FLEXIBLE PLATFORM APPROACH FOR CS27/29 FLY-BY-WIRE SYSTEMS

Stephan Korn, Rolf-Rekke Riebeling, Simon Görke, Reinhard Reichel, stephan.korn@ils.uni-stuttgart.de,
University of Stuttgart, Germany

Abstract

Due to high complexity and development costs, implementations of fly-by-wire systems are rarely found in class CS27/CS29-helicopters. This paper presents an approach which is aimed at reducing the development effort and hence allows more cost-effective system realizations. The proposed design process is based on the Flexible Platform technology developed by the University of Stuttgart. This technology is characterized by the following features: 1) The software architecture provides a clear separation between system management (i.e. platform management) and the applications (i.e. flight control laws). 2) The platform management provides transparency of distribution, redundancy, fault tolerance etc. for the applications. 3) It is composed of a generic middleware and model-based upper management layers, both offering a high degree of specialization capability. Consequently, the system management of any fly-by-wire system to be developed can be realized by specialization of the platform management. 4) At implementation level, this specialization appears in form of software components that allow extensive parameterization. Appropriate binding of these implementation-level parameters is performed automatically via a dedicated tool-suite using the system-level input of a systems engineer. The content of this paper is the introduction to the technology of the Flexible Platform, previously developed for the fixed-wing aircraft usage domain. Furthermore, it presents the extension of the Flexible Platform approach to the rotorcraft usage domain. This approach has been validated by the instantiation of a helicopter fly-by-wire system (CS27/29). In doing so, a representative hardware-in-the-loop fly-by-wire demonstrator was realized.

1. Introduction

The complexity of avionics systems in general and of fly-by-wire systems in particular, results from a variety of functional and non-functional reasons.

First of all, there is the complexity of the system functions itself, e.g. flight control or autopilot, interacting with many other systems.

Furthermore, non-functional requirements like safety requirements and the aspect of dissimilarity lead to highly redundant system structures, driving complexity as well.

Finally, the systems have to be operated in several modes such as:

- In-flight mode, pre-flight/aft-flight mode incl. BIT[1] mode,
- Interactive modes for failure reporting, debugging, maintenance, autorigging,
- Simulator modes etc.

It is worth mentioning the increasing degree of functional integration concerning modern system design. In order to reduce costs, more and more system functions will be integrated.

1.1. Avionics Architectures

FIG. 1 shows the classic configuration of a FBW[2] system based on a redundant centralized computer architecture. This structure can be considered representative for many avionics systems.

[1] BIT: built-in test
[2] FBW: fly-by-wire

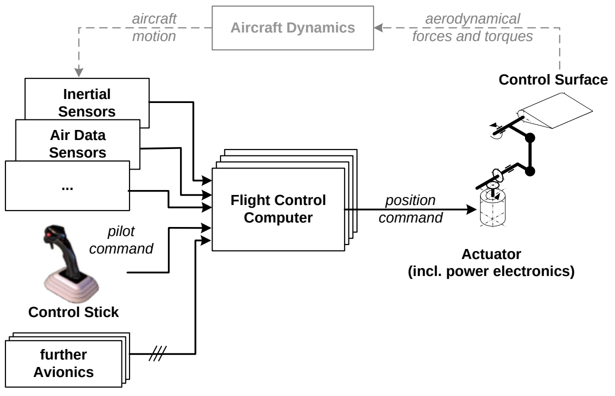

FIG. 1. Classic fly-by-wire system structure (as in [8])

Up until the 1990s, each system respective system function was allocated to a dedicated (simplex or redundant) computer system. The results were federated avionics architectures [1] as depicted in FIG. 2.

