FORMAL-BASED FRAMEWORK FOR ANALYSIS OF LOGICAL ARCHITECTURE

Maria Spichkova, Heinrich Schmidt
School of Science, RMIT University, Melbourne, AUSTRALIA
{Maria.Spichkova, Heinz.Schmidt}@rmit.edu.au

Abstract

This paper presents a formal framework for modeling and analysis of data and control flow dependencies between components or services within remotely deployed distributed systems. This work aims at elaborating for a concrete system, which parts of the system (or system model) are necessary to check a given property. The approach allows services and components decomposition oriented towards efficient checking of system properties as well as analysis of dependencies within a system: after analysis of the logical architecture of the system, an optimized architecture is developed following the specified algorithm. One of the key features of this approach is a semi-automatic support of the presented ideas on verification level.

Keywords: formal methods; static analysis; dependencies between services; decomposition; verification

1. INTRODUCTION

Nowadays, many system engineering approaches focus on different aspects of architecture and communication modeling. Modeling of communication and data dependencies within a system as well as developing corresponding system architecture have very deep roots. The first research in this area goes back to the late sixties. However, in those days software and systems architecture of the kind used in practice and research today were still nascent. The idea of hierarchical and modular design for functions were just being formulated by Dijkstra (1969), and the idea to take some concepts from architecture and civil engineering was barely forming by Naur (1968):

> *"... software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem."*

Nowadays, many system engineering approaches focus on different aspects of architecture and communication modeling. Our aim in this paper is to discover and formalize the aspects related to remotely deployed distributed systems. This approach originated from the analysis of two case studies from automotive area, which were developed together with industrial partners within DenTUM and Verisoft-XT projects. The first case study (Feilkas et al., 2009), developing an Adaptive Cruise Control system with Pre-Crash Safety functionality, was motivated and supported by DENSO Corporation, the second case study (Spichkova, 2010), developing a Cruise Control System with focus on system architecture and verification, was supported by Robert Bosch GmbH. A sample-property proven for the Cruise Control System was like follows:

If the driver pushes the Accelerate-button while the system is on and none of the switch-off constraints occurs, the system must accelerate the vehicle during the next time unit respectively to the current speed of the vehicle and the predefined acceleration schema.

Thus, a system must analyze the information from sensors to check whether any switch-off constraints occur, i.e. if the battery voltage is too low or if the gas pedal sensor fails. On the logical level, we model a system having the corresponding input channels, e.g., an input channel *requestAccelerate* to obtain an information whether the driver have pushed the Accelerate-button. Then, the property becomes a relation between the values of the specified input and output channels. On this basis, we can analyze which set of (sub)components of the system is sufficient to check this relation.

One of the essential questions we have investigated during this analysis was which part of the system functionality do we need to analyze to check a certain property in sense of monitoring, testing or formal verification. In most cases, we don't need the complete information about the system as a whole to analyze certain aspects or to check certain properties.

The problem is how to find which information (at the level of one or more components or subsystems) is sufficient for this check, and how to solve this problem on a formal level to allow the automatization of the analysis. The reason for this question has a very practical ground: any additional information about the system can make the whole process slower, more expensive or even infeasible, especially if we are speaking about verification. In the case of model checking this can lead to the state explosion problem. In the case of theorem proving this can lead to stack overflow or excessive or prohibitive amounts of time needed by verification engineers to complete the task.

On the logical level, this problem can be reformulated as follows: *What is the minimal part of model required to check a specific property?* We suggest an approach focusing on data and control flow dependencies between services.

Dependencies' analysis results in a decomposition that gives rise to a logical system architecture, which is the most appropriate for the case of remote monitoring, testing and/or verification. In the case where the check of properties depends on information in one location, obviously we can check the property without remote connection and without sending system-wide data for inclusion in the monitoring, testing, or verification process.

In the case of remote connection, an additional problem arises through the need of transferring large amounts of data, for example from sensor instruments or automated test generators and model checkers. Thus, it is crucial to analyze regarding to efficiency, which properties (or their parts) should be checked locally, and which should be sent to the cloud. Therefore, an appropriate model covering all these aspects of remotely deployed distributed systems is needed to fulfill the corresponding constraints and to make decisions on (logical) service-oriented system architecture at least semi-automatically. Our aim in this paper is to discover and formalize the aspects related to remote processes also in relation to formal analysis of system behavior using high-performance computing and cloud virtual machines. Both clouds and grids aim to reduce costs and increase flexibility and reliability of computing, monitoring, testing, and verification by using third-party operated resources, which are allocated remotely. Grid systems, cf. e.g. (Foster et al., 2002), enable the sharing of geographically distributed, autonomous resources with the advantage of solving large-scale problems to support collaborative science, engineering and business applications. Clouds, cf. (Vaquero et al., 2009; Buyya et al., 2008), by comparison to grids, provide the required resources using virtualized infrastructure, on demand, and in a way that leverages the concentration of cloud infrastructure across many third-party cloud users/clients by smoothing peaks in load and valleys of underutilization across these usages.

Contributions: In this paper, we present a formal approach that allows modeling of data and control flow dependencies between services. The aim of the approach is to allow decomposition oriented towards efficient checking of system properties, also taking into account such characteristics as performance, worst-case execution time, reliability, etc.

Another contribution of this approach is a semi-automatic support of these ideas on verification level. Applying ideas introduced in this paper within specification and proof methodology (Spichkova, 2013b), and taking into account human factor analysis within formal modeling and verification (Spichkova, 2013a; Spichkova, 2012), we obtain concise and at the same time readable representation of the data and control flow dependencies.

To support this approach on verification level, we have built the corresponding set of theories in Higher-Order Logic. It allows checking dependency relations using an interactive theorem prover Isabelle/HOL (Nipkow et al., 2002). To discharge proof goals automatically, we apply Isabelle's component Sledgehammer (Blanchette et al., 2011) that employs resolution based first-order automatic theorem provers (ATPs) and satisfiability modulo theories (SMT) solvers.

A shorter version of this paper, cf. (Spichkova, Schmidt, 2014), was presented at the 2014 Asia-Pacific Services Computing Conference in Fuzhou, China. In our current work we extend and generalize the approach to allow the formal analysis of dependencies between services and also components within a system (or within a system of systems).

Outline: This paper is structured as follows. In Section 2 we introduce a formal theory, which allows to reason about dependencies between components. In Section 3 we discuss how to use this formalization for tracing purposes and reliability analysis. In Section 4 we present the main ideas of system architecture optimization, based on the provided formal model and the algorithms for identification of strongly connected components. Then, in Sections 5 and 6, we introduce solutions for efficient checking of properties, and present an extension of our formalization with aspects of remote computation. In Section 7 we describe the support of our approach on verification level. In Section 8 we discuss our approach and provide an overview of related work. Finally, Section 9 concludes the paper with a summary of what has been achieved.

2. INTER-/INTRASERVICE DEPENDENCIES

In this paper, we follow the definition of services introduced in (Broy, 2005), where a service/component *S* is defined as a partial function from input streams *I(S)* to output streams *O(S)*. Thus, the behavior of a service can be defined partially (only for a subset of its possible inputs), in contrast to the totality of a component interface behavior. In our approach, we need only an abstract definition of a service: we focus on the *analysis of dependencies* within and between services/components, where the behavior of a (sub)services/(sub)components can be specified using different languages.

First of all, we specify the type *CSet* of services/components and the type *chanID* of the system channels. The set of the services/components (more precisely, over their identifiers) at the abstraction level *L* (i.e. level of refinement/decomposition) is denoted by $CSet^L$ *set*. The set of the system channels is defined to be the unchanged for all abstraction levels. We also need to define the set *varID* of the local variables that are important for modeling of the dependencies between data and control flows (cf. Section 2.2 for an example).

