Ontology-based Runtime Reconfiguration of Distributed Embedded Real-Time Systems

Oliver Höftberger
Vienna University of Technology, Austria
Email: oliver.hoeftberger@tuwien.ac.at

Roman Obermaisser
University of Siegen, Germany
Email: roman.obermaisser@uni-siegen.de

***Abstract*—Embedded real-time systems with dynamic resource management capabilities are able to adapt to changing resource requirements, resource availability, the occurrence of faults and environmental changes. This enables better resource utilization, more flexibility and increased dependability. Depending on the application domain, reconfiguration decisions must be found and applied within temporal bounds. Although semantic techniques are used to react to unexpected events in standard IT systems, they exhibit a computational complexity and temporal unpredictability that is not suitable for real-time systems. This paper describes a temporally predictable framework for reconfigurable embedded real-time systems. It uses a service-oriented approach to dynamically reconfigure component interactions. Knowledge about the system structure and semantics is provided in a system ontology with relevant information for embedded real-time systems (e.g., transfer delay times, accuracy of relations). The ontology allows to automatically generate service substitutes by exploiting implicit redundancy in the system. Furthermore, an algorithm is presented that searches the ontology for semantically equivalent implementations of failed services. The process of substitution search and substitute service generation is demonstrated with an example from the automotive domain.**

I. Introduction

Dynamic resource management enables a system to dynamically adapt to changes in the environment (e.g., take off or in-flight in a plane) and to changes in resource demands or resource availability (e.g., power, time (scheduling), communication bandwidth, memory) while ensuring real-time constraints. It enables better resource utilization, improved dependability, and it makes power-aware system behavior possible. Primarily due to safety concerns, reconfiguration of safety-critical systems is often reduced to selecting system-wide modes out of statically defined scheduling tables, which impairs the flexibility of resource management, but provides good analyzability and determinism.

In standard IT systems, semantic techniques and service oriented architectures have been successfully employed to support self-adaptation, context awareness and fault-tolerance, e.g., in business-to-business transactions [1].

Although self-adaptation in real-time systems has been addressed in prior work [2], [3], [4], the extension of semantic techniques, usually used on large-scale IT systems, for dynamic reconfiguration in embedded real time systems is an unsolved research challenge. The major research gaps are as follows:

- *System architecture for runtime adaptation in real-time networked embedded systems.* Architectural building blocks are required to manage a knowledge base with facts about the system (e.g., constituent components, interfaces, services, semantics) and exploit this knowledge for the self-adaption of the system (e.g., substitution of failed services).
- *Ontology for Embedded Real-Time Systems:* Methods for modeling embedded systems based on semantic techniques are required. The interfaces of embedded systems are substantially different to those of IT systems; so the semantics used to describe an interface from existing languages (e.g., [5]) must be adapted to the specific requirements of embedded systems (e.g., precise temporal interface specifications, deadlines of services, temporal accuracy of information [6, p. 116]).
- *Temporally predictable and reliable algorithms for self-adaptation.* Algorithms for deriving a new configuration within predictable time are required in order to perform fault recovery actions within the temporal bounds of the real-time application. The state-of-the-art of semantic technologies does not support temporal guarantees for inference.

This paper provides contributions towards the above challenges. Section II introduces a system architecture for runtime adaptation in real-time networked embedded systems. A system ontology for embedded systems is introduced in Section III. In Section IV we propose a temporally predictable substitution algorithm, which identifies new system configurations where failed services have to be replaced. Section V demonstrates this approach with an automotive use case. Sections VI and VII conclude the paper with a discussion and prospects for future work.

II. Model of Reconfigurable System

An appropriate system architecture supporting runtime reconfiguration has to provide means to dynamically adapt the behavior and interactions of software components to changing environmental conditions and requirements. This involves online planning of component interactions, allocation of software tasks to computational nodes in a distributed system, scheduling of tasks and resources. Different proposals for such architectures have been made in the past, e.g., [7]. For the proposed reconfigurable system framework, a service-oriented architecture was found to be the most suitable approach, as this allows to dynamically reconfigure applications by replacing individual services. Additionally, services can easily be shared between distinct applications without creating undesired dependencies. The usage of service-oriented architectures [8] for embedded systems has already been proposed in several publications, like [9], [10] or [11]. Within the service-oriented approach the reconfigurable system is composed of *components* which are self-contained pieces of software or hardware with periodic input and output behavior. A *service* is the specified behavior of a component. Components provide their services to other components using a well-defined message-based interface.

Applications are defined by the composition of services that have to be executed such that the task of the application is fulfilled. Composing two services means, that the output of the component that is providing the first service is linked with the input of another component that is providing the second service. The service composition is then the concatenation of services that form an application. Dynamic reconfiguration is done by substitution of individual services of a service composition.