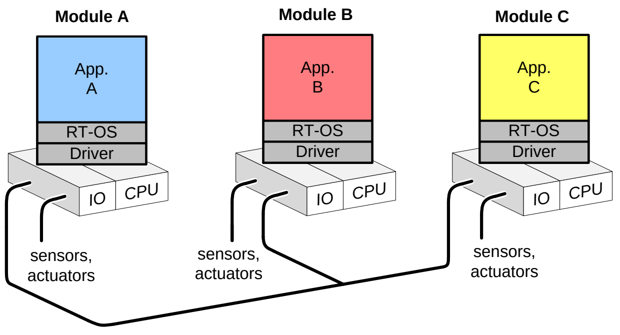

FIG. 2. Federated avionics architecture (as in [8])

Increasing system functionality (more functions, more complexity, more functional co-operation) led to integrated avionics architectures like IMA[3] [2], as applied in the Boeing 777 or the Airbus A380. As shown in FIG. 3, IMA integrates several system functions as applications on common avionic resources (modules). Partitioning means [4] provided by each resource prevent operational interference between different system-functions and allow incremental certification [7].

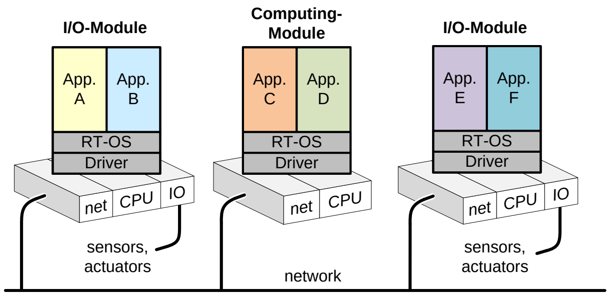

FIG. 3. Integrated avionics architecture – IMA (as in [8])

As a central part of the integrated approach, a uniform API (typically based on the ARINC 653 standard) is provided for all applications. For the applications, the operating system provides abstraction of the underlying hardware, the I/O-interfacing and the communication between avionics resources. These abstraction functionalities are realized in a generic way, so that when an IMA-module is used in a new system, the abstraction layers are specialized to fulfill the given requirements. This form of reusability is a central feature of a platform (as of the Flexible Platform [8]) which benefits cost, development risks and flexibility of the life cycle.

1.2. Flexible Platform

An aspect contributing massively to the complexity of aircraft systems is fault tolerance based on redundancy (replica). This requires a failure/redundancy management with respect to avionics resources, system functions and system aggregates (sensors, actuators). No abstraction layers are provided by integrated avionics architectures like IMA for the failure/redundancy management of the system and its aggregates, and therefore are completely left to the applications. Especially in highly redundant systems, these aspects significantly contribute to complexity, development cost and risk.

The focus of this paper is a platform approach with an advanced integrated avionics architecture. It provides additional abstraction layers (as depicted in FIG. 4) which manage all resources of the avionics architecture in a fault tolerant way without burdening the applications. These abstraction functionalities are part of the Flexible Platform and implemented in the generic platform management.

[3] IMA: Integrated Modular Avionics

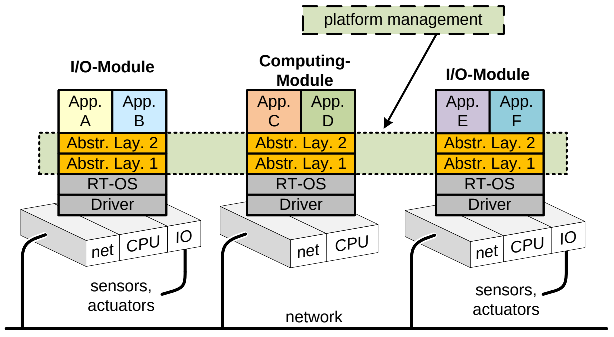

FIG. 4. Integrated avionics architecture based on the Flexible Platform (adapted from [8])

The approach of the Flexible Platform is applicable to even highly safety critical systems such as FBW systems. It is applicable to centralized hardware structures as well as to distributed ones. Key aspects of the Flexible Platform are:

1) *System respective platform instance transparency*
   The platform management comprises all management functions necessary to operate the realized system in a fault tolerant way. It is clearly separated from the laws, i.e. core applications such as flight control laws. Complexity, distribution, fault tolerance and redundancy are fully transparent to the laws. Consequently, the laws can be designed in a "simplex minded" way.
2) *Flexibility of the platform management*
   The platform management software is realized in a widely generic way. Adaption to system-individual requirements is done by specialization, which translates to the parameterization of configurable software components for the predominant share of the platform management. A smaller part of the specialization is achieved by model scaling.
3) *Tool based configuration*
   The more complex a system gets, the more challenging is the task of the platform specialization. Somehow, any application-relevant information has to be reflected in the specialization data. To reduce the effort of the specialization process for the system developer, the implementation-level parameter data is automatically generated. A manually created system description, defining the system's properties at a very high level of abstraction, serves as input for this process. Using this description, a tool-suite performs the instantiation of the parameter data in a multi-step refinement process. Each refinement step is based on a set of meta-models and transformation rules. Together they represent the system- and software-architectural knowledge required for the respective specialization step.

This paper is organized as follows: Section 2 introduces basic definitions as used within the context of the Flexible Platform, the hardware architecture and gives a basic overview on the platform management's functionality. Section 3 covers the software architecture and the internal design of the platform management as previously developed for fixed-wing aircraft applications. Section 4 describes the automatic platform specialization part of the system design process. Finally, sections 5 and 6 cover the extension of the Flexible Platform approach for the rotorcraft usage domain.

2. Platform Architecture Introduction

2.1. Definitions

The basic system design process utilizing a platform approach is depicted in the figure below (FIG. 5). This provides a complement overview for the subsequent terms as they are applied in the context of this paper:

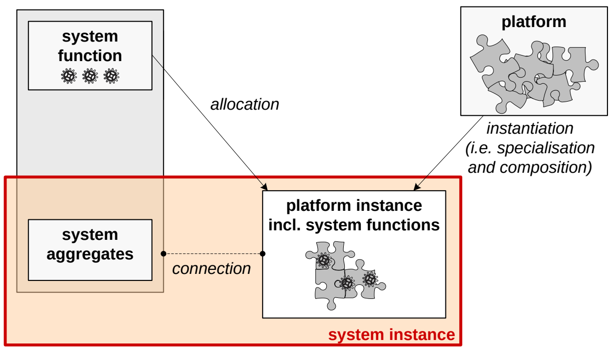
system function
platform
allocation
instantiation (i.e. specialisation and composition)
system aggregates
connection
platform instance incl. system functions
system instance

FIG. 5. Platform-based system design process

- *platform*
  The Flexible Platform can be understood as a library of hardware and software components. The focus of the Flexible Platform is on the software domain: drivers, OS[4] and the platform management are part of the platform. All these generic software components have to be specialized for a system realization.
- *platform instance (pfi)*
  A platform instance consists of 1) a specific arrangement of network-connected hardware modules (*cpm*, *acm*, *iom*, *net* – see subsection 2.2) for a given system realization and 2) its individually specialized platform management software. The independently developed application software (laws) is not considered as part of the instance, but integrated into it.
- *system*
  A system realized as one platform instance consists of 1) its system-function, e.g. a flight control law 2) the platform instance to which the system-function is allocated 3) its aggregates.
- *system-function*
  It is the functional core of the system. It does not contain any system management functions.
- *law (application)*
  A law represents a system-function instantiated as single software component and allocated to one (or more) hardware modules of a *pfi*.
- *system-function path (sfp)*
  A set of *pfi*-modules and aggregates, including a suitable subset of the *pfi*-network, is called a system-function path, as long as it ensures correct operation of the system function at least at minimum performance level. Generally, a *pfi* with redundant modules and aggregates will comprise a set of system-function paths, each having the capability to run the system-function correctly, though possibly at different degrees of performance.
- *mapp*
  Several laws (apps) can be grouped into a mega-application called *mapp*. In case of failure during flight, mapps can be dynamically reallocated among the modules of a *pfi*. This paper's scope is restricted to a single *mapp*, and dynamic reallocation of multiple mapps is not covered in the following.
- *quality of service*[5] *(qos)*
  In the context of this paper, *qos* represents avionics specific metadata characterizing the quality of data (sensor data, output data of a module etc.), quality of performance of aggregates, degree of performance of a system-function path etc. *Qos* is a fundamental part in the generalization of platform management tasks.
- *aggregates*
  Aggregates are sensors, actuators or simply external systems communicating with the platform instance. Aggregates are not part of the platform instance.

