Learning Middleware Models for Verification of Distributed Control Programs

Ali, Khalili[1], Massimo, Narizzano[1], Lorenzo Natale[2], Armando, Tacchella[1]

Abstract

A substantial effort is underway to make robots useful outside controlled environments and without direct human supervision. In practice, this can happen only after a rigorous certification process which ensures that several requirements, e.g., operation safety, are met. Automated verification of control programs can be part of cost-effective methodologies to support certification, but it is widely recognized as difficult to attain in robotics, because of several traits that characterize robot design and implementation. One such trait is the widespread usage of middleware to implement control programs in a distributed fashion. In these cases, the challenge of verification is made steep mainly by the fact that the correctness of control software depends on middleware components whose structured models might not be available, or too difficult to obtain from their documentation.

Our proposal is to ease the application of automated verification techniques by identifying abstract middleware models in the form of finite-state automata. The identification procedure is itself largely automated, and the only prerequisite is for the middleware to be available for controlled experimentation. Once middleware models are computed, behaviors that would lead to unsafe operation can be spotted automatically on a composition of identified middleware and control software models using model checking techniques. The approach is based on our tool AIDE — Automata IDentification Engine — to identify abstract middleware models, and the model checker SPIN to verify control units. To validate our approach, we consider four different case studies built on YARP publish-subscribe middleware. Our results confirm that AIDE enables the extension of precise engineering methods to distributed control software in robotics.

Keywords: Distributed Control Software, Software Testing and Verification, Automata-based Inference, Model Checking

Email addresses: khalili.ir@gmail.com (Ali, Khalili), massimo.narizzano@unige.it (Massimo, Narizzano), lorenzo.natale@iit.com (Lorenzo Natale), armando.tacchella@unige.it (Armando, Tacchella)

[1]DIBRIS, Università degli Studi di Genova, Via Opera Pia 13 16145, Genoa, Italy

[2]iCub Facility, Istitto Italiano di Tecnologia, Via Morego 30 16163, Genoa, Italy

1. Introduction

The domains of consumer and civil robotics are expanding markets that are considered to have high potential growth in a near future. It is considered that this growth will be propelled by new robots designed for manufacturing and service applications produced in low-volumes by SMEs [1]. In these domains robots will operate in close interaction with untrained human operators in everyday environments. Due to ethical and legal implications, safety and functional soundness need to be demonstrated for such robots to become viable commercial products [2]. For instance, the standard ISO 13482:2014 for robotic assistants posits several requirements, among which clause 5.10 concerning "Incorrect autonomous actions" stands out as peculiar with respect to other cybernetic systems. The clause recites:

> *A personal care robot that is designed to make autonomous decisions and actions, shall be designed to ensure that wrong decisions and incorrect actions do not cause an unacceptable risk of harm.*

Compliance to this requirement can be obtained by enforcing limits for speed and force according to interaction zones, along with physical separation between robot and human. However, such limits can restrict the scope of the interactions, ruling out some of the envisaged applications for collaborative robots. To minimize the need for such restrictions, the control software running in these robotic platforms must be subjected to rigorous verification processes. More specifically, since traditional verification techniques like visual inspection and unit testing can hardly spot ill behaviors in the context of autonomous actions, computer-aided verification can be used to systematically explore the state space in search of requirement violations.

Automated verification can be useful and cost effective, but its application is widely recognized to be challenging in robotics, because of several traits that characterize robot design and implementation. One such trait are distributed control programs, which are pervasive in modern industrial and research robots. Middleware is a fundamental enabler of such architectures, managing hardware heterogeneity, improving software quality, simplifying design and reducing development costs. Middleware for robotics is a rich and constantly evolving field: a survey published in 2008 cites 15 different such systems [26], but more have emerged since then. Different kinds of middleware are characterized by different levels of support, ease of use and applicability to robotics. Generic middleware like CORBA[3], Ice[4], ØMQ[5] and DDS[6] provide complete communication backbones, but are rarely employed in robotics because they lack specific components and have steep learning curves that prevent adoption. Robotic-specific

[3] `http://www.corba.org/`
[4] `http://www.zeroc.com/ice.html`
[5] `http://zeromq.org/`
[6] `http://portals.omg.org/dds/`

middleware — like OROCOS[7], Player [27], YARP [12], Orca [28], ROS [29], OpenRDK [30], Mira [31] and LCM [32], to mention just a few — provide complete functionality for a subset of communication paradigms including RPC, publish-subscribe, transparent inter-process and intra-process communication. In this paper, we will be focusing on distributed control programs built on robotic-specific middleware which simplifies many aspects tied to code development and deployment in robotics, but introduces further issues. Examples include misuse of middleware components leading to unpredicted behaviors, or configuration mistakes leading to execution errors. Insofar all components of control software could be assigned a precise semantics, e.g., in terms of finite-state machines, computer-aided verification is made possible by techniques such as model-based testing and model checking. However, developing a structured model can be difficult for a "black-box" component, i.e., an overly-complex, poorly-documented, or closed-source piece of software. This becomes critical when such component is located in middleware APIs used, e.g., to orchestrate uniform access to hardware resources.

A viable solution to the problem of obtaining compact models of middleware components is automata-based learning — see, e.g., [3] for a comprehensive list of references. The key idea is that the internal structure of a black-box component can be inferred by analyzing its interactions with an embedding context. Learning algorithms supply the component with suitable input test patterns to populate a "conjecture" automaton by observing the corresponding outputs; then, they check whether the conjecture is behaviorally equivalent to the actual component. Once identification of different kinds of abstract models of middleware is enabled by automata-identification techniques, the correctness of the whole control software can be checked resorting to various techniques. In this contribution we explore the pairing between automata-based identification and model checking — see, e.g., [4] for an overview. In short, once the abstract model of a middleware component is obtained, it is combined with the abstract model of the code relying on the component, and the composition is checked against some prescribed property. Model checking, when practically feasible, ensures an exhaustive exploration of all possible behaviors, and thus returns a formal correctness statement about the whole control software.

Our work is inspired by previous contributions, mainly in the field of automata-based learning and verification. The pioneering work in this domain is *Black-box checking* (BBC) [5] which challenges the problem of verifying black-box systems trough model checking. Since a prerequisite for model checking is the provision of a model which is not available in black-box systems, BBC combines well known automata learning algorithms — namely, Angluin's L^* [6] — and model checking techniques together. *Adaptive model checking* (AMC) [7] is an extension of BBC where it is assumed that some model of the system to be verified exists, but it might be inaccurate or partially obsolete. AMC research shows that initializing the learning algorithm with existing information can im-

[7] `http://www.orocos.org/`

prove the performances of the whole verification process when compared to BBC approach. Applications of automata-based learning in robotics exists, but are widely different from ours in terms of scope and purpose. For instance, an approach to teaching by demonstration is about enabling a robot to learn and refine representations of complex tasks [8]. Through generalization, the robot can incorporate several demonstrations of the same task into a single graph-like representation. Natural feedback cues provided by the teacher through speech allow the robot to further refine this representation. Map learning is another application where automata-based inference was applied [9].

Considering the current state of the art, our main contributions are the following:

- The application and experimental evaluation of automata-based inference and verification to distributed control programs — whereof some preliminary evaluations can be found in previous works of ours [10, 11].
- Relevant case studies about the identification of YARP [12] components, and several examples to demonstrate how the identified models can support bug-finding in control programs.
- An extensive experimental analysis is carried out with our tool AIDE [15] — Automata IDentification Engine, combined with the state-of-the-art model checker SPIN [16]. AIDE was developed entirely in the context of this research project, and SPIN is integrated in our verification work-flow to support YARP developers with automated bug-finding.

Our choice of YARP is dictated by several reasons, including a deep knowledge of the platform, and a fairly large installed base due to the adoption of YARP as the standard middleware of the humanoid iCub [13] and COMAN [14]. From a technical point of view, YARP provides a publish-subscribe framework which offers the advantage of a relative simplicity, yet presents many of the verification challenges that are intrinsic of middleware in distributed control architectures. As mentioned above, while BBC and AMC works tackle problems similar to ours, their contribution is mainly theoretical and oriented to computer-aided design of integrated circuits. On the other hand, previous application of automata-based inference in robotics are not connected with the problem of software verification. To the best of our knowledge, this is the first contribution about automata-based identification from black-box middleware that shows a potential to enable precise assessment of distributed control code built on top of publish-subscribe middleware.

The remainder of this paper is organized as follows. In Section 2, a short summary of background will be provided. Section 3 introduces and motivates our YARP-based case studies by a working example. Section 4 presents our experiments on identification, and Section 5 considers verification of identified models plus software built on top of them. Finally, concluding remarks and possible directions of future works are given in Section 6.

