Safe Run-time Reconfiguration for Event-driven Systems with Pub/Sub in ROS

Changjian Zhang[1], David Garlan[1], and Eunsuk Kang[1]

Carnegie Mellon University, Pittsburgh PA 15213, USA
changjiz@andrew.cmu.edu, garlan@cs.cmu.edu, eunsukk@andrew.cmu.edu

Abstract. Run-time reconfiguration is a critical feature for enabling modern software systems to evolve in response to changing requirements and environmental conditions. There are two challenges to carrying out run-time reconfiguration in a safe manner: (1) how to avoid anomalous behaviors that may be introduced by reconfiguration and (2) how to minimize disruption to the system. Prior work has proposed solutions for certain classes of systems, such as transactional systems. As far as we know, however, no existing approach is designed to enable reconfiguration of *event-driven* systems where components communicate with each other indirectly through mechanisms such as a publish-and-subscribe (Pub/-Sub) pattern. Re-configuring these types of systems is challenging due to the loosely coupled nature of component interactions, but is nevertheless becoming increasingly important in emerging domains such as robotics and the Internet-of-Things (IoT).
This paper presents an approach for safe reconfiguration of event-driven systems that rely on a publish-subscribe architecture. In particular, the approach targets a large class of systems that are deployed on the Robot Operating System (ROS), a popular framework for robotics applications. A formal specification of the proposed reconfiguration method along with its desired safety property is provided in the Alloy language. In addition, a replace algorithm is provided to illustrate how to reconfigure a system without violating the safety property as well as minimize the system disruption. Finally, a prototype implementation in ROS is described.

Keywords: Run-time Reconfiguration · ROS · Formal Methods · Alloy.

1 Introduction

Software systems in emerging domains such as robotics, automotive, and the Internet-of-Things (IoT) are increasingly being deployed into dynamic environments with changing agent behaviors and requirements. Since shutting down or re-deploying the entire software is not a desirable option in many of these systems, they must be able to dynamically *adapt* to changes. *Run-time reconfiguraton* is a type of adaptation activity that involves modifying the configuration parameters of a system (e.g., replacing a component with an updated version or activating a certain feature). For example, in the mobile robotics domain, if a

robot moves from a well-lit area to a darker one, it may need to change its sensor configuration to accommodate this environmental change (e.g., by enabling a heat motion sensor). Run-time reconfiguration becomes critical for safety and reliability if engineers cannot access the robot when it is in mission; or even if they could, it is often more efficient to reconfigure the system dynamically than to recall it and perform the reconfiguration at the development site.

There are two main challenges to performing run-time reconfiguration in a safe manner:

(i) **Ensuring system safety during run-time reconfiguration**: Informally, safe reconfiguration means a system should not exhibit anomalous behavior during a reconfiguration process. However, since the system is still active, a change to the original configuration may disrupt the on-going computations in an unexpected manner, possibly resulting in anomalous outputs or failures. For example, when replacing the motion sensor for a navigational robot, neglecting to stop its movement while the robot is temporally blind may lead to a crash into a nearby object.

(ii) **Minimizing disruption to the system**: According to the prior work [11], part of the system being reconfigured may become unavailable or degraded, but the rest of the system should remain operational. In the context of this work, *disruption* refers to the level of unavailable or degraded functionalities, and reconfiguration strategies should aim to minimize it. In the robotics example, its movement may temporarily be disabled when the motion sensor is being replaced, but its radio communication should remain fully operational.

Prior research has proposed solutions to these challenges for certain classes of systems. For example, Kramer and Magee [11] used a transaction-based model of a system and identified *quiescence* as a sufficient criterion to ensure the consistency of system behavior during run-time reconfiguration. In their model, a transaction is a sequence of message exchanges between two nodes. Their proposed algorithm guarantees that all the initiated transactions can complete within a bounded period of time, while it requires to block all the dependent transactions which may cause more disruption than necessary. This work has been further extended with the approaches that attempt to minimize the disruption to the system [14, 18].

Unfortunately, these existing approaches do not apply to non-transactional systems directly. In particular, our interest is *event-driven* systems, where a set of loosely coupled components interact with each other through indirect communication mechanisms such as a publish-and-subscribe (Pub/Sub) pattern. A key added challenge to re-configuring these types of systems is the loosely coupled nature of their computational model, where components continuously respond to newly generated system events and where there exists, in general, no guarantees about the completion of a task. As far as we are aware, no existing approach to run-time reconfiguration is designed to handle these challenges.