Generally, the Flexible Platform approach allows allocation of several systems on one and the same platform instance. As the laboratory FBW demonstrator presented in this paper (section 6) is restricted to a single system, i.e. the flight control system, the presentation in this paper is restricted to single system applications as well.

2.2. Hardware Architecture

To explain the core philosophy of the hardware architecture, a simplified FBW platform-instance and its aggregates are depicted below (FIG. 6) to give an overview for the described terms. The key hardware components of the Flexible Platform are *cpms* (core

[4] OS: Operating System

[5] QoS as definied in this paper

processing modules), *ioms* (input output modules), *acms* (actuator control modules) and *net* (platform network).

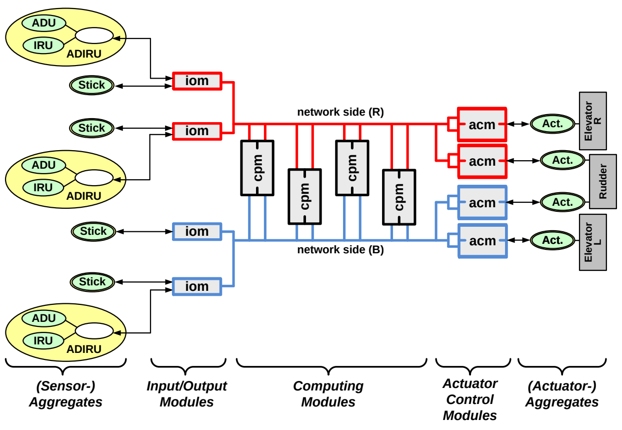

FIG. 6. Platform-instance example with aggregates

cpm (core processing module):

- A *cpm* is built up as a dual lane module.
- The lanes run their software replica in parallel and apply cross-comparison mechanisms in order to achieve fail/passive behavior to a very high degree.
- A *cpm* performs basically the main laws of a system (i.e. flight control laws etc.) and the key services of the platform management.
- A *cpm* is the only module of the platform where every lane has full access to all channels of the platform network. Segregation means are implemented to prevent single point faults.
- A *cpm* has no I/O-interfacing (except the interface to the platform network).

iom (input/output module):

- *Ioms* considered in this paper are single lane *ioms* (one lane per *iom*).
- An *iom* runs parts of the platform management and contributes to the overall platform management.
- An *iom* has only partial access to the platform network. In the demonstrator (section 6) it has access to one dual channel bus.
- An *iom* has the capability to run complex applications.
- The basic services of an *iom* are:

  a) Data acquisition of aggregates (sensors or actuators).
  b) Unification of data representation.
  c) Determination of meta-data, including *qos*.
  d) Metadata representation is unified with respect to syntax and semantics.
  e) Output of data to other systems or aggregates and the corresponding monitoring (e.g. data wrap-around checks).

acm (actuator control module):

- An *acm* is basically a duplex *iom*. It is optimized with respect to actuator control and monitors the actuator(s) and itself in order to provide fail/passive characteristics.
- Input-/output data as well as metadata will be unified with respect to their presentation. This is done analogously to *ioms*.
- An *acm* runs parts of the platform management and contributes to the overall platform management.
- An *acm* has only partial access to the platform network. In the demonstrator (section 6) it has access to one dual channel bus.
- An *acm* has the capability to run complex applications such as actuator control loops, analytical models of actuators. etc.

net (network):

- Each platform instance consists of (at least) two separate networks. For recent applications, the flexray[6] bus technology was chosen. In this case, each network consists of two redundantly operated communication channels.
- Only *cpms* are allowed to have access to both networks.
- In the laboratory demonstrator (section 6), an additional (third) network has been added to perform the communication with the less safety critical part of the avionics suite.