A. Interface specification

Components have to interact with their environment and need to cooperate with other components in order to utilize their services. Hence, different component interfaces are needed to compose the system. In [12] and [13, p. 15ff] the authors elaborate on the interface specification of system components in order to ease design and composition of real-time systems. The behavior of a component (i.e. the component's service) has to be entirely defined by the specification of its interactions at the interfaces of the component, while the internal component implementation can be seen as a black-box. A component is considered as correct, when it provides its service. This means that its interactions at the component interfaces are conform to the service specification of the component. Contrarily, a component fails when its interface behavior deviates from the service specification. From the point of view of dynamic reconfiguration two different types of interfaces are of interest:

- **Linking Interface:** The linking interface is used to exchange messages between cooperating components. Thereby the specification of the linking interface comprehends the whole semantics of the component's service.
- **Local Interface:** At the local interface a component interacts with its physical environment by accessing external devices like sensors, actuators, IO devices or fieldbuses. Due to the heterogeneous nature of such devices, local interfaces typically do not have a common structure or temporal behavior. The functionality of external devices (e.g., input from a sensor, output towards an actuator) is provided as service to other components via the linking interface.

In order to automatize the substitution and composition of services, the linking interface of a component has to follow a defined structure. The linking interface specification defines for all input and output messages of a component the following properties:

- **Value domain:** definition of the data structure of a message with its data types, value ranges and maximum rate of change.
- **Temporal requirements for input messages:** maximum allowed time between two consecutive incoming messages (i.e. maximum message period).
- **Temporal behavior for output messages:** time between two consecutive outgoing messages (i.e. period of message).
- **Accuracy:** value of accuracy required for input messages or provided for output messages. In measurement theory accuracy is considered to be the *"Closeness of the agreement between the result of a measurement and the true value of the measurand [14]"*. Hence, the accuracy is a measure for the worst case deviation of the message content at the moment it is received (i.e. in case of a required input message) or sent (for output messages) by the component from the actual value – in the real physical system. Reduced accuracy might originate from inexact measurements (e.g., input from sensors with extenuated precision) or when a service implements an inexact model of the real world (e.g., intentional omission of terms in an equation to reduce computational effort). On the other hand, the accuracy can be increased by measures like sensor fusion. The relative measurement error between the measured value $m_{use}(t)$ and the actual value $m_{act}(t)$ is usually defined based on the value range r of the measured entity [15]. Hence, the relative measurement error $e_{rel}(t)$ at time t is:

$$e_{rel}(t) = \left| \frac{m_{use}(t) - m_{act}(t)}{|r|} \right| \tag{1}$$

As components not just output measurement data from sensors, but also processed data from other services, also the temporal accuracy of data within messages has to be considered. The temporal accuracy is a function of the age Δt of a measured entity (i.e. time between its value was observed (sampled) and the time the value is used) and the maximum rate of change $\frac{dm}{dt}$ of the entity [6, p. 103]. The accuracy value a_m for message m is then:

$$a_m = 1 - \underset{\forall \Delta t}{max} \left(e_{rel}(t) + \left| \frac{dm}{dt} \Delta t \right| \right) \tag{2}$$

B. Service composition

In this system model applications are divided into sets of loosely coupled interacting services implementing distinct functionalities. For instance, sensor elements are connected to (or are part of) components which service it is to provide the actual sensor value at the linking interface in order to be used by other services. Correspondingly, actuator services take inputs from other services and control according actuators, while processing services use inputs from some services and produce output for distinct services. The data-flow in this model is conform to *Kahn Process Networks* described in [16, p. 155ff].

Services are composed by connecting the output of one component with the input of other components. Thereby, for each required input at the linking interface of a component another component is determined which output specification contains a corresponding value. If all requirements (i.e. data type compatibility, temporal behavior and data accuracy) can be fulfilled, both components can be connected, and hence, services are composed. An application is independent from other applications if it is formed by a group of services, where for each service all required input messages are provided by at least one other service within the same group. In case of an erroneous component or the removal of a service from the system, an application can be kept operational by substituting the missing service with a semantically equivalent service. Such a semantically equivalent service can either be an existing service that provides the same information as the disappeared service (e.g., replacing some sensor service by a redundant sensor service), or a transfer service is generated that takes inputs from other services and calculates the required information. In this case a component has to be instantiated that provides the generated transfer service, and therefore, outputs the required information at the component's interface.

In order to keep the focus of this paper on semantic service substitution it is assumed that all components are executed with the same period. Additionally, components are statically scheduled with a defined offset within that period. Hence, there is no influence of other periods on the temporal accuracy of output values of a service, when they are used by another service. This simplifies the substitution search during reconfiguration. As this assumption might be infeasible for some distributed embedded systems, it will be part of future work to overcome this restriction.

Dynamic reconfiguration can be facilitated by using deterministic execution frameworks like the ACROSS platform [17], which inherently provides a message-based interface between components. In this platform a common notion of time is established among all components that allows the temporal coordination of interacting components. Furthermore, the time-triggered communication paradigm with reconfiguration facility ensures predictable component interactions and segregation between independent components.

III. System Ontology

Expert knowledge about the system structure, relations and interactions between subsystems of the distributed embedded real-time system is formally described in the system ontology. In case of a service failure, the ontology can automatically be searched for groups of services that permit to reconstitute the failed service by a semantically equivalent service. The proposed ontology description framework restricts the design space compared to existing ontology languages (e.g., [18], [19], [20] or [21]) in order to reduce the complexity of reasoning algorithms, and hence, allow predictable service substitution. The resulting system ontology is composed of entities that are classified as one of two basic building blocks: *concepts* and *relations*.

