HAL
open science

Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates

Sönke Holthusen, Sophie Quinton, Ina Schaefer, Johannes Schlatow, Martin Wegner

To cite this version:

Sönke Holthusen, Sophie Quinton, Ina Schaefer, Johannes Schlatow, Martin Wegner. Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates. Workshop on Pre- and Post-Deployment Verification Techniques, Jun 2016, Reykjavik, Iceland. 10.4204/EPTCS.208.3 . hal-01426654

HAL Id: hal-01426654

https://inria.hal.science/hal-01426654v1

Submitted on 5 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates

Sönke Holthusen[1], Sophie Quinton[2], Ina Schaefer[1], Johannes Schlatow[1], Martin Wegner[1]
[1]TU Braunschweig
{s.holthusen, i.schaefer}@tu-bs.de, schlatow@ida.ing.tu-bs.de, wegner@ibr.cs.tu-bs.de
[2]Inria Grenoble - Rhône-Alpes
sophie.quinton@inria.fr

In this paper we address the issue of change after deployment in safety-critical embedded system applications. Our goal is to substitute lab-based verification with in-field formal analysis to determine whether an update may be safely applied. This is challenging because it requires an automated process able to handle multiple viewpoints such as functional correctness, timing, etc. For this purpose, we propose an original methodology for contract-based negotiation of software updates. The use of contracts allows us to cleanly split the verification effort between the lab and the field. In addition, we show how to rely on existing viewpoint-specific methods for update negotiation. We illustrate our approach on a concrete example inspired by the automotive domain.

1 Introduction

Critical embedded systems currently offer limited support for software updates. In the automotive domain, this leads to outdated software being used on mechanically durable vehicles. Adaptable vehicle electronics could be used, e. g., for increasing safety via software patches, or to limit costs by a better usage of computational resources. In a different context, aerospace computing platforms are currently migrating from fixed, ground-verified configurations to re-configurable platforms, and from functionally separated solutions to integrated systems with the capability to change during operations. These two examples illustrate the fact that we need to support *flexible* adaptation of safety-critical embedded systems via continuous change after deployment.

At the moment software updates in critical embedded systems are prepared via intensive verification and tests performed in the lab. Unfortunately, this lab-based verification process is becoming increasingly difficult, in particular because of complex platform dependencies between changing applications—for example through timing, fault handling, security mechanisms, etc. Every particular system configuration is different and there are just too many of them to perform exhaustive a priori verification, even using abstraction techniques. For this reason, our goal is to use *in-field* formal analysis to determine whether a specific update in a given system configuration may be safely applied.

Although promising, in-field verification of software updates for safety-critical embedded systems is also challenging because it requires an automated process able to handle multiple viewpoints such as functional correctness, timing or security, with access to limited resources. For this purpose, we propose a *contract*-based methodology. Such an approach is particularly well-suited to cleanly split the verification effort between the lab (pre deployment) and the field (post deployment). In addition, it lends itself to our original approach whose specificity is to address multi-viewpoint verification based on a combination of existing viewpoint-specific methods. Our work is motivated by the project *Controlling*

L. Aceto, A. Francalanza and A. Ingolfsdottir (Eds.): Workshop on
Pre- and Post-Deployment Verification Techniques (PrePost 2016)
EPTCS 208, 2016, pp. 31–45, doi:10.4204/EPTCS.208.3

Concurrent Change (CCC)[1], which addresses new methods to develop and control embedded system platforms integrating changing applications under high requirements to real-time, safety, availability, and security. The methodology is currently being implemented as a complete tool chain. We strive to present our current results both at a high level of abstraction, so that our results can be reused or adapted to other contexts, and at a lower level of abstraction so that our theory matches the practical needs of the CCC project. This dual approach is reflected in this paper.

This paper is organized as follows. Section 2 introduces the general methodology that we propose. Section 3 then presents the actual context in which we develop this approach. In Section 4 we show the effectiveness of our methodology on a concrete example in the automotive domain. Finally, Section 5 discusses the state of the art and Section 6 concludes.

2 Contracting for software update negotiation

We present here our general approach to in-field negotiation of software updates.

2.1 Problem statement

Let us state first that, independent of verification, software updates require a component-based approach. The overall objective of in-field negotiation is then to guarantee that conformance to system requirements is preserved during the update. It is however likely that establishing conformance monolithically will not be feasible in practice; hence the use of contracting, which clearly splits responsibilities between a component and its environment so as to make design and verification easier. As a result, our methodology assumes that the following primitives are given.

Component framework. We suppose a given notion of component so that one can specify requirements and verify that a given system conforms to them.