As a *stream* we understand here both data and control flows. On the level of logical architecture, we can see a

stream as an abstract *channel* with the name of the type *chanID*. The sets of the input and output streams' identifiers, *I(S)* and *O(S)*, for a service/component *S*, are defined as the corresponding functions over the set of the services/components identifiers:

$I: CSet \rightarrow chanID\ set$

$O: CSet \rightarrow chanID\ set$

The function *Var(S)* specifies for a service/component *S* the corresponding set of the local variables:

$Var: CSet \rightarrow varID\ set$

To represent the (de)composition relations between components, we specify a function *SubComp* over the set of the services/components identifiers:

$SubComp: CSet \rightarrow CSet\ set$

Lemma 1: If any two components/services are related via *SubComp*, they cannot belong to the same abstraction level, i.e. for all abstraction levels and all components should hold that a component and its subcomponents should be defined on different abstraction levels:

$\forall\ S\ .\ \forall\ C \in SubComp(S).\ \forall\ \mathrm{L}\ .$
$S\ \in CSet^L \Rightarrow C \notin CSet^L$

Lemma 2: If a local variable belongs to a subcomponent/subservice of a component/service *S*, then this variable should be also defined as a local variable of *S*, i.e., for all abstraction levels and all components should hold that if a local variable belongs to one of the subcomponents, it also belongs to the composed component:

$\forall\ S\ .\ \forall\ x \in Var(C).\ \forall\ \mathrm{L}\ .$
$C \in SubComp(S) \Rightarrow x\ \in Var(S)$

Lemma 3: For all abstraction levels and all components should hold that after correct decomposition of a component each of its local variables could belong only to one of its subcomponents:

$\forall\ S\ .\ \forall\ x \in Var(S).\ \forall\ C_1, C_2 \in SubComp(S)\ .$
$x\ \in Var(C_1) \wedge x\ \in Var(C_2)\ \Rightarrow C_1 = C_2$

Lemma 4: For all abstraction levels and all components should hold that if a component does not have any local variables, none of its subcomponents has any local variables:

$\forall\ S\ .\ Var(S) = \varnothing \Rightarrow \forall\ C \in SubComp(S)\ .\ Var(C) = \varnothing$

Lemma 5: For all abstraction levels and all components should hold that if C is a subcomponent of a component *S* on some abstraction level *L*, it couldn't be a subcomponent of another component on the same level *L*:

$\forall\ \mathrm{C}, S, L\ .\ C \in SubComp(S) \wedge S\ \in CSet^L.$
$\forall\ X\ \in CSet^L\ .\ X \neq S \Rightarrow C\ \notin SubComp(X)$

To keep the focus on the direct (de)composition relations between components/services, we do not allow the transitivity property for the *SubComp* function.

While modeling communication between services/components on the level *L*, we specify the dependencies by the function

$Sources^L: CSet^L \rightarrow CSet^L\ set$

This function returns for any service *A* the corresponding (possibly empty) set of services that are the sources for the input streams of *A*. More precisely, all the dependencies can be divided into the direct and indirect ones.

Similar to the function $Sources^L$, we define direct dependencies by the function

$DSources^L: CSet^L \rightarrow CSet^L\ set$

For example, $C_1\ \in DSources^L(C_2)$ means that at least one of the output channels of C_1 is directly connected to some of input channels of C_2. In more complicated situations, we need to reason about the (transient) dependencies within many or all of the services/components of the system.

The direct sources for a service *C* can be defined as follows:

$DSources^L(C) = \{\ S \mid \exists\ x \in\ I(C) \wedge x \in O(S)\}$

If we do not have to take into account the indirect dependencies, the function $Sources^L$ is defined to be equal to the function $DSources^L$, otherwise we define this function recursively over the set of services using the following fixed-point construction:

$Sources^L(C)\ =$
$DSources^L(C) \cup \bigcup_{S\ \in\ DSourcesL(C)} \{\ S_1 \mid S_1\ \in Sources^L(S)\}$

The functions $DSources^L(C)$ and $Sources^L(C)$ have a number of crucial properties. In this paper we introduce three of them as lemmas.

Lemma 6: If a service *X* does not belong to the $DSources^L$ set of any service *C* of the system, it also does not belong to the $Sources^L$ set of any service *C*:

$\forall\ C \in CSet^L.\ X\ \notin DSources^L(C) \Rightarrow$
$\forall\ C \in CSet^L.\ X\ \notin Sources^L(C)$

Thus, if *C* belongs to the $Sources^L$ set of a service *S*, then exist some service *Z*, such that *C* belongs to its $DSources^L$ set (as a special case, *Z* could also be equal to *S*):

$C \in Sources^L(S) \Rightarrow \exists\ Z.\ C\ \in Sources^L(Z)$

This also imply that if $Sources^L$ set of a service *C* is empty, its $DSources^L$ set is empty too, and vice versa:

$DSources^L(C) = \varnothing \Leftrightarrow DSources^L(C) = \varnothing$

Lemma 7 (Transitivity): If a service *C* belongs to the $Sources^L$ set of a service *S*, which itself belongs to the $Sources^L$ set of another service *Z*, then the service *C* also belongs to the $Sources^L$ set of a service *Z*.

$C \in Sources^L(S) \wedge S \in Sources^L(Z) \Rightarrow C\ \in Sources^L(Z)$

Lemma 8: If we have mutual dependencies between services, then their $Sources^L$ sets are equal and contain these services themselves (*XS* and *ZS* denote here some sets over $CSet^L$):

$Sources^L(C) = (XS \cup Sources^L(S)) \wedge$
$Sources^L(S) = (ZS \cup Sources^L(C))$
$\Rightarrow Sources^L(C) = Sources^L(S) = XS \cup ZS \cup \{\ C, S\ \}$

In general, values of an output channel $y \in O(C)$ of a service *C* do not necessarily depend on the values of all its input streams. This means that an optimization of system/ services' architecture may be needed in order to localize these dependencies. To express any restrictions we use the following notation: $I^D(C, y)$ denotes the subset of $I(C)$ that *y* depends upon.

There are three possible cases to consider:

1. *y* depends on all input streams of *C*:
   $I^D(C, y) = I(C)$

2. *y* depends on some input streams of *C*:
   $I^D(C, y) \subset I(C)$

3. *y* is independent of any input stream of *C*, i.e.,
   $I^D(C, y) = \varnothing \neq I(C)$

While determining these sets, we should take into account not only the direct dependencies between input/output values, but also the dependencies via local variables of the service. For example, let *C* be a service with a local variable *st*, representing its current state, and an output channel *y*, which depends only on the value of *st*. If *st* is updated depending to the input messages the service receives via the input channel *x*, then $x \in I^D(C, y)$. To be more precise, we mark this special case by an upper index of *x*, i.e., $x^{st} \in I^D(C, y)$ indicating that *x* influences the values of the variable *st*, and the channel values *y* via the values of the variable *st*.

2.1 Elementary Services

Based on the definitions above, we can decompose services to have for each output channel the minimal subservice/component, which computes the corresponding values for this channel (we call them *elementary services/components*).

A service/component is called to be *elementary*, if

- it has a single output channel (in this case this service can have no local variables), or
- all its output channels are correlated, i.e. mutually depend on the same local variable(s).

Thus, we cannot simply remove any part of an elementary service/component to a separate service/ component without introducing a new local channels.

Let *C* be a service with *m* input channels $x_1, \ldots, x_m$ and *n* output channels $y_1, \ldots, y_n$ as well as *k* local variables $l_1, \ldots, l_k$. For each output channel y_i, $1 \le i \le n$ we check the corresponding set $I^D(C, y_i)$ using the following constraints:

(1) If $I^D(C, y_i)$ contains only direct dependencies from some input channels, i.e. $I^D(C, y_i) \subseteq I(C)$, we remove the corresponding computations from *C* to a single subservice C_i, which has a single output channel y_i.

(2) If $I^D(C, y_i)$ contains not only direct dependencies, but also dependencies via local variables, we should check whether there are other output channels depending on these variables, because the output channels depending on the same local variables will belong to the same subservice/ subcomponent of *C*, otherwise we will need to repeat the computation for these variables or to have additional local channels to share the computed values between the components.

Thus, we have two cases to consider:

(2a) If y_i is the only output channel depending on these variables, we remove the corresponding computations from *C* to a single subservice C_i, which has a single output channel y_i.

(2b) If there are other *jm* output channels $y_{j1}, \ldots, y_{jm}$ depending on these variables, we remove the corresponding computations from *C* to a single subservice C_i, which has *jm+1* output channels $y_i, y_{j1}, \ldots, y_{jm}$.

$I^D(C, y_i)$ becomes a set of input channels of the new subservice C_i.

After this decomposition, we will have a (flat) system architecture, where each component/service is elementary.

Lemma 9: If we do not apply any additional decomposition strategies, then all components/services on the abstraction level *L1* do not have any subcomponents/ subservices: $\forall\, C \in CSet^{L1}.\ Var(C) = \varnothing$

2.2 Running Example

In this section we illustrate the presented ideas by an example. First of all, we show how each service can be decomposed to optimize the dependencies within each single service, and after that we optimize the architecture of the whole system.