In this paper, we propose a solution for safe run-time reconfiguration of event-driven systems that employ a Pub/Sub architecture. In particular, our approach

targets a large class of system that are deployed on the Robot Operating System (ROS), a popular open-source framework for robotics applications. The key idea behind this approach is a novel algorithm that given an application-specific definition of a safety, determines the minimal set of components that will be affected by a configuration change, and places these components into a degraded mode while ensuring that the safety property remains satisfied.

Specifically, this paper makes the following contributions. First, we propose a semantic model of the behaviors of an event-driven system and its safety property. Second, we define a set of atomic actions for modifying the system structure and present a novel replacement algorithm, which decomposes a high-level reconfiguration task into a sequence of these atomic actions. Third, we present a prototype implementation of the reconfiguration algorithm in ROS. Moreover, previous work did not provide a formal specification or analysis of their reconfiguration strategies. To make our approach more rigorous, we use Alloy [7] to specify our semantic model and verify its correctness.

The rest of the paper is organized as follows. Section 2.1 uses *Quiescence* as an example to identify the key elements of building a reconfiguration strategy for a particular class of systems. Then, Section 2.2 provides a brief introduction to ROS. Section 3 presents our model for event-driven systems. Section 4 defines the safety property of this model and the actions for changing the system structure. Section 5 discusses the implementation in ROS. Finally, Section 6 studies the related work and Section 7 concludes the paper.

2 Background

In this section, we first use *Quiescence* as an example of safe reconfiguration for transactional systems. In the second part, we give a brief introduction to ROS architecture and clarify that why prior work on transactional systems cannot directly apply to ROS systems.

2.1 Definition of Quiescence

Quiescence [11] uses a transaction-based model. A *node* is a basic processing unit which can initiate and service transactions, and a *transaction* is a sequence of message exchanges between two nodes. A transaction should complete in bounded time, and the initiator of the transaction should be aware of its completion. A transaction would cause its participants changing from one consistent state to another. When a node is consistent, it should satisfy the system's global invariants. This property definition is similar to ACID [6] in the database field where a transaction should be atomic and consistent.

In run-time reconfiguration, the on-going transactions may be interrupted leaving the system into an inconsistent state. Therefore, it is critical to ensure all the initiated transactions to complete, and Kramer and Magee identified *Quiescence* as a sufficient state to achieve it. In their solution, a node works in two operation modes: active and passive.

Active: *a node in active state can initiate, accept, and service transactions.*
Passive: *a node in passive state must continue to accept and service transactions, but*

(i) *it is not currently engaged in a transaction that it initiated, and*
(ii) *it will not initiate new transactions.*

Passive mode ensures that a node would not actively change its state, but other nodes can still initiate transactions on it. Therefore, Kramer and Magee defined *Quiescence* as a stronger state:
Quiescence: *a node N is quiescent if:*

(i) *it is not currently engaged in a transaction that it initiated,*
(ii) *it will not initiate new transactions,*
(iii) *it is not currently engaged in servicing a transaction, and*
(iv) *no transactions have been or will be initiated by other nodes which require service from this node.*

To conclude, *Quiescence* ensures the correct behavior of a system by letting affected components not initiate any new transaction. However, this approach requires the system to be transactional.

2.2 Robot Operating System

ROS [1] is a robot software framework providing the communication infrastructure for nodes and a set of building and debugging tools. In ROS, a *node* is a basic computation unit which should be a process. Then, developers can focus on building application nodes under standard communication protocols.

ROS defines two communication protocols: topics and services. *Topic* implements a topic-based Pub/Sub style, and *Service* implements a RPC-based SOA style. Developers can apply various computation models, such as event-driven systems with topics or transaction-based systems with services. Moreover, developers can also build transactional systems with topics. In fact, ROS provides *Action* [1] which is a transaction-like computation model built upon topics, and [9, 16, 19] have shown how developers can build more complex transactional models with the Pub/Sub style.

However, ROS systems mainly use event-driven pattern with the Pub/Sub style, which is not transactional. According to Kramer and Magee's definition, a transaction should complete in bounded time and the initiator should be aware of its completion. But in event-driven systems, components continuously respond to system events and the completion of a computation is unclear. Thus, the transaction model cannot directly apply to event-driven systems.

Pub/Sub is well adapted to scalable and loosely coupled systems [5], and it is suitable for building event-driven systems [3]. Most often, such systems are considered to be easily reconfigurable, adding/removing a publisher/subscriber would not "affect" other nodes. However, this is not always true. Imaging a robot system with an obstacle detector publishing obstacle information and a

motor controller subscribing to this event to adjust the speed. For this system, removing the detector would not directly cause an error, but the function of the motor controller is affected which leads to anomalous behavior or potential failure.