- *component*: In this paper we address software updates so the term component refers to software components. How these components are specified depends on the actual framework, but at this point we do not make any assumption about them. A set of components form what we call a *software model*.
- *system model*: A software model alone is not sufficient to describe a system if one wants to verify non-functional properties. Therefore, we define a system model as a triple made of:
  - a software model;
  - a *platform model*, which is not subject to change;
  - a *configuration* that describes all runtime parameters relating components and platform which may be changed during negotiation.
- *system requirements*: The whole verification problem supposes that there exists some formal representation of the expected behavior of a system, which we suppose given as a set of requirements. In addition, we need a formal relation between systems and requirements, that we call *conformance*, to allow establishing that a given system conforms to its specified requirements.

[1] `http://ccc-project.org/`

Contract theory. We further assume primitives relating to contracting.

- *contract*: A contract for a component K is a pair (A,G) consisting of an *assumption* A on the system in which K will execute and a *guarantee* G on the way K will behave in such a system.
- *contract satisfaction*: Components and contracts are related via a contract satisfaction relation. Intuitively, a component *satisfies* a contract $\mathscr{C} = (A,G)$ if, assuming that A holds, then G holds, too.
- *compatibility of contracts*: This concept is derived from the above primitives. A set of contracts is *compatible with respect to* a platform model if there exists a *feasible configuration*, i. e., a configuration such that, in any system made of components satisfying the given contracts, if assumptions on the environment of the system hold then all assumptions and therefore all guarantees hold.

Contract compatibility is the cornerstone of the contract-based approach as it does not require the actual components to be available, only their contracts. Because of that, we use from now on the term *software model* to refer to the set of contracts abstracting the components.

Besides, note that an additional verification step is in principle needed after compatibility has been established in order to derive conformance to system requirements from assumptions and guarantees. In our case, system requirements are always expressed directly as assumptions or guarantees so we will largely ignore this step in the rest of the paper.

In a context where a component framework and a corresponding contract theory have been developed, we can now formulate our problem as follows.

Definition 1 *An* update request *is a pair* $(changeType, \mathscr{C})$ *where*

- *changeType describes the change to undertake which is one of the following* $\{add, remove, update\}$*;*
- $\mathscr{C}$ *is the contract attached to the component undergoing change.*

The objective of the *negotiation process* is to determine whether the update request can be granted and if that is the case then for which configuration.

Definition 2 *A* negotiation process *is a function taking a system model* $sys = (software, platform, config)$ *and an update request* $(changeType, \mathscr{C})$ *as parameters and returning an* answer*: either* no*, or* yes *along with a feasible configuration c.*

This means that negotiation is a synthesis problem: we need to synthesize a configuration that makes all contracts compatible.

2.2 Design and verification flow

The use of contracts effectively splits the verification process between the lab and the field, as we show in the following.

In the lab. At design-time (see Figure 1), during software development and test in the lab, additional data for helping the in-field negotiation process are prepared and formalized as contracts. They will typically include properties of the software components which can only be obtained using involved verification methods, e. g., requiring manual intervention such as theorem proving, or intensive computation, for example in the case of model checking. Assumptions on the environment in which the system will function may also be specified. Contract satisfaction is then established in the lab, as both the component and its contract are available. This step is performed for every component independently and saves the effort to verify every possible combination of components and possibly versions of components.

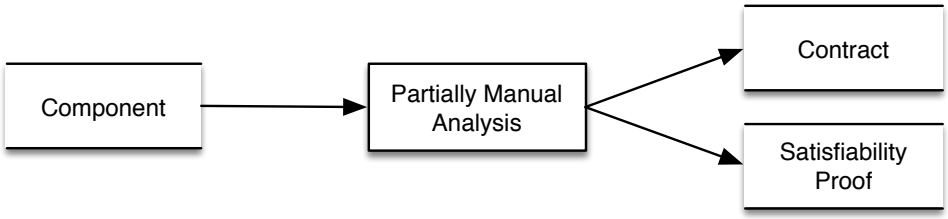

Figure 1: In the lab a contract and a proof for satisfaction is generated.

In the field. After deployment, whenever an update is requested, formal verification is performed to determine whether the update may take place without risking a violation of system requirements. This step precedes the actual update and the system architecture is designed to ensure that it does not disrupt the running system. A negotiation process takes place, aiming at finding a system configuration which will guarantee that all system requirements will be satisfied after the update. This is the challenge of the approach as this synthesis and verification phase must be performed automatically, with limited memory and in reasonable time. Updates will typically be performed while the system is suspended (e. g., overnight while the car is parked) so that the time constraint exists but is less tight than the memory constraint.

