Towards Continuous Evolution through Automatic Detection and Correction of Service Incompatibilities

Benny Akesson*†, Jack Sleuters*, Sven Weiss‡, and Ronald Begeer*
*ESI (TNO), Eindhoven, the Netherlands
†University of Amsterdam, the Netherlands
‡Independent, Helmond, the Netherlands

***Abstract*—Systems with long life times need to continuously evolve after deployment in response to changing technology and business needs. Lacking this ability not only prevents systems from quickly reacting to these changes, but also increases risk, as many small updates are collected into big infrequent upgrades. Service-oriented architectures support continuous evolution by decoupling the application from a particular product, technology, and implementation using service interfaces that hide the component implementing the service. However, this arrangement results in a large number of possible interactions between different components and versions, making it difficult and time-consuming to detect and correct incompatibilities caused by updating service interfaces.**

This paper has three main contributions towards enabling continuous evolution in service-oriented architectures: 1) the state-of-the-art in the areas of specification of service interfaces, and detection and correction of incompatible service interactions is surveyed, 2) directions for a methodology to detect and correct incompatible interactions that is currently under development are discussed, and 3) the methodology is discussed in the context of a simplified industrial case study from the defense domain.

***Index Terms*—Continuous Evolution, Model-based Engineering, Service-oriented Architecture, Compatibility, Adapter Generation, Interface Specification**

I. Introduction

Complex systems need to *continuously evolve*, both during development and after they have been deployed. This is often in response to changing business needs, e.g. market trends, competition, legislation, industry roadmaps, and customer requirements. Evolution can also be driven by changes in technology, such as introduction of new technology, standardization, and new interoperability requirements. The need to continuously evolve is particularly important for systems with long life time, e.g. in the defense domain [1]–[3]. Lacking the ability to handle change efficiently results in systems that evolve slowly, as many smaller updates are collected into big risky upgrades, once every 10-15 years, that are largely driven by the need to counter obsolescence. Such upgrades are expensive and time-consuming and may require the system to be taken out-of-service for several years.

The required efficiency and flexibility to address continuous evolution of complex and dynamic systems is facilitated by *componentization* and *service-oriented architectures*. Componentization implies breaking down large monolithic applications into smaller *component-based applications*. This enables more efficient system development by separating concerns, reducing testing and maintenance costs, and facilitating reuse between products. Flexibility can be provided by service-oriented architectures, where components provide and require services corresponding to a particular functionality. This allows applications to declare the services they depend on without knowing which components provide them, since all components providing a service must implement the same *service interface*. The notion of services supports continuous evolution by decoupling the application from a particular product, technology, and implementation. The service dependencies are resolved at run-time by a service framework, allowing applications to declare their needs dynamically depending on the situation. Run-time resolution of dependencies also allows newer versions of components, or even completely new components, to provide a service needed by an old application.

While componentization and service-oriented architectures facilitate continuous evolution of systems, it is not without associated challenges. Having a system built from a large number of services that are reused across many products makes updating services difficult. The components implementing the service must be updated to reflect changes to the interface, but the main challenge is to detect and correct incompatibilities for all components of different versions that may depend on the service. This issue slows down the evolution of systems based on service-oriented architectures.

This paper presents initial *applied research* towards improving continuous evolution in service-oriented architectures by addressing the problem of detecting and correcting incompatible service updates. The paper has three main contributions:

1) A survey of the state-of-the-art in the areas of interface specification, and detection and correction of incompatible services
2) Directions for a methodology to detect and correct component incompatibilities that is currently being developed is presented. It includes modelling both the structure and behavior in service interfaces, which enables the use of formal analysis techniques, in our case based on Open (Petri) Nets [4], to automatically detect and correct incompatibilities after a service update.
3) The methodology is discussed in the context of a running example based on a simplified industrial case study of a service in a radar system.

The rest of this paper is structured as follows. Section II introduces the industrial case study that provides the context of this work. The state-of-the-art in relevant areas is surveyed in Section III. Section IV then describes the directions for our methodology to detect and correct incompatibilities. Lastly, conclusions are drawn in Section V.

II. Case Study

This section presents the industrial case study considered in this work. First, we introduce the service-oriented architecture in Section II-A, followed by an example service from a radar system requiring an update in Section II-B.

A. Service-Oriented Architecture

A service-oriented software architecture for complex distributed and evolving systems, based on the open Intelligent Robust Architecture for Time Critical Systems (INAETICS) [5], provides the context of this work. We continue by introducing a somewhat simplified terminology related to this architecture.

A system based on the INAETICS architecture provides functionality by executing one or more independent *applications*. Applications are a dynamically combined collection of independent *components*. A large application, such as a multi-face radar, may have thousands of such components. Components can only be accessed through their defined *services*, which can be provided either through remote procedure calls or message passing. In the former case, the *service interface* is formally specified by a number of method signatures. In the latter case, which is the focus of this work, it is formally specified using *message interfaces* that define sets of valid *message types*. The protocol behavior is not formally specified in the service interface in either case, but is represented in code and documentation.