A. Concepts

Concepts denote entities in the system which have particular properties and which are in relationship with other concepts. Using different types of relations (cf. Subsection III-B) hierarchies and instantiations of concepts can be defined. Thereby, instantiated concepts inherit properties and relations of higher level concepts. In return, higher level concepts represent classes of entities that share the same properties and relations. In the system ontology the following three different types of concepts have to be distinguished (see Figure 1):

- **structure concepts:** facilitate decomposing a system into structural parts with defined properties, connections and interdependencies. For instance, a car consists of different parts like the engine, gear box, wheels, etc. that have different properties and might interact with each other. Hierarchies of structural concepts can be modeled in order to group concepts that are sharing the same properties and interconnections.
- **property concepts:** model measurable static (e.g., length of vehicle) or dynamic attributes (e.g., speed of vehicle) in the system. Input and output parameters of services are directly mapped to property concepts of the system ontology. This means, that a service is either interested in the value of the property concept, or the service provides the value of that property concept as output to other services. As the value of a property concept can be provided by different services simultaneously and each service might use a distinct method to determine that value, the accuracy of the provided value strongly depends on the actual service that provides the value. Hence, not a single accuracy value for a property concept exists, but for each providing service an accuracy value is given. This accuracy value is part of the linking interface of the service (cf. Section II-A).
- **transfer concepts:** model the relationship between concepts by describing the transfer behavior from input property concepts to output property concepts. Transfer concepts have at least one property concept as input and another property concept as output. Various implementation choices of the transfer behavior model of transfer concepts are possible, like simple 1-to-1 mappings, mathematical functions, finite-state machines, look-up tables, or transformation codes using a programming language. Transfer concepts contain a transfer delay time that measures the duration needed until the transfer behavior model output reaches 90% of its stable value after a step-change at the transfer behavior model input [6, p. 7]. Furthermore, each transfer concept is assigned an accuracy function that defines the influence of the transfer behavior on the accuracy of the output value. Thereby it relates the accuracy of each input to the transfer behavior with the output accuracy. Accuracy reductions due to imperfections of the transfer behavior model (e.g., terms with negligible influence on the equation are omitted to reduce computational effort) are also reflected by the accuracy function.

B. Relations

Relations are binary connections between concepts. Different types of relations are distinguished that can be used to construct the system ontology. More complex relations between several concepts have to be modeled using transfer

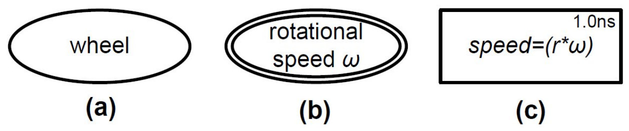

Fig. 1. Examples of distinct types of concepts: (a) structure concept, (b) property concept, (c) transfer concept with output property *speed* and input properties r and ω.

concepts (cf. Subsection III-A). In the following the defined relations are presented:

- **input:** connects property concepts with transfer concepts. Thereby the connected property concept acts as input for the transfer function of the transfer concept.
- **output:** connects property concepts with transfer concepts. Thereby the connected property concept acts as output for the transfer function of the transfer concept.
- **has-property:** assigns property concepts to structure concepts. In case the structure concept is instantiated, the property concepts connected to that structure concept are inherited.
- **is-part-of:** is used to hierarchically structure a system. It defines the hierarchy between structure concepts such that one of them is a subcomponent of the other one. This implies that all property concepts of the higher level structure concept are identical for the lower level structure concept.
- **isA:** is also used to define a hierarchy between structure concepts. But in contrast to *is-part-of*, this relation has the semantics of a generalization (e.g., *wheel* is a generalization of the *left front wheel* of a car). All lower level structure concepts inherit the higher level structure concept's relations to other concepts. The number and types of property concepts are inherited, but the actual property value of each instance is different (e.g., the *left front wheel* probably has a different speed than the *right front wheel*).
- **provides:** declares that the value of a property concept is provided by some structure concept. Provision of property values can, for instance, be conducted by reading sensor measurements, data processing or simulation. Property concepts that are provided by structure concepts are essential for the service substitution algorithm (see Section IV), as the algorithm searches until appropriate provided property concepts have been found.
- **controls:** describes the influence of a structure concept onto a property concept. This is typically the case when an actuator or output device interacts with its environment.

IV. Substitution Algorithm

The substitution algorithm plays a crucial role in the self-adaptation process of the reconfigurable system. After a service vanishes from the system (i.e. due to a component fault or its intentional removal), the substitution algorithm has to determine if the semantics of the service can be substituted by combining other services in the system. In the positive case these services have to be composed to provide a semantic equivalence of the missing service. Thereby, the challenge is that for real-time systems the process of service substitution has to be conducted within the temporal bounds determined by the real-time system.

All inputs and outputs of services are mapped to property concepts in the system ontology. If a service disappears the property concepts that are mapped to the output of the service are not provided any longer. Hence, a semantically equivalent service has to provide the same property concepts as the original service, but it might use different property concepts at the input. When a component failure has been detected or a newly integrated service requires the value of a property concept, which is currently not provided, as input, the service substitution algorithm is called for each required property concept. In order to find semantic equivalences for these property concepts, the substitution algorithm processes the system ontology. It searches for combinations of provided property concepts that allow to deduce the value of the required property concept. After a combination of such provided property concepts has been found, a transfer service is generated automatically. The transfer service takes the individual property concepts of the found combination as input and deduces the value of the required property concept. This is done by using the models represented by the transfer concepts which connect the property concepts found for substitution. At the end a component has to be instantiated that provides the transfer service. This component can then be used to provide the required property concept based on the values of other property concepts in the system ontology.