To succeed, the component in charge of the negotiation, which we call the *Negotiation Controller* (NC), can rely on the contracts that have been prepared at design time. A first specificity of our approach is that we do not try to tackle all viewpoints together as in e. g., [3]. Instead, we rely on *viewpoint-specific analysis engines*, each of which uses a model that is best adapted to the particular viewpoint it is dealing with. This also means, we do not encode the search for a configuration into one big constraint-solving problem including all viewpoints. We reduce the possible variables for the main problem to a minimum, while the viewpoints can solve viewpoint-specific problems on their current partial configuration. Besides, the brute-force approach that involves checking all possible configurations until a feasible one is found is not viable due to the size of the configuration space. To tackle this, a second specificity of our approach is that we rely on added capabilities of the viewpoint-specific analysis engines. Indeed, our experience is that these engines, although used in general for checking one given configuration, can be extended to provide information about sets of configurations. Our strategy is therefore to start from an over-approximation of the set of feasible configurations and iteratively update this set based on the results provided by the analysis engines. More precisely we proceed as shown in Figure 2.

1. The NC computes a set of *configuration candidates* which is represented as a set of constraints.
2. The NC then picks one candidate and triggers the viewpoint-specific engines so that they check if that candidate configuration is feasible w. r. t. the considered viewpoint — and as such a witness to the software model being compatible with respect to the platform model.
3. If the configuration is not feasible, one engine returns a set of additional constraints to the NC. If the configuration is feasible for all viewpoints then the update can be accepted.
4. The last two steps repeat until the NC can no longer find a candidate or if a feasible configuration has been found.

In the rest of this paper, we show in more detail how we follow this approach in a specific, applied context.

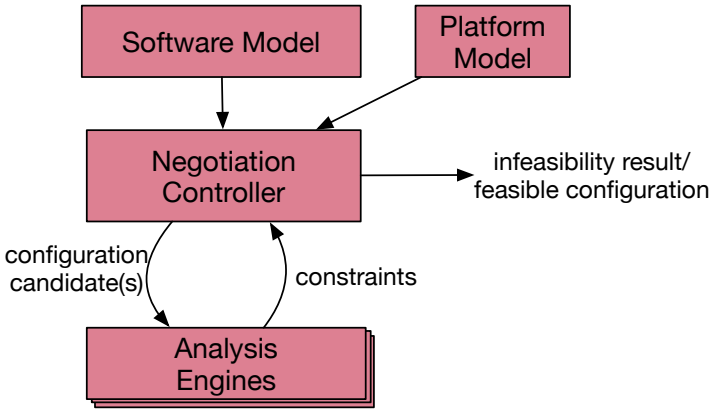

Figure 2: With the contracts from the software model and the available hardware resources from the platform model, the contract negotiation takes place in field.

3 Software update negotiation in the CCC project

In this section, we present the architecture that we are currently developing to support negotiation in the context of the CCC project. Our goal in this section is to define all the concepts used in Definition 2, namely: a software model represented as a set of contracts, a platform model and a notion of configuration — and to show how we use them to perform contract negotiation.

Platform architecture. Similar to integrated architectures like AUTOSAR [2] or ARINC 653, our approach offers a clear separation between software/application and hardware/platform components through a Runtime Environment (RTE). We use the Genode OS Framework [6] as RTE, which allows for the design of component-based systems where access control is applied via so-called capabilities. Our architecture complements the RTE with a so-called *Multi-Change Controller (MCC)* which acts as negotiation controller. Similar to the EPOC [9] project, the MCC is divided into a *model domain*, in which analyses are performed without actually modifying the system, and an *execution domain*, in which the update is implemented. In this paper, we focus on the model domain.

Component framework. Our software is built as a set of *functions* made of *components*. For convenience, we only consider here functions with a single component so most of the time, we will abstract functions away. Components are interconnected via *services*: components can offer services to each other, or require services from one another. Matching between offered and required services is made via comparison of the corresponding *service interfaces*, which describe the services' expected type of communication.

Communication between components can be performed synchronously via *Remote Procedure Calls (RPCs)* or asynchronously using *signals*. More precisely, RPCs allow passing arguments to the component being called, but the caller cannot continue its execution until the callee has finished handling the request. In contrast, signals can be used asynchronously, only notifying the callee and ending the communication afterwards.

Components are implemented as one or more *threads* such that each thread can be activated by: (1) another component requiring a service via an RPC or a signal; (2) another thread of the same component; or (3) a timer.

Requirements. Although the scope of the CCC project is larger, we consider in this paper the following viewpoints.

- *functional dependencies*: This viewpoint addresses issues related to the structural information presented above, e. g., to guarantee that all required services are indeed provided.
- *timing*: The timing viewpoint is typically concerned with establishing that a function reacts to an input within a specified delay.
- *functional correctness*: Functional correctness focuses on control flow properties, e. g., expressing that a given service cannot be called before some initialization step has been performed.