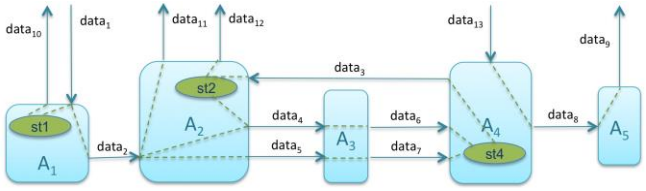

Figure 1. System S: Dependencies and I^D sets

Given a system *S* (cf. also Figure 1) consisting of five services, where the set *CSet* on the level *L0* is defined by $\{A_1, \ldots, A_5\}$. The set of all element of the chanID type is defined by $\{ data_1, \ldots, data_{13}\}$. The ranges of the *I*, *O* and *Var* functions for the level *L0* (i.e. for the $CSet^{L0}$ as the domain of these functions) can be specified as follows:

$I(A_1) = \{ data_1\}$
$I(A_2) = \{ data_2,\ data_3\}$
$I(A_3) = \{ data_4,\ data_5\}$
$I(A_4) = \{ data_6,\ data_7,\ data_{13}\}$
$I(A_5) = \{ data_8\}$

$O(A_1) = \{ data_2, data_{10}\}$
$O(A_2) = \{ data_4, data_5, data_{11}, data_{12}\}$
$O(A_3) = \{ data_6,\ data_7\}$
$O(A_4) = \{ data_3,\ data_8\}$
$O(A_4) = \{ data_9\}$

$Var(A_1) = \{ st1 \}$
$Var(A_2) = \{ st2 \}$
$Var(A_3) = \varnothing$
$Var(A_4) = \{ st4 \}$
$Var(A_5) = \varnothing$

Table 1. Dependencies within the system *S*

	$DSources^{L0}$	$Sources^{L0}$	$\mathbb{I}^{\mathcal{D}}$
A_1	$\varnothing$	$\varnothing$	$data_2 : \{data_1\}$ $data_{10} : \{data_1^{st1}\}$
A_2	$\{A_1, A_4\}$	$\{A_1, A_2, A_3, A_4\}$	$data_4 : \{data_2, data_3^{st2}\}$ $data_5 : \{data_2\}$ $data_{11} : \{data_2\}$ $data_{12} : \{data_3^{st2}\}$
A_3	$\{A_2\}$	$\{A_1, A_2, A_3, A_4\}$	$data_6 : \{data_4\}$ $data_7 : \{data_5\}$
A_4	$\{A_3\}$	$\{A_1, A_2, A_3, A_4\}$	$data_3 : \{data_6^{st4}, data_7^{st4}\}$ $data_8 : \{data_{13}\}$
A_5	$\{A_4\}$	$\{A_1, A_2, A_3, A_4\}$	$data_9 : \{data_8\}$

The sets I^D of data and control flow dependencies between the services are shown in Table 1. We represent the dependencies graphically using dashed lines over the service box. Now we can decompose the system's services according to the given I^D specification. The result of the decomposition will be a logical architecture of the system on the abstraction level *L1* (cf. Figure 2), where all services are elementary and have no subservices:

A_1 has two output channels, $data_2$ depending on the input channel $data_1$ directly and $data_{10}$ depending on the same input channel but via a local variable *st1*. Therefore, it should be decomposed in two subcomponents: A_{11} with input $data_1$ and output $data_2$, and A_{12} with input $data_1$ and output $data_{10}$. Thus,

$SubComp(A_1) = \{ A_{11}, A_{12}\}$.
$I(A_{11}) = I(A_{12}) = \{ data_1\}$
$O(A_{11}) = \{ data_2\}$ $O(A_{12}) = \{ data_{10}\}$
$Var(A_{11}) = \varnothing$ $Var(A_{12}) = \{ st1 \}$

A_2 has four output channels: $data_4$, $data_5$, $data_{11}$, $data_{12}$. $data_5$ and $data_{11}$ both depend on the input channel $data_2$ directly, the corresponding computations should be removed to two separate subservices. $data_{12}$ depends on the input channel $data_3$ via local variable *st2*, but there is another output channel $data_4$ depending on this local variable, therefore, computations for these two outputs should belong to the same subservice. Thus, A_2 should be decomposed in three subservices:

$SubComp(A_2) = \{ A_{21}, A_{22}, A_{23}\}$.
$I(A_{21}) = I(A_{23}) = \{ data_2\}$ $I(A_{22}) = \{ data_2 , data_3 \}$
$O(A_{21}) = \{ data_{11}\}$ $O(A_{22}) = \{data_4, data_{12}\}$
$O(A_{23}) = \{ data_5\}$
$Var(A_{21}) = Var(A_{23}) = \varnothing$ $Var(A_{22}) = \{ st2 \}$

A_3 has two output channels: $data_6$, $data_7$. The channels $data_6$ and depends on the input channel $data_4$, the channel $data_7$ depends on the input channel $data_5$. Therefore, A_3 should be decomposed in two subservices:

$SubComp(A_3) = \{ A_{31}, A_{32}\}$.
$I(A_{31}) = \{ data_4\}$ $I(A_{32}) = \{ data_5\}$
$O(A_{31}) = \{ data_6\}$ $O(A_{32}) = \{ data_7\}$

$Var(A_{31}) = Var(A_{32}) = \varnothing$

A_4 has two output channels, $data_8$ and $data_3$. It should be decomposed in two subservices, because $data_8$ depends on the input channel $data_{13}$ directly and $data_3$ depends on the input channels $data_6$ and $data_7$ via local variable *st4*. Thus,

$SubComp(A_4) = \{ A_{41}, A_{42}\}$.

$I(A_{41}) = \{ data_6, data_7\}$ $I(A_{42}) = \{ data_{13}\}$

$O(A_{41}) = \{ data_3\}$ $O(A_{42}) = \{ data_8\}$

$Var(A_{31}) = \{$ st4 $\}$ $Var(A_{42}) = \varnothing$

A_5 has only one output channel, which means that no decomposition on this level is required; this service is already an elementary one.

We obtain the following set of the services/components at the abstraction level *L1:*

$CSet^{L1} = \{ A_{11}, A_{12}, A_{21}, A_{22}, A_{23}, A_{31}, A_{32}, A_{41}, A_{42}, A_5\}$

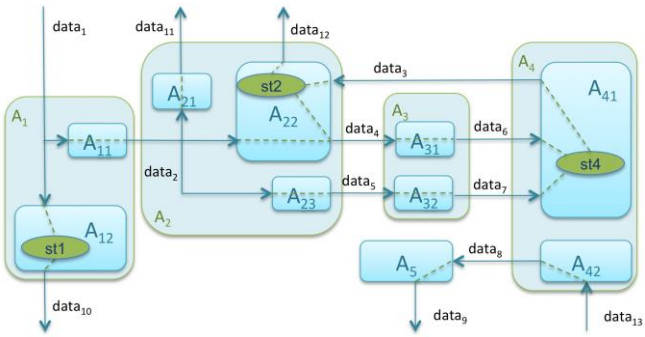

Figure 2. Services' decomposition (level L1)

3. RELIABILITY ANALYSIS AND TRACING

The functions $Sources^L$ and I^D allow us to trace back which parts of the system provide information to a certain service. This provides a basis for identifying of elementary services and the corresponding optimization of the logical architecture. To trace which services are affected by the results produced by the service, we introduce another two functions, O^D and Acc^L. For any service *C*, the function O^D (dual to I^D) returns the corresponding set $O^D(C, x)$ of output channels depending on input *x*. This helps us to solve following problems:

- if there are some changes in the specification, properties, constraints, etc. for *x*, we can trace which other channels can be affected by these changes;
- having for each output channel the minimal subservice computing the corresponding results, we can check the critical paths in the system by allocating to each subservice /output channel the corresponding worst case execution time (WCET). For this purpose we can use representation of the system architecture as a directed graph (cf. Section 4).

If the input part of the service's interface is specified correctly in the sense that the service does not have any "unused" input channels, the following relation will hold:

$\forall\, x \in I(C).\ O^D(C, x) \neq \varnothing$

On each abstraction level L of logical architecture, we can define a function

$Acc^L: CSet^L \rightarrow (CSet^L\ set)$

For any service A the function Acc^L returns the corresponding set of services (names) $B_1, \ldots, B_{AN}$ that are the acceptors for the output streams of A. This set can be empty. The function Acc^L is dual to the function $Sources^L$:

$x \in Acc^L(y) \Leftrightarrow\ y \in Sources^L(x)$

This model allows us to analyze the influence of a service's failure on the functionality of the overall system: if a service *C* fails, the set of affected function will be $Acc^L(C)$. For example, for the system from Figure 2, on the abstraction level *L1* the services A_{12}, A_{21}, A_5, have no acceptors, where the set of acceptors of the service A_{11} is

$\{ A_{21}, A_{22}, A_{23}, A_{31}, A_{32}, A_{41} \}$.

Thus, the failure of the service A_{12} causes wrong behavior only of A_{12} itself, where the failure of the service A_{11} has influence on the results of seven services, including the service A_{11}. On this basis, we can mark each service *C* by a number of services, which are affected by *C* (and, respectively, by its failure). Thus, the *impact number* of the service A_{12} will be 1 (the minimal value), where the impact number of A_{11} will be 7.

Moreover, on this basis we can apply results of our previous work on efficient hazard and impact analysis for automotive mechatronics systems (Dobi, et al., 2013), which was done in collaboration with industrial partners from ITK Engineering AG. Many faults in software and system development can be reduced to the errors in manual or automated transformations of the requirements specifications to the deployed software and systems. A safety engineer needs to understand the relationship between software, electronic hardware and physical components' faults, and their consequences in terms of hazards (hazardous system failures). Dobi, et al. (2013) describe the relationship between system safety goals and component safety and reliability requirements formally. Our approach goes further and allows analysis of the correlations between hazardous system failures.

