Automatic Translation of Service Specification to a Behavioral Type Language for Dynamic Service Verification

Shanshan Jiang, Cyril Carrez, and Finn Arve Aagesen

Department of Telematics
Norwegian University of Science and Technology (NTNU)
N-7491 Trondheim, Norway
{ssjiang, carrez, finnarve}@item.ntnu.no

Abstract. Networked *services*, constituted by the structural and behavior arrangement of *service components* are considered. A service component is executed as a *generic software component*, denoted as an *actor*, which is able to download and execute different EFSM (Extended Finite State Machine) based functionality. The functionality of an actor is denoted as its *role*, while a *role session* is a projection of the role with respect to the interaction with one other actor. We propose an approach for verification of the services, based on interface verification techniques for the verification of the role sessions. The service component specifications used for actor execution are based on XML representations, while the verification of the role sessions is based on a *behavior type language.* This language has a sound theoretical basis, and is used to avoid "message-not-understood" errors. Rules are given for automatic translation from the XML manuscripts to this behavioral type language. This translation first makes projection to the role session, using hidden actions. Those hidden actions are then removed so a sound verification can take place.

1 Introduction

Networked *services* constituted by *service components* are considered. A service component is executed as a software component in *nodes*, which are physical network processing units such as servers, routers or switches, and user terminals such as phones, laptops and PDAs. Traditionally, the nodes and the service components have a predefined functionality. Concerning both the nature of nodes and the software engineering principles, changes are taking place. From being a static component, the service component can be based on generic software components, which are able to download and execute different functionality depending on the need. Such generic programs are from now on denoted as *actors* (by analogy with the actor in the theatre). The functionality of an actor is denoted as its *role*, while a *role session* is a projection of the role with respect to the interaction with one other actor. *Role* and *role session* need the power of Extended Finite State Machines (EFSMs) in order to describe a complex protocol of interaction (which is not the case with most Web-based services as they are request-reply services specified as procedure calls).

There are basically two different approaches to service verification. One is to model the composite behavior of the whole service [Hol90], but it leads to state explosion and has limited applicability for complex systems. Another approach is the decomposition of the service system and isolated verification of the decomposed parts (new services that reuse existing components can take full advantage of this compositional verification). Within this approach we have the sub-approach focusing on the interfaces between the service components, with interface type languages [CFN05,LX04]. Most of these approaches use behavioral type systems [Nie95,NNS99,RV00], where a type specifies a *non-uniform* interface, meaning the set of operations (or messages) the interface accepts depends on the context. Indeed, this type is viewed as an abstract behavior of the component, and is used during compositional verification to ensure liveness and safety properties of the application. We aim for a quick compositional verification, restricted to the compatibility verification of connected interfaces. This will keep the number of states very low, instead of verifying the compatibility of the whole behavior of the components. We used the type language developed in [CFN05], preferred because of its high level of abstraction. In this setting, each component must satisfy a contract, which specifies the behavioral type of its interfaces; an assembly of components is sound if connected interfaces are compatible.

This work is part of the TAPAS project (*Telematics Architecture for Play-based Adaptable System*), which goal is to enhance the flexibility, efficiency and simplicity of system deployment, operation and management by enabling dynamic configuration of network-based service functionality. See [AHAS03] and the URL http://tapas.item.ntnu.no. We propose an approach to verify the services, based on interface verification techniques for the verification of the role sessions. We provide an automatic translation from XML-based EFSM service component specification to the behavioral type language applied. The projection process has two steps. First, make projections that preserve the binding between the role sessions related to each service component by using hidden actions. Then remove the hidden actions so a sound verification can take place.

The paper is organized as follows. The context of our service specification is described through the related TAPAS concepts (Sect. 2). The behavioral type language used in the verification is described in Sect. 3. Section 4 gives the methodology of translation. Related work and Conclusion end the paper.