All the above system requirements can be expressed at the component or function level so we specify them directly in our contracts. Conformance to these requirements is thus established if contract compatibility is guaranteed.

Contract framework. To express assumptions and guarantees related to our three viewpoints we have developed a contract language capturing all the relevant information. Rather than using a formal definition, in the following we choose to describe our language through an example.

Functional dependencies. Listing 1 shows the contract for a component T, which requires a service `object_recognition` (line 3) and provides a service `trajectory_calculation` (line 4). The names of the required and provided services are IDs, linking to a more detailed service description in a global repository. Knowing the required and provided services of all components, functional dependencies can be resolved [14].

Timing. The timing requirement is specified at lines 20–22 and states that the RPC call to `object_recognition` must complete within 100 time units. In addition, to enable timing analysis [7], contracts must contain enough information to extract a task graph [8]. A task graph is a directed graph in which nodes represent tasks and edges represent the asynchronous (or synchronous) activations/calls and returns. For that purpose the thread structure as well the nature of communication (RPC or signal) must be known. In our example, the component is made of two threads called `trajectory_calculation_get` and `trajectory_calculation_init`. The former is triggered via RPC (line 7 in the listing). When activated it first proceeds with executing some local processing task which takes between 1 and 5 time units to complete on a computing resource of type `CPU_type_1` (lines 8 to 10). It then performs an RPC call to use a service (line 11) and finally executes another task before completing (lines 12–13).

```
1  component T
2    services
3      requires object_recognition
4      provides trajectory_calculation
5    threads
6      thread trajectory_calculation_get
7        on RPC trajectory_calculation.get()
8          task tc1
9            onto CPU_type_1
10             wcet=5 bcet=1
11         RPC object_recognition.get()
12         task tc2
```

```
13          onto CPU_type_1
14            wcet=5 bcet=1
15    thread trajectory_calculation_init
16      on RPC trajectory_calculation.init()
17        task tci
18          onto CPU_type_1
19            wcet=10 bcet=5
20  timings
21    timing 100
22      object_recognition.get()
23 control_flow
24  not trajectory_calculation.get()
25    until trajectory_calculation.init()
```

Listing 1: An example contract.

Functional correctness. Finally, the functional requirement is specified at lines 23–25. It requests that the trajectory calculation must always be initialized before it is used. Such control flow properties can be easily checked on a control flow graph extracted from the contracts.

Note that our contracts do not explicitly distinguish assumptions from guarantees. In addition, a significant part of the guarantees hold independent of the assumptions and can thus be regarded as part of a specification rather than a contract. We have chosen this presentation to improve readability. In the contract for component T, the assumptions correspond to the three viewpoint-specific requirements. The most notable guarantees regard worst-case execution time bounds, which can only be formally established using involved verification techniques including abstract interpretation [18].

System model. Let us synthesize the information presented above to define our system model. Remember that a system model is made of a software model, a platform model and a configuration.

- *software model*: this is the set of contracts used as an abstraction layer on top of components. Note that only the components that are actually part of the chosen configuration will be loaded when a new configuration is set up.
- *platform model*: for contract negotiation we need some abstraction of the platform as well. The only non-functional viewpoint that we address here is timing, so information about the available processing resources and their scheduling policy is sufficient, as will be detailed later. As already mentioned, we assume that the platform model is fixed.
- *configuration*: this describes how the selected software components are connected to each other and to the hardware. It consists of a set of connections (functional dependencies) between offered and required services, a mapping of tasks to the platform resources, and priorities assigned to tasks.

Denote *software* the set of software contracts that make up the software model. The set of services (and whether they are required or offered by a given component) as well as the set of tasks that correspond to a given component can easily be extracted from the component's contract.

Denote *platform* the set of processing resources that constitute the platform model. In our case all resources have the same scheduling policy, namely static-priority preemptive, so the only additional information we need about them is their type in order to guarantee an appropriate mapping of tasks.

Definition 3 *A* configuration *config is a tuple* (C, SC, M, Π) *where:*

- *$C \subseteq$ software represents the set of selected components, i. e., components which must be loaded and run on the platform.*
- *SC is a set of* service connections*, i. e., of triples (c_1, s, c_2) with $c_1, c_2 \in C$ representing a connection between two components c_1 and c_2 via a service s .*
- *M is a function associating to each task the resource onto which it is mapped.*
- *Π is a total priority order on tasks.*

Note that not all these configurations are *well-formed* as we discuss below.