Assignment to sides:

- The system can be structured according to the two networks, i.e. into side(B) and side(R)[7].
- *Ioms* and *acms* are linked to only one network. Their side-assignment will be accordingly.
- Although a *cpm* is linked to both networks, each *cpm* is assigned to one side as well.
- In a flexray-based platform instance, modules assigned to one side will be synchronized with the corresponding flexray bus.

2.3. Platform Management

The major task of the platform management is to ensure a correct active system-function path in spite of faults or failures in aggregates or the platform instance. This requires services at different levels of system platform operation:

- Management of redundant sensors or redundant data sources, respectively.
- Management of redundant actuators or redundant actuator valves, respectively.

---

[6] The Flexible Platform is generally not limited to time-triggered bus technology (such as flexray).
[7] B…Blue, R…Red

- Management of *pfi*-modules with respect to detection of faulty modules and passivation of them.
- Redundancy management of all *cpms* of the so-called *pfi*-core.

As many decisions are made in the *pfi*-core this aspect will be the focus of this section.

Concerning an effective realization of dissimilarity, it was decided to rely on an active/standby replica control philosophy for *cpms*. All *cpms* run basically the same tasks but the commands of only two *cpms* are transmitted to the *acms*. One *cpm* obtains the "active" or "master" state, denoted as *cpm(m)*. Another *cpm* of the opposite side will be selected as *cpm(sl)* representing the "standby" or "slave" state. All *cpms* not selected as *cpm(m)* or *cpm(sl)* will become "shadow" *cpm*, denoted as *cpm(sh)*. They operate in "standby" similar to the slave but without transmitting commands to the actuators.

For *cpm(m)* and *cpm(sl)*, the *acms* monitor the received commands and return the corresponding evaluated status to the *cpms*. Concerning local control, only the *cpm(m)* commands are used by the *acms* further on.

Cpm(m) and *cpm(sl)* evaluate by means of the status feedback of *ioms* and *acms* in combination with further *qos* data the degree of performance for their individual system-function path (*sfp*), represented by *qos(sfp)*. In fault-free condition, the following shall be true:

$$qos(sfp)_{master} \geq qos(sfp)_{slave}$$

As long as this condition holds, no reconfiguration will be done. Consequently, the main rules for *cpm*-replica control can be derived:

- There must be a single *cpm* in status master providing the maximum *qos(sfp)* compared to *cpm(sl)*.
- If a master is established and there is a correct *cpm* available on the other side, this *cpm* has to switch into the slave status.
- All other *cpm* have to switch into the shadow status.
- If *cpm(m)* is lost due to a fault or if $qos(sfp)_{master} < qos(sfp)_{slave}$ holds, then *cpm(sl)* initiates transition to become *cpm(m)*.
- If *cpm(m)* is lost and there is no *cpm(sl)*, a *cpm(sh)* of the same side will initiate transition to become *cpm(m)*.

With respect to the master/slave/shadow-reconfiguration, FIG. 7 and FIG. 8 depict two different reconfiguration scenarios. The first scenario shows the platform reaction on module failures, the second scenario concerns inconsistent communication failures between *cpm(m)* and *acms* or *cpm(sl)* and *acms*, respectively.