2. Background

This section provides a short review and some background definitions related to automata-based identification, model checking and middleware for robotics. In the third subsection we also briefly introduce YARP, the middleware whereon our case studies and experimental results are based.

2.1. Automata-based Identification

Broadly stated, the goal of model identification is to build a mathematical model of an unknown target system. Automata-based identification (also, automata learning) amounts to learn a model of a system in terms of some finite-state machine. We consider a setting in which the target system can be experimented with, and experimental results are collected to learn a model — also known as *active learning*. This field was pioneered by Gold [17], and later refined by Angluin [6] with her L^* algorithm to learn deterministic finite-state automata (DFA). An adaptation of L^* for identifying deterministic Mealy machines (DFMs) was first developed by Niese [18] and it was further extended by Shahbaz [3], and by Irfan [19]. DFMs differ from DFAs because, besides making state transitions on input symbols, they also emit some output symbol given the current state and current input. For this reason, DFMs are more adequate than DFAs to describe components that interact through input/output channels. In the following, we consider yet another finite-state model known as interface automaton. Interface automata fit our context better than DFMs, because they can model reactive systems that interact with their surrounding environment in an asynchronous way. Formally, an *Interface automaton* (IA) as a quintuple $P = (I, O, Q, q_0, \rightarrow)$ where I is a set of *input actions*, O is a set of *output actions*, Q is a set of states, $q_0 \in Q$ is the *initial state* of the system, $\rightarrow \subset Q \times (I \cup O) \times Q$ is the *transition relation*. The sets O, I and Q are always finite, nonempty and mutually disjoint. Provided that reasonable technical conditions are met, algorithms to learn DFMs can also be adapted to learn IAs [20]. In order to learn IAs corresponding to middleware components we use our open-source tool AIDE — Automata IDentification Engine[8]. Currently, owing to its learning algorithm for non-deterministic Mealy machines [15], AIDE is the only publicly available automata-based identification tool which features learning algorithms for behavior non-deterministic IAs. While we did not use such distinctive feature in this work, it could be useful for future extensions to deal with domain abstractions that introduce non-deterministic behaviors — see [15] for relevant examples.

2.2. Model Checking

A fundamental problem in most system engineering tasks is to check whether a given design will satisfy all the intended requirements. Model checking [21, 22]

[8] AIDE is developed in C#; source code and documentation can be downloaded from `https://aide.codeplex.com/`.

is a computer-aided reasoning technique that checks requirements expressed in temporal logic on systems modeled as finite state automata — for a review of model checking and related techniques see [4]. In principle, model checking entails the exploration of the whole space of behaviors generated by executing the automaton starting from some initial conditions. If a violation of the property is found among such behaviors, then a counterexample reproducing the offending behavior is returned. In practice, the space of possible behaviors can be huge, even for relatively small models. For this reason, several techniques have been introduced to make model checking feasible on designs of practical interest. Among such techniques, one may distinguish among those which maintain an explicit representation of explored states [16] — so called *explicit state* model checking — and those which encode states as logical formulas and explorations as operations on such formulas [23] — so called *symbolic* model checking. While both formulations have their own pros and cons, they both succeeded in making model checking a practical computer-aided technique to verify circuits and protocols — see, e.g., [24].

In this work, we use model checking in order to assess properties of software built on top of black-box middleware. The target model is obtained by composing (*i*) IAs learned by AIDE from middleware with (*ii*) automata obtained from software. In the latter case, it is important to notice that we assume that software can always be modeled as a finite state machine, because the overall model checking problem could be undecidable otherwise — see, e.g., [25]. Our tool of choice is SPIN [16], an explicit-state model checker which inherently supports a parallel computation model based on asynchronous composition and shared memory. In SPIN, models are specified in a language called PROMELA (Process Meta Language) and correctness claims are specified in the syntax of standard linear temporal logic (LTL). As such, properties of interest that can be checked with SPIN include absence of blocking synchronization, deadlock and starvation, as well as mutual exclusion preservation. While not exhaustive of the set of properties that might be desirable, they are by far the most common causes of software malfunction in concurrent and distributed systems, so we chose to focus on them in our investigation.

2.3. Publish-subscribe middleware

The publish-subscribe paradigm has become increasingly dominant in robotics because it naturally supports the development of asynchronous and distributed control applications. There are several variant of publish-subscribe architectures each implementing different communication patterns. With reference to the terminology introduced in [33] publish-subscribe offers three levels of decoupling mechanisms: *space decoupling*, *time decoupling* and *synchronization decoupling*. *Space decoupling* is achieved when components produce messages without being explicitly aware of the number and location of the receivers. *Time decoupling* guarantees message delivery even if senders and receivers are not active or connected at the same time. Finally, *synchronization decoupling* requires that messages are sent and received asynchronously by the communicating entities. When communication is asynchronous the issue is whether to guarantee

that messages are correctly received by slow recipients. Persistence is therefore another key property of publish-subscribers architectures [33]. Robotics applications are by nature driven by the need to implement real-time control loops: for this reason robotic middleware often implements policies that attempt to minimize latency at the cost of dropping messages.

Given its importance in robotics we focus our case studies on a publish-subscribe architecture, in particular using the YARP library [12]. YARP is a port-based middleware that provides a portable abstraction for the operating system and implements a variant of the publish-subscribe paradigm, i.e. the *observer pattern* [34], which is a type of distributed publish-subscribe providing space and synchronization decoupling. YARP implements a port-based software abstraction [35]. Objects of the `Port` class can deliver messages of any size and type across a network, using a number of underlying protocols — including shared memory. Ports can be configured to implement publish-subscribe with different levels of decoupling and dynamically reconfiguration of connections and protocols.

In YARP ports can be configured for synchronous or asynchronous communication through read and write primitives. In asynchronous communication the latter can be blocking or non-blocking. Using `Port` objects, a component which performs a synchronous write waits until all receivers confirm reception of the message. Similarly a component which performs a synchronous read waits until a new message is received by the port. By default `Port` objects in YARP are configured for both synchronous read and write because it guarantees correct delivery of messages without extra code. To support synchronization decoupling YARP provides the `BufferedPort` class. This is a type of port configured for asynchronous operations, i.e., `BufferedPort` objects are able to store and handle messages internally — either for transmitting or receiving them — using dedicated threads. In this case, different policies can be used to buffer messages: First In First Out (FIFO) policy uses a queue that grows and guarantees that no message is dropped; Oldest Packet Drop (OPD) policy uses fixed size buffers in which new messages overwrite old ones to guarantee minimum latency. Read operations in a `BufferedPort` object can be blocking in case we want execution to wait for incoming packets or non-blocking for fully asynchronous operations.

To summarize, in this paper we will consider the following objects and configurations:

- `Port` objects in which read and write are blocking and sender and receiver strictly synchronized;
- `BufferedPort` objects in which read and write are asynchronous, read operations are blocking, and buffering policy is FIFO or OPD;
- `BufferedPort` objects in which, in addition, read operations are non-blocking and buffering policy is FIFO or OPD.

These different communication modalities can have a large impact on the behavior of the components and, if not properly understood, can lead to unexpected behavior and introduce subtle bugs. We will see that our approach

```
1: procedure WRITER
2:     Input P1, P2, Q1: BufferedPort
3:     !Initialize P1 and P2 as FIFO
4:     Connect P1 to Q1
5:     while true do
6:         for i = 1 to N do
7:             Write message m to P1
8:         end for
9:         Read message from P2
10:    end while
11: end procedure
```

```
1: procedure READER
2:     Input Q1, Q2, P2: BufferedPort
3:     Initialize Q1 and Q2 as FIFO
4:     Connect Q2 to P2
5:     while true do
6:         for i = 1 to N do
7:             Read message m from Q1
8:         end for
9:         Create a message m'
10:        Write m' to Q2
11:    end while
12: end procedure
```

Figure 1: An example code client (left) and server (right). In the server it is easy to introduce a subtle bug: if the buffered port is not configured to use FIFO buffering (line 3) the server may loose a packet and wait forever.

enables learning models of different types of ports and testing user code that relies on them.

3. An Explanatory Example

In order to understand how the techniques described in the previous section are applied, we introduce a simple example, yet one that contains all the ingredients that we deem relevant.[9] Let us assume that we have two components WRITER and READER whose pseudo-code is presented in Figure 1. WRITER sends some constant number of packets, say N, and READER waits for them, processes them, and finally returns an acknowledgment to WRITER. After the acknowledgment packet is received, the procedure starts again. Inside a robotic control architecture, WRITER could be a component providing some service, e.g., an *Object Detector* producing coordinates of the detected object in the 3D space; READER could be a component which relies on such service, e.g., a *Gaze Controller* receiving object coordinates to control the head of the robot to gaze at the target point.