Negotiation process. In this section we formally define how the configuration space is constrained during the contract negotiation. The objective of the negotiation process is to find a feasible configuration. To achieve this the MCC picks a configuration from a set of candidates called the *configuration space* and runs it through the viewpoint-specific analysis engines to determine whether it is feasible. Initially the configuration space contains all *well-formed* configurations (see below). It is then iteratively restricted based on the feedback provided by the viewpoint-specific analysis engines.

It has been shown in [14] that the functional dependency resolution problem can be formulated entirely as a set of constraints, so there is no need for a specific, dedicated analysis engine. Such constraints include in particular:

1. For all $(c_1, s, c_2) \in SC$, $c_1 \neq c_2$ and the service s is indeed required by component c_1 and offered by component c_2.
2. For every selected component $c_1 \in C$ requiring a service s, there is a unique selected component $c_2 \in C$ such that $(c_1, s, c_2) \in SC$.

Regarding mapping and priority assignment, not all configurations are well-formed either as the following constraints must be met:

3. A task τ can only be mapped onto a resource r if r is of the type for which execution time bounds are provided in the contract corresponding to τ.
4. Tasks from the same thread must have the same priority.

Definition 4 *A configuration (C, SC, M, Π) is* well-formed *if and only if it satisfies the four above mentioned conditions.*

In contrast, functional correctness may require intricate analysis which is best kept separate from the main set of constraints. Whenever a configuration is rejected by the functional correctness analysis engine checking control flow properties, a new constraint is added to restrict *SC*. Such constraint will typically forbid any configuration in which c_1 is selected and (c_1, s, c_2) is a service connection if it has been established that c_1 is "misusing" the service offered by c_2. Note that this removes a fairly large set of configuration candidates, namely all possible mappings and priority assignments for each *SC* eliminated.

Finally, timing analysis is the most challenging part of our negotiation scheme. Encoding the response-time analysis as an ILP (integer linear programming) problem has been done [17], but at the cost of either a prohibitive complexity or pessimism, meaning that configurations which may be feasible are discarded. Our strategy is to perform timing analysis using state of the art timing analysis tools such as e. g., pyCPA [12] but to better exploit the provided result. Suppose for example that timing analysis cannot guarantee that a given task τ will meet its deadline in the worst case. This means that for the current mapping any configuration that keeps the same priority for τ and higher-priority tasks will violate τ's timing requirement. A full theory of how such constraints can be generated is in progress and out

of the scope of this paper but it is particularly interesting to remark that such issues have been relatively ignored so far in the real-time systems research community.

4 Application to an automotive example

Let us now illustrate our approach to contracting for update negotiation on an example taken from the automotive domain. We consider a car in which a function for parking assistance is deployed and show how it can be updated to add a function for lane detection. We first describe the software components we use and their relevant properties before elaborating on the negotiation process.

4.1 System model and viewpoints before the update

The component P for parking assistance depends on `trajectory_calculation`, a service for calculating a trajectory for the car (i. e., a list of vectors to steer the vehicle). That service, which is provided by component T, itself needs an `object_recognition` service to determine which objects must be avoided. Suppose that two components offering this service are available (O_1 and O_2), with O_2 additionally offering a service `object_masking` for masking objects.

Initial configuration. Assume that during deployment all the above components have been uploaded onto a unique hardware resource called CPU1. Figure 3 shows how functional dependencies are solved in the current configuration. Note that only P, T and O_2 are selected — meaning that the code and contract of O_1 are available but not currently in use in the system. The rest of the configuration, namely task mapping and priority assignment will be discussed later.

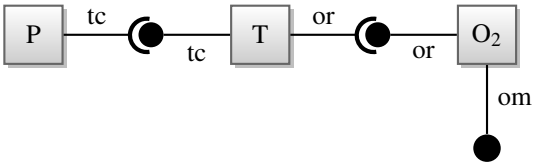

Figure 3: Resolution of functional dependencies chosen during deployment. Boxes represent components, circles and semicircles represent services that are offered and required, respectively. Service names are abbreviated to their acronym.

Initial requirements. Let us describe our system requirements. Listing 2 shows the contract for component P. Note that two keywords have not been introduced before, namely **`initialization`** and **`time`**. The former refers to an initial *operational mode* that precedes the normal mode in which all other calls are performed. The latter term is used to introduce an *activation pattern* which describes how often a given thread is activated based on a timer. In our case, thread `park_assist` is activated periodically every 200 time units with a possible jitter of at most 5 time units.