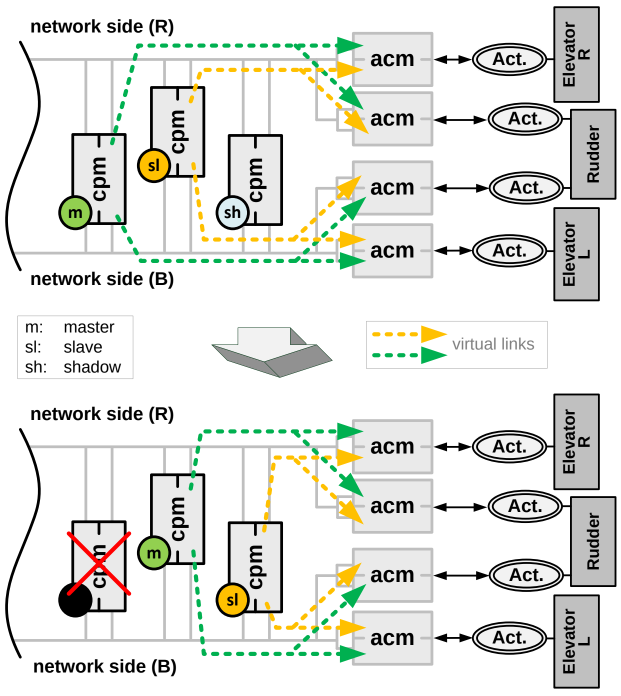

FIG. 7. Master reconfiguration due to *cpm* faults

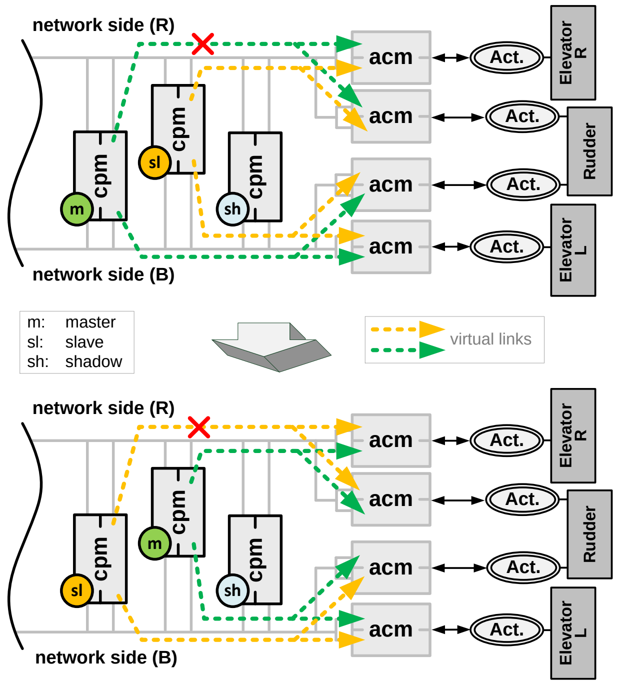

FIG. 8. Master reconfiguration due to an inconsistent failure between *cpm(m)* and *acm*

The end-to-end communication within a platform instance is based on so-called virtual links. With respect to the *cpms*, these virtual links are not related to the modules but rather to the respective state as "master" or "slave" (FIG. 7 and FIG. 8). This grants the capability to reconfigure the virtual links during flight from a set of preconfigured links.

Decisions such as the determination of the status master/slave/shadow cannot be done by a single *cpm*. They have to be achieved by all *cpms* together, denoted as “distributed replica control”. This requires consensus [9] properties between the *cpms* under the constraint that *cpms* are not synchronized[8]. Basically, consensus is achieved as follows:

1. The application of broadcast and data evaluation mechanisms in all *cpms* featuring reliable broadcast (according to [3] & [9]) between all *cpms*.
2. Implementation of consensus mechanisms relying on item 1. and taking the “asynchronism” between the *cpms* into account.

In this way, consensus is provided for all *pfi*-relevant decisions and ensures consistent and correct *pfi*-operations.

In summary, these platform management mechanisms have the following significant impacts on the platform characteristics:

- Due to the utilization of dynamic virtual links, the complexity of the *pfi*-core, i.e. the number of *cpms* and the arrangement of *cpms*, is fully transparent to *acms* and *ioms*. This supports the scalability of the system and simplifies the design of *acms* and *ioms*.
- The master/slave/shadow replica control facilitates the implementation of highly credible dissimilarity. In particular, it enables the use of two different types of *cpms*, type(A) providing the maximum *qos(sfp)*, type(B) providing only a reduced *qos(sfp)*, without any changes in platform management mechanisms.

3. Software Architecture

FIG. 9 gives an overview of the layered software architecture of a module lane as described in the following sections.

3.1. OS and Drivers

As part of the OS, the driver infrastructure and communication stacks provide means for module hardware access out of the platform management middleware, i.e. an implementation of the OSI-Layers 2, 3 and 4. This covers:

- X-lane[9]-communication within a redundant module.
- Network communication between modules.
- Data bus interfacing of other avionics domains or complex sensors (e.g. IRUs[10]).