Given the code in Figure 1, a problem may arise if `BufferedPort` objects are not configured correctly. Since communication is asynchronous, packets must be queued if READER is not fast enough to match WRITER. To do this, ports

[9] This example is already described as a case study in Section 3 of [11]. Notice that the total run-time herein reported for identification of various models is at least one order of magnitude larger than the one reported in [11]. This discrepancy is due to a different value of T_{max} which is set to 50 ms in [11] and to 500 ms in our current experiments. The reason of the different choice is entirely due to the different interfacing mechanism used with YARP: direct software connection in the case of [11] and network connection in this paper. While more general, network connection forces us to consider a large enough observation time to be sure that network delays do not cause mistakes in the identification process — see Section 4 for more details.

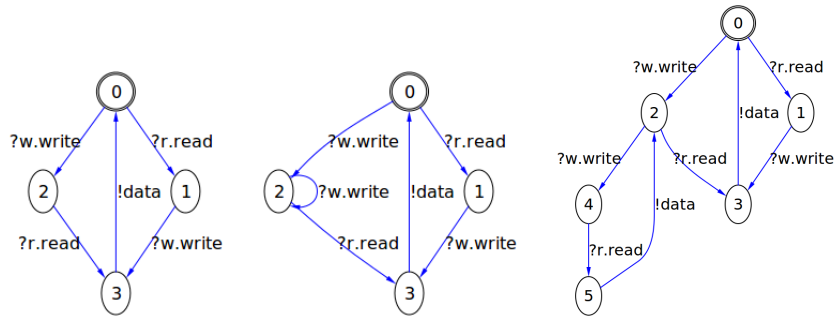

Figure 2: Behavior of a `Port` with one writer w and one reader r (left), behavior of a `BufferedPort` with OPD (center) and FIFO (right) policy. The input actions *r.read* and *w.write* denote the receiver thread reading from a port and sender thread writing to a port. The *data* refers to output action of delivering data to the receiver. By convention, we append to the name of a transition the symbol ? (resp. !) to denote that it is an input (resp. output) transition and the initial state is represented with double circle.

must be configured for FIFO buffering policy explicitly because asynchronous operations in buffered ports adopt the default OPD policy to comply with real-time constraints. Unfortunately, the code in Figure 1 relies on the fact that all N packets sent by Writer must be received by Reader lest the two process deadlock. In order to detect misconfiguration of YARP in such settings, a model checker requires the automata corresponding to Reader, Writer and also a model of the underlying communication services provided by YARP.

To infer a model of read/write services through YARP, we consider the behavior of one port with just one writer `w` and one reader `r`. In particular we identify the port model in three different configurations: standard `Port` objects and asynchronous `BufferedPort` objects with FIFO and OPD policy. The resulting models are depicted in Figure 2. In the case of a standard port — Figure 2 (left) — identification requires 600 CPU seconds overall.[10]. Here, the type of communication is of a "send/reply" type, wherein the writer and the reader are tightly coupled. In the case of a `BufferedPort` object — Figure 2 (center, right) — the writer and the reader enjoy more decoupling, in the sense that YARP takes care of the lifetime of the objects being transmitted through the ports by pooling them, and handling the pool in accordance with the decided policy. By default, a `BufferedPort` will follow a OPD policy, i.e., it will keep the newest message received only — Figure 2 (center). This is because in the implementation of control loops, it is often preferable to reduce the communication latency at the cost of dropping (late) packets. If FIFO mode is enabled — Figure 2 (right) — YARP keeps all received messages. Since in this mode the resulting model would be infinite-state, we limit the system to send

[10] All the experiments have been carried out on a Core2Duo 2.26GHz PC with 4GB of RAM on Ubuntu 12.04.

proc	stmt	cur	act	next
WRITER:	Write m_1 to P_1	0	?w.write	2
WRITER:	Write m_2 to P_1	2	?w.write	2
READER:	Read from Q_1	2	?r.read	3
READER:	(receives m_2)	3	!data	0
READER:	Read from Q_1	0	?r.read	1

Figure 3: Trace leading to a deadlock in the example shown in Figure 1. The trace shows which process is running (**proc**), which statement is executed (**stmt**) and what happens in the YARP IA model: **cur** is the current state, **act** is the (input/output) action, and **next** is the next state.

N packets at most, so that the buffer will never exceed the maximum size of N messages. In Figure 2 (right), we can see the model of a `BufferedPort` (FIFO) with $N = 2$, whose identification required 3900 CPU seconds. It is important to notice that in all these cases, most of the time to infer the model is spent on network communication between the learner and a system wrapper, system resets, and other bookkeeping tasks. Overall, less than 1% of the total time is spent inside the core learning algorithm.

Both WRITER and READER can be translated into an interface automaton and they can be composed with one of the identified automata of the port component shown in Figure 2, considering a specific number of packets N. For $N = 2$, the trace shown in Figure 3 leads to a deadlock if `BufferedPort` is configured in OPD mode as in Figure 2 (center). As it can be seen, two consecutive writes on BufferedPort (with OPD policy) will take the IA into state 3 (here, packet m_1 is dropped); one read followed by data transmission will bring the IA back in the initial state. Finally, the second read will bring the IA into state 1. Unless there is a further write, the IA will not exit state 1. However, WRITER is waiting for an acknowledgment from READER and thus will not perform any further write. On the other side, READER stalls waiting for another packet — which was dropped. If the port were operating in FIFO mode as in Figure 2 (bottom), then the trace above would not result in a deadlock, because both messages will be queued and received in due course.

4. Identification of YARP Components

In this section, we present the results of our experiments to identify some components in YARP middleware with our tool AIDE. To this end, we made some assumptions outlined in the following.

Active and passive states are separated. In all our experiments, it is assumed that active and passive states are separated in our target, i.e., any single state cannot have both input and output actions enabled at the same time. From a practical point of view, when learning a system we wait for some output only

Table 1: Identification of different YARP ports. We consider synchronous `Port` objects as well as asynchronous `BufferedPort` objects including non-blocking reading for both FIFO and OPD mode. The reported measures include number of states $|Q|$ and transitions $|T|$ of the identified model, number of output ("#MM") and equivalence ("#EQ") queries in the learning algorithm, number of experiments on the SUL ("#Exp"), number of cache hits ("%hit") and total time spent on learning.

Model	$\|Q\|$	$\|T\|$	#MM	#EQ	#Exp	%hit	Time (×1000s)
Port	4	5	18	1	218	81%	0.6
BufferedPort, OPD	4	6	42	2	243	61%	1.8
BufferedPort, FIFO	8	11	144	5	280	51%	3.9
BufferedPort, OPD, non-blocking read	4	8	45	2	246	36%	3.4
BufferedPort, FIFO, non-blocking read	8	16	160	4	414	32%	3.3

within a predefined T_{max} interval. If there is no output in such interval, then we conclude that the current state is passive and some input should be given to the system. Technically, we say that *quiescence* of the system is observable by the learning algorithm [20, 36]. Whenever the system is not quiescent, the current state is considered as active and no input is tried in that state. In our experiments, we consider T_{max} as 500ms since this value is large enough to observe all the responses from the System Under Learning (SUL), even those that are severely delayed. Clearly, choosing an even larger T_{max} would not hurt the correctness of the identified model, but it would degrade performances substantially. On the other hand, while a smaller value for T_{max} would increase the performances of identification in terms of total run-time, it may also cause delayed responses from the SUL to be assigned as outputs to states that do not exist in the real system.

System is deterministic. We assume that our system is *output deterministic*, meaning that there is only one observable event in any active state. If the system were not output deterministic, then the — deterministic — learning algorithm would fail. While AIDE incorporates an algorithm to learn non-deterministic models [15], all the case studies that we present in this paper do not require such extension.

Queries are cached. Since the system is assumed to be deterministic, the result of the experiments — also called queries — performed on the SUL can be cached. AIDE implements this mechanism in a way which is transparent for the core learning algorithm. In practice, the cache maintained by AIDE is a tree of execution traces which is traversed each time an experiment is to be run on the SUL, so that calls to the real systems are deferred as much as possible. Caching queries, also helps in determining if a state is active or passive, based on the history of traces executed on the SUL.