From the above example, we find the semantic dependencies among nodes are also critical to ensure the correct behavior of a system. Similarly, in the transactional model, all the participants of a transaction are semantically related. However, such relationships in event-driven systems are not clear. Moreover, prior work on transactional systems used sequence diagrams to analyze such semantic relationships, but we didn't find a widely accepted notation for event-driven systems which we can apply this analysis. Therefore, this paper first defines a model to specify the semantics for event-driven systems and identifies its safety property.

3 Semantic Model for Event-Driven Systems

This section introduces the semantic model for event-driven systems. We also set up an example system to better illustrate the concepts.

3.1 Problem Setting

The architecture of the example robot system is shown in Figure 1. In this system, an obstacle detector node reads image data from a camera device periodically to compute whether there are objects in the front. It publishes the obstacle information to topic t every second. Then, a motor controller node subscribes to this topic and adjusts the speed accordingly. For safety concern, if the obstacle detector is unavailable (e.g., being removed from the system), then the motor controller should stop the robot.

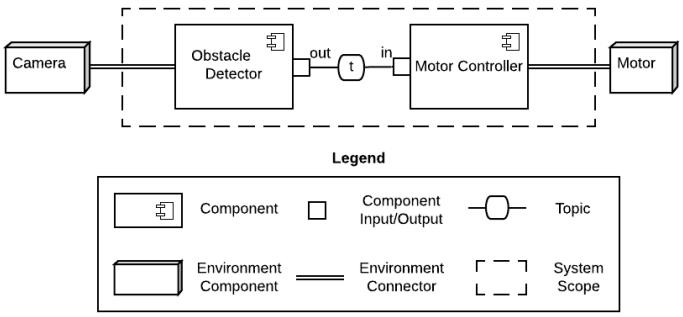
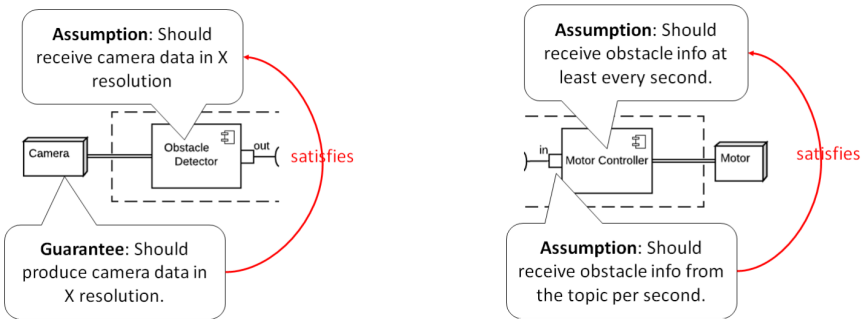


Fig. 1: C&C view of the example robot system

3.2 Functionality, Assumption, and Guarantee

A component should provide a set of functionalities. A *functionality* defines a computation process which takes input data and produces output. In programming languages, it is often represented as a function. In our example, the obstacle detector provides the functionality that takes the camera data and produces the obstacle information every second. The controller takes the obstacle information and produces the proper motor speed.

To make a functionality work normally, it should have the correct implementation and the input data should satisfy some predicates. We assume the implementations are always correct and define the predicates over the input data as *assumptions*. Then, when a functionality works, it changes the environment or produces new data through its associated outputs. The output data should also satisfy some predicates which we define them as *guarantees*. For example, the detection functionality may assume that it should receive camera data in a particular resolution and guarantee to produce the obstacle information every second through the port *out*.



(a) (b)

Fig. 2: (a) shows how the assumption made by a functionality is satisfied by the environment, (b) shows how the assumption is satisfied by an input port.

The system scope delineates the components which can be modified, and components outside the scope are called *environment components*. An environment component can be a hardware device or another system module which would not be modified. These components provide guarantees which can satisfy the assumptions made by the functionalities of components in scope. Figure 2a, for instance, shows the camera device satisfies the detection assumption that it should receive image data.

In an event-driven system, an input port (a subscriber) receiving data can trigger the functionalities of a component. Developers often make assumptions

on the data of the input ports, and these assumptions should satisfy the assumptions of the triggered functionalities. For example, Figure 2b shows that the assumption of input *in* satisfies the assumption of the motor control functionality that it should receive obstacle information.