In the following the substitute search and transfer service generation algorithms, which form the major parts of the substitution algorithm, are detailed. Beside these algorithms, also algorithms for fault detection as well as service and communication scheduling are required. Such algorithms have been discussed in many publications, e.g., [6, p. 126ff], [22] or [23], and thus, are not part of this paper.

A. Substitute Search

The substitute search algorithm has to find possibilities to derive the value of a property concept from distinct property concepts in the system ontology, which have to be provided (e.g., by a dedicated service or given as constants in the system). Starting from the required property concept, paths in the system ontology have to be found that lead to provided property concepts.

Algorithm 1 specifies the substitute search algorithm which performs a depth first search on the system ontology. In order to improve the search characteristics of the algorithm different heuristics for ontology traversal are possible. However, basic mechanisms for identification of semantically equivalent substitutes will remain the same.

The presented algorithm checks for each concept n that has a direct relation towards the required property concept r whether it is a transfer concept, or not. In case of a transfer concept this means, that n could output a value that is semantically equivalent to the value of r. But this is only

Algorithm 1 Substitute Search algorithm

```
 1: function SUBSTITUTESEARCH($r$)
 2:     for all $n \in r.inputs$ do
 3:         if $n = TRANSFER\_CONCEPT$ then
 4:             if $checkRequirements() \neq ACCEPT$ then
 5:                 continue
 6:             end if
 7:             $S \leftarrow \{\}$
 8:             $T \leftarrow \{n\}$
 9:             $provided \leftarrow true$
10:             for all $i \in n.inputs \setminus \{r\}$ do
11:                 if $i.provided = false$ then
12:                     if $i.visited = false$ then
13:                         $n.visited \leftarrow true$
14:                         $[s,t] \leftarrow SubstituteSearch(i)$
15:                         if $[s,t] = \{\}$ then
16:                             $provided \leftarrow false$
17:                             break              ▷ No substitute for $n$
18:                         else
19:                             $i.provided \leftarrow true$
20:                             $i.service \leftarrow s$
21:                             $i.tree \leftarrow t$
22:                             $S \leftarrow S \cup s$
23:                             $mergeTree(T, t)$
24:                         end if
25:                     else
26:                         $provided \leftarrow false$
27:                         break                  ▷ No substitute for $n$
28:                     end if
29:                 else
30:                     $n.visited \leftarrow true$
31:                     $S \leftarrow S \cup i.service$
32:                     $mergeTree(T, i.tree)$
33:                 end if
34:             end for
35:             if $provided = true$ then
36:                 return $[S, T]$
37:             end if
38:         else
39:             if $n.provided = false$ then
40:                 if $n.visited = false$ then
41:                     $n.visited \leftarrow true$
42:                     $[s,t] \leftarrow SubstituteSearch(n)$
43:                     if $[s,t] \neq \{\}$ then
44:                         $n.provided \leftarrow true$
45:                         $n.service \leftarrow s$
46:                         $n.tree \leftarrow t$
47:                         return $[s,t]$
48:                     end if
49:                 end if
50:             else
51:                 $n.visited \leftarrow true$
52:                 return $[n.service, n.tree]$
53:             end if
54:         end if
55:     end for
56:     return $\{\}$                              ▷ No substitute found
57: end function
```

true under the condition that all input concepts to transfer concept n can also be provided (e.g., by a dedicated service, as system constant or deduction from other property concepts). Before checking that all input concepts i of n are provided, requirements on the achievable accuracy on the search path – in the temporal and value domain – have to be verified. In the algorithm this is done by calling $checkRequirements()$. This function uses the accuracy functions of the transfer concepts on the actual search path to calculate the maximum achievable accuracy. Afterwards it checks if the achievable accuracy is within the required accuracy bound for input values of the required property concept r. Additionally, the transfer delay times of the transfer concepts are summed up and compared with the temporal requirements for input values of r. In case the requirements check shows that the path leading to n does not fulfill the requirements of r, the algorithm tracks back and searches for other paths. Otherwise, it is determined whether all of the input concepts i of n are provided, or not. If i has already been marked to be provided (lines 30ff.), then the set of services $i.service$, on which the value of i depends, is added to the set S of services on which the required property concept r will depend on after a successful substitution. Furthermore, the tree structure T is updated (call of $mergeTree(T, i.tree)$), which holds the transfer concepts that connect the services in S. In case that i has not yet been marked as provided (lines 12ff.), two situations have to be distinguished: (1) i has already been visited, (2) it has not yet been visited.

(1) As the substitution search algorithm has already been applied for concept i and no successful substitution had been found, i need not be checked again. Consequently, due to the fact that an input for n exists that cannot be provided, n cannot be used to deduce the value of r.

(2) The algorithm is called recursively with i as required property concept. If it is not possible to provide i (line 15), then it is also not possible to deduce the value of r by using n. Otherwise, i is marked as provided and the list of services and the tree structure are updated accordingly.

