what happens, e.g., if a robot is physically blocked? How long should the actuator wait for the target speed to be reached?). Such a design would also significantly increase the complexity and coupling of the system as a whole as a key question now becomes what should happen if a new request comes in while the RPC from the previous request has not yet returned. ROS queues up these requests and does not service a subsequent request until the current request has completed.[2] This makes adding of, e.g., emergency stop functionality impossible through the same interface. If the RPC service, on the other hand, allowed re-entrancy, this would negate the simplicity of the client, as any command might now be interrupted and overridden by a subsequent command, even before the previous command RPC had returned (adding significant handling complexity to the client).

[2] ROS enables programatic mechanisms to define the queues and their behaviour and calls to enable the call-backs like `ros_spin()` and `ros_spin_one()`.

```
struct wb_speed
{
    /// speed in mm/s
    int32_t  speed;
};
```

Figure 5: The speed actuator/sensor blackboard message.

```
class:wb_speed, atomic, Speed_Control, "Speed_Control", Actuator
class:wb_speed, atomic, Speed_Status,  "Speed_Status",  Sensor
```

Figure 6: Separate control and status message slots.

We argue here that physical sensors and actuators often impose an end-to-end control and verification requirement that cannot conveniently be hidden by a simplistic interface. In fact, such end-to-end control can naturally be mapped to an asynchronous PULL model where concerns of sensor and actuator (server) components are separated from each other and from the concerns and functionality of control components (acting as RPC clients). Figure 5 shows a blackboard message for the speed controller (akin to the temperature message in Fig. 1, but without boilerplate code and C++ convenience constructor). The key difference to the ROS-service definition is that only a single speed field is defined for both the (actuator) request and the (sensor) response. Instead of modelling a synchronous RPC with a parameter and a return value, two idempotent, asynchronous messages are used, a Control message for the actuator and a Status message for the sensor (Figure 6). Both the actuator and the controller components us the PUSH paradigm to query their corresponding message slots.

This approach facilitates decoupling through natural encapsulation of concerns (e.g., an actor software module only needs to be concerned with actuator hardware; other concerns such as reading sensor values are *factored out* into a sensor module, and control and error handling can be moved into a separate controller module). Moreover, the control/status approach complements very well deterministic arrangements of logic-labeled finite-state machines (LLFSMs) (Estivill-Castro and Hexel, 2013a; Estivill-Castro and Hexel, 2013b; Estivill-Castro et al., 2014) where each LLFSM is a module, as the arrangement

1. is scheduled sequentially (e.g., in a single thread), so messages are automatically atomic and do not require explicit synchronisation, and
2. transitions in LLFSMs are labeled by Boolean expressions, not events (so they are not event-driven).

4 CASE STUDIES

4.1 ROS, bride and smach vs Simpler Model-driven Development

As we mentioned, the ROS framework has become the de-facto standard for component-based robotics and robotic software frameworks. It also offers a software environment for networked robotics. In a sense it completely covers all areas of software engineering for experimental robotics (Brugali, 2007).

Its main tools for describing and developing behaviours on robots are smach (Bohren and Cousins, 2010) (a ROS-independent Python library to build hierarchical state machines) and bride (a model driven engineering tool chain based on Eclipse). We compare control/status messages (with LLFSM) in the exact scenario presented in the bride-tutorial: the predator/prey setting. The corresponding model can be developed with control/status messages with just 3 logic-labeled finite-state machines (LLFSMs). This is demonstrated in a one-minute video.[3] The impact of status/control messages can be observed in this video. Moreover, the three finite-state machines execute in only one thread, and they constitute an executable model; that is, they are exactly the software that is actually running. The complete system is visible and transparent.

One LLFSM is dedicated to the behaviour of the prey, a second one to the behaviour of the predator, and the third one controls the ROS simulator that creates the animation. Since the simulator is a central tool in ROS, we are forced to use ROS-msg and ROS-srv to interact with the turtle simulator. However, the controller LLFSM (that keeps track of the status of the simulation) uses our approach of separation of concerns using status and control messages. It spawns the prey, spawns the predator and receives a control signal when the predator finds the trace of the prey (in order to delete the prey from the simulator and halt the simulator).