Equivalence is Conformance Testing. Once the learning algorithm has enough information about the traces of the SUL, a conjecture about the corresponding IA can be extracted from those traces. At this point, AIDE must establish

Table 2: Scalability analysis for learning `BufferedPort` with FIFO buffering and different maximum buffer size (N). Legend as in Table 1.

N	$\|Q\|$	$\|T\|$	#MM	#EQ	#Exp	%hit	Time ($\times 1000s$)
1	4	5	18	1	218	86%	0.6
2	6	8	72	3	279	52%	3.1
3	8	11	144	5	280	51%	3.9
4	10	14	255	6	395	55%	6.8
5	12	17	504	7	689	64%	10.7
6	14	20	1218	8	1476	72%	22.0
7	16	23	1440	8	1841	69%	35.0

whether the conjecture is equivalent to the SUL or not, but there is no exact way to do so in practice. AIDE approximates equivalence checks with random walk testing: the SUL is experimented with random input sequences, and it is checked whether those sequences are simulated by the conjecture automaton. If a trace cannot be simulated, then the learning algorithm restarts from that trace to build an extended conjecture. In our implementation, the maximum number of tests is a user-defined parameter ζ. The length of tests is randomly selected between $|Q|$ an $K*|Q|$, where $|Q|$ is cardinality of the set of states in the conjecture automaton, and K is also a user-defined parameter. In this way, as the size of the conjecture automaton grows, the length of the tests also grows. In our experiments we set $\zeta = 200$ and $K = 3$, which are always sufficient to learn an accurate model of the SUL. These settings resulted from some preliminary experiments wherein we tried increasing the values of ζ and K until we could make sure that the learned model was accurate, at least for the case studies we consider. Increasing ζ and K even further would not bring any benefit to the accuracy of the learned model, but it could degrade learning performances. On the other hand, combination of ζ and K such that either $\zeta < 200$ or $K < 3$ do not guarantee an accurate learning process in all cases.

Overall, the trade-off between accuracy and performances in the learning process is regulated by the three parameters T_{max}, ζ and K. While the choices we made are adequate for all the case studies we consider, it should be noticed that their setting depends on the SUL and on the learning environment. From the point of view of sheer performances, these parameters should be always set to the least possible value which does not compromise on accuracy. Since verification of software built on top of SULs is our ultimate target, we have chosen to compromise on performances while ensuring that the learned model is accurate. In the end, the learned model is going to be used to check several pieces of software, and therefore the price we pay in learning can be amortized over several verification sessions.

4.1. Identification of YARP ports

We have already showed an example of identification in Section 3 for three conditions: `Port`, `BufferedPort` with OPD and `BufferedPort` with FIFO. In

Table 3: Learning an emulated `BufferedPort` component based on previously identified conjectures in three different scenarios.

N	S1 (ms)	S2 (ms)	S3 (s)
1	33	70	185
2	44	160	1015
3	49	251	2030
4	55	499	3318
5	99	770	4815
6	120	1095	6854

the following, we consider further elements related to YARP port abstractions. In particular we consider the case in which the read primitive is *non-blocking*. Complete identification results for all configurations are presented in Table 1. Considering the learned automata, we see that the behavior of `Port` is similar to `BufferePort` with OPD. In a `Port` object, both read and write are blocking, whereas in a `BufferedPort` with OPD, write is not a blocking primitive: if the output buffer is not empty, the next call to write will overwrite the previous message. In `BufferedPort` with FIFO policy enabled, writing to the port is still non-blocking, but the new message is queued in the buffer and older packets are not dropped — in Table 1 the size of the buffer is $N = 2$.

In case of `BufferedPort`, the size of the buffer clearly matters, because we must ensure that the model inferred can support verification of the code relying on top of it. If such code requires a buffer whose size cannot be bounded at compile-time, then learning and verification with finite-state models are not feasible. On the other hand, if the buffer size can be bounded, our approach can still be used to validate the code. To see the effects of buffer size on identification, we have also considered the effects of increasing the maximum buffer size (N) in a `BufferedPort` when buffering policy is set to FIFO. These results are shown in Table 2. For $N = 1$, the observed model is the same as normal Port objects, and by increasing the size, the size of the model grows gradually. Notice that, if $e(N)$ denotes the number of experiments as a function of buffer size N, then $e(N)$ grows more than proportionally with N.

In order to measure how much of the learning time is due to interaction with the SUL, as opposed to the time spent in core learning tasks, we consider learning in three different scenarios. In each scenario, an emulator is used to interact with the learner in place of the real black-box system. The emulator is "programmed" using a previously-identified model of YARP. In our case, the model is `BufferedPort` with different buffer sizes. In the first scenario, denoted as "S1", the emulator is connected to the learner directly, via native method calls; in the second scenario, denoted as "S2", the emulator is connected remotely via TCP/IP, but the learner can investigate about existence of an event, i.e., quiescence can be queried about; finally, in scenario "S3", quiescence is still observable, but this time the learner must wait a prescribed amount

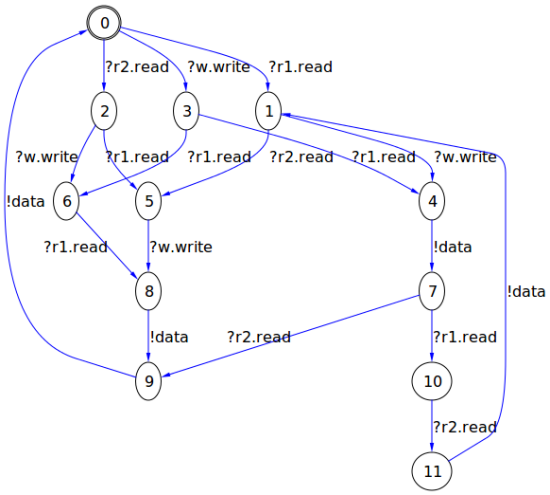

Figure 4: Port with two reader and one writer, in the case in which reader 1 gets connected to the port before reader 2. This is an extension of the model presented in Figure 2 which was with one reader and one writer.

of time to assess whether the state is quiescent or not. Table 3 presents the learning time for scenarios S1, S2, S3. Comparing these results to Table 2, one can see that a tiny fraction of the time spent to learn the real system is required to identify an emulated model. Still, the main portion of learning time is the waiting time required for each action, as witnessed by the fact that in $S1$ and $S2$ times are millisecond whereas in $S3$ learning is measured in seconds.

All the experiments above are carried out considering a configuration with a single writer and a single reader to-from a YARP port, as in Section 3. However, we would like to remark that this is not because of any intrinsic limitation of AIDE. For example, in Figure 4, a model of port communication with two readers and one writer is presented — identification of this model required 342 membership queries and 4 equivalence queries which took about 3.5 CPU hours overall. This specific example also contributes to show that learning abstract models with AIDE is useful to uncover specific and/or undocumented features which affect YARP abstractions. In the case of Figure 4, the unintuitive aspect is that when two readers connect to the same port, the order of connection matters. In particular, after the writer sends a packet to the port, the reader which connected later (reader 2) has to wait for the the first reader (reader 1) to get the packet from the port. Failing to recognize such behavior in code implemented in the readers, might easily cause deadlocks.

4.2. Identification of YARP semaphores

Besides ports, another widely used YARP component is the *semaphore* abstraction. In YARP a semaphore is implemented according to the POSIX standard [37]. In particular, a semaphore has an internal counter which can be

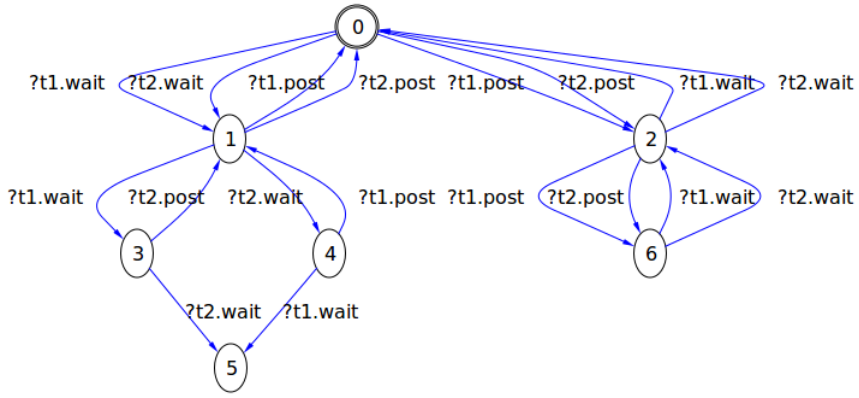

Figure 5: Identified model of YARP semaphore with maximum value 3.

safely incremented or decremented by multiple threads. Indeed, the two basic operations on semaphores are *wait* — for counter decrement — and *post* — for counter increment. The behavior of *wait* is different according to the internal state of the semaphore. If a thread calls *wait* on a semaphore with initial value greater than one, then the semaphore is decremented and the calling thread resumes immediately; if the initial value of the semaphore is 0, then the calling thread must wait until the semaphore is restored to a value greater than 0. An additional *reset* operation can be used to initialize the semaphore. Unless specified otherwise, we assume a default initial value of one, i.e., the semaphore marshals exactly one shared resource. Notice that, since the internal counter of the semaphore can increase, the state space of a semaphore is not bounded at compile time. Like for asynchronous ports with FIFO policy, if we are to learn semaphores using finite-state models, we have to limit the difference between the number of executed *wait* and *post*, i.e., the maximum value of the internal counter, to a constant integer N.