Messages in the INAETICS architecture can be sent either *synchronously* or *asynchronously*. In the synchronous case, the component sending a message blocks until it has received a response. In contrast, asynchronous communication posts messages in a message queue, which allows the component to continue executing immediately after sending a message. In this case, the component checks its message queue for incoming messages at appropriate times during its execution.

B. Example Service

This work considers a case study based on a service in a radar system. The service has been somewhat simplified for ease of presentation. For reasons of confidentiality, we refer to the service as PeriodicTask. The PeriodicTask service schedules an optional task to be performed by the radar system periodically. The client component using the service can specify a maximum budget that has to be considered by the radar system when performing the task to prevent it from spending too many resources on it. The budget actually used by the radar when executing the PeriodicTask service should be less than or equal to the specified maximum budget. PeriodicTask periodically reports the actual used budget back to the client.

The PeriodicTask service communicates asynchronously with client components. The provided service interface is defined in terms of message types and protocol behavior, as illustrated by the communicating finite state machine (FSM) in Figure 1. In the figure, a label -x represents a message of type x being sent and +x received, respectively. As seen in the figure, a client component sends the PTON command message, specifying the maximum budget as an argument. PeriodicTask responds to this command by sending the PTSTATE response message with a status argument ACK back to the client, acknowledging the reception of the command. Now that PeriodicTask is activated and scheduled by the radar system, it periodically reports its used budget to the client. The same PTSTATE response message is used to report this UPDATE status, along with the actually used budget. When PeriodicTask is no longer required, the client can turn it off by sending the PTOFF signal. PeriodicTask responds to this command by sending the ACK response to the client. While in the ON state. it can also receive a new PTON message with a different maximum budget.

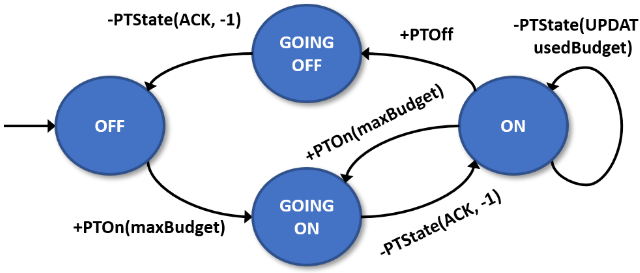


Fig. 1. The behavior of the PeriodicTask service illustrated as a communicating FSM.

As described above, the current version of the PeriodicTask service uses the PTSTATE response message for two different purposes: 1) reporting status information to the client, and 2) periodically reporting the used budget. This has the disadvantage that the ACK response message, which only acknowledges the reception of a command, also has to report a used budget value even though it does not have a valid value yet. In this case, a used budget value of -1 is reported to signal that the value should not be considered. A better solution would be to introduce a separate PTPERFORMANCE message for periodically reporting the used budget and to remove it from the PTSTATE message. However, changing the service interface in this way would cause incompatibility with all components using the service. We will use this service update as a running example through this paper and discuss how the compatibility issue can be efficiently resolved, promoting continuous evolution of the system.

III. STATE-OF-THE-ART

Before presenting the directions for our proposed methodology for detection and correction of incompatibilities between services and their client components, we review the state-of-the-art in related areas. First, we briefly discuss what it actually means to be (in)compatible in Section III-A. Section III-B then discusses related work on interface specification, before methods for detection and correction of incompatibilities are surveyed in Section III-C.

A. Compatibility

Compatibility between services and their client components may be defined in many different ways, but two kinds of compatibility are widely recognized and distinguished in the literature [6]–[9]:

1) *Structural compatibility* means that messages specified in the service interface, and their constituent fields, match those used by the client component in terms of name, type, and semantics. A similar notion of structural compatibility exists for services implemented through remote procedure calls, i.e. assuming matching names, parameters, and types in method signatures. Structural incompatibility may cause a client component to fail because of a type error or cause incorrect functional

behavior if a compatible data type with, e.g. different precision, is used.

2) *Behavioral compatibility* means that the service and the client components agree on the protocol, i.e. the ordering requirements between messages or method calls. Behavioral incompatibility may affect functional correctness, e.g. through deadlock, livelock, or message loss.

Applying this terminology to the PeriodicTask service in Section II-B reveals that the proposed update to the service interface causes both structural and behavioral incompatibility for the client components. The structural incompatibility is caused by separating parts of the PTSTATE message into a separate PTPERFORMANCE message. The new message type affects the protocol of the service, causing a related behavioral change.

To be able to (semi-)automatically detect and correct both types of incompatibilities, we need to be able to formally specify both structure and behavior in service interfaces. Existing approaches for this are surveyed in the next section.

B. Interface Specification

An important part of detecting incompatibilities is to have good means to specify and validate conformance with the service interface. Interfaces have been modeled in many programming, specification, and modelling languages over the years. In fact, a recent survey of the state-of-the-practice of Model-Based Engineering (MBE) in the embedded systems domain suggests that modelling interfaces is one of the most common functional aspects covered by current industrial applications of MBE, with structural aspects being more commonly specified than behavioral ones [10]. This suggests that problems and benefits related to interface specification are widely recognized in industrial practice. Industrial interest going back a long time is also suggested by IBM publishing in this area in the 90's [11] and Motorola mentioning interface specification as a part of their MBE-based development process more than a decade ago [12].