By contrast, the bride model-driven approach generates some of the code, mostly the corresponding signatures and methods with empty bodies, and many code sections that are not necessary and unused. The software developer is required to then edit the code and complete several other functions and procedures in an environment invisible to the modelling tools. That is, it is a one-way direction, the models generate templates of the code, but once the code is completed, one cannot use the modelling tools anymore.[4]

More drastic is the contrast of the complexity of artefacts. The bride tutorial requires the developer to become familiar with

1. Node and computation specification diagrams for capacity building; and as already mentioned, expertise to fill in the code in the generated templates.
2. Interconnection diagrams for system deployment.
3. smach diagrams for coordination building.

The bride-tutorial results in at least 5 (five) ROS nodes (one predator, one prey, one turtle manager, one turtle mover, and one paint detector), and 6 (six) communication protocols. Each node implies at least one thread, each communication protocol implies at least one message and topic type and channel, corresponding to management of 6 message-queues, and corresponding call-back coding. The story does not finish there, we further need a smach model with two finite-state machines who also need instruments for synchronisation and the connection diagram!

Also, a very serious problem is that, in this and earlier bride-tutorials, many aspects of the rates of publishing and call-back response are hidden. In fact, if the *launch* does not get lucky, messages may be lost, and the simulation may fail!

We believe this case study alone provides justification to the title of this paper. While bride and smach are tools going in the right direction, they start to suffer a blowing up effect similar to UML's. Tools should be simple to support (but not simplistic as to actually complicate) the tasks at hand. One would perhaps suggest that more sophisticated and complicated tools are necessary for large, and more complicated challenges and missions of field and service robotics. However, our point here is that if the cognitive map of even simpler systems is so large, and its semantics so cumbersome, the actual potential for fragile, error-prone scenarios in large systems is a real obstacle. We are of the opinion that such tools would not scale. Software developers would then simply resort back to tools closer to the programming code. Part of this is the simplistic (but unworkable) publish/subscriber model whose issues we analysed earlier (see Sections 2 and 3). A simplistic approach attempts to hide fundamental issues without solving them (and therefore, they eventually re-emerge).

Another aspect is the enormous consumption of resources that the resulting software typically requires

[3] youtu.be/fX7ANt03Xsc

[4] The tool used to design the machines in the prey-predator demo of LLFSM for clfsm is called MiCASE. It offers many levels of abstraction, allowing to close an open states, some features of MiCASE appear in another video youtu.be/F8K4V78vUbk.

due to the optimistic nature of the underlying assumptions. In other words, in order to keep the probability of failure (due to resource over-utilisation) low, CPU and other resources have to be over-specified. More dramatically, since multithreading and uncontrollable multitasking is happening under the hood and behind the scenes, it is close to impossible to validate and formally verify the behaviour of such systems (and as a consequence, even with massive resource over-specification, the system as a whole remains a best-effort system that cannot formally guarantee responsiveness under specific load or fault scenarios).

The expertise required with `ROS` tools like `bride` and `smach` is not unattainable by system integrators or application engineers, but certainly will exclude end users and application architects from direct participation in specifying projects for robotic systems. Simpler tools (control/status messages and LLFSMs) allow communicating behaviour to a broader audience than those traditionally involved with the design, implementation, and deployment of robotic behaviour. We argue that this will become an even greater necessity in the foreseeable future with a dramatic increase in the number of applications in service robotics.

4.2 Teleo-reactive Models vs LLFSMs

Another trend in modelling software that describes behaviour in robots is that of teleo-reactive programs (Nilsson, 2001; Morales et al., 2014). This formalism has been expanded in many directions, more recently in an attempt to enable the formal verification for robotics that must meet real-time deadlines (Dongol et al., 2014). The formalism is considered central to describing behaviour in goal-oriented agents. We argue that such a formalism suffers from similar challenges as modelling tools such as Behaviour Trees[5], in that their semantics for concurrent elements seems simple, but, in fact, results in serious complications because of its open concurrency model. Besides the possibility of undefined behaviour (Hayes, 2008), the problems are similar to those of UML state-diagram semantics (Estivill-Castro and Hexel, 2013a).

