DOI: 10.15439/2018F64 ISSN 2300-5963 ACSIS, Vol. 15

Reverse Engineering of Legacy Software Interfaces to a Model-Based Approach

Mathijs Schuts*, Jozef Hooman†, Ivan Kurtev‡ and Dirk-Jan Swagerman§
*Philips, Best, The Netherlands
Email: mathijs.schuts@philips.com
†ESI (TNO), Eindhoven, The Netherlands, and
Radboud University, Nijmegen, The Netherlands
‡Altran, Eindhoven, The Netherlands
§Philips, Best, The Netherlands

***Abstract*—Cyber-physical systems consist of many hardware and software components. Over the life-cycle of these systems, components are replaced or updated. To avoid integration problems, good interface descriptions are crucial for component-based development of these systems. For new components, a Domain Specific Language (DSL) called Component Modeling & Analysis (ComMA) can be used to formally define the interface of such a component in terms of its signature, state and timing behavior. Having interfaces described in a model-based approach enables the generation of artifacts, for instance, to generate a monitor that can check interface conformance of components based on a trace of observed interface interactions during execution. The benefit of having formal interface descriptions also holds for legacy system components. Interfaces of legacy components can be reverse engineered manually. In order to reduce the manual effort, we present an automated learner. The learner can reverse engineer state and timing behavior of a legacy interface by examining event traces of the component in operation. The learner will then generate a ComMA model.**

I. Introduction

The high-tech industry creates complex cyber-physical systems. The architectures for these systems consist of many hardware and software components. These components can be self-created or made by a third party supplier. Components interact with each other using software interfaces. Good interface descriptions are crucial for component-based development of cyber physical systems. Typically, however, software interfaces are only described in terms of their signature, i.e., the set of operations. Sometimes also the allowed sequence of operations is specified, for instance in terms of a state machine or a few example scenarios. The timing behavior of an interface is almost never described. For instance, the expected frequency of notifications and the allowed time between the call of an operation and the corresponding response. Violations of assumptions about timing behavior, however, are an important source of errors over the complete life cycle of these systems.

To overcome the drawbacks of current interface definitions, we have developed a Domain Specific Language (DSL), called ComMA as an abbreviation for Component Modeling and Analysis. ComMA [1] is currently used at the business unit Image Guided Therapy (IGT) of Philips for the formal definition of signature, state and timing behavior of software interfaces. ComMA specifies the signature of a server, i.e., the operations it offers to clients and the notifications it can send to clients. In addition, a ComMA interface definition includes a state machine which specifies the allowed interactions between client and server, timing constraints on sequences of operations, and data constraints on the parameters of operations.

Based on a ComMA specification, a large number of artifacts are generated automatically, for example:

- A visualization of state machine, timing and data constraints by means of plantUML[1].
- A Microsoft Word document according to the prescribed Philips template with the interface specification; this also uses comments in the ComMA specification including Doxygen-style comments[2].
- A simulator of the interface based on the state machine.
- Proxy source code in C++ and C# for the middleware technology SSCF of Philips IGT for transparent deployment of software components. SSCF is an abbreviation of Simple Service Communication Framework.
- A monitor which can be used to check whether an implementation of the interface conforms to the specification. This is done based on an execution trace that is recorded or sniffed during the usage of the implemented interface. The monitor checks conformance to the specified state machine and the timing and data constraints.

The monitor is very useful to check interface compliance after software updates or hardware upgrades. The monitor stores the timing information from the trace that is used to check the timing constraints. This information can be visualized to obtain insight in the timing characteristics. This is, for instance, useful when an updated hardware component is obtained from a supplier. Then the impact on the Philips part of the interface can be determined based on the differences between the characteristics of the old and the updated component.

Given the benefits of the ComMA approach, all new major system interfaces of Philips IGT are modeled and checked using ComMA. There are, however, hundreds of existing interfaces and it would be beneficial to apply the power of

[1]www.plantuml.com
[2]www.doxygen.org

the ComMA framework also to these interfaces. A manual transformation would require a large reverse engineering effort. Hence, the goal of the work described here is to support this transformation automatically such that the manual effort is reduced significantly.

Our approach is based on model learning techniques to obtain a first version of an interface state machine in ComMA. The main contribution is that we also learn the timing constraints. Since the learned interface may not be complete and states will not have meaningful names, manual changes will be needed. These changes are validated by the monitor generated by ComMA.

Concerning the model learning techniques, we have experimented earlier with active learning which stimulates the system under learning actively and infers an hypothesis based on the responses of the system [2]. Active learning requires the implementation of an adapter to connect the System Under Learning (SUL) with the learner. This adapter has to deal with behavior of the SUL that does not match the assumptions of the learning techniques, such as a SUL which is not input enabled or a SUL which sends no output or multiple outputs after a stimulus. This technique also requires frequent resets of the SUL which may be time consuming. Furthermore, nondeterminism of the SUL is a problem for active learning.

To avoid these issues, the approach described here is based on passive learning [3] where traces of SUL behavior are used to derive an hypothesis about the state behavior. Our algorithm is based on regular inference [4]. In particular, we use the algorithms described in [5], [6].

A disadvantage of passive learning is that only the behavior that is represented in the used traces will be in the resulting state machine. Hence, compared to the active learning approach, the model might be less complete. In our case, however, this is acceptable since the learned model is intended as a starting point for subsequent manual editing.