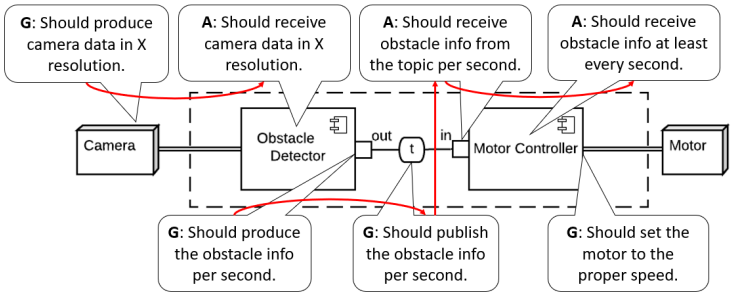


Fig. 3: The annotated C&C view with assumptions and guarantees

We define connectors as first-class entities. A connector (a topic) should provide messages satisfying some guarantee. Hence, an output port (a publisher) can connect to a connector when its guarantee satisfies the guarantee of the connector; and an input port (a subscriber) can connect to a connector when the connector's guarantee satisfies its assumption.

Finally, it forms a satisfaction chain of assumptions and guarantees. Figure 3 shows all the assumptions and guarantees of the example system. To conclude, a functionality works iff its assumptions are satisfied. Assumptions made by a functionality are satisfied by the environmental guarantees or by the input assumptions. Then, the input assumptions are satisfied by the guarantees of the connected connectors, and the connectors are satisfied by the connected output ports. At last, the output ports satisfy the connector guarantees when their corresponding functionalities work.

4 Reconfiguring the semantic model

This section identifies the safety property of the model and presents how to ensure this property during reconfigurations.

4.1 Safety Property

We assume that when all the functionalities in a system work (their assumptions are satisfied), the system should perform the correct behavior. Therefore, we define the safety property, that *the assumptions of all the functionalities should be satisfied*. Apparently, a change to the system configuration may cause violations against it. For example, when removing the obstacle detector from the system, the assumption of the control function is not satisfied.

4.2 Degraded Mode

Affected components may become unavailable or work in a degraded mode in order to ensure the safety property. For example, the passive mode in *Quiescence* requires a node to stop initiating new transactions. In our model, unsatisfied assumptions would cause a functionality becomes unavailable. Thus, this functionality can be degraded by not relying on these assumptions, and its associated output ports should provide degraded guarantees. Therefore, we extend our model as follows:

(i) Each functionality should have a *normal* mode and a *degraded* mode.
(ii) A functionality in its degraded mode should not rely on any assumption.
(iii) If the output guarantee of a functionality in degraded mode satisfy an assumption/a guarantee, then the output guarantee in normal mode should also satisfy it.

For example, the degraded mode for the motor control function could be stopping the motor, which does not rely on receiving the obstacle information.

The current definition of degraded mode is a naive solution. Without any assumption, the functionality is always satisfied but might cause a larger disruption to the system. Another issue is that the degraded function is defined by developers. Developers are responsible for ensuring that the degraded functions can still guarantee the correct behavior of the system.

4.3 Reconfiguration Actions

Previous work [10,15] have identified four primitives for modifying the structure of systems. For our model, we extend them with two actions, and in total they are *remove*, *create*, *detach*, *attach*, *upgrade*, and *degrade*. Any reconfiguration process should be decomposed into a sequence of these actions. Each action must ensure a system to move from one valid state to another.

We specify the model and the actions in Alloy [7] and check their validity and consistency with bounded model checking. Alloy is a specification language based on first-order relational logic. It is suitable for this task because: 1) our model is mainly about the relations among functionalities, assumptions, and guarantees; 2) the Alloy model checker provides an automated analysis for checking assertions. We've also explored TLA+ [12] to specify the model. However, Alloy provides an easier way to assert the correctness of the actions under any system configuration within a bounded model. The follows describe those actions in details[1].

[1] In the following Alloy code, *System, Component, Connector, Port,* and *Functionality* are *signatures*. A *signature* defines a set of entities in the universe. The Alloy keyword **pred** defines a predicate, and actions are defined as predicates by using pre-post-conditions. The keyword **in** refers to operator $\subseteq$, keyword **all** refers to $\forall$, keyword **some** refers to $\exists$, and **no** is the shorthand for $\neg\exists$.