The representation of dependencies between services by a directed graph (like on Figure 3 in Section 4.2) allows also

to find the *worst case execution time* (WCET) needed for the concrete output based on the WCETs of the system's services, as well as analyze the influence of a service's failure on overall system. For example, it is easy to see from Figure 3 that for the outputs of the services A_{11} and A_{12} the worst case computation time is equal to the WCETs of these services, where for the outputs of the services A_{21} the WCET is equal to the sum of WCETs of A_{11} and A_{21}, etc.

4. Strongly Connected Components/Services

After the decomposition discussed in the previous section, we obtain a (flat) architecture of system. The main feature of this architecture is that each output channel (within the system) belongs the minimal subservice of a system computing the corresponding results.

We represent this (flat) architecture as a directed graph and apply one of the existing distributed algorithms for the decomposition into its *strongly connected components and services*, e.g. FB (Fleischer et al. 2000), OBF (Barnat et al, 2008), or the coloring algorithm (Orzan, 2004). For our goals, we extend them by a preliminary simplification of the graph (cf. below). This optimization is algorithm independent and is also applicable for the case another decomposition is chosen.

Let us introduce some basic terms and definitions to explain the main ideas of the decomposition we apply in our approach.

A *directed graph G* is a pair *(V, E)*, where *V* is a set of vertices, and $E \subseteq V \times V$ is a set of directed edges.

If $(u,v) \in E$, then *v* is called a *successor* of *u*, and *u* is called a *predecessor* of *v*.

A vertex *t* is called *reachable* from vertex *x* if $(x,t) \in E^*$, where E^* is a transitive and reflexive closure of *E*. If x_k is reachable from x_0, then there is a *path* (sequence of vertices) $x_0, \ldots, x_k$, such that $\forall\ 0 \le i < k.\ (x_i, x_{i+1}) \in E$.

A set of vertices $SC \subseteq V$ is *strongly connected* if for any vertices $u, v \in CS$ holds that *v* is reachable from *u*.

A *strongly connected service (SCS)* is a maximal strongly connected set of vertices, i.e. after extension of this set by any additional vertex the set is no more strongly connected.

An SCS is *trivial* if it consists of a single vertex. We call an SCS *leading (terminating) trivial* if it consists of a single vertex that has no predecessors (successors).

In our representation of a flat architecture as a directed graph, services become vertices and channels become edges. We distinguish here

- channels that are system's input/output; within the graph, we represent them by dashed edges, cf. also Figure 3, because they are less important on this level of (de)composition – we do not need to take them into account by identifying the SCSs;
- channels that are local for the system, i.e. representing dependencies between services.

Vertices that are leading trivial SCSs are labeled by LT, vertices that are terminating trivial SCSs are labeled by TT, and vertices that are *commonly trivial* SCSs (trivial, but neither leading nor terminating) are simply labeled by T.

In addition, we define another special king of trivial SCSs: services that has neither successors, no predecessors; this means that all their input/output channels are system's input/output channels. We call them *disconnected trivial* SCSs and label by DT.

Two vertices *u* and *v* of an undirected graph *G* are connected, if there is a path from *u* to *v* (for the case of directed graph we need to ignore the orientation of its edges); otherwise, they are *disconnected*. A graph *G* is *connected* if every pair of vertices in *G* is connected; otherwise, it is called *disconnected*.

If the graph *GL1* representing a flat system's architecture is *disconnected*, i.e. consists of *N* graphs that are either connected graphs or single vertices (we denote this set by *GN*), then we identify SCSs in each of these graphs separately and could also parallelize these computations (cf. also example below).

4.1 Service/Components Decomposition

In this section we present the algorithm for the decomposition of services/components into the sets of strongly connected services/components.

Step 1: Construct the set *GN* from the system's architecture on level *L1*. Denote the graph representing a flat system's architecture by *GL1*. In the case *GL1* is *connected*, *GN = { GL1 }*, otherwise *GN* will have more than one elements.

Repeat steps 2-4 until the set *GN* is empty:

Step 2: Chose one element (graph) of the set *GN*, denote it *X*, delete it from *GN*. Identify DT services in *X* and delete these vertices from the graph. This is non-recursive procedure with time complexity *O(n)*, where *n* is a number of vertices.

Step 3: Apply the recursive elimination technique *One-Way-Catch-Them-Young* (OWCTY), to remove the LT services and the reversed OWCTY technique to remove the TT services, cf. (Fisler et al., 2001) for the description of these techniques.

Step 4: The removed services become single services on the level *L2*. To the rest of the graph *X*, we apply one of the existing algorithms for the decomposition into SCCs, e.g., FB (Fleischer et al. 2000), OBF (Barnat et al, 2008), etc.

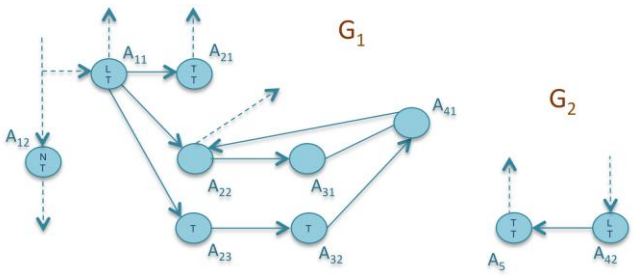

Figure 3. System S (level L1): Detection of the strongly connected services

4.2 Example of the Decomposition

To illustrate the ideas introduced in Section 4.1, we continue with the example from Section 3. We represent the system shown on Figure 2 as a directed graph to apply a decomposition algorithm, cf. Figure 3. This graph is disconnected and consists of two connected graphs, G_1 and G_2, therefore we can analyze them in parallel.

Step 1: $GN = \{ G_1, G_2 \}$.

For G_1 we need to perform the following steps:

Step 2: We identify a single DT service in G_1, namely, A_{12}. We delete it from the graph, and it becomes system's subservice S_1 on the level *L2*.

Step 3: We apply the OWCTY elimination technique to identify and remove the LT and TT services:

- On the first run of the algorithm, we identify A_{11} and A_{21} as LT and TT services respectively and delete them. They become S_2 and S_3 on the level *L2*.
- After this, A_{23}, which is commonly trivial in G_1, becomes an LT service and should be deleted, becoming S_4 on the level *L2*. Then, the same situation will be with A_{32}: it should be deleted as an LT service and becomes S_5 on the level *L2*.
- The rest of the graph (vertices A_{22}, A_{31}, A_{41}) with corresponding edges contains neither LT nor TT vertices. This is an input for the SCC-decomposition algorithm.

Step 4: In this example, we apply a variant of the FB algorithm. The time complexity of this algorithm is $O(n*(n+m))$, where n is a number of vertices and m is the number of edges in this graph. The FB algorithm identifies that the vertices A_{22}, A_{31}, A_{41} build a single SCS, which becomes S_6 on the level *L2*.

For G_2 we need to perform the following steps:

Step 2: G_2 has only two vertices; none of them can be identified as a DT service.

Step 3: We apply the OWCTY elimination technique in its simplest version, because no recursion is needed in this case: the vertex A_{42} is identified as a leading trivial SCS, and the vertex A_5 is identified as a terminating trivial SCS.

Step 4: On the level L_2 we have two subservices of the system: S_7 and S_8.

Figure 4 presents the result of the architecture optimization.

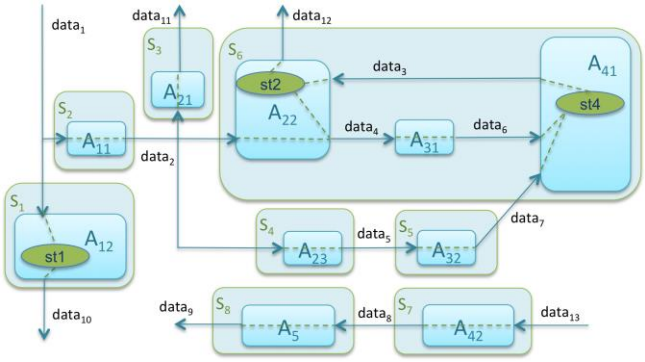

Figure 4. Architecture of S (level L2)

5. Efficient Checking of Properties

A property can be represented by relations over data and control flows on the system's channels. Let for a relation r, I_r and O_r be the sets of input and output channels of the system used in this relation. For each channel from O_r we recursively compute all the sets of the dependent input channels. Their union, restricted to the input channels of the system (for the case we exclude properties' specification over local channels), I^D_r should be equal to I_r, otherwise we should check whether the property was specified correctly (i.e. exclude (human) error in the specification of the property, e.g., when a wrong channel identifier was used) and if so, precise it or eliminate unnecessary constraints:

(1) If a channel x belongs to I^D_r but not to I_r, we need to extend r explicitly specifying that this property should hold for any values on x.