Most interface specifications focus purely on structure. This is the case for programming languages, such as Java and C++, which typically specify method signatures provided by classes implementing the interface. Similarly, many Interface Definition Languages (IDLs), focus on purely structural aspects, such as declaring types and method signatures, allowing them to describe structural aspects of service interfaces. An example of an IDL in this category is Apache Avro[1], which is used in the INAETICS architecture [5].

In contrast, there are also modelling languages and formalisms that focus on behavior and can be used to model the behavior of a service, or the behavior on its interface. This includes different kinds of transition systems, such as FSMs or Petri Nets [13]. Both of these formalisms have been extended for the case of communicating systems. Communicating Finite State Machines [14] are a variation of regular FSMs, where transitions have been labeled with communication actions that indicate that a message of a particular type is being sent or received. The model of the PeriodicTask service in Figure 1 is an example of a communicating FSM. Open (Workflow) Nets [4] are a variation of Petri Nets that include input and output interface places to model asynchronous messages of different types being received and sent on the interface of the service. Open Nets can be composed by connecting and fusing corresponding input and output places with matching labels (message types). If all input and output places of Open Nets have been connected, it becomes a closed net, which is amenable to behavioral analysis.

A benefit of behavioral formalisms like communicating FSMs and Petri Nets is that they have clear and unambiguous semantics describing their execution. This allows correctness properties relevant to behavioral compatibility, such as deadlock-freedom, to be established for the models expressed in the formalism. An important distinction between behavioral formalisms is whether they assume synchronous or asynchronous communication semantics. Strict synchronous communication semantics require that state transitions in communicating processes happen atomically [6], [11] to ensure that the state of the system is always consistent. Synchronous communication does not assume any message queues, so it is essential for deadlock-freedom that communicating services are designed such that one process is always ready to receive at least one message sent by the other.

The main benefit of synchronous communication semantics is that they are easier to analyze and validate than the asynchronous case [6], [8]. A reason for this is that deadlock-freedom is undecidable in the asynchronous case, unless there are guarantees that all messages sent by the processes can be buffered in a message queue until they are received. This in turn requires the maximum number of messages present at any time in the message queue to be bounded, and the message queue to be dimensioned accordingly, creating a dependency on the underlying platform. However, synchronous communication semantics also have a dependency on the system as they require these semantics to be supported in the system under analysis. This requirement may or may not be satisfied for a given system, and may be an undesirable constraint for systems that are built from scratch.

There are also MBE approaches that specify both structure and behavior of components. An industrial MBE approach to ensure correct behavior of software in light of changing components and interfaces is to apply the formal technique Dezyne, previously known as Analytical Software Design (ASD), supported by tools from the company Verum[2]. In Dezyne, interfaces and software designs are modelled as state machines. The Verum toolset supports model checking to ensure freedom of deadlock, livelock, race conditions, and that all components are compliant with their interfaces. Code can be generated from the models, creating a link from specification to implementation of correct code. The benefits and limitations of this approach have been evaluated in case studies at Philips [15], [16]. The main results suggest that the Verum approach improved defect density and software productivity compared to industrial standards. It was also concluded that the developed components were stable and reliable against frequent changes in requirements. The main limitation of the approach is that it only applies to event-based control components, as it does not support data-dependent behavior. It furthermore requires careful component design to

[1] https://avro.apache.org/

[2] https://www.verum.com/

avoid state-space explosion that prevents the model checking from completing in reasonable time.

Component Modelling and Analysis (ComMA)[3] is another MBE approach developed in an industrial context that models both structure and behavior of components [17], [18]. The ComMA approach addresses the issue of system errors being caused by components that are not compliant with their interfaces. This is done through a modular and reusable set of Domain-Specific Languages (DSLs) for specifying interface signatures, behavior, timing, and data constraints. These DSLs, which are implemented in Xtext[4], are used in several phases of the design process. In early phases, transformations to UML diagrams enable visualization of behavior, providing early feedback. Later phases benefit from generation of middleware code and synthesized monitors that verify compliance with the interface. The latter is done by providing execution traces to the monitor, which checks whether the events and their associated data values and timings are valid according to the interface specification for the current state of the component.

ComMA interfaces can be specified manually or supported by tooling that reverse-engineers the interface model from a set of valid execution traces [19]. Although the former method is suitable in environments where components are modeled from scratch, the latter is helpful when introducing the approach in environments with a large number of legacy components. It is also possible to automatically generate both ComMA and Dezyne models from Rhapsody models [20], providing a migration path for users of that tool. A key differentiator between ComMA and Dezyne is that ComMA only specifies and verifies compliance with interfaces and does not generate or validate correct implementations. As a result, ComMA imposes fewer restrictions on the implementation, allowing it to be applied to a wider class of systems.