Remove: Remove a component c from the system. Line 3 defines c should be an existing node in the system[2]. Then, Line 4 defines that component c should be isolated. Here, *isolated* means that the component is not connected to other components through any connector. When the node is isolated, it cannot affect other nodes even if it is still producing data. Thus, it is safe to remove it.

```
1  pred remove[s, s': System, c: Component] {
2    // Preconditions
3    c in s.comps
4    all p: (c.inputs + c.outputs) | no conn: s.conns | p in conn.roles
5    // Postconditions
6    s'.comps = s.comps - c
7    s'.avail_funcs = s.avail_funcs - c.funcs
8    // Rest unchanged
9    ...
10 }
```

Create: Add a component c to the system. Line 4 defines that all the functionalities of component c should be in degraded mode. When a node is created, it is initially isolated and its functionalities may not be satisfied. Thus, if all its functionalities are in degraded mode, it is safe to add it to the system.

```
1  pred create[s, s': System, c: Component] {
2    // Preconditions
3    c not in s.comps
4    all f: c.funcs | f.degraded = True
5    // Postconditions
6    s'.comps = s.comps + c
7    s'.avail_funcs = s.avail_funcs + c.funcs
8    // Rest unchanged
9    ...
10 }
```

Detach: Detach a port p from a connector *conn*. There are two cases: detaching an input port and detaching an output port. For detaching an input port, Line 5 define that the corresponding component c should be able to work without it. It means that the functionalities of c should be satisfiable when excluding this input from the component, as shown in Line 18-19. As a result, these functionalities should be either in degraded mode or satisfied by other input ports of the component.

For detaching an output port, if the guarantee of connector *conn* cannot be satisfied by other connected outputs excluding p, as shown in Line 25-28, then all the connected input ports would also become unsatisfied. Thus, all the components should be able to work without these input ports.

```
1  pred detach[s, s': System, conn: Connector, p: Port] {
2    // Preconditions
3    conn in s.conns
4    p in conn.roles
5    p in Input => some c: s.comps | p in c.inputs and safe_no_input[s, c, p]
6    p in Output and not conn_satisfied_no_out[s, conn, p] =>
7       all c: s.comps | safe_no_input[s, c, (conn.roles & Input)]
8    // Postconditions
9    some conn': Connector {
10     conn'.roles = conn.roles - p
11     // Rest unchanged
```

[2] In Alloy, any element is treated as a set. Thus, c is a set of component with one value, and *s.comps* returns the all the components in system s.

```
12      ...
13    }
14  }
15
16  pred safe_no_input[s: System, c: Component, no_i: Input] {
17    all f: c.funcs, a: f.cur_mode.mode_asms |
18      (some i: (f.assoc_inputs - no_i) |
19        satisfy[i.input_asm, a] and input_satisfied[s, i])
20      or
21      satisfy[s.env_guars, a]
22  }
23
24  pred conn_satisfied_no_out[s: System, conn: Connector, no_o: Output] {
25    some c: s.comps, f: c.funcs, o: (f.assoc_outputs - no_o) {
26      o in conn.roles
27      satisfy[f.cur_mode.mode_guars[o], conn.conn_guar]
28      f in s.avail_funcs
29    }
30  }
```

Attach: Attach a port p to a connector *conn*. Line 6-7 define if the port is an output, then its current guarantee should satisfy the guarantee of *conn*. Line 9 defines if the port is an input, then the guarantee of connector *conn* should satisfy its assumption.

```
1   pred attach[s, s': System, conn: Connector, p: Port] {
2     // Preconditions
3     conn not in s.conns => no conn.roles
4     p in (s.comps.inputs + s.comps.outputs)
5     p not in conn.roles
6     p in Output => { some c: s.comps, f: c.funcs | p in f.assoc_outputs and
7         satisfy[f.cur_mode.mode_guars[p], conn.conn_guar]
8     } else {
9       satisfy[conn.conn_guar, p.input_asm]
10    }
11    // Postconditions
12    some conn': Connector {
13      conn'.roles = conn.roles + p
14      // Rest unchanged
15      ...
16    }
17  }
```

Degrade: Degrade a functionality f of a component c. The associated outputs of f would produce their degraded guarantees. If the degraded guarantee of an output does not satisfy the guarantee of its connected connector (Line 6-7) and the connector cannot be satisfied by other outputs excluding the associated of f (Line 4), then this connector and all its connected input ports become unsatisfied. Therefore, all the components should be able to work without these input ports.

```
1   pred degrade[s, s': System, c: Component,  f: Functionality] {
2     f.degraded = False
3     let aff_conns = { conn: s.conns |
4           not conn_satisfied_no_out[s, conn, f.assoc_outputs]
5           and
6           no o: f.assoc_outputs | o in conn.roles and
7             satisfy[f.degrad_mode.mode_guars[o], conn.conn_guar]
8         }
9     | all c: s.comps | safe_no_input[s, c, (aff_conns.roles & Input)]
10      some c': Component, f': Functionality {
11        f'.cur_mode = f.degrad_mode
12        f'.degraded = True
13        // Rest unchanged
```

```
14        ...
15      }
16  }
```