[8] “Not synchronized” means in this context: The services of different *cpms* are not synchronized with each other – they do not refer to a common global time.

[9] x-lane: Cross-Lane or inter-lane

[10] IRU: Inertial Reference Unit

- Access to plain aggregate hardware such as position pick-offs or electro-mechanical switches.

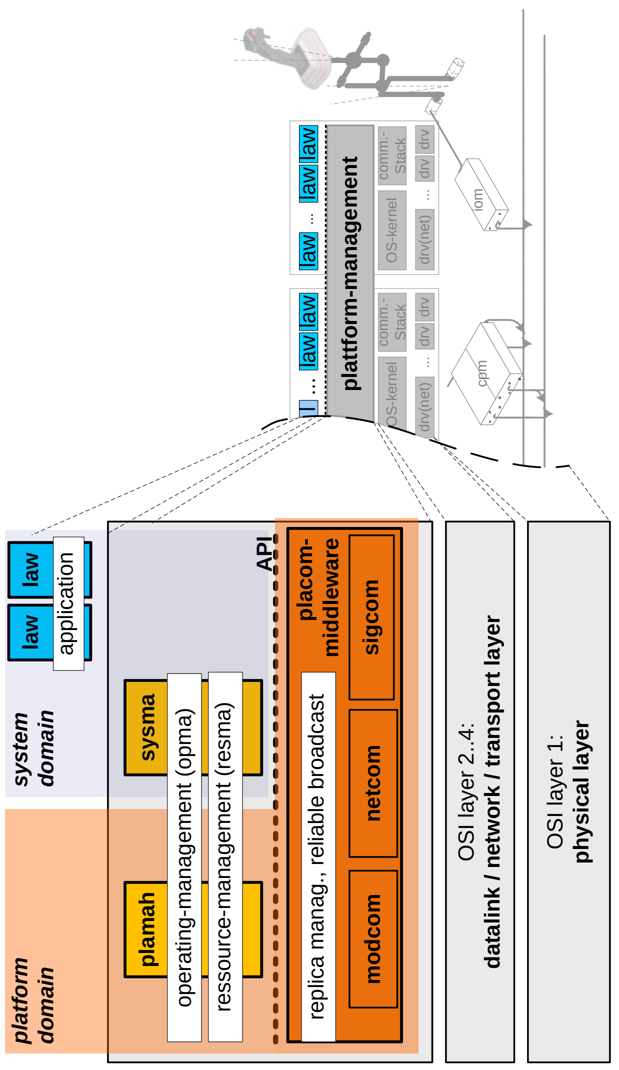

FIG. 9. Overview of the software architecture (as in [8])

3.2. Platform Communication Middleware (placom)

A specific platform communication layer (*placom*) is placed on top of OSI layer 4. It represents the major part of the platform management in the *pfi* and basically covers three communication domains:

- *modcom*
  This function realizes x-lane-related tasks within an internally redundant module, i.e. cpm or acm. Basically, it comprises mechanisms for cross-comparing the lanes’ data and ensuring their consistency.
- *netcom*
  This domain covers the management of inter-module communication within a *pfi* on top of OSI-layer 4 (see FIG. 9). The main tasks of *netcom* are:

a) Communication failures or failures of the data sources must not contaminate the receiving module.
b) Reliable broadcast [9] is achieved between *cpms*.
c) Metadata for each network and each module are generated in unified representation with respect to syntax and semantics.

- *sigcom*
  This function performs the acquisition of data and metadata of aggregates and their transformation to simplex data with *qos* in a fault tolerant way. The specific tasks are:
  a) Evaluation of operating status and failure messages provided by the aggregate (e.g. sensor) or by components involved in the signal transfer path beginning at the original source up to each particular signal sink (e.g. *pfi*-core).
  b) Using voting functions to generate a single signal from redundant signals.
  c) Monitoring the signals and adapting the voting/monitoring mechanisms.
  d) Providing all metadata in order to prepare reconfiguration decisions, i.e. permanent passivation, intermediate passivation with reacceptance in the membership, BIT control, etc.
  e) Providing *qos* information for each signal or signal group, respectively.
  f) Unification of data with respect to syntax and metadata with respect to syntax and semantics.
  g) Routing of data including *qos* information to each designated application and its API, respectively.