Only if all inputs for n are provided, the algorithm returns the list of services S needed to deduce r as well as the tree structure T connecting these services.

In case n is no transfer concept, it means that there exists a direct mapping between r and n. If n is already provided, then also r can be provided by the same services (line 52). Else, if n is not provided and it has not yet been visited, the substitution search algorithm is called recursively, and in case of a successful substitution, the same results from the recursive call can be applied to r.

B. Transfer Service Generation

After substitutes for a property concept have been found, a transfer service has to be generated that calculates the value of the required property concept from a set of values from distinct property concepts that are provided. The transfer service generation algorithm takes the tree generated by the substitution search algorithm and transforms it into the transfer service that is then provided by a dedicated component. Each node within the tree is a transfer concept that has a specified transfer behavior which transforms the concept's inputs to an output. At the leaves of the tree transfer concepts are located which only have provided property concepts as input. The output of the tree's root represents the value of the required property concept. Hence, in order to generate the transfer service, the transfer behavior of transfer concepts in the tree has to be concatenated from the leaves towards the tree's root. Algorithm 2 presents the algorithm for the generation of a transfer service.

Algorithm 2 Transfer Service Generation algorithm

```
1: function GENERATETRANSFERSERVICE($T$)
2:     $behav \leftarrow \{\}$
3:     for all $i \in T.input$ do
4:         if $i.is\_service = true$ then
5:             $[b, v] \leftarrow getInput(i)$
6:         else
7:             $[b, v] \leftarrow generateTransferService(i)$
8:         end if
9:         $behav \leftarrow concatBehavior(behav, b)$
10:        $bindVariable(T.behav, v)$
11:    end for
12:    $behav \leftarrow concatBehavior(behav, T.behav)$
13:    $[b, o] \leftarrow getOutput(T.behav)$
14:    $behav \leftarrow concatBehavior(behav, b)$
15:    return $[behav, o]$
16: end function
```

The input to the algorithm is the root T of the tree structure holding the transfer concepts of the found substitution path. For each successor of the actual root node T, which denotes an input for the transfer concept at the root, it is checked whether the input is a service or another transfer concept (line 4). In case i being a service, it means that this input to the transfer concept is already available in the system and it is provided as output of the service i. Therefore, by calling $getInput(i)$ additional behavior b is generated that loads the output value of service i into variable v (line 5). Otherwise, if i is a transfer concept, the transfer service generation algorithm is called recursively. This results in the generation of behavior b that stores its output into variable v. Behavior b is then appended to the current input behavior already stored in the overall transfer behavior variable $behav$. This input behavior subsumes the behavior required to provide the inputs for the transfer concept at root node T. Additionally, variable v is bound to the actual transfer behavior of T. Hence, whenever the behavior at the root (i.e., $T.behav$) reads the output of b (which either loads the output of an existing service or it is the behavior of a transfer concept), the output variable v is accessed. After all inputs of T have been processed, the behavior of T itself is appended to the overall behavior $behav$. Finally, output behavior for T is generated that writes the results of $T.behav$ into output variable o, and the algorithm returns the behavior of the transfer service as well as the output variable.

V. USE CASE EXAMPLE

A practical example for applying ontology-based runtime reconfiguration can be found in the automotive domain, where modern cars integrate a multitude of sensors, actuators and networked electronic control units (ECUs). While safety and availability of automotive systems are of utmost importance, cost-efficient solutions are required to satisfy the needs of this mass market. Instead of expensive implementations with explicitly redundant system components, already available implicit redundancy can be exploited to increase reliability or save hardware cost. Thereby, the same system property (e.g., the steering angle) can be derived by different means from several other system properties. For instance, the steering angle can be directly measured by a dedicated sensor at the steering linkage, but it can also be derived from the curve radius of the car, which in turn can be deduced from the lateral acceleration and the car speed or from the difference in the angular speed of the inner and the outer wheel in the curve. The reasonableness of applying runtime reconfiguration in this application domain has also been identified in [24], [25]. In these papers the solution is based on predefined configurations with different behavior, which can be switched at runtime. In contrast to this, the ontology-based approach presented in this paper allows to automatically derive valid configurations from the system ontology at runtime.

Figure 2 depicts a small excerpt of a system ontology that could be used for automotive systems. This ontology is designed to highlight the usage of ontology-based reconfiguration, and therefore, should not be regarded to be complete. For invertible transfer functions of transfer concepts input and output relations are combined to a single input/output relation to reduce the number of relations and concepts. The system ontology presents knowledge about the interconnections and interactions of the steering mechanism of a car with its wheels as well as distinct other properties of the car (i.e. speed and accelerations). Transfer concepts are labeled with equations which explicitly or implicitly connect different properties in the system. Alternatively, $1:1$ in the transfer concepts means, that all properties of one structure concept can be equally used for the other structure concept connected by the transfer concept. With the ontology in the figure it will be demonstrated how the proposed algorithm derives a solution for the example of the steering angle above. In order to facilitate exercising the example, some of the property concepts are highlighted with gray background color.