Upgrade: Upgrade a functionality f of a component c to its normal mode. Line 6 defines that all the assumptions made by the normal mode of the functionality should be satisfied by the current system configuration.

```
1  pred upgrade[s, s': System, c: Component,
2               f: Functionality]
3  {
4    // Preconditions
5    f.degraded = True
6    all a: f.norm_mode.mode_asms | asm_satisfied[s, f, a]
7    // Postconditions
8    some c': Component, f': Functionality {
9      f'.cur_mode = f.norm_mode
10     f'.degraded = False
11     // Rest unchanged
12     ...
13   }
14 }
```

Finally, we use **assert** and **check** keywords to verify the validity of the above actions, that is from any valid system configuration with 5 elements, after executing an action, the system still satisfies the safety property. Then, we use **run** keyword to check the consistency, that is it is possible to generate an instance of an action to demonstrate that its preconditions are reachable. The following code use *detach* as an example:

```
1  assert SafeDetach {
2    all s, s': System, conn: Connector, p: Port |
3      valid_sys[s] and detach[s, s', conn, p] => valid_sys[s']
4  }
5  check SafeDetach for 5
6  run Detach {
7    some s, s': System, conn: Connector, p: Port |
8      valid_sys[s] and detach[s, s', conn, p] and valid_sys[s']
9  } for 3 but exactly 2 System
```

4.4 Replace Algorithm

With the formal specification of the model and the actions, we are confident that when the preconditions are satisfied, executing the actions won't violate the safety property. Thus, a reconfiguration process should change the system in a proper sequence where the preconditions of each step are satisfied. There may be more than one possible sequences to change from one configuration to another, and the algorithms to generate such sequences should minimize the system disruption. As a counterexample, an algorithm which degrades all the functionalities can still satisfy the safety property.

This section introduces an algorithm to replace a node c with c'. In general, it defines the following process:

1. Detach all the ports of c, and before each detaching, degrade the affected functionalities.
2. Remove c.

3. Start the new node c' in its degraded mode.
4. Attach all the inputs of c' to their corresponding connectors.
5. Upgrade functionalities of c'.
6. Attach all the outputs of c' to their corresponding connectors, and upgrade the degraded functionalities of other nodes.

Algorithm 1 Replace algorithm

```
1: procedure DegradeByConn(conn)
2:     inputs ← connected inputs of conn          ▷ conn could be a set of connectors
3:     cs ← corresponding components of inputs
4:     for all c in cs do
5:         for all f in AffectedFuncs(c, inputs) do
6:             Degrade(c, f)
7:         end for
8:     end for
9: end procedure
10:
11: procedure Degrade(c, f)
12:     conns ← AffectedConns(f)
13:     DegradeByConn(conns)
14:     ExecDegrade(c, f)
15: end procedure
```

The pseudo-code in Algorithm 1 shows the most critical procedures for degrading functionalities. In a system without cyclic dependencies, the algorithm finds the affected functionalities recursively by reusing the expressions defined in Section 4.3. In particular, these functions are:

– AffectedFuncs(c, i) in line 5 returns the set of functionalities of component c which cannot be satisfied when excluding the input(s) i, that is

```
{ f: c.funcs |
    some a : f.cur_mode.mode_asms |
      (no i2: (f.assoc_inputs - i) |
        satisfy[i2.input_asm, a] and input_satisfied[s, i2])
      and not satisfy[s.env_guars, a]
}
```

– AffectedConns(f) in line 12 returns the set of connectors which cannot be satisfied when f is degraded, that is

```
{ conn: s.conns |
    not conn_satisfied_no_out[s, conn, f.assoc_outputs]
    and
    no o: f.assoc_outputs | o in conn.roles and
      satisfy[f.degrad_mode.mode_guars[o], conn.conn_guar]
}
```

Justification of correctness: Degrading a functionality won't violate the safety property. This theorem is trivially true because the algorithm finds out and

degrades the functionalities which violate the preconditions before executing an action.

Justification of minimized disruption: According to the algorithm, if a node is not connected to the node to change through any connector path, it is not in the search space, and thus would not be degraded. If a node is connected, the algorithm finds out the ones which violate the preconditions and only degrades these functionalities. Thus, functionalities that are still satisfiable are not degraded.