In Table 4, we present the results related to the size of identified models and the time required to learn them. Our identification experiment considers synchronization between two threads for different maximum values of the internal counter. As shown in Figure 5 for $N = 3$ and two threads, we assume that the input alphabet consists of the primitives *wait* and *post* executed by the two threads — a total of four input actions. In the initial state $q = 0$, the counter value is one. From this state it is possible to either decrement or increment the counter, consistently with the POSIX standard definition [37]. States $q = 2$ and $q = 6$ correspond to one and two consecutive post actions, respectively. Since we do not allow the internal counter to have a value higher than $N = 3$, identification does not generate further states. On the other hand, if a wait action is executed in $q = 0$, then we know that the next call will be blocking. Once in $q = 1$, if thread t1 executes wait the semaphore blocks in $q = 3$ until the other thread executes post; symmetrically, if t2 executes wait the semaphore blocks

Table 4: Scalability analysis for learning semaphore of different maximum count N. Legend as in Table 1.

N	$\|Q\|$	$\|T\|$	#MM	#EQ	#Exp	%hit	Time ($\times 1000s$)
1	7	12	65	1	265	38%	0.93
2	10	18	152	3	367	32%	1.90
3	13	24	300	5	533	42%	2.78
4	16	29	620	7	883	54%	4.15
5	19	37	1406	9	1704	61%	7.07
6	22	42	2451	7	3210	64%	11.5

in $q = 4$. In such states, a further wait request by the thread which is still not blocked — either $t2$ in $q = 3$ or $t1$ in $q = 4$ — causes a deadlock represented by state $q = 5$. Notice that the model identified by AIDE suffers from a potential deadlock if state $q = 5$ is reached (Figure 5). Indeed, if the real system uses the semaphore in a wrong way, i.e., if a user tries to acquire two times the semaphore without freeing it, this is exactly the behavior that will occur.

5. Case Studies

In this section, we consider verification of some case studies building on the models identified in the previous one. These cover practical examples in which mistakes in the code can result in unsafe robot behavior. We verify the following properties: correct access to critical sections and absence of deadlocks, starvation and missing messages (packets drop). In the first case study a planner sends via points to a controller using shared memory. Incorrect handling of critical sections may results in corrupted position commands received by the controller. The controller may even skip via points if messages are dropped. Both situations lead to unpredictable behavior and, potentially, to collisions and consequent damages. A deadlock in this case is not immediately dangerous for the robot. However resetting the system to resume execution may require expensive operations, especially in the context of manufacturing. In the second case study the robot tracks a target. Here, a deadlock may have dangerous effects on the tracking performances, requiring the intervention of safety measures like watchdogs or limit checks and consequent need to restart the system. The third case study investigates smooth termination of a software component. In this case incorrect software behavior may prevent proper execution of parking routines leaving the system in an undefined, possibly dangerous, state. Finally, the last example considers a distributed system for resource management. Besides problems due to deadlock and starvation already illustrated, the resource manager (arbiter) may incorrectly assign the same resource multiple times. If the manager is regulating how software components have access to, e.g., the motors, this can result again in unpredictable robot behavior.

In all such cases, YARP components are considered as black-box, whereas the overall application to be verified is white-box, i.e., the source code is used

```
1: procedure PLANNER
2:     Input S_1 ... S_n : Semaphore
3:     Input B_1 ... B_n : Queue
4:     while true do
5:         Produce message m_1 ... m_n
6:         for i ← 1 to n do
7:             if B_i is not full then
8:                 WAIT(S_i)
9:                 Enqueue m_i in B_i
10:                POST(S_i)
11:            end if
12:        end for
13:    end while
14: end procedure
```

```
1: procedure CONTROLLER
2:     Input S_i: Semaphore
3:     Input B_i : Queue
4:     while true do
5:         if B_i is not empty then
6:             WAIT(S_i)
7:             Dequeue m from B_i
8:             POST(S_i)
9:         end if
10:        Process m
11:    end while
12: end procedure
```

Figure 6: Planner - Controller(s) example using YARP semaphore/mutex.

to build a model for verification. To verify applications that rely on YARP, we have to "compose" them with the identified models. This can be done in two steps. The first one is *code abstraction*, i.e., the code of applications which use YARP — usually C++ — is translated into PROMELA code for SPIN. Here, the translation is manual, but in principle, it can be done in an automated fashion. The second one is *translation of identified models*, i.e., conversion of inferred IAs into a PROMELA models. Every model is translated into a process which communicates with two unbuffered channels to simulate synchronous communication. The task of the channels is to receive input actions from environment, and emit output events correspondingly. To make the composition flexible, the input and output channels are the parameters of the process. At any time, the next state is determined by the received input action or the emitted event, and all the transitions are performed as atomic actions. In addition to PROMELA, learned automata can be exported as DOT graphs — for visualization purposes — and C++ — to be used, e.g., as a stub for testing. Once both the application code and the YARP components are translated into PROMELA, it is SPIN which takes care of their composition.

5.1. Planner - Controller(s)

In this case we assume that data coming from a single (high-level) *Planner* module must be dispatched to one or more (low-level) *Controller* modules. For instance, when a complex movement is requested, the planner converts it to a number of task to be sent to joint controllers for execution. We assume that tasks are put in FIFO buffers by the planner for the controllers to read and execute. YARP semaphores can be used to synchronize access to the queues; we are implicitly assuming that the planner and the controllers communicate using a shared-memory model, so that YARP ports are not needed — see Section 5.4 for a more complex scenario involving both semaphores and ports. In Figure 6

we show the pseudo-code for one planner and n controller modules. According to the code, the planner enqueues messages in n different queues. A message for controller i is enqueued only if the queue B_i is not full, with semaphore S_i ensuring synchronized access. Controllers extract message from the queues as long as there is one. Also extraction from the buffer is considered a critical section and it is guarded by a pair of POST and WAIT primitives.

Examples of properties that we would like to check in this case, include mutual exclusion guarantee, absence of starvation, deadlock, non-blocking synchronization and absence of packet loss. Considering one planner and two controllers as in Figure 6, SPIN can verify that all such properties hold within 10ms of CPU time. This requires the specification of additional *fairness constraints*, i.e., a set of directives which forces SPIN to consider only execution traces in which all threads get a fair chance of execution. If we drop this assumption, the example in Figure 6 still guarantees mutual exclusion in the access to buffers, but it could be subject to other fallacies. For instance, starvation may occur because there is nothing in the code that prevents one of the two controllers to be never called. If one of the two controllers is starving, then also packet loss will occur, because the planner will keep producing messages which will be dropped as soon as the buffer associated to the starving thread becomes full. Packet loss may occur also in the absence of starvation, if the planner gets much more processing time than the controllers: in this case the buffers might become full while the planner keeps generating (and discarding) messages. Deadlock, on the other hand, will not happen even under unfair scheduling, unless we change the code to ensure that no packet is loss. In this case, the planner should wait until each controller gets the message before exiting the critical section, but if a controller is starving, then it will never allow the planner to do so, resulting in a deadlock. We checked all these scenarios in SPIN by removing fairness assumptions. Notice that, if the semaphore is used incorrectly, i.e., two consecutive calls by the same user without a release in the middle, than a deadlock arises. The example presented in Figure 6 uses the semaphore correctly and the deadlock never happens. However, also an incorrect use of the semaphore can be spotted by SPIN.