The contract for T has already been introduced in Listing 1. Due to space constraints the contracts for components O_1 and O_2 are omitted. In addition to functional dependency constraints, there are only two timing requirements and one control flow property to verify:

1. Each activation of the `park_assist` thread must complete within 150 time units — this requirement is specified in Listing 2.

```
component P
  services
    requires trajectory_calculation
  threads
    thread init
      on initialization
        RPC trajectory_calculation.init()
    thread park_assist
      on time (period=200 jitter=5)
        task p1
          onto CPU_type_1
            wcet=3 bcet=1
        RPC trajectory_calculation.get()
        task p2
          onto CPU_type_1
            wcet=7 bcet=1
  timings
    timing 150
      park_assist
```

Listing 2: The contract of P

2. A call to `object_recognition_get` must return within 100 time units.
3. `trajectory_calculation_get` must not be called before at least one call to `trajectory_calculation_init` has been made.

The last two requirements are from the contract for component T.

Timing viewpoint. The task graph presented in Figure 4 shows two task chains corresponding to the normal mode of operation of P and its initialization mode, respectively. This graph has been obtained by unfolding the task graphs inside the thread description with the concrete task graphs of called threads. This means the RPC to *trajectory_calculation.init()* in Listing 2 is replaced by the corresponding task graph from the contract for component *T* (Listing 1). Furthermore, the call of *object_recognition.get()* is replaced by the corresponding task. This continues until all RPCs are replaced by tasks. Note that O_2 has a task which is never activated (namely the task corresponding to the `object_masking` service) and is thus omitted in the task graph.

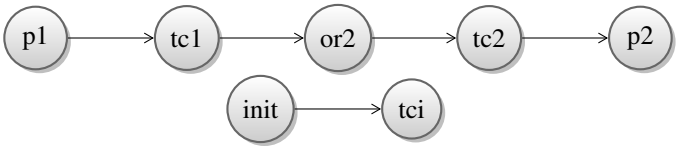

Figure 4: The task graph before the update. Note that or2 corresponds to the task performing the `object_recognition` service in component O_2.

Because our platform model consists of only one resource, all tasks are mapped to CPU1. In presence

of several operational modes timing analysis is performed separately for each mode [15]. Therefore in the normal mode of operation only the first chain of tasks is considered. Our two timing requirements translate at the task level into latency constraints: one from a call to task p1 to the end of p2; the other from a call to tc1 to the end of tc2. We will detail how such requirements are verified in the description of the negotiation process.

Control flow graph. In our example we have chosen a very simple functional correctness requirement, which can be directly derived from the semantics of the keyword **`initialization`** in our contract language. We therefore omit the description of the control flow graph that can be derived from the contracts and focus on the more intricate timing aspects of our example.

4.2 Update scenario

Our objective is to add to the system a component L implementing lane detection, for which it requires services for `object_recognition`, `object_masking` as well as a service for `steering` provided by a component S which is also to be uploaded. We omit the contract for S but show the contract for L in Listing 3: component L is made of a single thread which sequentially calls services `object_recognition`, `object_masking` and `steering` while performing some internal computation in between calls.

4.3 Contract negotiation

Now we can show how our negotiation process is performed to allow the addition of the lane detection function.

Functional Dependency. The first step of contract negotiation aims at finding the set of partial configurations guaranteeing that every service required by a component in the software model is provided by another component in that model. In our case we need to account for the services required by the component to be added to the software model, namely L (note that S does not require any service). Functional dependency is thus solved on the software model consisting of components P, T, O_1, O_2, S and L. Figure 5 illustrates the result, as we explain now. Full lines between offered and required services represent links for which there is no alternative. For example, T is the only component offering service `trajectory_calculation` required by P, making the connection between the two mandatory. In contrast, the `object_recognition` service required by T can be provided by O_1 and O_2, which we denote using dashed lines. Any configuration which matches one of those represented in Figure 5 will guarantee compatibility at the functional dependency viewpoint. For the sake of simplicity we will assume that the service interfaces for `object_recognition` prevent O_1 and O_2 from offering that service to more than one component. In that case there are only two possible solutions left.

Functional correctness. The part of the control flow graph which is relevant for the requirement imposed by T does not change after the update so the functional correctness viewpoint does not need to constrain the configuration space: any configuration which satisfies the functional dependency constraints also passes the functional correctness test.

Timing. The last, more elaborate viewpoint we consider for our example is timing. Figure 6 shows the task graph corresponding to one of the possible functional configurations. The second possible task graph

```
component L
  services
    requires object_masking
    requires object_recognition
    requires steering
  threads
    thread lane_assist
      on time (period=100 jitter=5)
        task la1
          onto CPU_type_1
            wcet=3 bcet=1
        RPC object_recognition.get()
        task la2
          onto CPU_type_1
            wcet=3 bcet=1
        RPC object_masking.get()
        task la3
          onto CPU_type_1
            wcet=10 bcet=5
        RPC steering.setAngle(int value)
        task la4
          onto CPU_type_1
            wcet=4 bcet=1
  timings
    timing 75
      lane_assist
```

Listing 3: The contract of L

is obtained by swapping tasks or1 and or2. The values indicated next to the tasks are their worst-case execution time and therefore a property of the task and not of the task graph.