C. Detection and Correction of Incompatibilities

Having discussed different approaches for specifying the structure and behavior of services and their interfaces, we continue by looking at approaches to detect and correct incompatibilities when a service is replaced by a newer version, or by an alternative service. To reason about compatibility across such replacements, a notion of *substitutability* is defined for communicating FSMs in [8]. Substitutability is a relation between two services that represents whether some notion of compatibility with partner services is preserved as the replacement service substitutes the original. Two types of substitutability are distinguished: 1) *context-dependent*, which focuses on compatibility between a particular pair of services, and 2) *context-independent*, where compatibility has to be preserved for all possible services that were compatible with the original. Intuitively, two services are substitutable if their interfaces are structurally compatible and if they are *observationally indistinguishable* [8] from each other. This means that there may be internal differences in behavior, but that these are completely transparent to an outside observer. The concept of context-free substitutability is similar to the notion of *accordance*, defined for Open Nets in [21], although this notion limits compatibility to consider deadlock-free execution.

If incompatibility between two services has been detected, there are three ways to correct the problem [21]: 1) replace one of the services with a compatible alternative, 2) change implementation of one of the services, and 3) introduce an adapter that mediates between them. This section focuses on the latter option.

1) Structural Incompatibilities: Much work on adapter generation has been done that considers the problem of interoperability between services that were not explicitly developed to interact. The problem of resolving structural incompatibilities is strongly related to the problem of schema matching. The schema matching problem has been widely studied and several approaches and supporting tools, some of which are commercial, have been developed.

A commonality among many adapter generation approaches is that they require some kind of mapping rules that specify how messages sent and received by one service relate to those of another [7], [11], [22]–[24]. These mapping rules bridge the structural differences between the services. For example, a message sent by one service may need to be transformed into another type, split up into multiple messages, merged with other messages, or hidden, to ensure a structural fit and provide correct functional behavior. This specification is typically done manually [7], [11], [22], [23], although semi-automatic approaches based on schema matching have been proposed and implemented in tools [24].

A survey of work towards partial automation of schema matching is provided in [25]. The authors claim that full automation is generally not possible, because most schemas have semantics that are not expressed, or even documented. Matchers should hence propose candidate matches between schemas for manual review and selection. The survey presents a taxonomy that classifies existing approaches according to five criteria: 1) whether matching is based on the schema itself or data values from schema instances, 2) whether matching considers individual elements or more complex structures, 3) whether the matchers use a linguistic approach based on similarity of names and textual descriptions or a constraint-based approach based on keys and relationships, 4) the cardinality of the matching (1:1, 1:n or n:m), and 5) whether they use auxiliary information, such as information about previous matches. The survey concludes by discussing seven implementations and positions them with respect to the proposed taxonomy.

2) Behavioral Incompatibilities: Two methods for determining whether two services represented as Open Nets are in accordance are presented in [21]. The first method is based on *projectional inheritance*, which intuitively means to check if the services are observationally indistinguishable unless the differences between them, e.g. any newly added features, are explicitly used. Another way too look at this is as determining whether the updated service is a sub-protocol of the original, in the sense familiar from object-oriented programming. This notion of sub-protocol is also used to determine compatibility for communicating FSMs in [11]. Projectional inheritance between two Open Nets can be established by checking if they are branching bi-similar. This approach is a sufficient, but restrictive, method to verify accordance. The second approach is a precise condition based on *operating guidelines*, which is a characterization of the set of all compatible services. A benefit of both approaches is that they do not require any

[3] http://comma.esi.nl/

[4] https://www.eclipse.org/Xtext/

knowledge about the clients using the services, i.e. they are context-independent.

There has been a lot of work in the area of semi-automatic correction of behavioral incompatibilities, in particular protocol mismatches. A survey of different approaches to generate protocol adapters is provided in [6]. A distinguishing feature among different approaches to adapter generation is the formalism they use to model behavior. The most commonly used formalisms for this purpose are workflow languages [9], process algebras [7], communicating FSMs [11], [23], [24], Open Nets [22], and pseudo code [26]. The mentioned works based on workflow languages, pseudo code, process algebras, and Open Nets handle both synchronous and asynchronous communication semantics, while the approaches based on communicating FSMs only apply to the synchronous case.

Most approaches to generate protocol adapters execute at design time [9], [11], [22], [24], [26] and produce a model of an adapter in form of e.g. a communicating FSM or an Open Net, that can be connected to models of the services to make them interoperable. Although not frequently mentioned, these models can be used as input for code generation to generate an implementation of an adapter that connect the two services. In contrast, [23] proposes a service adaptation machine that executes at run time. In essence, this machine looks at messages that are expected to be sent and received by the communicating services in their current state. If there is no obvious match, it consults the set of specified mapping rules and see if there is any rule, or even sequence of rules, that can produce an expected message so that the interaction can progress. A benefit of this approach is that it does not generate a fixed adapter for a given pair of incompatible services, but dynamically adapt between any services, given the appropriate mapping rules. However, a drawback of the approach is that the adaptation is best effort, as it is not possible to a priori state whether or not the adaptation will be successful.