5.2. Tracker - Sensor - Controller

In control programs organized on publish-subscribe middleware, it is common to send sensory information and commands on distinct channels. If this is the case, components may receive information from multiple sources and must synchronize their activities on data received from such connections. In the context of a grasping application, a practical example are the modules whose pseudo-code is shown in Figure 7. Here a *Tracker* identifies the position of the object in the workspace — for example, using stereo vision in the form of three-dimensional coordinates. Through the connection between ports Q_1 and Q'_1, this information reaches a *Controller* which in turn computes the torque commands for the motors. Another component, a *Sensor*, reads sensory data from a Force/Torque sensor placed in the kinematic chain, and publishes it on a separate channel. The controller relies on this information to detect collisions

```
1: procedure TRACKER
2:     Input Q_1, Q'_1: BufferedPort
3:     Input N_1: Natural
4:     Set buffering policy for Q_1 to OPD
5:     Connect Q_1 to Q'_1
6:     for i = 1 to N_1 do
7:         Track object coordinates
8:         Pack object coordinates in m
9:         Send message m to Q_1
10:    end for
11: end procedure
```

```
1: procedure SENSOR
2:     Input Q_2, Q'_2: BufferedPort
3:     Input N_2: Natural
4:     Initialize BufferdPort Q_2 for OPD
5:     Set buffering policy for Q_2 to OPD
6:     Connect Q_2 to Q'_2
7:     for i = 1 to N_2 do
8:         Read torque
9:         Pack torque in m
10:        Send message m to Q_2
11:    end for
12: end procedure
```

```
1: procedure CONTROLLER
2:     Input Q'_1, Q'_2 : BufferedPort
3:     Input N_3: Natural
4:     Set buffering policy for Q'_1 and Q'_2 to FIFO
5:     for i = 1 to N_3 do
6:         Read message m_1 from Q'_1
7:         Read message m_2 from Q'_2
8:         Control joints on the basis of m_1 and m_2
9:     end for
10: end procedure
```

Figure 7: Tracker - Sensor - Controller example, using YARP `BufferedPort` for communication.

Table 5: Results of verification in the Tracker - sensor - Controller example. N_i is the number of iterations performed in the processes of Figure 7: "N_1" in Tracker, "N_2" in Sensor, and "N_3" in Controller; "N" is the maximum size of the buffer in the port; "#States" is the size of the state space explored (combined YARP model + code model); "Time" is the amount of CPU time consumed (in seconds); "Memory" is the amount of memory consumed (in Megabytes); "Conclusion" is the final verdict of SPIN.

N_1	N_2	N_3	N	#States	Time(s)	Memory(MB)	Conclusion
∞	∞	∞	1	85	<0.01	128	OK
100	∞	∞	1	1915	0.01	128	blocking
∞	∞	∞	6	1268	0.01	235	OK
100	∞	∞	6	1939	0.01	235	blocking
1000	∞	∞	6	19039	0.07	236	blocking
100∗	∞	∞	6	117463	0.16	244	OK
1000∗	∞	∞	6	1196563	1.56	335	OK

and control the force exerted at the end-effector — connection between Q_2 and Q'_2. The programmer of the controller must decide how to read data from both connections. The crucial point is that these connections can become inactive, e.g., when no valid target is detected by the *Tracker*, or in a situation where the *Tracker* was closed by the user or died unexpectedly. To reduce latencies, the default behavior of YARP is to have readers wait for data on a port (blocking behavior). In this case, attempting to read data from both channels using the default mode introduces an unexpected blocking synchronization when *Tracker* does not produce any data.

```
1: procedure CONSUMER
2:     Input Q'_1: BufferedPort
3:     Input done: Bool
4:     Set done=false
5:     while !done do
6:         Read a message m from Q'_1
7:         Decode m and do something
8:     end while
9:     Cleanup code
10: end procedure
```

```
1: procedure PRODUCER
2:     Input Q_1, Q_2, Q'_1: BufferedPort
3:     Input N: Natural
4:     Connect Q_1 to Q'_1
5:     for i = 1 to N do
6:         Do something
7:         Prepare message m
8:         Send message m to Q_1
9:     end for
10:    Send message quit to Q_2
11: end procedure
```

```
1: procedure MONITOR
2:     Input Q'_1, Q_2, Q'_2: BufferedPort
3:     Input done: Bool
4:     Connect Q_2 to Q'_2
5:     Read message quit from Q'_2
6:     Set done = true
7:     Call interrupt method on Q'_1 (only in scenario S2)
8: end procedure
```

Figure 8: Producer-Consumer case study with a Monitor using the *interrupt* method.

In Table 5, the result of verification is presented. We consider scenarios where all three modules are executed indefinitely ($N_1 = N_2 = N_3 = \infty$), or where the tracker stops after generating some packets, e.g., $N_1 = 100$. This is to simulate an unexpected termination of one "producer" while the other "producer" and the "consumer" keep running. In all scenarios, YARP ports feature FIFO buffered mode with buffer limit set to either 1 or 6. As shown in the table, when the tracker stops after some packets are sent, a blocking synchronization will occur. Increasing (i) the size of the buffer and/or (ii) the number of steps before the tracker stops makes the verification problem more difficult. In the last two rows of Table 5 we consider non-blocking read configuration for ports. In this case, even if *Tracker* stops, blocking synchronization does not happen. As a final remark, we report that the presence of a blocking synchronization can be always verified by SPIN in less than 2 CPU seconds.

5.3. Handling smooth termination: Producer Consumer with a Monitor

In our experience, a typical example that confuses users is how to manage proper termination of components that use YARP ports. In control programs, components are most often driving hardware devices directly or holding handles to device drivers. It is therefore fundamental that all components are able to perform smooth cleanup of resources on exit. Cleanup in this case involves operations that are critical for the safety of the system and include, e.g., relinquishing ownership of a device driver, or execution of parking routines to leave the robot in a safe configuration. Failure to do so may leave the system in a corrupt state, e.g., a device driver that remains locked or crashes, and even cause physical damages, e.g., motors are powered down while the robot is still moving or it is in a statically unstable configuration. Proper shutdown is something

that people rarely take seriously because when dealing with pure software it is safe — although a bad practice — to rely on the fact that the operating system can kill a process and free all the resources it uses.

To handle process termination properly, YARP ports have a special primitive called *interrupt* which enables stopping all pending read operations on the ports, as well as any future read operation. This primitive is usually called inside a signal handler or upon reception of a specific message on a port before requesting termination of a component. Avoiding this primitive, or misusing it, is most often causing ill behaviors on the controlled robot. As a slightly simplified case study of incorrect process termination, we consider again a producer-consumer scenario. We suppose that we do not know in advance the number of packets produced by *Producer* and we would like to control termination of the components independently using a third component called *Monitor*. In a real application, *Monitor* would be a graphical interface dedicated to monitor and manage execution of components. Requests to start or terminate components would be initiated by a human operator and delivered to the individual components via dedicated YARP messages. In our case study, *Producer* signals termination to the monitor after sending N packets.

The pseudo-code is reported in Fig 8. *Producer* terminates spontaneously after a given amount of iterations N and, before closing, it notifies *Monitor* by sending the message *quit*. *Consumer* cycles through a **while** loop to read messages from *Producer*. It terminates when the flag *done* is set to *true*, executing, after the last iteration, a cleanup routine. *Monitor* is a procedure executed by a thread that shares the same address space of *Consumer*. Its job is to 1) wait for the *quit* message from *Producer*, and 2) ask *Consumer* to terminate by setting the flag *done=true*. In addition, it invokes the *interrupt* primitive of port Q_1' thus unblocking on-going and future calls to read on that port (line 6 in *Monitor*). This is in fact the crucial point: simply forgetting this line of code results in a race condition that may lead to a deadlock. To see this, assume that *Consumer* has parsed all the messages, then checks the value of *done*, finds it to be still *false*, and therefore keeps waiting for new messages on line 6. At the same time, *Producer* has finished sending N messages and sends *quit* to the *Monitor* (line 10). Even if *Monitor* can now set the shared variable *done* to *true* (line 6), *Consumer* is blocked waiting messages from *Producer* and has lost forever its chance to exit the **while** loop and execute the cleanup routine, i.e., a deadlock condition.

The result of verification is presented in Table 6, for the two scenarios. In Scenario 1 ($S1$) *Monitor* only sets the variable *done=true* and it skips invocation of the *interrupt* primitive on Q_1'. In Scenario 2 ($S2$) *Monitor* correctly invokes the *interrupt* on Q_1'. We ran the verification on SPIN for different model sizes and different numbers of packets sent to the consumer. In all cases, the model checker correctly identified the deadlock condition in Scenario 1, and correct execution of the cleanup routine in Scenario 2.

Table 6: Result of verification of Producer-Consumer with Monitor and invocation of *interrupt*. N is the number of iterations performed by *Producer*, the *Size* of the buffered ports is 2.