In particular, teleo-reactive programs can nest to create concurrency (Dongol et al., 2014, Fig 2). Namely, a teleo-reactive behaviour is a description of guarded programs of the form $c \rightarrow M$ where c is a state predicate (synthesised involving a Boolean expression on the sensors of the robot, and the state of the world) and M is either a (primitive) durative action of a *sequence of guarded programs*. Thus, for example, in the Herbert's world (Brooks et al., 1988), codification by teleo-reactive programs (Dongol et al., 2014, Fig 2) we have the top sequence description behaviour (named `robot`) execute the sub-behaviour `collect` if the environment variables indicate

$$(\neg \texttt{holding} \wedge \texttt{depot\_empty}) \vee \neg \texttt{open} \wedge \texttt{at\_depot}.$$

But then, the `collect` behaviour has its own sub-behaviour, `fetch`, that happens when the environment variable `see_can`, is true. We have 3 behaviours running concurrently, observing shared variables describing the environment and creating all sorts of race conditions, because as soon as the conditions become false in the top-level teleo-reactive program, the `collect` behaviour is to be suspended, which in itself is to suspend the `fetch` behaviour.

Moreover, in implementations like `ROS`, the PUSH model (with call-backs to subscribers or servers) suffers the challenges illustrated earlier with our `speed-controller` example (Figures 3 and 4), but now with actual information about 5 aspects of the world (`holding` – the sensor that determines if there is a can in the grasper, `depot_empty` – the sensory information about whether there is space in the table, `open` – the sensor that determines if the joint of the grasper is closed, `at_depot` – the sensory information of the robot's location, and `see_can` – whether a can is visible in the front camera). The clarification of the semantics of concurrency like this is usually terribly laborious and cumbersome. Just see the lengthly description in the UML standard on nested states, the inconsistent semantics regarding launching threads in Behaviour Trees, or complicated formulation of the semantics of teleo-operated programs. We believe all this leads to ineffective model-driven engineering.

Herbert's world (at least at the level of (Dongol et al., 2014, Fig 2) where grasping a can of soda is considered a primitive action) can be solved with three concurrent LLFSMs executing in a single thread, using the sequential semantics of LLFSMs. The video youtu.be/qs-jmjxOXLI demonstrates this.

Again, the modelling is rather simple. One LLFSM is responsible for searching through spinning. This behaviour is slightly different if it is to find a soda can or to find the table. When finding a can, the robot spins to its left. When aligning itself to the table, because of odometry, it can turn to the side that is shorter (in the video, after picking up the second can, it spins to its right). What to look for in the search behaviour is controlled by a super-LLFSM that governs the overall behaviour. The communications is via control/status messages.

Also, the localisation information of the robot is communicated via status/control messages. The distance to an object in the environment (in polar coordinates for the table) is the content of these messages.

[5](Billington et al., 2011) discusses the concurrency issues of Behaviour Trees and how LLFSMs eliminate them.

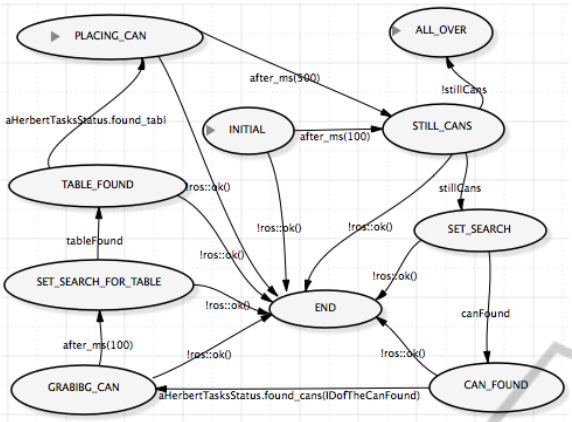

Figure 7: The master LLFSM of Herbert's world shows the use of status messages in some of the transitions.

Thus, this highly decoupled paradigm does pay off.