Supposed the component providing the value of the steering angle fails to operate as specified. After this has been detected by a fault detection mechanism, the substitute search algorithm is triggered with the *steering angle* α property concept (dark gray ellipsis at the left of the ontology) as input. From this concept the algorithm searches towards the *steering* concept, and it will eventually reach *steering gear*, as none of the other concepts connected to steering provides the possibility to directly deduce the steering angle. Due to the inheritance property of the *is part of* relation, the steering gear implicitly possesses the same property concept *steering angle* as the concept *steering*. The steering gear mechanically translates the angle of the steering wheel into an angular displacement of the front wheels (denoted by *wheel angle* β). In the example this is a simple mapping of one complete rotation of the steering wheel to a $40°$ angle at the car's front wheels. Hence, only one of both values (i.e. *steering angle* or *wheel angle*) is required to deduce the other one. The algorithm proceeds towards the *wheel angle* to check if this is already provided by some service, or not. As this is not the case, the next transfer concept is encountered, which relates *wheel angle* β, *wheel base a* and the *curve radius r*. The wheel base is a constant in the system, and thus, is considered to be provided. In contrast, the curve radius is neither measured by a sensor, nor is it provided by

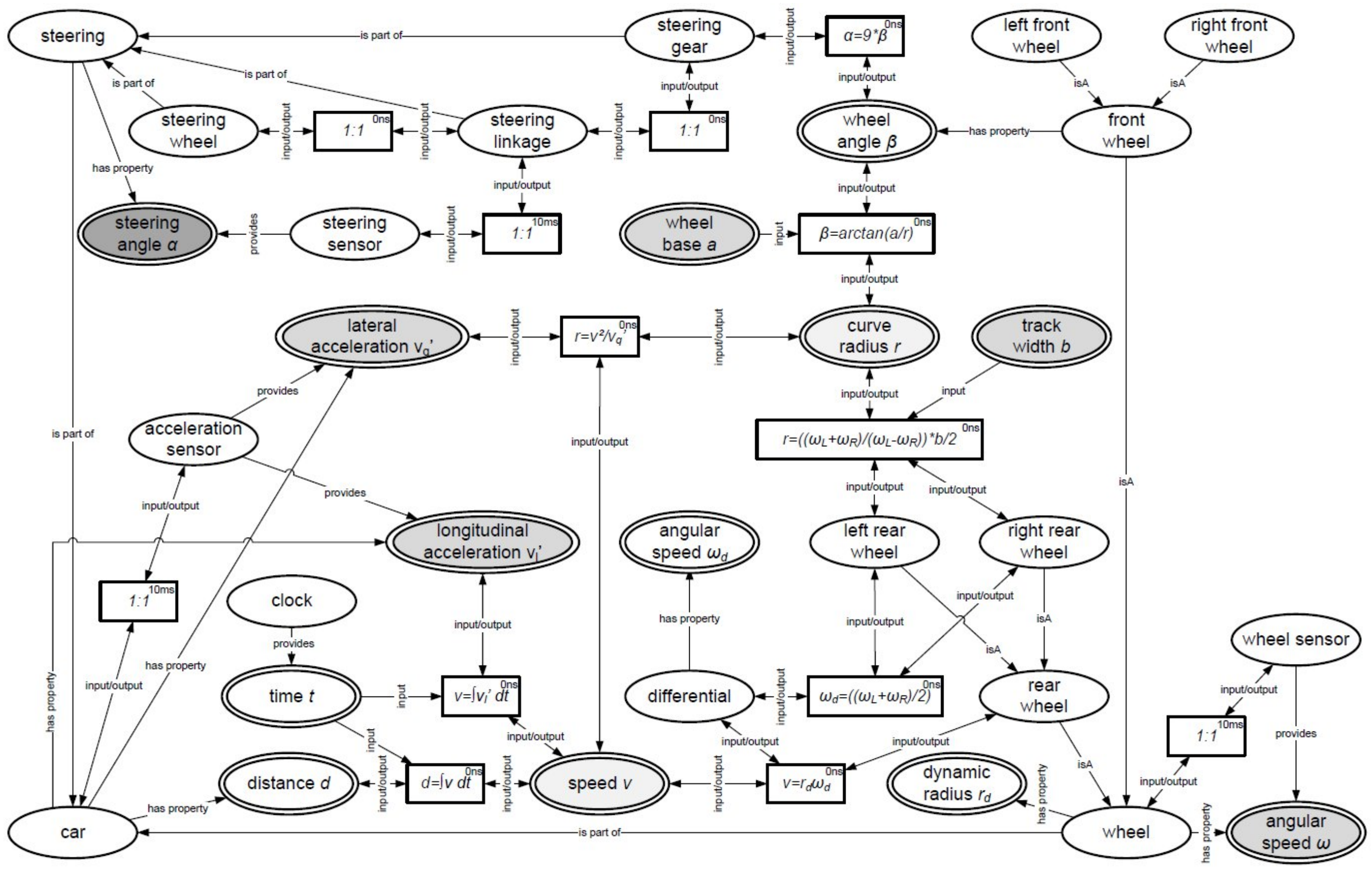

Fig. 2. Exemplary part of system ontology from automotive domain.

some other service. But if it was possible to obtain the value of the curve radius, then also the wheel angle and in succession the steering angle could be calculated. Depending on which relation the algorithm checks first (it depends on the search heuristics which input is taken first), the curve radius can either be derived from the *lateral acceleration* v_q' and the *speed v*, or from the *track width b* and the *angular speeds* ω_L and ω_R of the left and right rear wheels. While in the latter case the algorithm concludes that *steering angle* α can be derived from *wheel base a*, *track width b* and *angular speeds* ω_L and ω_R, in the first case the algorithm still has to derive the speed from other property concepts. Here, this is only possible by integrating the *longitudinal acceleration* v_l' over the time. Due to the missing *dynamic radius* r_d (i.e. the radius of the wheel that depends on the load and pressure within the tire) the speed cannot be calculated by the *angular speed* ω_d of the differential gearbox.