Related work

There are several model-based techniques to formally describe interfaces. Related to our approach is the Analytical Software Design (ASD) method [7] which includes formal interface specifications represented as state machines. An ASD interface model plays a similar role as a protocol state machine of UML [8]. An ASD interface not only describes the services offered by the server; it also specifies the operations allowed by the client. So it can be seen as a contract between client and server, similar to the Design by Contract approach [9]. Franca[3] is a related domain-specific language for the definition and transformation of interfaces.

All these approaches lack the ability to describe the timing aspects of the interface behavior and to check if an existing implementation conforms to an interface specification which includes timing constraints. Testing of real-time behavior by means of UPPAAL-TRON is described in [10]. In an industrial case, a timed automata model is obtained by first manually modeling the behavior of the system and next manually tightening the timing tolerances in an iterative ways using model-based testing.

An approach to obtain timing information of a component from execution traces is described in [11]. Models include worst case execution times of method calls. Downside of this approach is that the source code needs to be instrumented to acquire the execution traces and by doing so the timing behavior is influenced. In addition, only the time of a method call is captured, not the timing between events. In our approach the code does not need to be instrumented and timing between all event types is captured.

Structure of this paper

The paper is organized as follows. Section II provides a brief overview of the definition of interfaces in ComMA. Next we describe in Section III how an interface model can be obtained for an existing interface by manual editing. Automated support for reverse engineering of state behavior is presented in Section IV. Next, Sections V & VI, addresses the reverse engineering of state and timing behavior respectively. Results of experiments with our approach are presented in Section VII. Section VIII concludes the paper.

II. Model-based Definition of Interfaces

In this section, we introduce ComMA as far as needed to understand the remainder of this paper. The ComMA framework consists of the following four main languages:

- A language to describe the signature of an interface, see Section II-A.
- A language to capture observed interface interactions in the form of timed traces, see Section II-B.
- A language to describe the behavior of an interface, see Section II-C.
- A language to specify the generators to be used, see Section II-D.

The languages are illustrated by a test interface, called ITest, of a power control unit, see Figure 1. For the sake of explanation we made few modifications to the language instances. A predecessor of this unit has been introduced in [12].

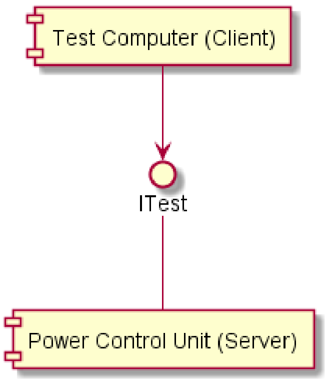

Fig. 1: Interface ITest of a power control unit

[3]franca.github.io/franca/

A. *ComMA Signature*

In a ComMA interface three types of operations are distinguished:

- *Commands* are synchronous operations from client to server. The client receives a *reply* from the server.
- *Signals* are asynchronous operations from client to server. Signals do not have a reply.
- *Notifications* are asynchronous operations from server to client. Notifications do not have a reply.

Listing 1 shows the signature of the ITest. First it defines two enumeration types, Stimulus and State. Next two commands are defined: 1) operation InjectStimulus with one parameter of type Stimulus; it replies a boolean value 2) operation GetState which replies a value of type State. Finally, the notification StateUpdate with one parameter State is defined. Observe that this example does not include any signal.

```
signature ITest {
  types
  enum Stimulus { VideoOnButton SystemOffButton ..}
  enum State { VideoOn VideoOnTransitioning SystemOff ..}

  commands
  bool InjectStimulus(Stimulus s)
  State GetState

  notifications
  StateUpdate(State state)
}
```

Listing 1: Example of a signature

B. *ComMA Trace*

The trace language is used to represent observed interface interactions. They can be, for instance, the captured network traffic or events written to a log file. An event is the occurrence of an operation. The language is independent of the technology used to record interactions; converters transform a technology-specific sequence of observed events to an instance of the ComMA trace language. An example of a ComMA trace is given in Listing 2. This example is based on an interface with the signature described in Listing 1. The listing shows two events, a command and its reply. Note that the time delta (in microseconds) between this event and its predecessor is denoted by “Timestamp” and the keyword “OK” indicates that this is a reply of the preceding command.

C. *ComMA Interface*

The behavior of an interface in terms of the allowed sequences of operations can be expressed in ComMA by the combination of a state machine and a number of constraints. The state machine describes the allowed order of the events between server and client. As an example, Listing 3 presents interface “ITest” which imports the signature of Listing 1.

Listing 3 shows the following:

- A variable “systemStateNotificationPending” is defined and initialized.
- The initial state is “SystemOff”.

```
Timing: 1464181458.066471
Timestamp: 0.000000
src address: 192.168.32.1
dest address: 192.168.32.2
Interface: ITest
Command: InjectStimulus
Parameter: ITest::Stimulus : ITest::Stimulus::VideoOnButton

Timing: 1464181458.072651
Timestamp: 0.006180
src address: 192.168.32.2
dest address: 192.168.32.1
Interface: ITest
Command: InjectStimulus OK
Parameter: bool : true
```