(2) If a channel x belongs to I_r but not to I^D_r, this means that r is unnecessary strict, because it contains assumptions on an irrelevant input channel.

From O_r we obtain the set $OutComp_r$ of services having these channels as outputs, compute the union of corresponding sets $Sources^{L2}$. This union together with $OutComp_r$ gives us the minimal part of the system needed to check the property r. This allowed us, especially in the case of cloud-supported processing, to reduce the costs of monitoring, testing or verification.

Assume we have to check relation $R1(data_{10}, data_{13})$ specified for the system presented on Figure 4. It is easy to

see, that there is no dependency between $data_{13}$ and $data_{10}$. We have here I_{R1} = { $data_{13}$ } and O_{R1} = { $data_{10}$ }. $data_{10}$ is a single output channel of S_1, i.e. $O(S_1)$ = { $data_{10}$ }. This gives us $OutComp_{R1} = S_1$.

Thus, $I^D(S_1, data_{10})$ = { $data_1$ } which is already equal to I^D_{R1}, because $data_1$ is a system input.

First of all we need to exclude error in the specification of the property, e.g., the case where *R1* is meant to be specified over $data_{10}$ and $data_1$. If *R1* was specified without such errors, we need

(1) to extend it explicitly specifying that it should hold for any values on $data_1$, and

(2) eliminate unnecessary constraints on $data_{13}$ from *R1*.

Because $Sources^{L2}(S_1) = \emptyset$, we need only the service S_1 to check this property.

Assume now we have to check relation $R2(data_1, data_{12})$ specified for the same system. For this case, I_{R2} = { $data_1$ } and O_{R2} = { $data_{12}$ }.

$data_{12}$ is a single output channel of S_6, and therefore $OutComp_{R2} = S_6$.

$I^D(S_6, data_{12})$ = { $data_2$, $data_7$ }, both channels are local, $Sources^{L2}(S_6)$ = { S_2, S_5 }.

$data_2 \in O(S_2)$, which gives us $I^D(S_2, data_2)$ = { $data_1$ }; this is a system input channel, therefore it should be a part of I^D_{R2}. $Sources^{L2}(S_2) = \emptyset$.

$data_7 \in O(S_5)$, $I^D(S_5, data_7)$ = { $data_5$ } (also local), $Sources^{L2}(S_5)$ = { S_4 }. \\

$data_5 \in O(S_4)$, $I^D(S_4, data_5)$ = { $data_2$ } (local, cf. above), $Sources^{L2}(S_4) = \emptyset$.

I^D_{R2} = { $data_1$ } = I_{R2}, thus, there is no need to extend the property or eliminate some to its constraints. To check this property we need the following set of services:

$OutComp_{R2} \cup Sources^{L2}(S_6) \cup Sources^{L2}(S_2) \cup Sources^{L2}(S_5) \cup Sources^{L2}(S_4)$ = { S_2, S_4, S_5, S_6 }.

6. Worst Case Execution Times

Among other factors, the purposed representation gives a basis for a straightforward analysis of the worst case execution time (WCET) for the producing of the selected outputs. To calculate WCET required to produce an output *d* (let us denote it $WCET_d$), we recursively compute for *d* all the sets of the required services. The sum of WCETs of the services gives us the WCET for the producing of *d*. More precisely, the algorithm can be defined as follows:

Step 1: Identify the service having d as its output. Let us denote this service as S_d.

Step 2. Based on the definitions from Section 2, calculate sources of the service S_d as $Sources^{L2}(S_d)$.

Step 3: Define the set of all services required to produce the output *d* as $RequiredServices(d) = \{S_d\} \cup Sources^{L2}(S_d)$.

Step 4: Calculate $WCET_d$ as a sum of the WCETs of the required services:

$$WCET_d = \sum_{x \in RequiredServices(d)} WCET(x)$$

Let us present the calculations on the basis of our running example:

$RequiredServices(data_9) = \{S_7, S_8\}$,
$RequiredServices(data_{10}) = \{S_1\}$,
$RequiredServices(data_{11}) = \{S_2, S_3\}$, and
$RequiredServices(data_{12}) = \{S_2, S_4, S_5, S_6\}$.

Respectively,

$WCET_{data9} = WCET(S_7) + WCET(S_8)$,
$WCET_{data10} = WCET(S_1)$,
$WCET_{data11} = WCET(S_2) + WCET(S_3)$,
$WCET_{data12} = WCET(S_2) + WCET(S_4) + WCET(S_5) + WCET(S_6)$.

7. Remote Computation

In the previous section we have analyzed, which part of the overall information about the system and its input data is necessary to check the corresponding property. In the similar way, we can trace the influence of the system environment's properties on the properties of the system itself. However, the decision, which information need to be processed locally (local services) and which need to be sent to the cloud (remote services), should be made not only according to this analysis of a (logical) system architecture, but also taking into account the following aspects of data flows between services:

(1) measure for costs of the data transfer/ upload to the cloud *UplSize(f)*: size of messages (data packages) within a data flow *f* and frequency they are produced. This measure can be defined on the level of logical modeling, where we already know the general type of the data and can also analyze the corresponding service (or environment) model to estimate the frequency the data are produced;

(2) measure for requirement of using high-performance computing and cloud virtual machines, *Perf(X)*: complexity of the computation within a service *X*, which can be estimated on the level of logical modeling as well.

On this basis, we build system architecture, optimized for remote computation. We associate an *UplSize* measure to each channel (to each data flow), and a *Perf* measure to each elementary service in the system (i.e. for any service on the abstraction level *L1*).

HighPerf denotes a limit, above which a remote computation is desired. A limit of the *UplSize* measure, above which a limited up-/download is desired, is denoted by *HighLoad*, i.e. if the source of the data is deployed locally then the receiving services should preferably also deployed locally, and if the source is deployed in the cloud then the receiving services should preferably also deployed in the cloud.

We do not take into account costs of the transfer of software service themselves, assuming that this aspect is

already covered by the *Perf* measure. The *UplSize* measure should be analyzed only for the channels that aren't local for the services on abstraction level *L2*.

Using graphical representation, we denote channels with *UplSize* >> *HighLoad* by thick red arrows, and the services with *Perf* >> *HighPerf* by **white** color, where all other channels and services are marked blue. We use the same coloring notation when building the corresponding tables for the values above limits as well as when represented a system by the corresponding directed graph.

For the compressed table representation, which allows readable representation of the large systems' measures, we omit the concrete values of the measures leaving only the notes on relations to the defined limits.

While measuring *Perf* on the level *L1*, we mark a service by a sum of *Perf* measures of its subservices on the level *L2*. Thus, if the *Perf* measure of at least one subservice is high, the same hold for the service's *Perf*.

This analysis can also be done automatically, provided all the measures *UplSize* and *Perf* are defined for data and control flows as well as for (elementary) services. Using our approach, we can prove a number of properties for service composition according to these measures. We start with the architecture from abstraction level *L2* and represent it as a directed graph. Then we simply remove edges that correspond to channels with low *UplSize*, and obtain a set of connected graphs. If any two vertices (or graphs) are *connected* by a single edge that is *split* and has no source vertex (this represents a situation where a system input goes to many system's services), we see them as a connected graph too. Each of the graphs becomes a service on the abstraction level *L3* represents system architecture, optimized for remote computation.

Assume the following case for the system S from the previous examples (cf. also Figure 4):

- channels with *UplSize* value >> *HighLoad*: $data_1$, $data_4$, $data_5$, $data_6$, $data_7$, $data_8$.
- Services with *Perf* value >> *HighPerf*: A_{22}, A_{23}, A_{41}, A_{42}, and on the abstraction level *L2*: S_2, S_4, S_6, S_7.

$data_4$ and $data_6$ are local channels of S_6, their *UplSize* measure is not relevant for this case. We represent the system from the abstraction level *L2* as a directed graph GL_2 (cf. Figure 5). After removing edges that correspond to channels with low *UplSize*, we obtain a set GL_2' of connected graphs.

Figure 6 represents system architecture, optimized for remote computation. This corresponds to the abstraction level *L3*. On this level, S_1 and S_2 are composed together into a new service S_1': $data_1$ corresponds to a data flow, where messages (data packages) have large size and come with a high frequency, but $data_2$ has very low frequency (e.g., S_2 realizes a filtering function according to a given criteria), therefore it make more sense to deploy S_1 locally, together with S_2.

There are no changes for service S_3, but S_4, S_5, and S_6 are composed into S_4', and S_7 with S_8 into S_7'.

The services S_4', and S_7' have *Perf* measure higher *HighPerf*, therefore using high-performance computing and cloud virtual machines is required for these services.