However, it is not guaranteed to always find out the minimal set. The algorithm removes the old node first and then adds the new one, yet it is not always necessary. It may not need to degrade the connected nodes if adding and attaching the new node first. But if the system defines a constraint over the connector such as it should only have one output port connected, then this strategy would not work. The trade-off is that the current algorithm does not need to deal with such constraints which reduces its complexity.

5 Implementation in ROS

This section presents the ROS implementation of the reconfiguration strategy, which is developed in Python.

5.1 Assumptions and Guarantees

The most critical concepts in our semantic model are guarantees, assumptions, and their satisfaction relations. Assumptions and guarantees are in fact predicates against data. There are two common characteristics of data in ROS: (1) each topic is associated with a specific message type; (2) developers often specify a publisher producing messages in a particular rate. Therefore, we can use these two factors to compose predicates, such as:

(i) An output port produces nothing.
(ii) An output port produces messages in type T.
(iii) An input port receives messages in type S at x Hz ...

Then, we can build the satisfaction rules:

(i) A predicate (a guarantee or an assumption) p satisfied another predicate q iff p and q have the same message type, and the message rate of p is greater or equal to it of q or q does not rely on the rate.
(ii) If a guarantee p defines it produces nothing, it does not satisfy any predicate.

5.2 ROS Reconfiguration Framework

The implementation is based on the work of a CMU MSE-ESE student team in 2017. It introduces a configuration management node to manage the run-time

model of the current system configuration and to accept and execute reconfiguration commands. The user API registers the model information of each node to the management node and then invoke the ROS API to initialize the nodes and connectors.

We refactor the ROS tutorial [1] project as a demo system. The tutorial includes a talker node which publishes "hello word" messages at 10Hz and a listener node which subscribes to this topic. The following code show how to construct the talker node in our framework:

```
1  def talker():
2    comp = Component("talker") # Initialize node
3    pub = comp.create_output("out", String) # Initialize ports
4    # Initialize functionalities
5    talk = comp.create_functionality("talk", outputs=[out])
6
7    def normal():
8        hello_str = "this is %s %s" % (rospy.get_name(), rospy.get_time())
9        rospy.loginfo(hello_str)
10       out.write(hello_str)
11
12   def degraded(): # In degraded mode, the talker is silent.
13       pass
14
15   talk.set_normal_mode(normal, rate=10, outputs=[out])
16   talk.set_degraded_mode(degraded)
17   comp.start() # Run the component
```

`Component` in Line 2 initializes the node with name "talker". In Line 3, `comp.create_output` adds an output port with its name and message type[3]. Also, `comp.create_input` adds an input port with its name and relied assumptions. `comp.create_functionality` in Line 5 adds a functionality with its name and associated outputs and inputs. `set_normal_mode` and `set_degraded_mode` in Line 15 and 16 set the normal and degraded behavior of the functionality. In sum, our API requires developers to explicitly identify system structures and their semantic information. Then, it sends this model information to the configuration manage node, and the management node uses it to build the model.

We implement the replace algorithm defined in Section 4.4. It simply transforms the pseudo-code and Alloy expressions to Python code. Then, when receiving a command, it takes the current model, generates, and executes the reconfiguration process. For example, in the tutorial example, replacing the talker node with a new one would result in the following process: (1) degrade the listener node; (2) detach output *out* of the talker from its connector; (3) remove the talker; (4) create the new talker; (5) upgrade the new talker; (6) attach the output port of the new talker to its connector; (7) upgrade the listener.

In addition, we implement the feature to verify the current system configuration against our semantic model by dumping the model into an Alloy specification. Also, it can verify the correctness of a sequence of actions by checking the model after each reconfiguration step. This is useful when we have more sophisticated algorithms which are hard to proof or verify. For these algorithms, we can verify their generated sequences instead.

[3] `create_output` and `create_input` would not connect the port to its connector. We consider the attaching action should be managed by our framework.

6 Related Work

Kramer and Magee [11] first used a transaction-based model and identified *Quiescence*. However, it is too conservative which requires all the affected nodes to stop initiating any new transaction. With the same model, [14, 18] focus on the current progress of each transaction. They identified that it is safe to change a node if it has not begun its participation in a transaction or has completed its participation. Thus, affected components do not have to completely stop initiating new transactions, and the level of disruption is reduced. [2] considered transaction to be a too strong restriction. They focused on the direct interactions between two connected components and identified *dynamic dependency* to minimize the disruption.