Listing 2: Fragment of a ComMA trace

```
interface ITest{
  variables
  bool systemStateNotificationPending

  init
  systemStateNotificationPending := false

  initial state SystemOff {
    transition trigger: ITest::GetState do:
      reply(ITest::State::SystemOff)
      next state: SystemOff

    transition trigger: InjectStimulus(ITest::Stimulus s)
      guard: (s == ITest::Stimulus::VideoOnButton) do:
        systemStateNotificationPending := true
        reply(true)
      next state: VideoOnTransitioning

    ..
  }

  state VideoOnTransitioning {
    transition trigger: ITest::GetState do:
      reply(ITest::State::VideoOnTransitioning)
      next state: VideoOnTransitioning
    OR
      do: reply(ITest::State::VideoOn)
      next state: VideoOn

    transition guard: systemStateNotificationPending do:
      systemStateNotificationPending := false
      StateUpdate(ITest::State::VideoOnTransitioning)
      next state: VideoOnTransitioning

    ..
  }

  state VideoOn { .. }
}
```

Listing 3: Example of a ComMA state machine

- The first transition is triggered by the “GetState” operation. The replied state value is “SystemOff”. This is a self-transition.
- The second transition is triggered by “InjectStimulus” with parameter “VideoOnButton”. After replying value “true”, the state machine transitions to state “VideoOn-Transitioning”.
- The second state is “VideoOnTransitioning”.
- The first transition of this state is triggered by the “GetState” operation. The replied state value can be either “VideoOnTransitioning” or “VideoOn”. This non-determinism is indicated with the “OR” keyword.

- In the second transition there is “StateUpdate” notification with parameter “VideoOnTransitioning”. Observe that this notification happens only once in the “VideoOnTransitioning” state which is coded by the “systemStateNotificationPending” variable.

Note that implicitly any behavior that is not defined in the state machine is not allowed.

In addition, a ComMA interface definition allows the specification of the timing behavior as a set of timing constraints. Listing 4 shows two examples of timing constraints:

- TimingRule0 describes the allowed time between an occurrence of command “GetState” and its reply. The Lower Specification Limit (LSL) is 2.4 ms and the Upper Specification Limit (USL) is 3.8 ms.
- TimingRule1 shows how constraints on more than two events can be grouped. It describes the allowed time between an “InjectStimulus” event and its reply, and the allowed timing between the reply and an occurrence of the “StateUpdate” notification.

```
timing constraints

TimingRule0
command ITest::GetState
  and reply(ITest::State::SystemOff)
  -> [ 2.4 ms ..  3.8 ms ] between events

group TimingRule1
command ITest::InjectStimulus(
    ITest::Stimulus::VideoOnButton)
  and reply(true)
  -> [ 5.9 ms ..  7.3 ms ] between events
  - [ 76.7 ms ..  165.3 ms ] -> notification
    ITest::StateUpdate(ITest::State::VideoOnButton)
end group
```

Listing 4: Example of a few timing constraints

Note that the ComMA trace of Listing 2 satisfies constraint TimingRule1, since the observed time delta between command and reply in this trace is approximately 6.2 ms which is between 5.9 ms and 7.3 ms.

D. ComMA Generator Specification

ComMA contains a separate language to specify which artifacts should be generated and it also allows the definition of parameters for these generators. An example is given in Listing 5 for a project called “Test” which imports the ITest interface. The project includes multiple generators:

- A “Monitor” to check if a ComMA trace conforms to the ComMA interface; in this case it takes file “Test.traces” as input.
- “SscfHeader”, is a generator that is explained in Section IV of this paper. The generator takes a “ITest.sscfheader” file as input.
- “Minedmodel”, is a generator that is explained in Sections V & VI. The generator takes a “Test.traces” file as input, excludes some parameters and filters out some unsolicited events as explained later.

```
Project Test {
  Compound Interface ITest {
    version
    ``1.0"

    description
    ``Demo project with Test component."
  }

  Generate Monitor {
    trace files
    ``Test.traces"
  }

  Generate SscfHeader {
    header files
    ``ITest.sscfheader"
  }

  Generate Minedmodel {
    trace files
    ``Test.traces"

    exclude parameters int string

    unsolicited events
    "dummyCMD2M"
    "dummyM2CMD"
  }
}
```

Listing 5: Example of a generator specification

III. Manual Reverse Engineering

Existing interfaces can be modeled manually in ComMA. This manual approach is depicted in Figure 2 and consists of the following steps:

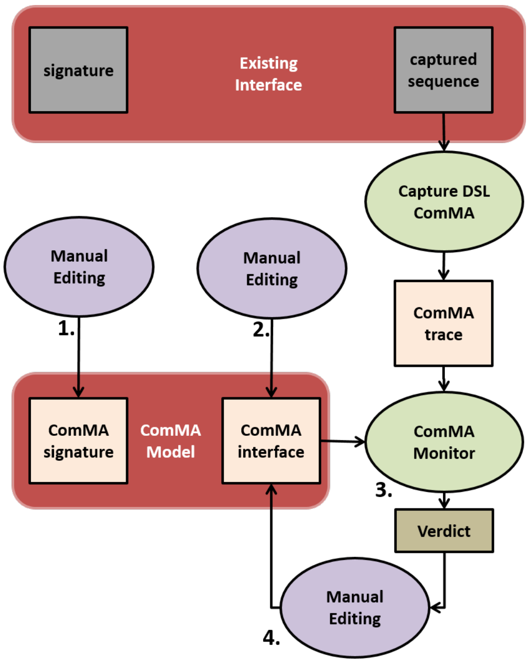

Fig. 2: Manual approach

1) The signature of the interface is defined manually.
2) A first version of the behavior of the interface is defined manually.