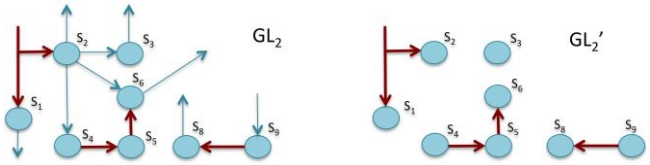

Figure 5. Service (de)composition according to the channel measures UplSize

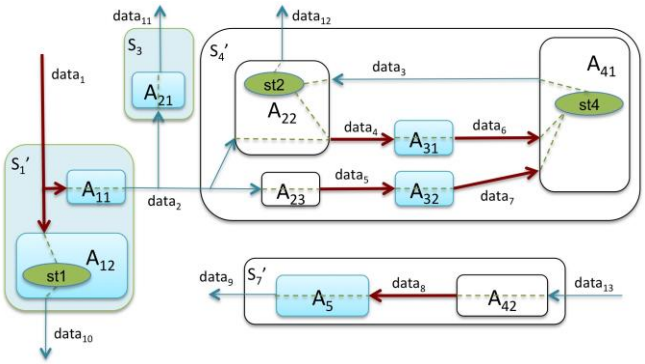

Figure 6. Optimized architecture of S (Level L3)

8. Semi-Automatic Formal Verification

To support our approach on verification level, we have also build a set of corresponding theories in Higher-Order Logic (HOL) using an interactive semi-automatic theorem prover (proof assistant) Isabelle, cf. (Nipkow et al., 2002). This proof assistant is based on polymorphic HOL extended with axiomatic type classes, and support the proof of arbitrary mathematical theorems in interactive manner. Proofs are constructed in the structured proof language Isar (Wenzel, 2013). The advantage of this proof language is that the proofs are easy readable for both human and machine, which is not the case for many proof languages.

In state-of-the-art industrial development, quality assurance is performed by extensive testing of the generated code. However, testing can only demonstrate the absence of errors for exemplary test cases, but not the correctness of the

system. In opposite to testing, formal verification delivers a correctness proof for safety critical properties but requires significant effort. Thus, only the most critical parts of a system must be verified, and the whole process of specification and verification must be set up in a way that minimizes the overall effort. On the other hand, testing and simulation must belong to the development process, because in some cases, even after verifying certain properties, inconsistencies can still remain in the specification, model or code. As nicely stated by Donald E. Knuth's famous saying:

"Beware of bugs in the above code – I have only proved it correct, not tried it."

There is a number of works on integration of different system models and verification techniques. For instance, the Ptolemy approach (Eker et al., 2003) introduces a general way to combine heterogeneous models of embedded systems. A prominent example of integration of verification techniques is a combination of Model Checking and Deduction for I/O-Automata done by Müller and Nipkow (1995).

However, to our best knowledge there are no other works on achieving a *pervasive formal development process* for embedded applications starting with informal textual specification and leading to verified code. This direction has been touched for the first time by Botaschanjan et al. (2008) though only for upper layer of automotive systems and focused on later verification phases. The first steps towards a methodology for development of verified embedded system have been done in (Botaschanjan et al., 2005; Botaschanjan et al., 2006). For example, a typical setting found in the automotive domain, a time-triggered operating and communication bus system, has been verified (Spichkova, 2006; Kühnel, Spichkova, 2006; Kühnel, Spichkova, 2007).

The work presented in (Spichkova et al., 2012) covered the entire seamless pervasive development process. In comparison to the problem frame approach of Jackson (2001) as well as the 4-variable model of Parnas and Madey (1995), the authors have presented a pervasive formal development methodology for embedded systems starting from an informal textual specification of the requirements and going all the way to verified application code.

The results of our current work can be seen as an extension of the specification and analysis parts of the development methodology presented in (Spichkova et al., 2012), where

- verification was performed by using the Isabelle theorem prover;
- requirements and architecture specifications were represented using FOCUS (Broy, Stølen, 2001), a stream-based formal specification language for development of distributed interactive systems; its basics are compatible with our formalization.

Case studies: The representation of our approach in Isabelle/HOL contains approx. 70 general axioms and lemmas, cf. (Spichkova, 2014b). These axioms and lemmas are necessary to analyze in Isabelle the dependencies within a system in a (semi-)automatic way. This allows us, e.g., to verify whether a given set of services is equal to the set of (in)direct sources of some service, or whether the set of services is equal to the minimal set needed to check a certain property.

To show a feasibility of the approach, we also have done a case study using our formalization. In this case study, we started with 9 services connected with 24 channels on the abstraction level *L0*, among them 4 services are specified using local variables). Figure 7 presents how the number of services was changed among the abstraction levels.

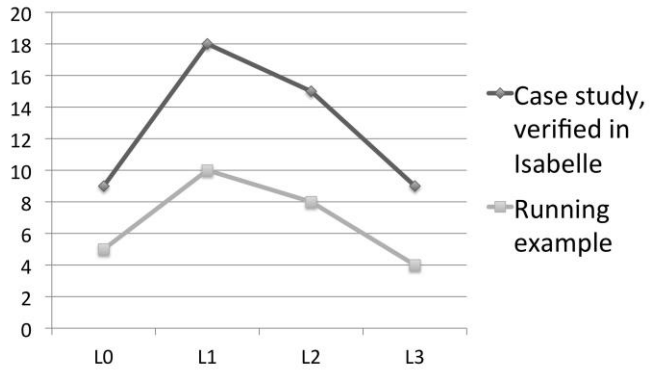

Figure 7. Correlation between the number of services and abstraction levels: Case study and the running example

Overall, the case study contains more than 300 lemmas; approx. 50% of them can be composed and proven automatically, using the predefined schema, which is a part of our formalization. All the technical details of formalization within the theorem prover Isabelle as well as the corresponding proofs for a case study to elaborate the approach are presented in (Spichkova, 2014b). Approx. 90% of the proofs (general as well as for a concrete system from the case study) are constructed using Isabelle's component Sledgehammer (Blanchette et al., 2011). Sledgehammer discharges proof goals applying to them resolution based first-order automatic theorem provers (ATPs) and satisfiability modulo theories (SMT), which makes the human-related part of the proof simpler and faster.

9. DISCUSSION AND RELATED WORK

The approach presented above extends our previous work in (Feilkas et al., 2009; Spichkova, 2010; Spichkova, 2011) by providing

(1) formal approach for modeling of data and control flow dependencies between components/services, and

(2) associated algorithm for architecture optimization towards efficient verification, testing, and monitoring of system's properties.

We leave here out of scope the automatization of the construction of the functions I^D, O^D, etc. for a concrete case. For the example presented in this paper as well as for the case study, verified in Isabelle, these functions were specified manually. This automatization is planned for the future work. However, one of the advantages of our current work is formal semi-automatic analysis of system architecture, and the sufficiency of the set of system's components to check a certain property.

Thus, in our work we touch the following research areas: modeling communication between components, system decomposition, as well as architecture modeling. In the rest of the section we discuss the related work on them.

8.1 Communication Modeling

Various languages and techniques have been proposed to represent communication between components/processes, for example, Bergstra's Algebra of Communicating Processes (Bergstra, Klop, 1985), Hoare's approach on Communicating Sequential Processes (Hoare, 1978) and its extension (Hilderink, 2003). Magee et al. (1995) tried to combine the ideas of operational semantics with Milner's π calculus, calculus of mobile processes, cf. (Milner, 1982; Milner, 1999). Reo, a channel-based coordination model for component composition, represents a co-algebraic view on this area (Arbab, 2004; Meng et al., 2012).

The work presented in (Vogel-Heuser et al., 2011) defines an extensive support to the components communication and time requirements, while the model discussed in (Hadlich et al., 2011) proposes general ideas on model for distributed automation systems.

Our approach, in contrast, is focusing not on the communication in general but on the dependency aspects and how their analysis can be used to increase the efficiency of properties checking.

8.2 System decomposition

There are a large number of approaches in the area of systems decomposition, cf., e.g., (Philipps, Rumpe, 1999; da Cruz, Penzestadler, 2008). In general we can say, that decomposition in many cases leads to a refinement of a system, where by composing a system from components we can implicitly build a new level of abstraction of system representation, cf. (Broy, 1997; Spichkova, 2008).

The main difference and the main contribution of our current work, also compared with our previous work on a formal decomposition (Spichkova, 2011), is

(1) extending of the specification approach to associate subspecifications with subservices or architectural components in a more usable, readable way,

(2) focusing on the aspects essential for the efficient checking of system's properties,

(3) taking into account aspects that are important for the case of using high-performance computing and cloud virtual machines.

To this end, our focus in the current paper goes also beyond ideas presented in (Spichkova, 2011), where readability and manageability of specification were discussed in relation to inconsistencies and incomplete specifications.