After the substitute search algorithm comes up with a solution – for instance the mentioned one with the *angular speeds* of the wheels – a transfer service has to be generated. This transfer service combines the values of the found property concepts in order to calculate the value of the required property concept (i.e. *steering angle* α). For the example use case, Figure 3 illustrates the tree structure that is generated by the substitute search algorithm. This tree is the input for the transfer service generation algorithm. Beginning at the leafs of the tree, which have to be provided property concepts, input behavior is generated that loads the value of the property concept into a variable. These variables are then bound to the transfer behavior of the according transfer concept. In the example this means, that the variables within the equations of the transfer behavior of transfer concepts are exchanged by the values of the property concepts. The output of each equation is again stored in some variable, which functions as an input for the next transfer behavior. After the tree has been traversed bottom-up, the transfer service generation is finished and the required property concept (i.e. *steering angle* α) will be stored in the output variable of the last transfer behavior equation. Finally, the transfer service itself is provided by a new component that provides input for other services.

VI. Discussion

In this paper a framework for runtime reconfiguration of distributed embedded real-time systems has been presented. The framework is based on the service-oriented approach, where the system functionality is provided by the services of individual components. Upon the failure of a component, the system functionality can be restored by exchanging the lost service with a semantically equivalent service. Such a semantically equivalent service can be automatically found by the proposed service substitution algorithm, which uses a system ontology to find equivalent substitutes. The development and application of these dynamically reconfigurable systems can be facilitated when predictable and deterministic frameworks like the ACROSS platform are used.

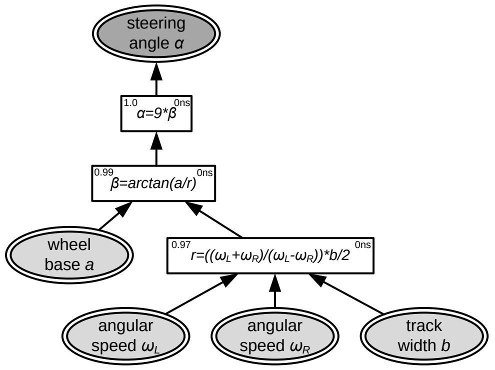

Fig. 3. Example tree for transfer service generation.

In order to be applicable for runtime reconfiguration of real-time systems, the substitution algorithm must have a deterministic temporal behavior. This means, that it has to terminate within a temporal bound dictated by the physical process that is controlled by the embedded system. In many state-of-the-art runtime reconfiguration frameworks (e.g., [3] or [7]) possible component interactions are modeled explicitly. Hence, runtime reconfiguration is reduced to the selection and switching between possible preconfigured configurations, which in turn results in a deterministic execution time for reconfiguration. But this approach is not flexible enough to react to unforeseen failures in the system. On the other side, reasoning in complex ontologies using expressive description logics typically has exponential complexity based on the size of the ontology [26]. Therefore, this approach cannot be used to find a solution for the substitution of a failed component in real-time.

Compared to the state-of-the-art of reconfiguration systems for embedded systems, the proposed framework and algorithms provide more flexibility by computing solutions for reactions to component failures from the system ontology. On the other hand, by reducing the expressiveness of the description language for the system ontology, the runtime behavior of the substitution algorithm becomes temporally predictable. The time complexity of the substitute search algorithm in a system ontology with n concepts and m relations, where no isA relations are used, is in $\mathcal{O}(m)$. This is due to the fact that each relation in the system ontology can be taken at most twice (i.e., once in each direction).

The isA relation defines a hierarchy between concepts, where properties of the higher level concept are instantiated for all lower level concepts. For instance, each of the four wheels of a car has an isA relation to the higher level concept $wheel$. Therefore, each individual wheel has its own instance of the properties of the high level wheel (e.g., angular speed, dynamic radius). When the car drives in a curve, the wheels will have different speeds. Due to the instantiation, during the substitution search the higher level concept might have to be accessed from different low level concepts (e.g., when the right wheel speed and the left wheel speed are accessed). Thus, the above mentioned complexity bound might not hold if the ontology contains isA relations. By preprocessing of the ontology, where all property concepts of the higher level structure concept of an isA relation are instantiated for the lower level structure concept, the complexity can be kept linear. However, the absolute number of concepts and relations increases due to instantiation.

VII. Future Work

The substitution search algorithm currently performs a depth first search on the system ontology. For large ontologies this might lead to a longer search time due to longer substitution paths. Thus, the resulting transfer service will also require more resources. Therefore a central topic for future work will be to develop different search heuristics for the substitution search algorithm and investigate their effectiveness.