3) a) An interaction sequence of the existing interface is captured during execution or testing, for instance by sniffed network traffic or logging of events. From this recorded sequence, a ComMA trace has to be created. Typically this is done by a dedicated DSL.
   b) From the ComMA trace and the manually defined ComMA interface, a monitor is generated using the existing ComMA generator. With this monitor we check if the captured trace conforms to the defined ComMA model.

   In Figure 2, "Verdict" is the outcome of the interface conformance check.
4) The verdict of monitoring leads to three possibilities, assuming the used trace is correct:
   - Fail and the ComMA generator lists the issues; fix the issues in the model.
   - Pass; there are two options:
     – Done, the model captures all required behavior; the engineer has to decide this based on domain knowledge or, for instance, design documents.
     – Not done, extend the model with new behavior.

IV. Automated Reverse Engineering Support

In this section, we describe our reverse engineering approach. It can be seen as an extension of the manual approach presented in Section III, where we automate steps 1 and 2 of Figure 2. The automated approach is depicted in Figure 3.

The automated approach consists of the following steps:

1) We assume the signature of an existing interface is available in some representation. This can, for instance, be an IDL file in case of a COM interface or a header file using macros in C++ for another technology. The aim is to generate a ComMA signature from this representation. This requires a parser that accepts instances of an interface representation. Next a generator to generate a ComMA signature file has to be constructed.

   At Philips IGT, most signatures are available in the SSCF format. Hence, we created a DSL for the translation of a C++ header file with SSCF macros to a ComMA signature file. Listing 6 depicts an example of the SSCF interface description. From this example, the generator will automatically generate Listing 1. We do not discuss this DSL in more detail since the transformation is trivial for the Philips specific SSCF technology. The generator is called "SccfHeader" and requires an SSCF header file as input. Listing 5 show how this generator can be used.
2) Similar to step 3 of the manual approach, the behavior of a legacy interface is manifested by some sequence of events which are translated into a ComMA trace. In this case, the so-called ComMA Learner is used to construct a state machine and timing constraints based on one or more ComMA traces. Hence, we assume that the traces used in the learning are correct.

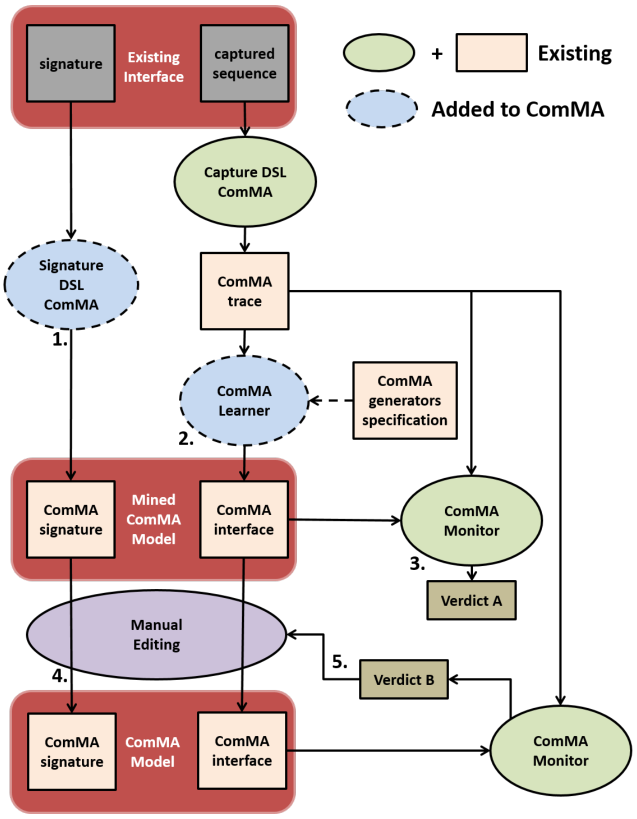

Fig. 3: Interface mining approach

```
SSCFTS1_BEGIN_INTERFACE(ITest)
  SSCFTS1_BEGIN_METHODS(ITest)
    SSCFTS1_INTERFACE_METHOD_1(bool, ITest,
      InjectStimulus, in(Stimulus))
    SSCFTS1_INTERFACE_METHOD_0(State, ITest, GetState)
  SSCFTS1_END_METHODS
  SSCFTS1_BEGIN_EVENTS(ITest)
    SSCFTS1_INTERFACE_EVENT_1(ITest, StateUpdate, State)
  SSCFTS1_END_EVENTS
SSCFTS1_END_INTERFACE
```

Listing 6: Fragment of an sscfHeader file

   a) The generation of a state machine by the ComMA Learner is described in Section V.
   b) The generation of timing constraints by the ComMA Learner is described in Section VI.

   Listing 5 shows how the ComMA Learner is called; the exclusion of parameters is explained in Section V.
3) Next, the existing generator of ComMA is used to generate a monitor and to check if the trace which is the starting point of step 2 indeed conforms to the learned interface. If the learner works correctly, the result should be a pass, so this is mainly a consistency check before continuing with the next steps.

The next two steps should be executed incrementally such that the changes on the model are small and can be easily reverted when they make the monitoring fail.

4) To create a more readable, complete and maintainable

version of the learned ComMA model, it is edited manually. For instance to add meaningful state names, reorder states, or to merge states and transitions.