2 Some TAPAS Concepts

Part of the TAPAS architecture relevant for the verification is illustrated in Figure 1. For a more comprehensive description of the architecture, see [AHAS03]. Concepts such as service, service components, actor, role and role session were defined in Section 1. The concepts of actor, director, role and manuscript are concepts from the theatre, where actors play roles according to manuscripts and a director manages their operations. An actor has two kinds of interfaces (Fig. 2):

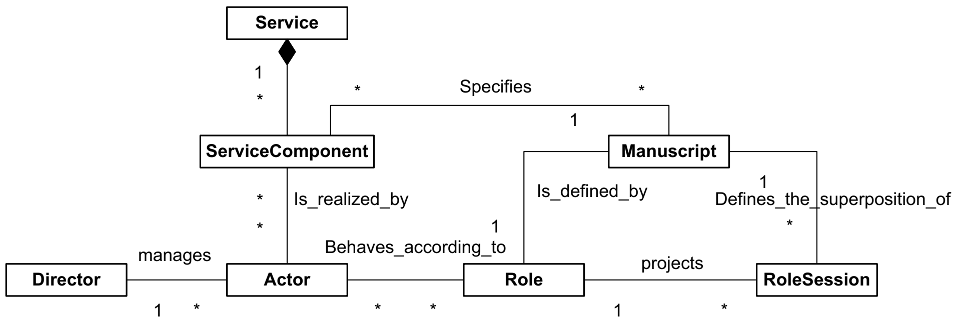

Fig. 1. Some TAPAS concepts related to the verification

Home Interface. This is a control interface between an actor and its director. Each actor is associated with one Director, who manages the performance of the actor through this control interface.

Application Interface. This is an interface where the role sessions between actors take place.

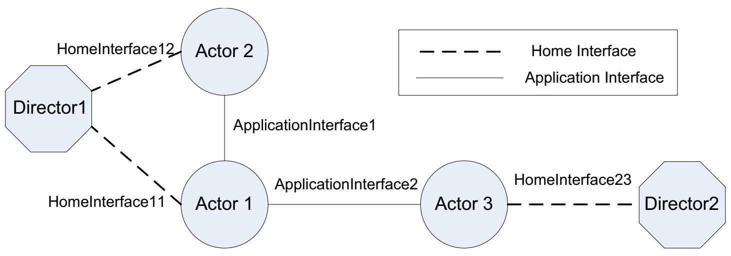

Fig. 2. Actor interfaces

Figure 3 shows the basic data structure of an XML manuscript. This manuscript is the specification of the EFSM based behavior of an actor. It contains the name of the EFSM, its initial state, data and variables, and a set of states. The state structure defines the name of the state and a set of transition rules for this state. Each transition rule specifies that for each input, the EFSM will perform a number of actions, and/or send a number of outputs, and go to the next state. The actions are functions and tasks performed during a specific state: calculations on local data, method calls, time measurements, etc. The `<actions>` list specifies only the action type (method name), parameters and action group (the classification of action types). This XML manuscript therefore specifies parameterized behavioral patterns. The detailed platform support and example implementation for XML service specification can be found in [JA03].