The master behaviour also resumes the LLFSM that drives the robot to a soda can, or back to the table. This sub-LLFSM behaviour that walks in the direction of a target is also different if aiming for a can (in which a reactive control keeps the robot aligned with the target strictly in front), while for the table, the behaviour is more tolerant about keeping the table in front (and although not immediately obvious, travels faster). Again, the modelling with LLFSMs exploits the decoupled but reliable communication that the control/status messages provide. The master LLFSM can issue requests to its sub-behaviours, using a control message, and the sub-behaviours can report their progress with the status message.

The generic states of the master-llfsm for this case study are shown in Figure 7. We also use control/status messages for indicating whether a can has been found, and more importantly, since `gusimplewhiteboard` (Estivill-Castro et al., 2014) enables messages to be `C++`-objects, we can invoke getters and setters on them. This is shown in Figure 7, where the transition from the state CAN_FOUND to the state GRABBING_CAN uses the variable `IDofTheCanFound` to extract only one component of the status message.

Moreover, this case study is a direct replication of the classic robotic mission that gave fame to the subsumption architecture (Brooks et al., 1988). We still have to use some `ROS`-msg and some `ROS`-srv because the video was produced using the `ROS`-drivers of the Kuka robot which is the standard platform for the RoboCup@Work league. The video has been produced in the `ROS` interface of the *Gazebo* simulator.

Teleo-reactive programs are considered useful for model checking, but we show here that the LLFSMs can also be readily formally verified. Our tools translate LLFSMs for standard model-checking tools such as `NuSMV`. For instance:

1. *The robot does not walk towards a can of soda unless it has found one.*
2. *The robot does not walk towards the table unless is has collected a can of soda.*

Formal representation of these properties as CTL or LTL formulas reveals that one has to be more precise. Nevertheless, we prove the first one as follows.

> If the value of `ros:ok()` holds true at least until we reach the **OnEntry** section of state SET_SEARCH (where we resume spinning looking for a can), then
>
> - as long as `ros:ok()` holds true
>   - we could stay in the state SET_SEARCH looking for a can for an indefinite amount of time (possible forever) or
>   - all transitions out of SET_SEARCH (with target some other state) in fact lead to the **OnEntry** section of CAN_FOUND (where we resume the walking straight).

We emphasise the insights that model-checking brings, as the LLFSM in Figure 7 actually implies 6 types of transitions from SET_SEARCH (plus the possibility of not taking any). For each of the two transitions we have an exit point after the **OnEntry** section, after the **OnExit** section without the execution of the **Internal** section and after the execution of a finite number of times of the **Internal** section and once the **OnExit** section. Although this may seem elaborate, it is certainly simpler than the teleo-reactive programs, which are models that actually do not execute.

5 CONCLUSIONS

In distributed systems, the notion of idempotent messages is crucial, ranging from embedded systems (Kopetz, 2011) to the scale of cloud computing. This means, the accidental or faulty re-sending of a message should not damage or impede communication or adversely impact a protocol. However, the PUSH approach does not achieve this. **With PUSH, you need to wait for an acknowledgement to ensure your message was received exactly once**, causing a plethora of issues on how long to wait for such an acknowledgement. The PULL model implements idempotence through the control/status approach. Moreover, the literature on fault-tolerant systems (Laprie, 1992) has already established that the PULL technology is robust to messages being lost, while the PUSH technology relies on delivery guarantees. **With**

PUSH**, you must have a channel that absolutely never loses messages**.

In Section 3, we used the example of the thermometer to highlight how **the** PUSH **technology implicitly creates more coupling** and affects the sphere of control in unsatisfactory aspects. Also, the PUSH approach does not scale up well. If the effect of a command is to be observed by more than one entity, the PUSH approach becomes susceptible to concurrency issues, e.g. if different subsystems have different priorities. Moreover, in the case of middleware, our implementations and their treatment of the case studies here illustrates that end-to-end control through the PULL approach is a simpler and more effective approach than the simplistic PUSH mechanism using a subscriber/server with call-backs. This is exemplified in in the first case study, where the PUSH approach results in 6 times the number of treads of the PULL approach. In the second case study, even if it were implemented, the teleo-operated approach would imply three times more threads. Minimising threads is crucial to enabling model-checking and formal verification. Additional threads exponentially explode the state space.