Scenario	N	#States	Time(s)	Memory(MB)	Conclusion
1	50	565	0.01	130	Deadlock
1	100	1115	0.01	130	Deadlock
1	150	1665	0.01	130	Deadlock
2	50	6618	0.01	130	OK
2	100	12818	0.01	131	OK
2	100	19018	0.02	132	OK

5.4. Centralized arbiter

When executing a complex task, it is often the case that a lot of modules participate and their access to shared resources must be synchronized. If the modules are spread across different machines, a shared-memory model like the one used in the Planner - Controller(s) case study is not applicable, and some kind of arbitration mechanism is required instead. YARP by itself does not offer such facility, but it is possible to extend YARP using basic functionalities provided by ports and semaphores. The simplest setting is that of a centralized arbiter: modules that ought to share a pool of resources communicate with the arbiter which disciplines access to the pool through a semaphore. Clearly, some kind of resource-sharing protocol must be implemented, and this is a potential source of bugs if the interaction with YARP components is not handled properly.

A simple but realistic example of synchronization relying on a centralized arbiter is presented in Figure 9. Here, the arbiter is implemented as two threads, namely ArbiterW and ArbiterP. We assume that there are n clients with n known a priori and fixed, so that each of the clients is assigned a unique identifier from 1 to n. A fixed number of resources m are assigned unique identifiers from 1 to m and are marshaled by the arbiter — of course assuming that $m < n$. To synchronize the clients, ArbiterW and ArbiterP share a semaphore for resource counting. Since all the resources are assumed to be available at the beginning, the semaphore is initialized to m. ArbiterW and ArbiterP also share an object of type *Table* to keep track of resource allocation. In particular, we assume that a *Table* object has three (thread-safe) primitives: Assign retrieves the identifier of a free resource to be allocated to a given client; isAssigned queries the table to check whether a given resource is assigned to a given client; and release frees the resources allocated to a given client. Communication between the arbiter and the clients is carried out through $2n$ bidirectional ports. Ports $W_1, \ldots, W_n$ are dedicated to resource allocations, and ports $P_1, \ldots, P_n$ to resource deallocations. The arbiter works by polling the client connections in a round-robin fashion in the ArbiterW thread. If a request is found waiting in some port W_i, then there are two cases. If there is still at least one resource available, the call to Wait returns and the resource returned by Assign is allocated to client i, which is also acknowledged. If

```
1: procedure ArbiterW
2:     Input W_1 ... W_n: Port (BufferedPort)
3:     Input S : Semaphore
4:     Input T: Table
5:     while true do
6:         for all i ∈ {1, ..., n} do
7:             if there is a message on W_i then
8:                 Read assignment request from W_i
9:                 Wait(S)
10:                r ← Assign(T, i)
11:                Write r to port W_i
12:            end if
13:        end for
14:    end while
15: end procedure
```

```
1: procedure ArbiterP
2:     Input P_1 ... P_n: Port (BufferedPort)
3:     Input S : Semaphore
4:     Input T: Table
5:     while true do
6:         for all i ∈ {1, ..., n} do
7:             if there is a message on P_i then
8:                 Read release request r from P_i
9:                 if isAssigned(T, i, r) then
10:                    Release(T, i)
11:                    Post(S)
12:                    Write acknowledgment to P_i
13:                end if
14:            end if
15:        end for
16:    end while
17: end procedure
```

```
1: procedure Client
2:     Input W, P : Port
3:     Input i : Natural
4:     Connect port W to W_i
5:     Connect port P to P_i
6:     while true do
7:         Write a request to W
8:         Read assigned resource r from W
9:         ... Perform some task with r ...
10:        Write a release request about r to P
11:        Read an acknowledgment from P
12:    end while
13: end procedure
```

Figure 9: A simple centralized arbiter implemented on top of YARP port and semaphore abstractions.

resources are exhausted, then ArbiterW blocks until at least one is freed, and so does the client waiting for the acknowledgment. ArbiterP works also by polling client connections cyclically to detect requests for resource deallocation. Notice that, even if ArbiterW is blocked on some request, ArbiterP can still answer to the clients and free resources. As an additional sanity check, ArbiterP releases a resource only if it was allocated to the client sending the release request.

In this case study, to ensure that the centralized arbitration mechanism works properly, we are interested to investigate about robust execution, boiling down to deadlock freeness, mutual exclusion preservation, and starvation freeness. In our experimental setup we consider two different scenarios. In the first scenario ($S1$), the arbiter threads use standard `Port` objects to connect the clients. As a consequence, read operations on ports are blocking and the order by which components are polled has an effect on the behavior of the system. This is a scenario wherein deadlock may occur and, as a result, clients can get blocked. In the second scenario ($S2$), the arbiter uses non-blocking FIFO `BufferedPort` objects which guarantee robust execution. In each scenario, we also consider different kind of models, i.e., systems having $n \in \{3, 4, 5, 6\}$ clients

Table 7: Result of verification for the centralized arbiter. For each scenario (S1, S2), n is the number of clients and m is the number of resources. "R" is the result of verification: "OK", if the property holds, "NO", if the property does not hold, "—" if SPIN exhausts available memory. "D" is the search space depth reached by SPIN, and "T" ("M") is the time (memory) used by SPIN for the verification.

System			Deadlock				Starvation				Mutual Exclusion			
	n	m	R	$D(\times 10^6)$	$T(\times 10^3 s)$	M(GB)	R	$D(\times 10^6)$	$T(\times 10^3 s)$	M(GB)	R	$D(\times 10^6)$	$T(\times 10^3 s)$	M(GB)
S1	3	2	NO	$< 10^3$	$< 10s$	2.798	OK	0.20	$< 10s$	2.86	OK	0.12	$< 10s$	2.29
	4	2	NO	$< 10^3$	$< 10s$	2.798	OK	2.75	0.08	4.02	OK	1.66	0.02	3.36
	5	2	NO	$< 10^3$	$< 10s$	2.798	OK	30.14	1.58	22.40	OK	18.41	0.30	12.24
	6	2	NO	$< 10^3$	$< 10s$	2.798	—	29.81	5.48	30.72	—	65.72	2.91	30.72
	4	3	NO	$< 10^3$	$< 10s$	2.798	OK	5.20	0.15	4.91	OK	3.14	0.03	3.78
	5	3	NO	$< 10^3$	$< 10s$	2.798	OK	80.42	5.23	25.34	OK	47.29	1.07	15.43
	6	3	NO	$< 10^3$	$< 10s$	2.798	—	32.65	3.08	30.72	—	66.47	2.28	30.72
S2	3	2	OK	0.09	$< 10s$	2.83	OK	0.15	0.01	7.421	OK	0.09	$< 10s$	2.83
	4	2	OK	1.21	0.01	3.28	OK	2.06	0.13	7.896	OK	1.21	0.01	3.28
	5	2	OK	12.56	0.25	10.25	OK	20.87	2.11	14.00	OK	12.56	0.25	10.15
	6	2	—	100.00	3.21	30.72	—	20.21	6.03	30.72	—	100.00	3.19	30.72
	4	3	OK	2.57	0.03	3.79	OK	4.55	0.25	8.41	OK	2.57	0.03	3.79
	5	3	OK	36.87	0.55	20.38	OK	64.19	5.11	23.69	OK	36.87	0.55	20.38
	6	3	—	100.00	2.53	30.72	—	59.28	6.75	30.72	—	100.00	2.54	30.72

and $m = \{2, 3\}$ resources.

The results of verification for all the properties of interest on both scenarios are presented in Table 7. The table is divided horizontally in two sections, corresponding to scenarios S1 and S2, respectively. The results about deadlock freeness, starvation freeness and mutual exclusion preservation, are reported into three different groups of columns where, for each property, we report verification results as well as data related to SPIN performances. In Table 7, we can observe that SPIN correctly reports that verification failed when the system may exhibit a deadlock configuration. As we mentioned before, this is always true in S1, and it can be checked with negligible time and effort: SPIN takes less than 10s CPU time to find a system trace that leads to deadlock. As we can see, all the other properties in S1 hold, but the effort to verify them can become substantial. For instance, with $n = 6$ clients it is not possible to verify starvation freeness and mutual exclusion preservation neither with $m = 2$ nor $m = 3$ resources, because SPIN exhausts the available memory — 32GB minus the space to run the operating system and SPIN. In these cases, the search space depth reached by SPIN, i.e., the maximum number of steps that SPIN explored along any trace before exhausting resources, is always in the order of hundreds of thousands of states. Therefore, even if SPIN did not reach a definite answer, the range of exploration is wide enough to increase the confidence of the developer that the system upholds the property for "typical" configurations. Scenario S2 results are similar in terms of CPU and memory usage to those of scenario S1 when it comes to starvation freeness and mutual exclusion preservation. In case of deadlock freeness, checking that the property is satisfied turns to be more expensive than finding violations as in S1. This is because, in principle, the state space of the system must be explored exhaustively in order to ensure that there are no "offending" behaviors, whereas a single trace is sufficient to witness one such

behavior.