A further part of future investigations will be the handling of state-based transfer behavior of transfer concepts (e.g., integration of speed to obtain the distance). In order to provide useful substitutions, a consistent state has to be maintained in the system. Possible solutions to this problem could be based on pro-active services which are permanently updating implicit system states, or state externalization and restoration techniques.

Acknowledgment

This document is based on the ACROSS project in the framework of the ARTEMIS program. The work has been funded in part by the ARTEMIS Joint Undertaking and National Funding Agencies of Austria, Germany, Italy and France under the funding ID ARTEMIS-2009-1-100208. The responsibility for the content rests with the authors.

References

[1] J. Cardoso, “Benchmarking a semantic web service architecture for fault-tolerant b2b integration,” in *Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. 26th IEEE International Conference on*, july 2006, p. 18.

[2] U. Brinkschulte, E. Schneider, and F. Picioroaga, “Dynamic real-time reconfiguration in distributed systems: timing issues and solutions,” in *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, may 2005, pp. 174 – 181.

[3] A. Rasche and A. Poize, “Dynamic reconfiguration of component-based real-time software,” in *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, feb. 2005, pp. 347 – 354.

[4] C. Prehofer and M. Zeller, “A hierarchical transaction concept for runtime adaptation in real-time networked embedded systems,” in *Proc. of the IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*, 2012.

[5] C. Bussler, E. Cimpian, and et al., “Web service execution environment (WSMX),” World Wide Web Consortium (W3C), Tech. Rep., 2005.

[6] H. Kopetz, *Real-Time Systems – Design Principles for Distributed Embedded Systems*, 2nd ed. Springer, 2011.

[7] O. Rawashdeh and J. Lumpp, “Run-time behavior of ardea: a dynamically reconfigurable distributed embedded control architecture,” in *Aerospace Conference, 2006 IEEE*, 2006, p. 15 pp.

[8] T. Erl, *Service-Oriented Architecture: Concepts, Technology and Design*, 5th ed. Prentice Hall, 2006.

[9] N. Milanovic, J. Richling, and M. Malek, “Lightweight services for embedded systems,” in *Software Technologies for Future Embedded and Ubiquitous Systems, 2004. Proceedings. Second IEEE Workshop on*, may 2004, pp. 40 – 44.

[10] A. Rezgui and M. Eltoweissy, “Service-oriented sensoractuator networks: Promises, challenges, and the road ahead,” *Computer Communications*, vol. 30, no. 13, pp. 2627 – 2648, 2007.

[11] P. Newman and G. Kotonya, “A Runtime Resource-aware Architecture for Service-oriented Embedded Systems,” in *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, aug. 2012, pp. 61 –70.

[12] H. Kopetz and N. Suri, “Compositional design of RT systems: a conceptual basis for specification of linking interfaces,” in *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, may 2003, pp. 51 – 60.

[13] R. Obermaisser and H. Kopetz, Eds., *GENESYS – An ARTEMIS Cross-Domain Reference Architecture for Embedded Systems*, 1st ed. Südwestdeutscher Verlag für Hochschulschriften, 2009.

[14] S. V. Gupta, *Measurement Uncertainties – Physical Parameters and Calibration of Instruments*. Springer, 2012.

[15] M. Bantel, *Grundlagen der Messtechnik – Messunsicherheit von Messung und Messgert*. Fachbuchverlag Leipzig, 2000.

[16] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*. http://LeeSeshia.org, 2011.

[17] C. El Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek, “The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems,” in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, sept. 2012, pp. 105 –113.

[18] I. Horrocks, B. Glimm, and U. Sattler, “Hybrid Logics and Ontology Languages,” *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 6, pp. 3 – 14, 2007.

[19] T. Lukasiewicz, “Expressive probabilistic description logics,” *Artificial Intelligence*, vol. 172, no. 67, pp. 852 – 883, 2008.

[20] E. Vassev and M. Hinchey, “Towards a formal language for knowledge representation in autonomic service-component ensembles,” in *Data Mining and Intelligent Information Technology Applications (ICMiA), 2011 3rd International Conference on*, oct. 2011, pp. 228 –235.

[21] B. Motik, B. C. Grau, I. Horrocks, and U. Sattler, “Representing ontologies using description logics, description graphs, and rules,” *Artificial Intelligence*, vol. 173, no. 14, pp. 1275 – 1309, 2009.

[22] K. Ramamritham and J. Stankovic, “Scheduling algorithms and operating systems support for real-time systems,” *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55 –67, jan 1994.

[23] K. Kotecha and A. Shah, “Adaptive scheduling algorithm for real-time operating system,” in *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on*, june 2008, pp. 2109 –2112.

[24] M. Trapp, R. Adler, M. Förster, and J. Junger, “Runtime adaptation in safety-critical automotive systems,” in *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, ser. SE’07. Anaheim, CA, USA: ACTA Press, 2007, pp. 308–315.

[25] R. Adler, I. Schaefer, M. Trapp, and A. Poetzsch-Heffter, “Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 20:1–20:39, Jan. 2011.

[26] K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer, “Comparison of Reasoners for large Ontologies in the OWL 2 EL Profile,” *Semantic Web*, vol. 2, no. 2, pp. 71–87, Apr. 2011.