5) As before, we use the generated monitor to check if the trace of step 2 still conforms to the edited ComMA interface. If not, the error has to be corrected, otherwise more changes can be made.

V. Learning State Behavior

In this section, we describe the learning of state machines. Figure 4 depicts the internal components of the Leaner. The "Serialize" component is used to format ComMA traces into a format which can serve as input for the "Algorithm" component. The "Deserialize" component converts the output of the "Algorithm" component into a ComMA interface.

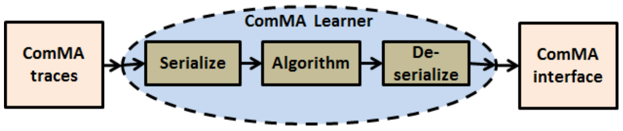

Fig. 4: Components of the learner

The general assumption is that the ComMA traces are correct, i.e., they represent valid behavior of the component.

A. *Serialize*

The "Serialize" component takes ComMA traces as input. It converts these traces into event strings. An event string starts with an interface name, followed by an event name, all parameter values, and finally the event type (command, reply, signal, or notification). Note that the conversion ignores all timing and address information in a trace.

B. *Algorithm*

The "Algorithm" component constructs a state machine based on the work described in [5], [6]. It uses a set of triggers, in our case *Commands* and *Signals*, and a set of listed actions, in our case *Replies* and *Notifications*. Triggers lead to transitions and action lists to states, following the pattern of a Moore machine where the output depends on the state only [13]. Based on one or more sequences of event strings, as a result of the previous component, the algorithm will construct a minimal (non-deterministic) finite state machine consistent with all input sequences. States with the same list of actions are merged, uniting the sets of their incoming and outgoing transitions. Note that this is different from (evidence-based) state merging [14] because the algorithm we use is linear and the resulting state machines might be non-deterministic.

C. *Deserialize*

The "Deserialize" component represents the output as a ComMA interface state machine. This means that the resulting Moore machine of the algorithm has to be transformed into a Mealy state machine where output depends on the state and the input trigger [15]. Moreover, a few restrictions on ComMA state machines have to be taken into account, such as limitations on the number of notifications on a transition. These restrictions are needed to enable the generation of monitors.

Listing 7 contains an example of a learned state machine for the "ITest" interface. Since the traces do not contain state information, the learned states are numbered. Notifications take place on transitions from a separate state with an underscore "_" in the state name. These states are added by the "Deserialize" component to meet the ComMA constraints mentioned in the previous paragraph. Observe the "OR" keyword which indicates that a reverse engineered state machine can be non-deterministic.

```
interface ITest{
  initial
  state s0 {
    transition trigger: ITest::InjectStimulus(
        ITest::Stimulus arg0)
      guard: (arg0 == ITest::Stimulus::VideoOnButton) do:
      reply(true)
      next state: s0_0_0
  }

  state s0_0_0 {
    transition do:
      ITest::StateUpdate(ITest::State::VideoOnTransitioning)
      next state: s1
  }

  state s1 {
    transition trigger: ITest::GetState do:
      reply(ITest::State::VideoOnTransitioning)
      next state: s1_0_0
        OR do:
      reply(ITest::State::VideoOnTransitioning)
      next state: s12
  }

  state s12 {
    transition trigger: ITest::GetState do:
      reply(ITest::State::VideoOn)
      next state: s13
  }
}
```

Listing 7: Example of a learned ComMA state machine

D. *Tuning the Learner*

The ComMA Learner can be tuned to ignore certain parameter values of events. For instance, an *int* parameter that acts like a cookie and is increased every transition might be excluded from the learning process. If we would not ignore the cookie, then the resulting state machine would become very large and restrictive. Hence, a new trace with different cookie values would not be accepted by the monitor. In such cases parameter values can to be ignored. Listing 5, shows how the parameters values for *int* and *string* are excluded from the learning algorithm. This means that the "Serialize" component does not include the *int* and *string* parameter values in the generated string.

VI. Learning Timing Constraints

In this section, we describe how we learn the timing constraints introduced in Section II. The timing constraints are created during step 2b of our automated reverse engineering approach.

Our algorithm assumes that the client initiates the observed interface communication. Hence, observed events from the server are assumed to be triggered as a consequence of a *command* or a *signal* sent by the client. Clearly this holds for a *reply* event since it is caused by a *command* from the client. Our assumption means that a *notification* event is triggered by a *command* or *signal* from the client.

This pattern is used to avoid race-conditions by design. The latter is consistent with the solicit communication scheme in other approaches like ASD [16]. To avoid that *notification* events are triggered by unsolicited events, e.g. periodic alive events, unsolicited events can be filtered out of a trace by instrumenting the ComMA Learner as has been done for the "dummyCMD2M" and "dummyM2CM" events in Listing 5.

As shown in step 2 of Figure 3, the algorithm is fed with a trace of event observations. To learn timing characteristics, it is useful if the trace is long and contains many instances of events occurring in timing constraints. The algorithm performs the following steps on this trace:

1) Step 1 of the algorithm groups events according to the occurrence of trigger events of the client. Hence every event group starts with either a *command* or a *signal*. When in the trace the next event is a *command* or a *signal*, a new event group is created. Otherwise, the event is either a *reply* or a *notification* and it is added to the current event group. *Replies* and *notifications* have two attributes that represent minimum and maximum time differences with the previous event. These attributes are called LSL (Lower Specification Limit) and USL (Upper Specification Limit). In step 1 they are equal and initialized to the value of "Timestamp" of the event (note that this represents the delta time with the previous event). Figure 5 illustrates the event grouping. The output of this step is a list of event groups.