For run-time reconfiguration of Pub/Sub systems, [4, 8, 17] assumed a broker network for dispatching messages in distributed Pub/Sub systems. Then, they focused on ensuring the safety properties of the broker network in run-time reconfigurations. However, our work focus on the safe reconfiguration of the components topology in Pub/Sub. [20] also focused this problem and analyzed how DDS (Data Distribution Service for Real-Time Systems) can preserve the system correctness in dynamic reconfiguration. However, it does not provide a general solution to this problem. [13] presented a formal model of Pub/Sub architecture style in Z and proved its safe reconfiguration strategy which can preserve the stylistic constraints.

7 Conclusion & Future Work

In this paper, we have introduced a solution for safely re-configuring event-driven systems with respect to the two challenges. For the first challenge that how to ensure the safety of a system, we find that the correct behavior of a system relies on its semantic dependencies among nodes. Thus, we present a model which specifies such relations through functionalities, assumptions, and guarantees. Then, we identify that safe reconfiguration means to ensure all the assumptions of functionalities being satisfied.

For the second challenge that how to minimize the disruption, we first define a set of actions to change the system structure and then present the replace algorithm to show how to find the minimal set of functionalities to be degraded. However, according to our justification, it is not guaranteed. In fact, finding the minimal set of affected functionalities of any reconfiguration becomes a more complex searching problem, which could be studied in future work.

Future work would first focus on a more complex demonstration of the strategy and the implementation. Second, we are exploring more fine-grained degraded modes which allow a functionality to work with assumptions which are still satisfiable in reconfigurations. Finally, we plan to design a DSL for ROS which allow developers to specify the system architecture as well as the semantic model. With such DSL, we may check a system's validity in design time, generate code, and initialize and reconfigure the system from the specification.

References

1. Ros - wiki, http://wiki.ros.org/
2. Chen, X., Simons, M.: A component framework for dynamic reconfiguration of distributed systems. In: International Working Conference on Component Deployment. pp. 82–96. Springer (2002)
3. Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R.: Documenting software architectures: views and beyond. Pearson Education (2002)
4. Cugola, G., Picco, G.P., Murphy, A.L.: Towards dynamic reconfiguration of distributed publish-subscribe middleware. In: International Workshop on Software Engineering and Middleware. pp. 187–202. Springer (2002)
5. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM computing surveys (CSUR) **35**(2), 114–131 (2003)
6. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM computing surveys (CSUR) **15**(4), 287–317 (1983)
7. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
8. Jaeger, M.A., Mühl, G., Werner, M., Parzyjegla, H.: Reconfiguring self-stabilizing publish/subscribe systems. In: International Workshop on Distributed Systems: Operations and Management. pp. 233–238. Springer (2006)
9. Jergler, M., Zhang, K., Jacobsen, H.A.: Multi-client transactions in distributed publish/subscribe systems. Proceedings - International Conference on Distributed Computing Systems **2018-July**, 120–131 (2018). https://doi.org/10.1109/ICDCS.2018.00022
10. Kramer, J., Magee, J.: Dynamic configuration for distributed systems. IEEE Transactions on Software Engineering (4), 424–436 (1985)
11. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering **16**(11), 1293–1306 (1990). https://doi.org/10.1109/32.60317
12. Lamport, L.: Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
13. Loulou, I., Jmaiel, M., Drira, K., Kacem, A.H.: P/s-com: Building correct by design publish/subscribe architectural styles with safe reconfiguration. Journal of Systems and Software **83**(3), 412–428 (2010)
14. Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., Lu, J.: Version-consistent dynamic reconfiguration of component-based distributed systems. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. pp. 245–255. ACM (2011)
15. Magee, J., Kramer, J., Sloman, M.: Constructing distributed systems in Conic. IEEE Transactions on Software Engineering **15**(6), 663–675 (1989)
16. Michlmayr, A., Fenkam, P.: Integrating distributed object transactions with wide-area content-based publish/subscribe systems. In: 25th IEEE International Conference on Distributed Computing Systems Workshops. pp. 398–403. IEEE (2005)
17. Parzyjegla, H., Mühl, G., Jaeger, M.A.: Reconfiguring publish/subscribe overlay topologies. Proceedings - International Conference on Distributed Computing Systems (2006). https://doi.org/10.1109/ICDCSW.2006.88
18. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Transactions on Software Engineering **33**(12), 856–868 (2007)
19. Vargas, L., Pesonen, L.I.W., Gudes, E., Bacon, J.: Transactions in content-based publish/subscribe middleware. In: 27th International Conference on Distributed Computing Systems Workshops (ICDCSW'07). p. 68. IEEE (2007)

20. Zieba, B., van Sinderen, M.: Preservation of correctness during system reconfiguration in data distribution service for real-time systems (dds). In: 26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06). p. 30. IEEE (2006)