6. Conclusions and Future Works

Enabling cost-effective verification of control programs for autonomous robots is one of the key steps towards turning robots from prototypes to products. In this paper, the main question that we seek to answer is whether the difficulties of certifying control software developed on black-box middleware can be overcome, and to what extent. In particular, we focus on distributed control software built on top of publish-subscribe middleware for which no structured models are available. We try to make verification of such software cost-effective in two steps: (*i*) automatizing the induction of middleware models through automata-based identification and our tool AIDE; (*ii*) verifying combined models through model checking and the state-of-the-art tool SPIN.

Considering a full-fledged middleware like YARP and several concrete case studies useful for robotics, the results are favorable. The combination of AIDE and SPIN would help to avoid many pitfalls that even experienced programmers tend to fall into once confronted with distributed resource management of the level of complexity required in modern robotics. Our approach could be also complementary to other techniques proposed in the literature to improve confidence in proper robot operation. For instance, the framework introduced in [38] could be used as a basis to integrate our work and produce model-driven development tools for robust autonomous robots.

As we claim success for the experiments we conducted, we are also aware that many limitations still exist that need to be overcome in the near future to make our approach within the grasp of robot developers. Our research agenda includes the problem of inducing data-intensive components, i.e., black-box modules in which data elements and control structure are intertwined, as well as easing the access to automated verification tecnology like model checking by "hiding" it underneath a model-driven development cycle.

References

[1] Strategic research agenda for robotics in europe 2014-2020, pp. 24-34. URL `http://www.eu-robotics.net/cms/upload/PPP/SRA2020_SPARC.pdf`

[2] K. Eder, C. Harper, U. Leonards, Towards the safety of human-in-the-loop robotics: Challenges and opportunities for safety assurance of robotic co-workers', in: Robot and Human Interactive Communication, 2014 RO-MAN: The 23rd IEEE International Symposium on, IEEE, 2014, pp. 660–665.

[3] M. Shahbaz, Reverse engineering enhanced state models of black box software components to support integration testing, Ph.D. thesis, Institut Polytechnique de Grenoble, Grenoble, France (2008).

[4] C. Baier, J.-P. Katoen, et al., Principles of model checking, Vol. 26202649, MIT press Cambridge, 2008.

[5] D. Peled, M. Y. Vardi, M. Yannakakis, Black box checking, in: Formal Methods for Protocol Engineering and Distributed Systems, Springer, 1999, pp. 225–240.

[6] D. Angluin, Learning regular sets from queries and counterexamples, Information and computation 75 (2) (1987) 87–106.

[7] A. Groce, D. Peled, M. Yannakakis, Adaptive model checking, Logic Journal of IGPL 14 (5) (2006) 729–744.

[8] M. N. Nicolescu, M. J. Mataric, Natural methods for robot task learning: Instructive demonstrations, generalization and practice, in: Proceedings of the second international joint conference on Autonomous agents and multiagent systems, ACM, 2003, pp. 241–248.

[9] A. Rieger, Inferring probabilistic automata from sensor data for robot navigation, Tech. rep., Universität Dortmund (1999).

[10] A. Khalili, L. Natale, A. Tacchella, Reverse engineering of middleware for verification of robot control architectures, in: Proceedings of 4^{th} International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), Springer, 2014, pp. 315–326.

[11] G. Cicala, A. Khalili, G. Metta, L. Natale, S. Pathak, L. Pulina, A. Tacchella, Engineering approaches and methods to verify software in autonomous systems, in: Proceedings of the 13^{th} Intelligent Autonomous Systems (IAS), 2014.

[12] P. Fitzpatrick, G. Metta, L. Natale, Towards long-lived robot genes, Robotics and Autonomous systems 56 (1) (2008) 29–45.

[13] G. Metta, L. Natale, F. Nori, G. Sandini, D. Vernon, L. Fadiga, C. von Hofsten, K. Rosander, M. Lopes, J. Santos-Victor, et al., The iCub Humanoid Robot: An Open-Systems Platform for Research in Cognitive Development, Neural networks: the official journal of the International Neural Network Society.

[14] A. Settimi, C. Pavan, V. Varricchio, M. Ferrati, E. Mingo Hoffman, A. Rocchi, K. Melo, N. Tsagarakis, A. Bicchi, A modular approach for remote operation of humanoid robots in search and rescue scenarios, in: J. Hodicky (Ed.), Modelling and Simulation for Autonomous Systems, Vol. 8906 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 192–205.

[15] A. Khalili, A. Tacchella, Learning nondeterministic Mealy machines, in: Proceedings of the 12^{th} International Conference on Grammatical Inference (ICGI), 2014, pp. 109–123.

[16] G. J. Holzmann, The model checker Spin, IEEE Transactions on software engineering 23 (5) (1997) 279–295.

[17] E. M. Gold, System identification via state characterization, Automatica 8 (5) (1972) 621–636.

[18] O. Niese, An integrated approach to testing complex systems, Ph.D. thesis, Universität Dortmund, Dortmund, Germany (December 2003).

[19] M. N. Irfan, Analysis and optimization of software model inference algorithms, Ph.D. thesis, Universita de Grenoble, Grenoble, France (September 2012).

[20] F. Aarts, F. Vaandrager, Learning I/O automata, CONCUR 2010-Concurrency Theory (2010) 71–85.

[21] J. Queille, J. Sifakis, Specification and verification of concurrent systems in CESAR, in: International Symposium on Programming, Springer, 1982, pp. 337–351.

[22] E. Clarke, E. Emerson, A. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems (TOPLAS) 8 (2) (1986) 263.

[23] J. Burch, E. Clarke, K. McMillan, Symbolic Model Checking: 10^{20} states and beyond, Information and Computation 98 (2) (1992) 142–170.

[24] M. Kaufmann, J. S. Moore, P. Manolios, Computer-aided reasoning: an approach, Kluwer Academic Publishers, 2000.

[25] E. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, Tools and Algorithms for the Construction and Analysis of Systems (2004) 168–176.

[26] N. Mohamed, J. Al-Jaroodi, I. Jawhar, Middleware for robotics: A survey, in: Robotics, Automation and Mechatronics, 2008 IEEE Conference on, IEEE, 2008, pp. 736–742.

[27] T. H. Collett, B. A. MacDonald, B. P. Gerkey, Player 2.0: Toward a practical robot programming framework, in: Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005), 2005, p. 145.

[28] A. Makarenko, A. Brooks, T. Kaupp, Orca: Components for robotics, in: International Conference on Intelligent Robots and Systems (IROS), 2006, pp. 163–168.

[29] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, Ros: an open-source robot operating system, in: ICRA workshop on open source software, Vol. 3, 2009, p. 5.

[30] D. Calisi, A. Censi, L. Iocchi, D. Nardi, Design choices for modular and flexible robotic software development: the OpenRDK viewpoint, Journal of Software Engineering for Robotics 3 (1) (2012) 13–27.

[31] E. Einhorn, T. Langner, R. Stricker, C. Martin, H. Gross, MIRA–middleware for robotic applications, in: Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on, IEEE, 2012, pp. 2591–2598.

[32] A. S. Huang, E. Olson, D. C. Moore, Lcm: Lightweight communications and marshalling, in: Intelligent robots and systems (IROS), 2010 IEEE/RSJ international conference on, IEEE, 2010, pp. 4057–4062.

[33] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131.

[34] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[35] D. Stewart, R. Volpe, P. Khosla, Design of dynamically reconfigurable real-time software using port-based objects, Software Engineering, IEEE Transactions on 23 (12) (1997) 759–776.

[36] J. Tretmans, Testing concurrent systems: A formal approach, CONCUR-99 Concurrency Theory (1999) 779–779.

[37] The IEEE and The Open Group, The Open Group Base Specification Issue 7 (IEEE Std 1003.1, `http://pubs.opengroup.org/onlinepubs/9699919799/` (2013).

[38] S. Bensalem, L. de Silva, F. Ingrand, et al., A verifiable and correct-by-construction controller for robot functional levels, Journal of Software Engineering for Robotics 2 (1) (2011) 1–19.