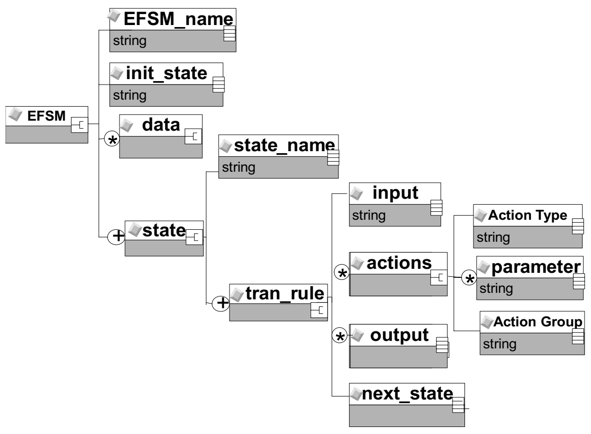

Fig. 3. Manuscript data structure

A fragment of an example XML manuscript is given in Fig. 4. This fragment comes from an example application called TeleSchool, which we used as an experiment of our approach. For lack of space, we only show simplified behavior description for one state. This service component specification will serve as example to demonstrate the translation rules later.

```
<state name="stInitUserInterface">
  <input msg="LogonEventInd"
         source="v_interface">
    <actions>
      <ActionType>ActorPlugIn
      </ActionType>
      <param>
        <name>role</name>
        <value>SchoolServer</value>
      </param>
      <ActionGroup>G1</ActionGroup>
      <store_return>v_server
      </store_return>
    </actions>
    <actions>
      <ActionType>setVariable
      </ActionType>
      <param>
        <name>value</name>
        <value>INPUT_MSG.school
        </value>
      </param>
      <ActionGroup>G2</ActionGroup>
      <store_result>v_currentSchool
      </store_result>
    </actions>
    <actions>
      <ActionType>setVariable
      </ActionType>
      <param>
        <name>value</name>
        <value>INPUT_MSG.user</value>
      </param>
      <ActionGroup>G2</ActionGroup>
      <store_result>v_currentUser
      </store_result>
    </actions>
    <output>
      <msg type="UserVerifyAccessReq">
        <param>
          <name>message</name>
          <value>INPUT_MSG</value>
        </param>
        <dest>v_server</dest>
      </msg>
    </output>
    <next_state>stPasswordIdentify
    </next_state>
  </input>
  <input msg="CancelEventInd" source=...>
    <actions>...</actions>
    <next_state>stInit</next_state>
  </input>
</state>
```

Fig. 4. Fragment of an example XML manuscript

3 Behavioral type language

We adopt the behavioral type language introduced in [CFN05]. This language describes messages that are exchanged on interfaces. We chose this language because it has a well defined semantics, and is based on a component model which is rather close to the Actor model of TAPAS. A component in [CFN05] has a set of ports. Each port interacts with a so-called *partner*, with which it sends and/or receives messages. Communication is asynchronous, and is made through an abstract communication medium containing FIFO queues (one for each port). A port will then be mapped to the interface in TAPAS, the main difference being that each port has its own queue, whereas in TAPAS there is one queue for the whole component. However, we think the two models are equivalent: retrieving, in a global queue, a message destined to an interface is similar to picking up the first message in the queue of that interface. The strong formal framework of the language in [CFN05] also allows us to avoid "message-not-understood" errors, and to ensure external deadlock freeness properties[1]. Moreover, a type not only imposes constraint on the interface it specifies, but also on its environment: it is possible to enforce the environment to send a message by specifying that the interface "**must** receive a message". Although this feature has not been used yet, we think it has an important impact on liveness properties when composing services.

In this paper the details of this language are not presented; the interested reader should consult [CFN05,CFN03], where a BNF table is developed, as well as semantics description and examples. However, we present an example of a bank account specification. The following type specifies the *operations* interface through which a client might perform credits and withdrawals:

```
operations = may ? [    deposit (real);  must ! [ balance (real); operations ]
                      + withdraw (real); must ! [    balance (real); operations
                                                   + neg_balance (real); negbal_operations ] ]
negbal_operations = must ? [ deposit (real);... + withdraw (real);...  ]
```

This type is read as follows: *operations* may receive (**may?**) *deposit* and *withdraw* messages. After receiving one of the two messages, the interface must send (**must!**) the balance of the bank account: message *balance* is sent when balance is positive, and the type becomes *operations* again. Message *neg_balance* is sent when the user is debtor, and then the type becomes *negbal_operations.* This latter type is similar to the *operations* type, with the exception of the modality: type *operations* **may** receive messages, whereas *negbal_operations* **must** receive messages. Hence, the client must perform some operations as long as he is debtor.

For the time being, we concentrate on the choice operator "**+**" and the sequence operator "**;**", so the resulting type is an abstract behavior of the component, which is roughly a projection of its behavior to a specific interface.

[1] The "message-not-understood" error avoidance is mainly due to the compatibility of the types of the interfaces. The deadlock freeness property is due to constraints on the internal behavior of the components, mainly on dependencies between interfaces (i.e. an interface waiting for a result on another one).