8.3 Architectural modeling

An introductory overview of foundations and applications of the model driven architecture can be found in (Rensink, Warmer, 2006). There is a large collection of approaches on architecture elaboration, with different aims and domain orientations, e.g., Medvidovic and Taylor (2000) introduce a classification and comparison framework for software architecture description languages, Malek et al. (2012) presents a framework for improving distributed system's architecture and their deployment, Broy (2005) focuses on specification and design of services and layered architectures.

A number of approaches, e.g., (Tyree, Akerman, 2005), propose several meta-models that introduce relationships between architectural design decision alternatives and activities related to them.

A number of architecture description languages have been developed to specify compositional views of a system on an abstract level, e.g., TrustME (Schmidt et al., 2001), which combines software architecture specification approaches with ideas of design-by-contract and allows capturing of behavioral interaction patterns between large-scale components of software and systems architectures, and gives a background for the Rich Architectural Description Language RADL (Schmidt, 2003) and its modeling of predictable component-based distributed control architectures by extending finite state automata and Petri nets (Schmidt et al., 2003), as well as a framework for a family of mutually-compatible parameterized contract models (Peake, Schmidt, 2011).

An architectural framework for developing systems of systems, where the development plants are geographically distributed across different countries, was presented in (Spichkova et al., 2015). A spatio-temporal architecture-based framework for testing services in the Cloud was proposed in (Liu et al., 2015a). A testing framework based on the logical architecture was presented in (Liu et al., 2015b). Requirements engineering aspects of a geographically distributed architecture were discussed in (Spichkova, Schmidt, 2015).

Our current work is focused on modeling of a logical service-oriented architecture. The main focus of our approach is on elaborating for a concrete system, which part of the system is sufficient to check a certain property, and how to optimize the logical architecture towards efficient monitoring, testing, and verification of system's properties, also for the case of remote connection.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a formal approach to modeling and analysis data and control flow dependencies between components. We described the theory, which lies behind the approach and presented a running example to illustrate the main ideas of the approach. We conclude that our approach allows

- to specify the dependencies within a system formally,
- to elaborate for a concrete system, which part of the system is sufficient to check a certain property,
- to analyze tracing and reliability aspects,
- to optimize the architecture of a given system, also taking into account such aspects as
- costs of data transfer/ upload to the cloud,
- and requirements of using high-performance computing and cloud virtual machines.

These results are especially important for analysis and optimization of remotely deployed distributed control systems: our approach allows system decomposition oriented towards efficient checking of system properties – it allows to send to or from the cloud only the information really needed for the monitoring, testing or verification of the properties of interest. Another contribution of our approach is a semi-automatic support of the presented ideas on verification level, an interactive semi-automatic theorem prover Isabelle/HOL.

Future Work: In the future work, we intent to automatize the construction of the functions that specify the dependencies (I^D, O^D, etc.) for a concrete case on each level of abstraction. One of the other possible directions of our future work is on combination of the ideas presented above with our previous work on analysis of crypto-based components (Spichkova, 2014a), to analyze the data dependencies between components wrt. secrecy/security properties.

11. REFERENCES

Arbab, F. (2004). Reo: a channel-based coordination model for component composition, *Mathematical Structures in Computer Science*, 14(3), pp. 329-366.

Barnat, J., Chaloupka, J., van de Pol, J. (2008). Improved distributed algorithms for SCC decomposition. *Electronic Notes in Theoretical Computer Science*, 198(1), pp. 63-77.

Bergstra, J.A., Klop, J.W. (1985). Algebra of communicating processes with abstraction, *ENTCS*, 37, pp. 77-121.

Blanchette, J.C., Böhme, S., Paulson, L.C. (2011). Extending Sledgehammer with SMT solvers. In N. Bjørner and V. Sofronie-Stokkermans (Eds.), *Automated Deduction* (pp. 116-130). LNCS, 6803. Dordrecht, The Netherlands: Springer.

Botaschanjan, J., Kof, L., Kühnel, C., Spichkova, M. (2005). *Towards Verified Automotive Software*. In Proceedings of 2nd International ICSE workshop on Software, ACM

Botaschanjan, J., Gruler, A., Harhurin, A., Kof, L., Spichkova, M., Trachtenherz, D. (2006). *Towards Modularized Verification of Distributed Time-Triggered Systems.* In Proceedings of Formal Methods, Springer-Verlag Berlin Heidelberg, Germany, pp. 163–178.

Botaschanjan, J., Broy, M., Gruler, A., Harhurin, A., Knapp, S., Kof, L., Paul, W., Spichkova, M. (2008). On the correctness of upper layers of automotive systems. Formal aspects of computing, 20(6), pp. 637-662.

Broy, M. (1997). Compositional refinement of interactive systems, *Journal of ACM*, 44(6), pp. 850-891.

Broy, M., Stølen, K. (2001): Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, New York, USA.

Broy, M. (2005). Service-oriented Systems Engineering: Specification and design of services and layered architectures. The JANUS Approach. In M. Broy, J. Grünbauer, D. Harel, T. Hoare (Eds.), *Engineering Theories of Software Intensive Systems* (pp. 47-81). Dordrecht, The Netherlands: Springer.

Buyya, R., Yeo, C.S., Venugopal, S. (2008). Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities, Proceedings of the IEEE International Conference on High Performance Computing and Communications, Dalian, China.

da Cruz, D.B., Penzenstadler, B. (2008). Designing, Documenting, and Evaluating Software Architecture, TU München, Technical Report, TUM-I1018. Retrieved April 17, 2016, from https://www4.in.tum.de/publ/papers/TUM-I0818.pdf

Dijkstra, E.W. (1969). Complexity controlled by hierarchical ordering of function and variability. Software Engineering: Report of a conference sponsored by the NATO Science Committee. Retrieved April 17, 2016, from https://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD236.html

Dobi, S., Gleirscher, M., Spichkova, M., Struss, P. (2013). Model-based hazard and impact analysis, Tech. Rep., TUM-I1333. Retrieved April 17, 2016, from http://mediatum.ub.tum.de/doc/1165553/1165553.pdf

Eker, J., Janneck,J. W. , Lee, E. A. , Liu, J., Liu, X., Ludvig, J. , Neuendorffer, S., Sachs S., Xiong, Y. (2003): *Taming heterogeneity - the Ptolemy approach.* Proceedings of the IEEE 91(1), pp. 127–144.

Feilkas, M., Hölzl, F., Pfaller, C., Rittmann, S., Scheidemann, K., Spichkova, M., Trachtenherz, D. (2009). A Top-Down Methodology for the Development of Automotive Software, TU München, Technical Report TUM-I0902. Retrieved April 17, 2016, from https://www4.in.tum.de/publ/papers/TUM-I0902.pdf

Fisler, K., Fraer, R., Kamhi, G., Vardi, M., Yang, Z. (2001). Is there a best symbolic cycle-detection algorithm? In T. Margaria and W. Yi (Eds.), Tools and Algorithms for the Construction and Analysis of Systems (pp. 420-434). LNCS, 2031. Berlin, Heidelberg, Germany: Springer.

Fleischer, L., Hendrickson, B., Pınar, A. (2000). On identifying strongly connected components in Parallel. In J. Rolim (Ed.), *Parallel and Distributed Processing* (pp. 505-511). LNCS, 1800. Berlin, Heidelberg, Germany: Springer.

Foster, I., Kesselman, C., Nick, J.M., Tuecke, S. (2002). The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, Technical Report.

Hadlich, T., Diedrich, C., Eckert, K., Frank, T., Fay, A., Vogel-Heuser, B. (2011). Common communication model for distributed automation systems,

Proceedings of the 9th IEEE International Conference on Industrial Informatics, IEEE INDIN, Caparica, Lisbon, pp. 131-136.

Hilderink, G. (2003). Graphical modelling language for specifying concurrency based on CSP, IEEE Software, 150, pp. 108-120.

Hoare, C.A.R. (1978). Communicating sequential processes, *Communications ACM*, 21 (8), pp. 666—677.

Jackson, M. (2001). *Problem Frames: Analysing and Structuring Software Development Problems.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Kühnel, C., Spichkova, M. (2006). Upcoming automotive standards for fault-tolerant communication: FlexRay and OSEKtime FTCom. *Proceedings of EFTS 2006 International Workshop on Engineering of Fault Tolerant Systems*. Universite du Luxembourg, Luxembourg.

Kühnel, C., Spichkova, M. (2007). Fault-tolerant communication for distributed embedded systems, *Software Engineering of Fault Tolerance Systems*, Series on Software Engineering and Knowledge Engineering, 19, pp. 175-198.

Liu, H., Spichkova, M., Schmidt, H. W., Sellis, T., Duckham, M. (2015). Spatio-temporal architecture-based framework for testing services in the cloud. In Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference (pp. 18-22). ACM.

Liu, H., Spichkova, M., Schmidt, H. W., Ulrich, A., Sauer, H., Wieghardt, J. (2015). Efficient testing based on logical architecture. In Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference (pp. 49-53). ACM.