IV. Directions for Methodology

Having reviewed the state-of-the-art, this section proceeds by discussing directions for a five-step methodology to detect and correct service incompatibilities. This methodology is currently under development. The general directions and technology choices presented below are finished, while implementation work towards a proof-of-concept demonstrator is ongoing. First, Section IV-A presents the five steps in a general sense and briefly mentions the technology choices we have made when applying an instance of the methodology to our case study. The motivation for our choices, as well as more details about interface specification, and detection and correction of incompatibilities are then provided in Sections IV-B and IV-C, respectively.

A. Overview

The essence of our methodology is based on specifying both structure and behavior in service interfaces to enable detection and correction of service incompatibilities using formal methods. An overview of the proposed five-step methodology is shown in Figure 2. We continue by briefly introducing the five steps:

1) *Service Interface Specification:* The structure and behavior of services is specified in the service interface. Interface specifications are created for each new service and are updated whenever changes to the specification are required. The ComMA approach [17], [18] has been chosen as the service interface specification framework for our application of the methodology.
2) *Generate Formal Service Model:* A formal model is generated from the service interface specification in a formalism supported by existing methods for analysis and synthesis. In our instance of the methodology, we have selected Open Nets [4] for this purpose.
3) *Check Accordance:* When a service is updated or replaced, the generated Open Net models of both the original and the updated service is provided as input to a method that checks whether or not they are in accordance. For this purpose, we have selected the context-independent method based on operating guidelines [27]. This approach does not require the two services to be completely equivalent, only that all possible previous clients are still supported. The updated service may hence allow a richer set of possible clients than the original. If the original and updated services are determined to be in accordance by this approach, the updated service can replace the original in all interactions with any possible clients without any further steps. If the two services are not in accordance, the following steps are taken.
4) *Generate Adapter:* For each partner service the updated service is interacting with, generate an adapter that ensures deadlock-free execution of the composition with the new adapter. For this step, we have selected the adapter generation approach based on controller synthesis [22] for our instance of the methodology. If no adapter can be found to correct the incompatibilities for any of the interacting services, the methodology has failed and no further steps are taken. Otherwise, if a model of an adapter is found for each partner service, the final step is taken.
5) *Generate Adapter Code:* This final step takes the models of the adapters and generates implementations for the target platform. In our instance, C++ code for the INAETICS architecture is generated.

The first step of the methodology is done at design time, as a service is first created or updated. The remaining four steps are done on the target platform as it is being updated. If the methodology fails to update a particular service, the update is rejected and the system continues executing with the existing version of the service.

B. Interface Specification

Formal service interface specifications in the INAETICS architecture describe structure, such as message types and method signatures, but not behavior. It is hence not possible to determine whether or not an updated service interface is compatible with the original, or if interacting with a previously compatible client components could result in deadlock. To resolve this issue, the proposed methodology uses the ComMA language [17] to specify both structure and behavior of service interfaces.

The ComMA approach was selected for the following reasons: 1) it supports specification of both structure and behavior, which is required to validate both aspects of compatibility,

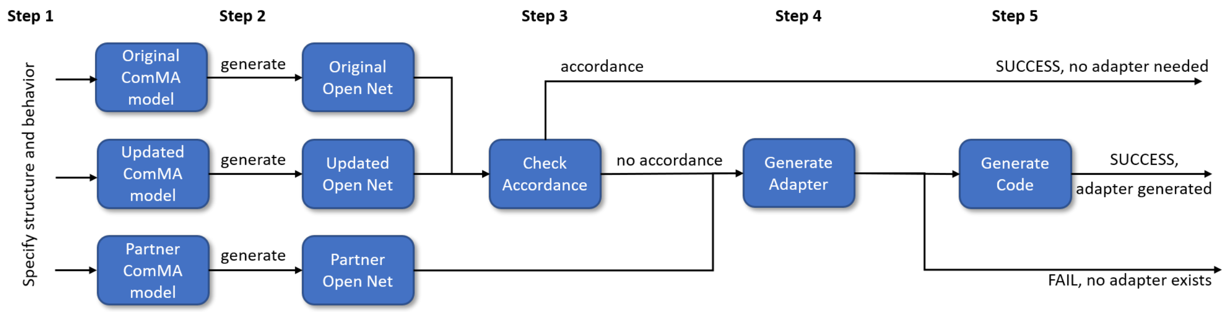


Fig. 2. Overview of proposed five-step methodology

```
enum Response {
   ACK, UPDATE
}

record PTResponse {
   Response response
   real usedBudget
}
```

(a) Types

```
signals
   PTOn(real budget)
   PTOff

notifications
   PTState(PTResponse response)
```

(b) Signature

Fig. 3. Structural service interface before update

2) it can model both synchronous and asynchronous communication and does hence not restrict the communication options offered by INAETICS, 3) it only specifies the service interface and does not unnecessarily constrain the implementation of the service, 4) it has been successfully applied in industry before, i.e. at Philips [17]–[20], 5) the support for automatic generation of interface specification from traces simplifies industrial adoption on a wider scale, and 6) the tooling is based on Eclipse, which is one of the most commonly used modeling tools in the embedded domain [10]. This makes it easier to gain industrial acceptance.