4 Translation methodology

The actions that an actor can perform are classified into three types:

Control functionality through Home Interface: management functionality including `ActorPlugIn`, `ActorPlugOut`, etc., defined in [JAHB99]. A request is sent to the Director and the Actor must wait for the result.

Role session through Application Interfaces: the application interactions use asynchronous message sending through Application Interfaces.

Internal actions: they are invisible in the interface descriptions.

The translation from service specification to interface language uses projection. Projection is an abstraction technique, which can produce a simplified system description or viewpoint by aggregating some of the system's functionalities while hiding others. It has been used in previous works [LS84,Flo03] to simplify the verification of protocols and validation analysis. In our approach, the projection process basically consists of two steps. The first step extracts the inputs and outputs for a specific interface. All other actions are considered as hidden (internal actions and interactions occurring at other interfaces). The second step removes those hidden actions so a sound verification can take place.

The automatic translation algorithm is as follows. It scans the XML manuscript once and extracts the interface interaction information, the translation procedure being carried out state by state. Each interface has a unique identifier (for example `I1`), which is assigned dynamically when the interface is created. For each interface, every state has a type name assigned, which is composed of the interface identifier and a number. For example, interface types `I1_*` are used for all the interactions with the HomeInterface (Director), where `I1_1` is the type of the first interaction, which will be transformed to `I1_2` after some I/O interaction. This dynamic creation of interface type is flexible and easy for implementation. Each behavior description will be translated to the equivalent interface type description (using the translation rules described hereafter), affecting one interface at a time. Finally, gathering of silent transitions and states (Sect. 4.3) is applied on the interface type descriptions. In order to simplify the translation, each state will receive only messages from one single interface (i.e. all the inputs for one state are from the same source). If inputs are from different interfaces, we create new states for processing them. In our first implementation, **must** and **may** modalities are not distinguished: all actions will be "**must**".

4.1 Messages

In TAPAS, all communications are through asynchronous message passing. The input and output operations are the visible actions through interfaces and are translated directly into interface types. An input message means a receiving interface type "**?**", while an output message means a sending interface type "**!**". Synchronous communication can be implemented by an output followed by an input message, thus translated into a sending interface type "**!**" followed by a receiving one "**?**".

We consider only the types of the parameters, not their values. The resulting message types are then finite, so the validation will be a finite-state verification (our verification is an optimistic one: it does not handle the cases where values are received outside the scope of their type).

The XML message structures are input, output, and control functionality.

<input> structure. The parameter "source" identifies the role session, and distinguishes the interface for this input operation. This structure is translated to "**?**M_i", with M_i the message type.

Example 1.

XML manuscript	Interface type
`<input msg="LogonEventInd" source="v_interface" >`	I2_2 = **must ? [** LogonEventInd**;** I2_3 **]** $I2\_*$ is used for the interface "v_interface"

<output> structure. The parameter "dest" identifies the role session for this output operation, and is used to find the binding interface. If it represents a new destination, a new interface will be created. This structure is translated to "**!**M_i", with M_i the message type.

Example 2.

XML manuscript

```
<output>
  <msg type="UserVerifyAccessReq">
    <param>
      <name>message</name>
      <value>INPUT_MSG</value>
    </param>
    <dest>v_server</dest>
  </msg>
</output>
```

Interface type

I3_1 = **must! [**UserVerifyAccessReq**;**I3_2**]**

$I3\_*$ is used for the interface "v_server"

Control functionality in <actions> structure. This is identified by a method name starting with "Actor" in `<ActionType>` substructure. This should be translated to "**!** M_i**; ?** M_j" for the HomeInterface.

Example 3.

XML manuscript:

```
<actions>
  <ActionType>ActorPlugIn</ActionType>
  <param>
    <name>role</name>
    <value>SchoolServer</value>
  </param>
  <ActionGroup>G1</ActionGroup>
  <store_return>v_server</store_return>    <!–of type "roleSessionId"–>
</actions>
```

Interface type:

I1_2 = **must ! [** ActorPlugIn (role)**; must ? [** RequestResult (roleSessionId)**;**I1_3**]]**

$I1\_*$ is used for the interface "HomeInterface"

4.2 Deactivation of interfaces

Interfaces can be dynamically created and deleted (deactivated). Deactivation is reflected by the "ActorPlugOut" control functionality. It plugs out an interacting actor, thus placing the corresponding role session into inactive state "**0**". The interface can then be deleted, the deletion being an internal behavior.