REFERENCES

Anzalone, S. M., Avril, M., Salam, H., and Chetouani, M. (2014). IMI2S: A lightweight framework for distributed computing. *Simulation, Modeling, and Programming for Autonomous Robots - 4th Int. Conf., SIMPAR*, Bergamo, v. 8810 *LNCS*, p. 267–278. Springer.

Billington, D., Estivill-Castro, V., Hexel, R., and Rock, A. (2011). Requirements engineering via non-monotonic logics and state diagrams. *Evaluation of Novel Approaches to Software Engineering*, v. 230, p. 121–135, Berlin. Springer Verlag.

Bohren, J. and Cousins, S. (2010). The SMACH high-level executive [ROS News]. *IEEE Robotics & Automation Magazine, IEEE*, 17(4):18–20.

Brooks, R., Connell, J., and Ning, P. (1988). *Herbert: A Second Generation Mobile Robot : By Rodney A.Brooks, Jonathan H.Connell and Peter Ning*. A.I. memo. Massachusetts Institute of Technology.

Brugali, D., ed. (2007). *Software Engineering for Experimental Robotics*, v. 30 *Springer Tracts in Advanced Robotics*. Springer-Verlag.

Chitic, S.-G., Ponge, J., and Simonin, O. (2014). Are middlewares ready for multi-robots systems? *Simulation, Modeling, and Programming for Autonomous Robots - 4th Int. Conf., SIMPAR*, Bergamo, v. 8810 *LNCS*, p. 279–290. Springer.

Dongol, B., Hayes, I. H., and Robinson, P. J. (2014). Reasoning about goal-directed real-time teleo-reactive programs. *Formal Asp. Comput.*, 26(3):563–589.

Estivill-Castro, V. and Hexel, R. (2013a). Arrangements of finite-state machines semantics, simulation, and model checking. *Int. Conf. on Model-Driven Engineering and Software Development MODELSWARD*, p. 182–189, Barcelona. SCITEPRESS.

Estivill-Castro, V. and Hexel, R. (2013b). Module isolation for efficient model checking and its application to FMEA in model-driven engineering. *ENASE 8th Int. Conf. on Evaluation of Novel Approaches to Software Engineering*, p. 218–225, Angers Loire Valley, France. INSTCC.

Estivill-Castro, V., Hexel, R., and Lusty, C. (2014). High performance relaying of C++11 objects across processes and logic-labeled finite-state machines. *Simulation, Modeling, and Programming for Autonomous Robots - 4th Int. Conf., SIMPAR*, v. 8810 *LNCS*, p. 182–194, Bergamo, Springer.

Hayes, I. J. (2008). Towards reasoning about teleo-reactive programs for robust real-time systems. *SERENE RISE/EFTS Joint Int. Workshop on Software Engineering for REsilient SystEms*, p. 87–94, Newcastle Upon Tyne, UK. ACM.

Huang, A., Olson, E., and Moore, D. (2010). Lcm: Lightweight communications and marshalling. *Intelligent Robots and Systems (IROS), IEEE/RSJ Int. Conf. on*, p. 4057–4062.

Kopetz, H. (2011). *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer.

Laprie, J. (1992). Dependability: Basic concepts and terminology. Laprie, J., ed., v. 5 *Dependable Computing and Fault-Tolerant Systems*, p. 3–245. Springer Vienna.

Morales, J. L., Sánchez, P., and Alonso, D. (2014). A systematic literature review of the teleo-reactive paradigm. *Artif. Intell. Rev.*, 42(4):945–964.

Nilsson, N. J. (2001). Teleo-reactive programs and the triple-tower architecture. *Electron. Trans. Artif. Intell.*, 5(B):99–110.

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

Sommerville, I. (2010). *Software engineering (9th ed.)*. Addison-Wesley Longman, Boston, MA, USA.