Parts of a simplified ComMA interface specifying the structure and behavior of the PeriodicTask service from Section II-B are shown in Figures 3 and 4, respectively. Figure 3a specifies relevant types that are needed to specify the signature of the interface in Figure 3b. The signature contains two signals, PTOn and PTOff, that are sent from the client to the PeriodicTask service, and a notification, PTState, that is sent from PeriodicTask back to the client component. Both signals and notifications are sent asynchronously. ComMA signatures also support synchronous commands, but these are not needed in the case study.

The behavior of the service interface is described in Figure 4. Behavior in ComMA is specified as an FSM, where events, such as commands and signals, trigger transitions between the states.

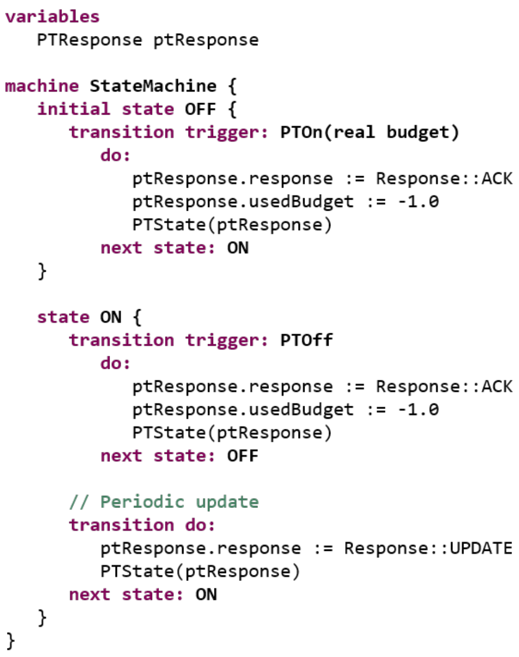
Unlike the model in Figure 1, we do not need the intermediate states for going on and going off, since ComMA can specify the received trigger signal and the response notification in a single transition. For brevity, Figure 4 does not include that the PTOn signal can be received in the ON state. This transition is analogous to the shown transitions triggered by the PTOn and PTOff signals. As indicated in the figure, PeriodicTasks starts in the OFF state and transitions to the ON state after receiving a PTOn signal with an associated maximum budget. The specification also shows that the service implementing the interface must reply by sending a notification with an acknowledgement and a used budget of -1, which is the undesired coupling of concerns previously discussed in Section II-B that is the reason for updating the service interface. Conformance with this specified behavior can be validated using a monitor that is automatically generated from the ComMA specification. The ON state specifies a similar transition back to the OFF state after receiving a PTOff signal from the client, as well as the periodic update. The periodic update is hence not triggered by an external event and does not have a transition trigger. It is possible to specify the period and maximum jitter constraints for the periodic update, but this has been omitted for simplicity. For the same reason, the current specification also does not constrain the used budget values to be non-negative. For more information about timing and data constraints, refer to [18].

Relevant parts of the service interface after the update are shown in Figures 5 and 6, respectively. Figure 5 shows that there has been a structural modification. The state message has been simplified to only include the response, as the used budget has been moved to the newly introduced PTPerformance notification. The updated behavior is specified in Figure 6. As seen in the figure, it is no longer necessary to monitor that the invalid budget is correctly sent in state messages when transitioning to the ON and OFF states. Similarly, the periodic update now sends a dedicated performance notification and does not have to include other state information. This decouples the concerns of responding to a signal from sending periodic updates, as desired in Section II-B, at cost of behavioral and structural incompatibility.

C. Detection and Correction

To address incompatibilities, we choose to base our approach on Open Nets [4]. The three main reasons for this choice are: 1) it is conceptually possible to translate a ComMA specification into a corresponding Open Net, 2) Open Nets can model both synchronous and asynchronous communication semantics, supported by both INAETICS and ComMA, and 3) there are existing analysis methods, supported by prototype academic tools[5], available for both detection and correction of

[5] http://service-technology.org/tools/index.html, https://github.com/nlohmann/service-technology.org



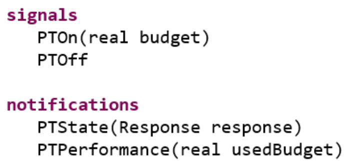
```
variables
   PTResponse ptResponse

machine StateMachine {
   initial state OFF {
      transition trigger: PTOn(real budget)
         do:
            ptResponse.response := Response::ACK
            ptResponse.usedBudget := -1.0
            PTState(ptResponse)
         next state: ON
   }

   state ON {
      transition trigger: PTOff
         do:
            ptResponse.response := Response::ACK
            ptResponse.usedBudget := -1.0
            PTState(ptResponse)
         next state: OFF

      // Periodic update
      transition do:
         ptResponse.response := Response::UPDATE
         PTState(ptResponse)
      next state: ON
   }
}
```