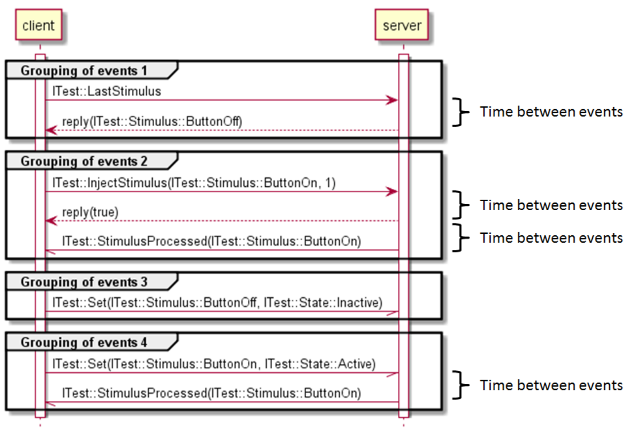
client
server
Grouping of events 1
ITest::LastStimulus
reply(ITest::Stimulus::ButtonOff)
Time between events
Grouping of events 2
ITest::InjectStimulus(ITest::Stimulus::ButtonOn, 1)
reply(true)
ITest::StimulusProcessed(ITest::Stimulus::ButtonOn)
Time between events
Time between events
Grouping of events 3
ITest::Set(ITest::Stimulus::ButtonOff, ITest::State::Inactive)
Grouping of events 4
ITest::Set(ITest::Stimulus::ButtonOn, ITest::State::Active)
ITest::StimulusProcessed(ITest::Stimulus::ButtonOn)
Time between events

Fig. 5: Example trace timing

2) The list of the previous step will typically contain many groups that have the same events. For instance, many groups consisting of signal S and notification N. Only the time difference between S and N might be different. In step 2 the first occurrence of such groups is placed in a new list. Every event group in the new list will become a timing constraint.
3) Next the algorithm iterates over the event groups list of step 1 and matches every event group in it to a unique event group in the list of step 2. When a match is found, the LSL value of an event of the unique group is compared with the matched group. If the LSL value of an event of the matched group is smaller than the LSL of the unique group, then the unique group LSL value is updated with the value of the matched group. Likewise, the USL value of an event of the unique group is compared with the matched group. If the USL value of an event of the matched group is larger than the USL of the unique group, then the unique group USL value is updated with the value of the matched group.
4) Finally, the LSL and USL values of the unique event groups are used to create the timing constraints. The resulting constraints can be reviewed and the LSL and USL values can be relaxed manually in step 4 of our automated reverse engineering approach of Figure 3.

A more detailed formulation of this algorithm is given by the following methods. Method *CreateEventGroups* implements step 1 of the algorithm. It creates the event groups.

```
$\mathit{CreateEventGroups}(events)$ ::=
$group \leftarrow \emptyset$
$groups \leftarrow \emptyset$
FORALL $evt \in events$ DO
  IF $evt.type$ is Command THEN
    IF $group \neq \emptyset$ THEN
      $groups \leftarrow add(groups, group)$
      $group \leftarrow \emptyset$
    FI
    $group.trigger \leftarrow \mathit{Command}(evt)$
    $previousEvt \leftarrow group.trigger$
  FI
  IF $evt.type$ is Signal THEN
    IF $group \neq \emptyset$ THEN
      $groups \leftarrow add(groups, group)$
      $group \leftarrow \emptyset$
    FI
    $group.trigger \leftarrow \mathit{Signal}(evt)$
  FI
  IF $group \neq \emptyset$ THEN
    IF $evt.type$ is Reply THEN
      $action.string \leftarrow \mathit{Reply}(evt, previousEvt)$
      $action.LSL \leftarrow evt.timestamp$
      $action.USL \leftarrow evt.timestamp$
      $group.actions \leftarrow add(group.actions, action)$
    FI
    IF $evt.type$ is Notification THEN
      $action.string \leftarrow \mathit{Notification}(evt, previousEvt)$
      $action.LSL \leftarrow evt.timestamp$
      $action.USL \leftarrow evt.timestamp$
      $group.actions \leftarrow add(group.actions, action)$
    **FI**
  **ELSE**
    *Trace does not start with Signal or Command.*
  **FI**
**OD**
**RETURN** $groups$
```

As an example, consider the trace of Listing 2. Observe that the time difference between the command and its reply is described in the value after the Timestamp keyword of the reply. This time stamp is stored into the LSL and USL attributes of the reply event.