A simple load analysis, performed, e. g., using pyCPA[2], will reject the functional configuration shown in Figure 6. The reason is that the task chain corresponding to the lane assist function is activated every 100 time units. If or1 is part of this chain, then that chain requires 90% of the available CPU time. However, the park assist chain has a 15% load (i. e., 30 time units every 200 time units). This means that no matter what the priority assignment is, the processor will be overloaded. The only option consists in swapping or1 and or2 in the functional configuration.

Our objective is now to find a priority assignment that satisfies the specified latency constraints. Remember that tasks inherit their priority from the threads to which they belong. In addition, we do not assign the same priority to multiple threads as this typically only adds more pessimism to the analysis result.

Let us focus on the end-to-end latency requirements imposed by P and L, which are respectively 150

[2] https://pycpa.readthedocs.org

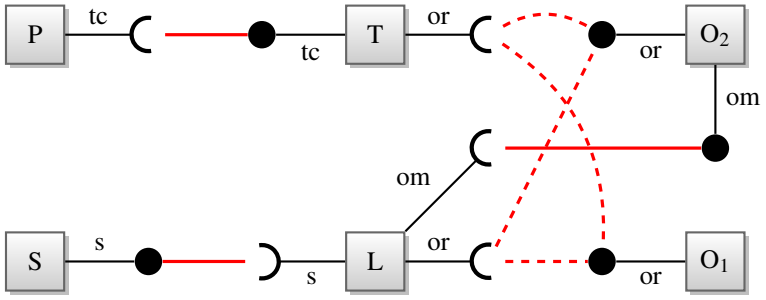

Figure 5: Possible partial configurations for our example. Full lines denote *must* connections while dashed lines denote *may* connections.

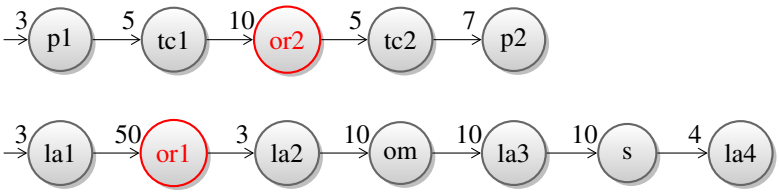

Figure 6: Task graph for one of the two possible functional configurations.

and 75 time units. The constraint on P is relatively loose and in fact it will be satisfied even if the park assist function is blocked by the complete lane assist task chain. Therefore, giving higher priority to the tasks involved in the lane assist function will always provide a feasible configuration. Interestingly, it is also possible to establish that the lane assist chain cannot afford to be blocked by the or1 task. In that case, a constraint stating that or1 must have lower priority than all the tasks involved in the lane assist function can be added to restrict the configuration space.

4.4 Discussion

The main purpose of our example was to illustrate how the contract negotiation process works, and in particular how incompatibility in one viewpoint can result in additional constraints to the configuration space. Therefore we kept the example quite simple. Moreover, note that the methodology is not yet fully implemented: the functional dependency analysis exists but it is not connected to the timing analysis engine at the moment. We currently mainly focus on providing useful timing analysis feedback.

5 Related work

Design of complex systems of components using contracts, or interfaces, has been widely studied since it was introduced by de Alfaro and Henzinger in [1]. A great variety of interface theories have been developed, which mostly focus on incremental design.

There has been active research lately on contract theories dealing with multiple viewpoints. For example, Reineke et al. focus on issues such as *consistency* between viewpoints [13]: if two viewpoints have some degree of overlap, how is it possible to guarantee that they do not contradict each other? Perssone et al. give a high-level classification of model-based approaches to multi-view systems, which also discusses additional challenges of multi-view modeling [11]: traceability of information between

views, reuse, automation, change propagation and extendability. In [10] Panunzio et al. give a more applied contribution as they propose a component model for separation of concerns backed by case studies originating from the European Space Agency. However, none of these results address dynamicity.

Another line of work related to this paper regards languages for contracts. The BCL language [5] has now been used in many safety-critical industrial applications. An alternative is the contract language from [4] which is based on temporal logic. Note however that our interest is less in the language aspects than in the methodology.