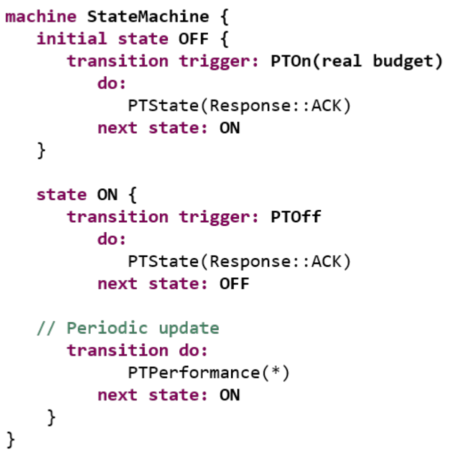
Fig. 4. Behavioral service interface before update



```
signals
   PTOn(real budget)
   PTOff

notifications
   PTState(Response response)
   PTPerformance(real usedBudget)
```

Fig. 5. Interface signature after update



```
machine StateMachine {
   initial state OFF {
      transition trigger: PTOn(real budget)
         do:
            PTState(Response::ACK)
         next state: ON
   }

   state ON {
      transition trigger: PTOff
         do:
            PTState(Response::ACK)
         next state: OFF

   // Periodic update
      transition do:
            PTPerformance(*)
         next state: ON
    }
}
```

Fig. 6. Behavioral service interface after update

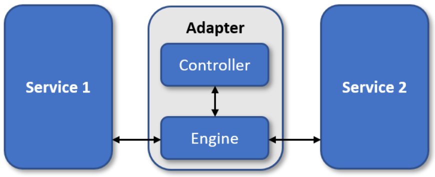


Fig. 7. Two services communicating via a generated adapter comprising an engine and a controller.

incompatibilities.

A disadvantage of choosing Open Nets is that this formalism is much less common than FSMs among industrial practitioners in the embedded domain [10]. This is a concern since best practices for MDE state that it is essential for acceptance that notation and terminology are familiar to the users [28], [29]. However, users of the proposed methodology specify behavior as FSMs in the ComMA language, and the transformation to Open Nets and the behavioral analysis happen automatically under the hood, which mitigates the concern. The generation of Open Nets will be implemented as a model-to-model transformation in the ComMA framework. The generated Open Net model will then pass through model-to-text transformations to create the required input models for the tools performing detection and correction.

Having chosen Open Nets as the formalism for our application of the proposed methodology, it makes sense to select the precise method for determining accordance based on operating guidelines. This method is implemented in the prototype tool Fiona [30].

For correction of incompatibilities, we select the adapter generation method based on controller synthesis [22]. We continue by giving a brief overview of this approach. The basic idea is to create an adapter between two services represented as Open Nets by synthesizing an adapter with two key components, an *engine* and a *controller*. This adapter architecture is shown in Figure 7. The engine is responsible for the flow and transformation of data and has the input and output places required to connect to the two services according to the set of specified mapping rules. It also has connections to the controller, which is responsible for ordering of events such that behavioral properties, like deadlock-freedom, are satisfied. The engine notifies the controller when messages have arrived and allows it to determine when to trigger transformations of the data, e.g. splitting up an incoming PTSTATE message into separate PTSTATE and PTPERFORMANCE messages in our case study, and in what order to send messages.

Adapter generation following this approach starts by composing the models of the two services with the model of the engine, which follows directly from the provided mapping rules. This composition results in a larger Open Net, where the only unconnected interface places are those between the engine and the controller. Using techniques for controller synthesis, a controller is then synthesized such that the desired behavioral property is satisfied. There are two possible adapter architectures that differ in whether the connections between the engine and controller are asynchronous or synchronous [31]. The latter typically results in smaller adapters with shorter synthesis times without any particular drawbacks.

In addition to being based on Open Nets, giving it the benefits listed above, this adapter generation approach has also been evaluated in a number of case studies [22], [31], [32] and shown to be able to efficiently generate adapters within seconds. This is particularly true for the architecture with synchronous controllers, which has been selected for our methodology. The adapter generation approach supporting asynchronous controller synthesis is available in the existing tool Marlene[6] [22].

V. Conclusions

Systems with long life time need to continuously evolve to quickly respond to changes in technology and business needs. Service-oriented architectures are enablers of continuous evolution by decoupling applications from a particular technology and implementation. However, manually managing compatibility as service interfaces evolve is expensive and time-consuming, since it is challenging to determine how all possible client components are impacted.

This paper presented initial applied research towards addressing this problem by surveying the state-of-the-art in the areas of interface specification, and detection and correction of service incompatibilities. Directions were presented for a five-step methodology for detection and correction of incompatible services. The methodology involves specifying both the structure and behavior in service interfaces at design time. Based on these specifications, a formal model of the behavior is generated and used to determine whether a service update is compatible with all possible clients. If this is not the case, an adapter is generated and deployed for each client component. The methodology was discussed in the context of a case study from the radar domain.

Acknowledgement

The research is carried out as part of the DYNAMICS project under the responsibility of ESI (TNO) with Thales Nederland B.V. as the carrying industrial partner. The DYNAMICS research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

References

[1] S. Neema, R. Parikh, and S. Jagannathan, "Building resource adaptive software systems," *IEEE Software*, vol. 36, no. 2, pp. 103–109, 2019.