Magee, J., Dulay, N., Eisenbach, S., Kramer, J. (1995). Specifying distributed software architectures," Proceedings of the 5th European Software Engineering Conference 1995, pp. 137-153. Springer.

Malek, S., Medvidovic, N., Mikic-Rakic, M. (2012). An extensible framework for improving a distributed software system's deployment architecture, *IEEE Transactions on Software Engineering*, 38(1), pp. 73-100.

Medvidovic, N., Taylor, R. (2000). A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering*, 26(1), pp. 70-93.

Meng, S., Arbab, F., Aichernig, B., Astefanoaei, L., de Boer, F., Rutten, J. (2012). Connectors as designs: Modeling, refinement and test case generation, *Science of Computer Programming*, 77 (7).

Milner, R. (1982). *A Calculus of Communicating Systems*. New York, NY: Springer.

Milner, R. (1999). *Communicating and mobile systems - the Pi-calculus.* Cambridge , United Kingdom: Cambridge University Press.

Müller, O., Nipkow, T. (1995). *Combining Model Checking and Deduction for I/O-Automata.* In Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS '95), Ed Brinksma, R. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen (Eds.). Springer-Verlag, London, UK, pp. 1-16.

Naur, P. (1968). The profiles of software designers and producers. Garmisch, Germany. Report of a conference sponsored by the NATO Science Committee. Retrieved April 17, 2016, from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

Nipkow, T., Paulson, L.C., Wenzel, M. (2002). *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS, 2283. Berlin, Heidelberg, Germany: Springer.

Orzan, S.M. (2004). On distributed verification and verified distribution, Ph.D. dissertation, Free University of Amsterdam.

Parnas, D. L., Madey, J. (1995): Functional Documents for Computer Systems. *Science of Computer Programming*, 25, pp. 41–61.

Peake, I., Schmidt, H. (2011). Systematic simplicity-accuracy tradeoffs in parameterised contract models. pp. 95-104, ACM.

Philipps, J., Rumpe, B. (1999). Refinement of Pipe-and-Filter Architectures. In J.M. Wing, J. Woodcock, and J. Davies (Eds.), *Proceedings of the FM'99* (pp. 96-115), LNCS, 1708. Berlin, Heidelberg, Germany: Springer.

Rensink, A., Warmer, J. (Eds.). (2006). *Model Driven Architecture – Foundations and Applications*, LNCS, 4066. Berlin, Heidelberg, Germany: Springer.

Schmidt, H., Poernomo, I., Reussner, R. (2001). Trust-by-contract: Modelling, analysing and predicting behaviour of software architectures, *Journal of Integrated Design & Process Science, 5(3)*, pp. 25–51.

Schmidt, H. (2003). Trustworthy components - compositionality and prediction. *Journal of Systems and Software. Component-Based Software Engineering*, 65 (3), pp. 215 - 225.

Schmidt, H., Peake, I., Xie, J., Thomas, I., Kramer, B., Fay, A., Bort, P. (2003). Modelling Predictable Component-Based Distributed Control Architectures. *Proceedings of the IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems.* IEEE Computer Society.

Spichkova, M. (2006). FlexRay: Verification of the FOCUS Specification in Isabelle/HOL. A Case Study. TU München, Technical Report TUM-I0602. Retrieved April 17, 2016, from http://www4.informatik.tu-muenchen.de/publ/papers/FoI_FlexRay_0206.pdf

Spichkova, M. (2008). Refinement-based verification of interactive real-time systems, *Electronic Notes in Theoretical Computer Science, 214*, pp. 131-157.

Spichkova, M. (2010). Architecture: Methodology of decomposition, TU München, Technical Report, TUM-I1018. Retrieved April 17, 2016, from https://www4.in.tum.de/publ/papers/TUM-I1018.pdf

Spichkova, M. (2011). Architecture: Requirements + Decomposition + Refinement. *In Softwaretechnik-Trends*, *32(4)*. Retrieved April 17, 2016 from http://pi.informatik.uni-siegen.de/gi/stt/31_4/index.html

Spichkova, M. (2012). Human Factors of Formal Methods. In *Proceedings of the IADIS Interfaces and Human Computer Interaction 2012 (IHCI 2012).* Lisbon, Portugal: IADIS Press.

Spichkova, M., Hölzl, F., Trachtenherz, D. (2012). Verified System Development with the AutoFocus Tool Chain. 2nd Workshop on Formal Methods in the Development of Software (WS-FMDS), Paris, France.

Spichkova, M. (2013a). Design of formal languages and interfaces: "Formal" does not mean "unreadable". In K. Blashki, & P. Isaias (Eds.), *Emerging Research and Trends in Interactivity and the Human-Computer Interface* (pp. 301-314). Hershey, PA: IGI Global. doi:10.4018/978-1-4666-4623-0.ch015

Spichkova, M. (2013b). Stream Processing Components: Isabelle/HOL Formalisation and Case Studies, *Archive of Formal Proofs*, ISSN 2150-914x. Retrieved April 17, 2016, from http://arxiv.org/abs/1405.1512

Spichkova, M. (2014a). Compositional properties of crypto-based components, *Archive of Formal Proofs*, ISSN 2150-914x. Retrieved April 17, 2016, from http://arxiv.org/abs/1405.3006

Spichkova, M. (2014b). Formalisation and analysis of component dependencies, *Archive of Formal Proofs*, ISSN 2150-914x. Retrieved April 17, 2016, from http://arxiv.org/abs/1405.3017

Spichkova, M., Schmidt H. (2014). Towards Logical Architecture and Formal Analysis of Dependencies Between Services, Proceedings of the 2014 Asia-Pacific Services Computing Conference (APSCC 2014), Fuzhou, China.

Spichkova, M., Liu, H., Schmidt, H. (2015). Towards quality-oriented architecture: Integration in a global context. In Proceedings of the 2015 European Conference on Software Architecture Workshops (p. 64). ACM.

Spichkova, M., & Schmidt, H. (2015, April). Requirements engineering aspects of a geographically distributed architecture. In Evaluation of Novel Approaches to Software Engineering (ENASE), 2015 International Conference on (pp. 276-281). IEEE.

Tyree, J., Akerman, A. (2005). Architecture decisions: demystifying architecture, *IEEE Software*, 22(2), pp. 19-27.

Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M. (2009). A break in the clouds: towards a cloud definition. SIGCOMM Computer Communication Review, 39 (1), pp. 50-55, ISSN 0146-4833, ACM.

Vogel-Heuser, B., Feldmann, S., Werner, T., Diedrich, C. (2011). Modeling network architecture and time behavior of Distributed Control Systems in industrial plant automation, *Proc. of the 37th Conference on IEEE Industrial Electronics Society*, Melbourne, Australia, pp. 2232 -2237.

Wenzel, M. (2013). The Isabelle/Isar reference manual. Retrieved January 7, 2015, from http://isabelle.in.tum.de/doc/isar-ref.pdf

Authors

Dr Maria Spichkova is a Lecturer at the RMIT University, Australia. She received a PhD in Computer Science in 2007 at Technical University Munich, Germany, where she worked in a researcher and lecturer role between 2003 and 2013. She was involved, both as participant and as project leader, in a large number of research industrial projects in collaboration with AVL, Microsoft, DENSO Automotive, Robert Bosch GmbH, BMW Car IT, etc. Maria Spichkova has supervised PhD, Bachelor and Masters Theses, taught a number of lecture courses (e.g., Applied Logic in Engineering, Modeling of distributed systems, Software Testing) as well as seminar courses. She has published 5 book chapters and many articles in international journals and conferences. She has also served as the reviewer of journals and conferences (such as IEEE Transactions on Industrial Informatics, Journal of Software and Systems Modeling, Acta Informatica, etc.), the Program Committee co-chair and member for international conferences and workshops. Currently, she conducts research activity related to specification, modeling, testing, and verification of safety-critical and distributed systems as well as to human factor related areas.

Prof Heinz Schmidt is Professor of Software Engineering in the School of Computer Science and Information Technology at the RMIT University, Australia, where he also directs the Australia-India Research Centre for Automation Software Engineering and the eResearch Office. He received his PhD from Bremen University, Germany. He is also Adjunct Professor at Maelardalen University, Sweden.

Professor Schmidt is internationally recognized in software engineering for parallel and distributed systems, notably systems composed of large numbers of interacting components in application areas ranging from industrial automation to telecommunication networking and artificial intelligence. Over the last few years he has focused on modeling, predicting and verifying extra-functional properties in software-intensive distributed systems. Professor Schmidt has published over 130 refereed articles on aspects of distributed and concurrent systems using Petri Nets, automata, abstract data type, and on practical methods/tools for constructing and testing such systems. He has published several books and book chapters.

He holds a patent in prediction of probabilistic worst-case time in software-intensive embedded control systems, jointly with ABB Germany. He has supervised and co-supervised over 25 higher-degree research students.