Changing constrained systems are in the focus of the FRESCOR project [16], in which contracts are used to ensure Quality of Service on the network-layer. However, the project focuses on bandwidth and latency constraints usually found in multimedia applications. In this context contracts may be broken (at times), enabling the test of novel configurations during runtime. This is not acceptable when updates have an impact also on safety-critical functions such as driver assistance systems in a car. The EPOC project [9] proposed for the first time a contract-based admission control framework for safety-critical systems, but it was restricted to timing aspects and could not handle multiple viewpoints.

6 Conclusion

It has become increasingly difficult to test and verify software updates for embedded systems, in particular because of the complex platform dependencies which exist between software components. In this paper we have presented a methodology to split the necessary effort between lab and field.

Performing in-field formal analysis to determine whether an update may be safely applied is challenging because it requires an automated process able to handle multiple viewpoints such as functional correctness, timing, etc. We have proposed an original methodology based on contract negotiation which enables this. In particular, instead of addressing all viewpoints together we rely on viewpoint-specific analysis engines, each of which uses a model that is best adapted to the particular viewpoint it is dealing with. A second specificity of our approach is that we rely on added capabilities of these viewpoint-specific analysis engines.

Finally, we have illustrated the benefits of our approach on a realistic scenario from the automotive domain, which we unfold from the specification of contracts until a safe update has been found.

References

[1] de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of SIGSOFT'01. pp. 109–120. ACM (2001), http://doi.acm.org/10.1145/503209.503226

[2] AUTOSAR Website, http://www.autosar.org/

[3] Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures. pp. 200–225 (2007), http://dx.doi.org/10.1007/978-3-540-92188-2_9

[4] Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Sci. Comput. Program. 97, 333–348 (2015), http://dx.doi.org/10.1016/j.scico.2014.06.011

[5] Ferrante, O., Passerone, R., Ferrari, A., Mangeruca, L., Sofronis, C.: BCL: A compositional contract language for embedded systems. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014. pp. 1–6 (2014), http://dx.doi.org/10.1109/ETFA.2014.7005353

[6] Genode Website, http://genode.org/

[7] Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., Ernst, R.: System level performance analysis — the SymTA/S approach. In: IEEE Proceedings Computers and Digital Techniques (2005)

[8] Henia, R., Rioux, L., Sordon, N., Garcia, G., Panunzio, M.: Integrating formal timing analysis in the real-time software development process. In: Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development, WOSP-C'15, Austin, TX, USA, January 31, 2015. pp. 35–40 (2015), http://doi.acm.org/10.1145/2693561.2693562

[9] Neukirchner, M., Stein, S., Ernst, R.: The EPOC architecture - enabling evolution under hard constraints. In: Organic Computing - A Paradigm Shift for Complex Systems, pp. 399–412 (2011), http://dx.doi.org/10.1007/978-3-0348-0130-0_26

[10] Panunzio, M., Vardanega, T.: A component-based process with separation of concerns for the development of embedded real-time software systems. Journal of Systems and Software 96, 105–121 (2014), http://dx.doi.org/10.1016/j.jss.2014.05.076

[11] Persson, M., Törngren, M., Qamar, A., Westman, J., Biehl, M., Tripakis, S., Vangheluwe, H., Denil, J.: A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems. In: Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013. pp. 10:1–10:10 (2013), http://dx.doi.org/10.1109/EMSOFT.2013.6658588

[12] pyCPA Website, http://pycpa.readthedocs.org/

[13] Reineke, J., Tripakis, S.: Basic problems in multi-view modeling. In: Proceedings of TACAS'14. LNCS, vol. 8413, pp. 217–232. Springer (2014), http://dx.doi.org/10.1007/978-3-642-54862-8_15

[14] Schlatow, J., Moestl, M., Ernst, R.: An extensible autonomous reconfiguration framework for complex component-based embedded systems. In: 2015 IEEE International Conference on Autonomic Computing, Grenoble, France, July 7-10, 2015. pp. 239–242 (2015), http://dx.doi.org/10.1109/ICAC.2015.18

[15] Sha, L., Rajkumar, R., Lehoczky, J.P., Ramamritham, K.: Mode change protocols for priority-driven preemptive scheduling. Real-Time Systems 1(3), 243–264 (1989), http://dx.doi.org/10.1007/BF00365439

[16] Sojka, M., Hanzálek, Z.: Modular architecture for real-time contract-based framework. In: IEEE Fourth International Symposium on Industrial Embedded Systems - SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8 - 10, 2009. pp. 66–69 (2009), http://dx.doi.org/10.1109/SIES.2009.5196196

[17] Wieder, A., Brandenburg, B.B.: Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In: 8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013. pp. 49–58 (2013), http://dx.doi.org/10.1109/SIES.2013.6601470

[18] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008), http://doi.acm.org/10.1145/1347375.1347389