Next we present a helper method that is used in subsequent methods. Method *AreTheSameGroup* determines if two event groups are the same, that is, they have the same trigger and actions.

```
$AreTheSameGroup(group0, group1)$ ::=
$areTheSameGroup \leftarrow true$
**IF** $group0.trigger = group1.trigger$ **AND**
    $group0.actions.size = group1.actions.size$ **THEN**
  **FORALL** $i \leftarrow 0; i < group0.actions.size; i := i + 1$ **DO**
    **IF** $group0.actions[i].name \neq$
        $group1.actions[i].name$ **THEN**
      $areTheSameGroup \leftarrow false$
    **FI**
  **OD**
**ELSE**
  $areTheSameGroup \leftarrow false$
**FI**
**RETURN** $areTheSameGroup$
```

The method *FindUniqueEventGroups* implements step 2 of the algorithm and returns a new list of unique event groups.

```
$FindUniqueEventGroups(groups)$ ::=
$uniqueGroups \leftarrow \emptyset$
**FORALL** $group \in groups$ **DO**
  $isUnique \leftarrow true$
  **FORALL** $group' \in uniqueGroups$ **DO**
    **IF** $AreTheSameGroup(group, group')$ **THEN**
      $isUnique \leftarrow false$
    **FI**
  **OD**
  **IF** $isUnique$ **THEN**
    $uniqueGroups \leftarrow uniqueGroups \cup group$
  **FI**
**OD**
**RETURN** $uniqueGroups$
```

The method *DetermineTiming* implements step 3 of the algorithm. It takes the output of steps 1 and 2 as input and returns an updated unique groups list.

```
$DetermineTiming(uniqueGroups, groups)$ ::=
**FORALL** $group \in uniqueGroups$ **DO**
  **FORALL** $group' \in groups$ **DO**
    **IF** $AreTheSameGroup(group, group')$ **THEN**
      $action.LSL \leftarrow min(action.LSL, action'.LSL)$
      $action.USL \leftarrow max(action.USL, action'.USL)$
    **FI**
  **OD**
**OD**
**RETURN** $uniqueGroups$
```

Using these methods, we create an algorithm to acquire timing constraints in the following way:
$groups = CreateEventGroups(events)$
$groups_{uniq} = FindUniqueEventGroups(groups)$
$timingRules = DetermineTiming(groups_{uniq}, groups)$

As a last step, the timing rules are added to the interface file after the state behavior.

VII. Results

In this section, we present the results of our experiments and an analysis of the results.

A. Experiments

To validate the ComMA Learner we used two cases for which we already constructed an interface manually earlier: the power control unit and a third-party operating table [17]. For the power control case we use a trace called "Trace 1". For the operation table, two traces were used, called "Trace 2" and "Trace 3". The latter two traces are recordings of two different scenarios. Table I shows the characteristics of these traces by listing the number of commands, replies, signals and notifications, together with the types of the parameters.

We experimented with the ComMA Learner on the three traces and the exclusion of certain parameter types. The experimentation results are shown in Table II. In the last column, "Verified" refers to step 3 of the approach described in Figure 3, i.e., the monitoring; "yes" means that we could create a monitor and the verdict was that the trace is accepted by our generated monitor while "no" denotes that we could not generate a monitor because of the size of the state machine.

As explained in Section V-B, the algorithm can take more than one trace as input. Observe that learning based on Trace 2 and Trace 3 separately leads to 33 and 32 unique event groups, respectively, when excluding string and int. Using both traces leads to 55 groups, hence 10 groups are part of both traces. In the "Verified" column, "yes & yes" means that the monitor accepts both traces.

```
interface ITest {
  in all states {
    transition trigger: ITest::dummyCMD2M
    transition do: ITest::dummyM2CMD
  }

  initial
  state s0 { .. }
}
```

Listing 8: Example of a generated ComMA state machine with unsolicited operations

TABLE I: Characteristics of traces

	Command		Reply		Signal		Notification			
Trace	Nr.	Arg. Types	Nr.	Arg. Types	Nr.	Arg. Types	Nr.	Arg. Types	Events Total	Transitions Total
1	39	enum	39	enum bool	0	-	11	enum	89	39
2	0	-	0	-	2964	enum bool string int	2125	enum bool string int	5089	2963
3	0	-	0	-	915	enum bool string int	600	enum bool string int	1515	914

TABLE II: Results of learning experiments

Experiment			Learner Output				
Trace	Excl.	Unique Groups Nr.	Timing Rules Nr.	States	Transitions	Time (in ms)	Verified
1	-	14	10	21	30	17	yes
1	bool	14	10	21	30	6	yes
1	enum	4	0	9	14	3	yes
1	all	4	0	9	14	2	yes
2	-	689	19	2636	3324	342	no
2	string int	33	1	92	124	140	yes
3	-	202	30	615	816	2	yes
3	string int	32	2	88	119	3	yes
3	all	29	0	82	110	3	yes
2 & 3	string int	55	0	163	217	11	yes & yes
2 & 3	all	49	0	146	194	17	yes & yes

TABLE III: Results of second learning experiment with filtering of periodic events

Experiment			Learner Output				
Trace	Excl.	Unique Groups Nr.	Timing Rules Nr.	States	Transitions	Time (in ms)	Verified
2	-	676	19	1971	2637	7	no
2	string int	25	1	74	93	1	yes
3	-	191	30	468	649	1	yes
3	string int	26	2	76	96	2	yes
3	all	25	0	75	96	2	yes

The model of the third party operating table is very large and unreadable. The main reason is the number of operations and the fact that the system components periodically exchange keep-alive events. These periodic events become part of the action lists which increases the number of possible states significantly. Because of this we have improved the instrumentation of the ComMA Learner by filtering out periodic events from a trace.