[2] B. Akesson, J. Hooman, R. Dekker, W. Ekkelkamp, and B. Stottelaar, "Pain-mitigation techniques for model-based engineering using domain-specific languages," in *Proc. Special Session on Model Management And Analytics (MOMA3N)*, 2018, pp. 752–764.

[3] B. Akesson, J. Hooman, J. Sleuters, and A. Yankov, "Reducing design time and promoting evolvability using domain-specific languages in an industrial context," in *Model Management and Analytics for Large Scale Systems*. Elsevier, 2019.

[4] P. Massuthe, W. Reisig, and K. Schmidt, *An operating guideline approach to the SOA*. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät , 2005.

[5] H. Bossenbroek and R. van Hees, "The INAETICS architecture - introducing INAETICS," White paper, Tech. Rep., 2015. [Online]. Available: https://www.inaetics.org/reference-material/

[6] R. Seguel, R. Eshuis, and P. Grefen, *An Overview on Protocol Adaptors for Service Component Integration*. Beta, Research School for Operations Management and Logistics, 2008.

[7] M. Dumas, M. Spork, and K. Wang, "Adapt or perish: Algebra and visual notation for service interface adaptation," in *International Conference on Business Process Management*. Springer, 2006, pp. 65–80.

[8] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella, "When are two web services compatible?" in *International Workshop on Technologies for E-Services*. Springer, 2004, pp. 15–28.

[9] A. Brogi and R. Popescu, "Automated generation of BPEL adapters," in *International Conference on Service-Oriented Computing*. Springer, 2006, pp. 27–39.

[10] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Software & Systems Modeling*, vol. 17, no. 1, pp. 91–113, 2018.

[11] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 2, pp. 292–333, 1997.

[12] P. Baker, S. Loh, and F. Weil, "Model-driven engineering in a large industrial context - Motorola case study," *Model Driven Engineering Languages and Systems*, pp. 476–491, 2005.

[13] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[14] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *Journal of the ACM (JACM)*, vol. 30, no. 2, pp. 323–342, 1983.

[15] G. H. Broadfoot, "ASD case notes: Costs and benefits of applying formal methods to industrial control software," in *International Symposium on Formal Methods*. Springer, 2005, pp. 548–551.

[16] A. Osaiweran, M. Schuts, J. Hooman, J. F. Groote, and B. van Rijnsoever, "Evaluating the effect of a lightweight formal technique in industry," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 1, pp. 93–108, 2016.

[17] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman, "Integrating interface modeling and analysis in an industrial setting." in *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, 2017, pp. 345–352.

[18] I. Kurtev, J. Hooman, and M. Schuts, "Runtime monitoring based on interface specifications," in *ModelEd, TestEd, TrustEd*. Springer, 2017, pp. 335–356.

[19] M. Schuts, J. Hooman, I. Kurtev, and D.-J. Swagerman, "Reverse engineering of legacy software interfaces to a model-based approach," in *2018 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2018, pp. 867–876.

[20] M. Schuts, J. Hooman, and P. Tielemans, "Industrial experience with the migration of legacy models using a DSL," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 2018, p. 1.

[21] W. M. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf, "Service interaction: Patterns, formalization, and analysis," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 2009, pp. 42–88.

[22] C. Gierds, A. J. Mooij, and K. Wolf, "Reducing adapter synthesis to controller synthesis," *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 72–85, Jan 2012.

[23] K. Wang, M. Dumas, C. Ouyang, and J. Vayssiere, "The service adaptation machine," in *2008 Sixth European Conference on Web Services*. IEEE, 2008, pp. 145–154.

[24] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-automated adaptation of service interactions," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 993–1002.

[25] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[26] A. Kumar and Z. Shan, "Algorithms based on pattern analysis for verification and adapter creation for business process composition," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2008, pp. 120–138.

[27] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg, "Analyzing interacting WS-BPEL processes using flexible model generation," *Data & Knowledge Engineering*, vol. 64, no. 1, pp. 38–54, 2008.

[28] D. Wile, "Lessons learned from real DSL experiments," *Sci. Comput. Program.*, vol. 51, no. 3, pp. 265–290, Jun. 2004.

[29] M. Voelter, "Best practices for DSLs and model-driven development," *Journal of Object Technology*, vol. 8, no. 6, pp. 79–102, 2009.

[30] P. Massuthe and D. Weinberg, "Fiona: A tool to analyze interacting open nets," 2008.

[31] A. J. Mooij and M. Voorhoeve, "Trading off concurrency to generate behavioral adapters," in *2009 Ninth International Conference on Application of Concurrency to System Design*. IEEE, 2009, pp. 109–118.

[32] A. J. Mooij and M. Voorhoeve, "Specification and generation of adapters for system integration," in *Situation Awareness with Systems of Systems*. Springer, 2013, pp. 173–187.

[6] https://github.com/nlohmann/service-technology.org