Formal Verification of Service Oriented Adaptive Driver Assistance Systems

Christian Schwarz and Dieter Zöbel
Universität Koblenz-Landau

Marco Wagner
Hochschule Heilbronn

APRES 2013
2013-04-08
Distributed Driver Assistance Systems

State of the art driver assistance systems

- static software and system architecture
- functional and electrical topology is known at design time.
- Examples:
  - ABS

Future driver assistance systems

- distributed over multiple entities
- system architecture changes at runtime

Examples:
  - Car-to-Car
  - Car-to-Infrastructure
  - DAS for articulated vehicles

Source: Daimler AG
Source: car-to-car.org
Case Study

**DAS for backing up a car/trailer-combination**
Case Study: Components

DAS for backing up a car/trailer combination

- sensors:
  - steering angle
  - hitch angle
  - rear view camera

- assistance logic: Computation of trajectories for
  - trailer
  - car/trailer combination

- output:
  - overlay
  - video output
Service Orientation

- handles:
  - high degree of distribution
  - the heterogeneity, and
  - the demand for reusability
- implements:
  - information hiding
  - well defined interfaces
  - service discovery

Automatic Re-Orchestration

- automatic adaptation of the software in case of a system change

![Service Orientation Diagram](Source: w3c.org)
Case Study: Service Oriented Model

- Hitch Angle Sensor Service
- Steering Angle Sensor Service
- Rear View Camera Service

Client

Compute Trajectory Trailer Service

Compute Trajectory Combination Service

Overlay Service

Video Out Service
Formal Verification

Verification

- safety critical systems must be validated
- in particular safety critical systems that are generated at runtime have to be validated at runtime
- approach: formal verification

Model Checking

- fully automated procedure
- checks a given specification for a given property (given in a temporal logic) (e.g. safety)
- either proves the property or gives a counterexample
Temporal Consistency as a Safety Property

**Def.: Temporal Consistency**

A system is temporal consistent with respect to a time period $\Delta t$ if its outputs do not rely on any inputs older than $\Delta t$.

**Case Study: A Safety Property**

The presented DAS for backing up a car/trailer combination is safe, if it is temporal consistent with respect to 100ms.
Case Study: Service Oriented Model

- Hitch Angle Sensor Service
- Steering Angle Sensor Service
- Rear View Camera Service

Services:
- Compute Trajectory Trailer Service
- Compute Trajectory Combination Service
- Overlay Service
- Video Out Service

Clients:
- Client

Christian Schwarz
Case Study: SoaML

SoaML

- UML-Profil for the specification of service oriented software
- contains (besides others) artifacts for interfaces and contracts
Case Study: SoaML

SoaML

- UML-Profil for the specification of service oriented software
- contains (besides others) artifacts for interfaces and contracts

| Consumer: | CalcOverlayRequest |
| OverlayService: | CalcOverlay |

```
<<Interface>>
GetTrajectoryComb

<<Interface>>
GetRearViewRequest

<<ServiceInterface>>
OverlayService

<<Capability>>
CalcOverlay

<<use>>

<<Exposé>>
```

- min: 1ms
- max: 4ms

- min: 1ms
- max: 4ms

- min: 7ms
- max: 22ms
Hybrid Systems

Discrete Systems
- discrete state changes
- finite state machines, activity diagrams, ...

Physical Systems
- continuous state changes
- differential equations

Hybrid Systems
- continuous and discrete state changes
- hybrid automata

\[ \dot{v} = g - \frac{kv^2}{m} \]

\[ \begin{align*}
\text{raising} & : x = 0 \\
\text{sync} & : x' = x \\
\text{falling} & : \dot{v} = g - \frac{kv^2}{m}
\end{align*} \]
Hybrid Automata

\[ I : \begin{align*} x &= 0 \\ x' &= x \\ v' &= -\alpha \cdot v \end{align*} \]

\[ F : \begin{align*} \dot{x} &= v \\ \dot{v} &= g - \frac{kv^2}{m} \end{align*} \]

\( x_0 = 3 \)
\( v_0 = 0 \)
Transformation of Contracts

Due to the lack of formal semantics of SoaML models, we have to make certain assumptions on the behavior of the specified services. We assume that whenever a service is executed (this might be event- or time-triggered), as a first step, it gathers all data needed to fulfill its task. Then there is some data processing step that produces the output. As a last step, it will send its reply.

In the following, we take a look at the artifacts of the SoaML models and how they are transformed. The **ServiceInterface** of a service describes which data (resp. other services) is needed to execute the task. Fig. 5 shows the transformation of the interface of the Overlay Service. The service is idle until it is activated, modelled by the synchronization label `act_p`. Then it attempts to gather its input data. We decided to model this by an indeterministic choice of all needed inputs. This is valid because the system is safe if and only if all possible executions are safe – including the worst choice. Next, the service might have to wait for an answer of the queried service. After receiving the answer, it signals (via `fin`) that the data processing may begin.

The **ServiceContract** determines the service type of the providing service. Moreover, it describes the timely behavior of both, the communication and the data processing step. Fig. 6 shows the transformation of the Overlay Contract. The contract automaton is idle until the date provided by the contract is needed (signaled via `req_c`). Then, it simulates the time needed to send the request. Next, it signals (via `act_p`) to start the gathering of inputs and waits for that to finish (`fin_c`). Afterwards, the time needed to generate the output and to send it to the requester is simulated. Finally, it signals that the requesting interface automaton can continue (`rep_c`).