As an example, Listing 5 specifies that the unsolicited events "dummyCMD2M" and "dummyM2CMD" have to be removed from the input trace. Then the generated state machine contains a part that allows the corresponding operations in all states. Listing 8 provides an example where "dummyCMD2M" is a *Signal* and "dummyM2CMD" a *Notification.*

Table III is an update of Table II where these two events are filtered from Traces 2 and 3. Observe that filtering reduces the number of states and transitions of the resulting model.

B. Analysis

When inspecting the learned models, we observed that the state machine for the power control case is quite readable. For this case, Listing 3 presents a fragment of the manually crafted model and Listing 7 presents a fragment of the generated model. Next we compare both state machines:

- States "s0" and "SystemOff" map because the VideoOnButton can be injected in this state. The "GetState" was not present in the observed trace and therefore not in the learned state machine
- The "VideoOnTransitioning" state in the manually crafted model is presented by the "s0_0_0" and "s1" states of the learned model. The learned state machine does not use state variables, but encodes this behavior in separate state. Observe that the learned model is more restrictive because "StateUpdate" needs to come before "GetState" while this is not required for the manual crafted model.
- States "s12" and "VideoOn" map because the GetState operation replies "VideoOn".

VIII. Concluding Remarks

We presented a manual and automated approach to reverse engineer existing legacy software interfaces. The benefit of the automated approach compared to the manual approach is that less manual labor is required for the creation of a ComMA model. Based on sequences of observed operations, a ComMA model is automatically generated that describes the external visible behavior of a software component in terms of its state and timing behavior.

We applied our approach on two cases for which we had manually crafted ComMA models and traces available. In our experiments, the ComMA monitor generated from a learned model accepts all traces that were used to learn the model.

We observed that the learned state machines can become very large and restrictive. For example, when an operation has an integer as a parameter and the trace has many occurrences of this operation with many different values for the integer, then the Learner will create a transition for every different value. However, this parameter value might be irrelevant for the state behavior of the learned component. In such situations, it is desirable to exclude integer values from the state machine learner and we instrumented the learner to allow this.

A general strategy could be to first learn a state machine without excluding any parameters and then incrementally exclude parameter types until the resulting state machine is manageable. The final step then would be a manual editing of the state machine.

With our approach the quality of the traces is very important. All behavior that is not in the input traces will not be in the resulting model.

In the future, we will apply our approach on legacy interfaces for which we do not have a manually crafted model.

Acknowledgments

We are grateful to the anonymous reviewers for useful comments and suggestions.

REFERENCES

[1] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman, "Integrating interface modeling and analysis in an industrial setting," in *MODELSWARD*, 2017. doi: http://dx.doi.org/10.5220/0006133103450352 pp. 345–352.

[2] F. Vaandrager, "Model learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, 2017. doi: http://dx.doi.org/10.1145/2967606

[3] W. van der Aalst, *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[4] T. Berg, B. Jonsson, and H. Raffelt, "Regular inference for state machines with parameters," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2006. doi: http://dx.doi.org/10.1007/11693017_10 pp. 107–121.

[5] I. Buzhinsky and V. Vyatkin, "Modular plant model synthesis from behavior traces and temporal properties," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017. doi: http://dx.doi.org/10.1109/ETFA.2017.8247578 pp. 1–7.

[6] ——, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 4, pp. 1521–1530, 2017. doi: http://dx.doi.org/10.1109/TII.2017.2670146

[7] G. H. Broadfoot, "ASD case notes: Costs and benefits of applying formal methods to industrial control software," in *Formal Methods (FM 2005)*, J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds. Springer, 2005. doi: http://doi.org/10.1007/11526841_39 pp. 548–551.

[8] G. Booch, J. E. Rumbaugh, and I. Jacobson, *The Unified Modeling Language user guide*, ser. Addison-Wesley object technology series. Addison-Wesley-Longman, 1999.

[9] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992. doi: http://dx.doi.org/10.1109/2.161279

[10] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: An industrial case study," in *Proceedings of the 5th ACM International Conference on Embedded Software*, ser. EMSOFT '05. ACM, 2005. doi: http://dx.doi.org/10.1145/1086228.1086283 pp. 299–306.

[11] Z. Gu, S. Wang, S. Kodase, and K. G. Shin, "An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software," in *24th IEEE Real-Time Systems Symposium, 2003 (RTSS 2003)*, 2003. doi: http://dx.doi.org/10.1109/REAL.2003.1253256 pp. 78–81.

[12] M. Schuts and J. Hooman, "Using domain specific languages to improve the development of a power control unit," in *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*. IEEE, 2015. doi: http://dx.doi.org/10.15439/2015F46 pp. 781–788.

[13] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata Studies, Annals of Math. Studies*, 1956.

[14] M. J. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, 2013.

[15] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, no. 34, p. 1045–1079, 1955. doi: http://dx.doi.org/10.1002/j.1538-7305.1955.tb03788.x

[16] J. Hooman, R. Huis in 't Veld, and M. Schuts, "Experiences with a compositional model checker in the healthcare domain," in *Foundations of Health Information Engineering and Systems (FHIES 2011)*, LNCS 7151. Springer, 2012. doi: http://dx.doi.org/10.1007/978-3-642-32355-3_6 pp. 93–110.

[17] I. Kurtev, J. Hooman, and M. Schuts, "Runtime monitoring based on interface specifications," in *ModelEd, TestEd, TrustEd*. Springer, 2017. doi: http://dx.doi.org/10.1007/978-3-319-68270-9_17 pp. 335–356.