4.3 Hidden actions and their removal

The first step of our projection on one interface replaces internal actions and interactions occurring at other interfaces by hidden actions, also called τ-transitions. The next step in the projection is to remove those hidden actions, by combining τ-transitions and states, as in Floch's work [Flo03].

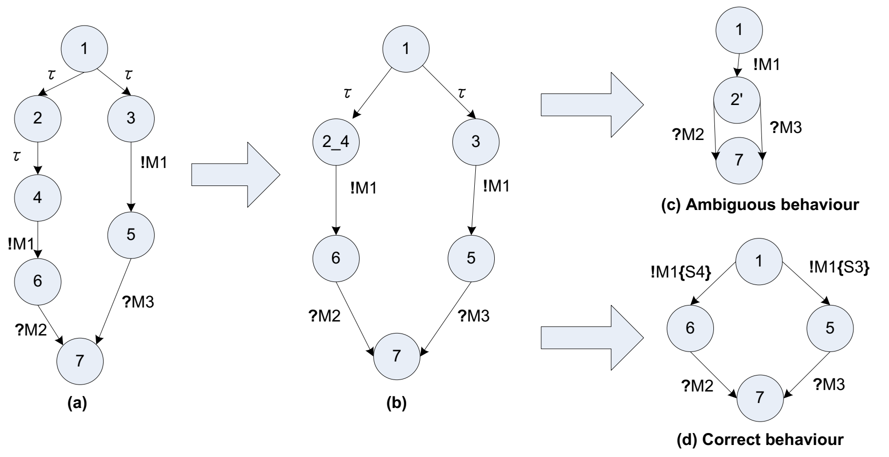

Fig. 5. Combination of τ-transitions

All the successive τ-transitions can be replaced by one single τ-transition, and the states are combined into one state, as shown in Fig. 5(b). If input and output sequences are the same for two states, these states may be combined. However, some ambiguous behavior may result, as shown in Fig. 5(c): from state 2' message $M2$ or $M3$ can be received, while originally $M2$ can be received only in state 6, and $M3$ in state 5 (This ambiguity may be due to some hidden parameter values, or to dependency between interfaces). We eliminate this ambiguity by adding state information in the name of the message (Fig. 5(d)): "**!** $M1\{S4\}$" means "output a message type $M1$ at the state $S4$". The final translated type for Fig. 5(a) could be expressed as follows, referring to Fig. 5(d):

I1 = **must ! [** M1{S4}**; must ? [** M2; I7 **]** + M1{S3}**; must ? [** M3; I7 **]]**
$I1$ and $I7$ are the interface type description for State1 and State7 respectively.

As states and input types (messages) are finite, our types have finite states, thus avoiding the infinite state verification problem. We also provide a more accurate description of interface behavior than the traditional interface definitions, which specifies signature of methods but not complex interface behavior.

5 Related work

Floch's PhD thesis [Flo03] provides a validation approach for dynamic service composition, which is similar to our work. Floch models the behavior of the service components as state machines using SDL; projection is used to transform it into interface behavior (also described as state machines using SDL-like notation). Our service specification has a higher level abstraction of behavior, as only action types are defined. Therefore, implementation details of the internal actions are already hidden, while at the same time keeping all the information about interface interactions. This simplifies the translation process. Another difference is that we provide translation by directly analyzing the XML data structure, whereas the transformation in Floch's work is based on state graphs.