---

**Fig. 6: Transformation of the ServiceContract of the Overlay Service (Fig. 4)**

- **Consumer:** CalcOverlayRequest
- **OverlayService:** CalcOverlay

**Overhead:**
- **F:** $\dot{t} = 1$, $\dot{A}_c = 1$
- **I:** $t \leq 4$

**Min:** 1ms
**Max:** 4ms

**Overhead:**
- **F:** $\dot{A}_c = 1$
- **I:** $t \leq 7$

**Min:** 7ms
**Max:** 22ms

**Overhead:**
- **F:** $\dot{t} = 1$, $\dot{A}_c = 1$
- **I:** $t \leq 4$

**Min:** 1ms
**Max:** 4ms
Transformation of Contracts

Due to the lack of formal semantics of SoaML models, we have to make certain assumptions on the behavior of the specified services. We assume that whenever a service is executed (this might be event- or time-triggered), as a first step, it gathers all data needed to fulfill its task. Then there is some data processing step that produces the output. As a last step, it will send its reply.

In the following, we take a look at the artifacts of the SoaML models and how they are transformed. The ServiceInterface of a service describes which data (resp. other services) is needed to execute the task. Fig. 5 shows the transformation of the interface of the Overlay Service. The service is idle until it is activated, modelled by the synchronization label act_p. Then it attempts to gather its input data. We decided to model this by an indeterministic choice of all needed inputs. This is valid because the system is safe if and only if all possible executions are safe – including the worst choice. Next, the service might have to wait for an answer of the queried service. After receiving the answer, it signals (via fin) that the data processing may begin.

The ServiceContract determines the service type of the providing service. Moreover, it describes the timely behavior of both, the communication and the data processing step. Fig. 6 shows the transformation of the Overlay Contract. The contract automaton is idle until the date provided by the contract is needed (signaled via req_c). Then, it simulates the time needed to send the request. Next, it signals (via act_p) to start the gathering of inputs and waits for that to finish (fin_c). Afterwards, the time needed to generate the output and to send it to the requester is simulated. Finally, it signals that the requesting interface automaton can continue (rep_c).
Transformation of Interfaces

Given these annotated models, we will now describe how to transform them automatically into hybrid automata, which can be used for model checking. The transformation presented here is modular – each component of the SOAOL model is transformed independently from the others. This allows a re-use of already transformed components and on the other hand enhances traceability if the verification attempt should fail.

We identified two types of services in our application domain, namely event-triggered and time-triggered services. The communication of event-triggered services is synchronous; a service requesting the data of such a service sends a request and waits for it to send a reply. This is the common service type in web services. For real-time systems this might not be optimal due to the additional delay caused by sending requests.

Time-triggered services on the other hand are triggered periodically and send their output to a buffer independently of a request. Periodic processes are standard in real-time systems engineering as they have superior timely behavior. On the other hand, they might result in unnecessary computation if the data is not actually needed.

We believe that being able to handle both service types in one architecture is beneficial, as it allows to handle both – sporadic and periodical events in an optimal way. In our transformation approach, we decided to treat both service types as uniform as possible. Basically, we treat time-triggered components to act_overlay.

$\dot{A}_p = 0$

$A_p = A_{RV}$

$\text{req}_{RV}$

$\text{req}_{TTC}$

$A'_{RV} = \text{rep}_{RV}$

$A'_{TTC} = \text{rep}_{TTC}$

$A_0$
Transformation of Interfaces

Given these annotated models, we will now describe how to transform them automatically into hybrid automata, which can be used for model checking. The transformation presented here is modular – each component of the SoaML model is transformed independently from the others. This allows a re-use of already transformed components and on the other hand enhances traceability if the verification attempt should fail.

We identified two types of services in our application domain, namely event-triggered and time-triggered services. The communication of event-triggered services is synchronous; a service requesting the data of such a service sends a request and waits for it to send a reply. This is the common service type in web services. For real-time systems this might not be optimal due to the additional delay caused by sending requests.

Time-triggered services on the other hand are triggered periodically and send their output to a buffer independently of a request. Periodic processes are standard in real-time systems engineering as they have superior timely behavior. On the other hand, they might result in unnecessary computation if the data is not actually needed.

We believe that being able to handle both service types in one architecture is beneficial, as it allows to handle both – sporadic and periodical events in an optimal way. In our transformation approach, we decided to treat both service types as uniform as possible. Basically, we treat time-triggered components to act_overlay fin_overlay
4 Transformation and Verification of SoaML models

Due to the lack of formal semantics of SoaML models, we have to make certain assumptions on the behavior of the specified services. We assume that the contract automaton is triggered whenever a service is executed (this might be event- or time-triggered), as a processing step that produces the output. As a last step, it will send its reply.

Time-triggered services on the other hand are triggered periodically and send their input or output data (resp. other services) is needed to execute the task. Fig. 5 shows the transformation of the SoaML model of the Overlay Service (Fig. 4) to the corresponding model checking system.

We identified two types of services in our application domain, namely event-triggered and time-triggered services. The communication of event-triggered services is synchronous; a service requesting the data of such a service sends a request and waits for it to send a reply. This is the common service type in web services.

The transformation presented here is modular – each component of the SoaML model is transformed independently from the others. This allows a re-use of the corresponding answers.

Moreover, it describes the timely behavior of both, the communication and the processing step that produces the output. As a last step, it will send its reply.

We believe that being able to handle both service types in one architecture is activated, modelled by the synchronization label
Future Work

• more complex properties to check
• make use of counter examples for the orchestration
Thank you for your attention!