The behavioral type language we used was first issued in [CFN03], and further developed in [Car03,CFN05]. Many type systems exist to capture the behavior of processes, actors or components, most of them based on process algebras like π-calculus. The closest type system to the one we used is the one of Najm et al. [NNS99]. The authors propose an actor calculus featuring regular or infinite-state behavior. Although they detect "message-not-understood" errors, communications are one-way. Ravara and Vasconcelos, in [RV00], were also inspired by Najm et al., but they did not make any distinction between inputs and outputs, hence their notion of error is rather loose and did not fit in our needs. They corrected this with Gay, providing so-called "session types" [GVR02], but distinguishing internal and external choice between actions (respectively client and server choices), which means we have to add messages to make sure those choices are made accordingly. De Alfaro and Henzinger provide another way of specifying interface behavior, using Interface Automata [dAH01]. Those automata specify the sequence of input/output allowed on an interface, but the compatibility they develop is too weak for our architecture. Indeed, two components are compatible if there exists an environment that can interact with the product automaton of the components' types; we believe this does not allow detecting "message-not-understood" in a plug-and-play environment such as TAPAS.

6 Conclusion

We have presented an approach for verification of the services, based on interface verification techniques for the verification of role sessions. We also provide an automatic translation from XML-based EFSM service specification to the behavioral type language applied. This language has a sound theoretical basis, and provides formal framework for compositional verification of component based systems. Especially, it allows us to capture "message-not-understood" errors while plugging a new component. The automatic translation provides an efficient and reliable way to extract interface types. An experiment has been carried out on an example application called TeleSchool.

For further work, the translated interface description can be used to compare specifications of behavior / service, so that dynamic service discovery can be done based on more accurate semantic interface behavior description comparison instead of simply signature matching. Furthermore, the dynamic assembly of components for validation needs to be further developed so as to provide safe plug-and-play techniques for components. Finally, we did not use all the features provided by the behavioral type language, and put aside the **may** and **must** modalities on the actions. We think about using the latter modality ("I have to send/receive") together with service-goal developed by Sanders and Bræk [SB04]: some actions can be specified as obligatory (**must**), so the service goal is fulfilled.

References

[AHAS03] F. A. Aagesen, B. E. Helvik, C. Anutariya, and M. M. Shiaa. On adaptable networking. In *ICT'03, Proceedings*, Assumption University, Thailand, 2003.

[Car03] C. Carrez. *Contrats Comportementaux pour Composants.* PhD thesis, ENST, Paris, France, December 2003.

[CFN03] C. Carrez, A. Fantechi, and E. Najm. Behavioural contracts for a sound composition of components. In *FORTE'03*, volume 2767 of *LNCS*. 2003.

[CFN05] C. Carrez, A. Fantechi, and E. Najm. Assembling components with behavioural contracts. *Annals of Telecomms*, 2005. To appear. Ext. of [CFN03].

[dAH01] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-01*, volume 26, 5 of *Software Engineering Notes*. ACM Press, 2001.

[Flo03] J. Floch. *Towards Plug-and-Play Services: Design and Validation using Roles.* PhD thesis, NTNU, Trondheim, Norway, February 2003.

[GVR02] S. Gay, V. T. Vasconcelos, and A. Ravara. Session types for inter-process communication. Preprint, Dept. of Computer Science, Univ. of Lisbon, 2002.

[Hol90] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, November 1990.

[JA03] S. Jiang and F. A. Aagesen. XML-based dynamic service behaviour representation. In *NIK'03, Proceedings*, Oslo, Norway, Nov. 2003.

[JAHB99] U. Johansen, F. A. Aagesen, B. E. Helvik., and R. Bræk. Design specification of the PaP support functionality. Technical Report 1999-12-10, Department of Telematics, NTNU, 1999.

[LS84] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[LX04] E. A. Lee and Y. Xiong. A behavioral type system and its application in ptolemy ii. *Formal Aspects of Computing*, 16(3):210–237, August 2004.

[Nie95] O. Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[NNS99] E. Najm, A. Nimour, and J.-B. Stefani. Infinite types for distributed objects interfaces. In *FMOODS'99, Proceedings*, Firenze, Italy, February 1999.

[RV00] A. Ravara and V. T. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR 2000*, volume 1877 of *LNCS*, pages 474–488. Springer, 2000.

[SB04] R. Sanders and R. Bræk. Discovering service opportunities by evaluating service goals. In *EUNICE'04, Proceedings*, Tampere, Finland, June 2004.