The design of *sigcom* turned out to be a very challenging task. The major challenge is that the semantics of metadata has to be interpreted in order to meet this task in a very generic way with a high degree of flexibility for specialization. This has to be done against a background of high degrees of diversity of sensor types and signal transport paths between aggregates and *pfi*-core.

3.3. High Level Platform Management (plamah)

Based on the *placom*-middleware and the respective API, *plamah* establishes the following properties:

- *Activation of system-function paths*
  By determining the master/slave/shadow status for each module, *plamah* contributes to the selection of the active system-function paths within the *pfi*.

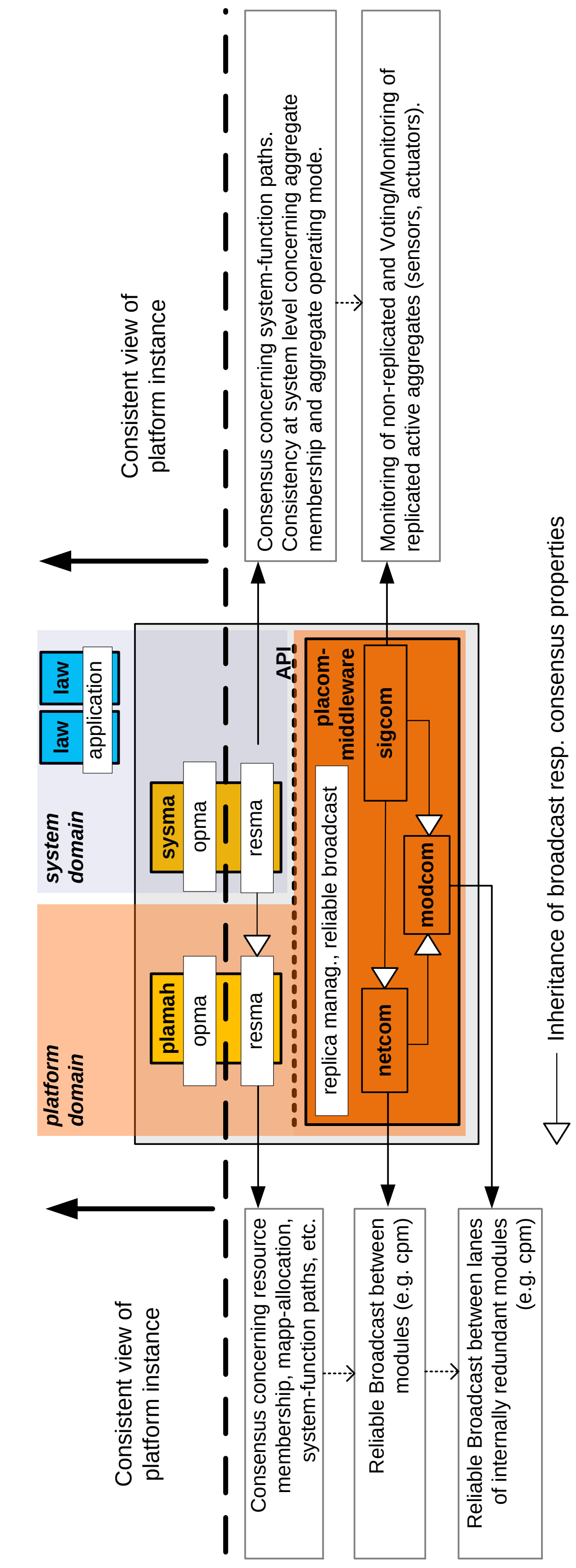
Consistent view of platform instance
Consensus concerning system-function paths. Consistency at system level concerning aggregate membership and aggregate operating mode.
Monitoring of non-replicated and Voting/Monitoring of replicated active aggregates (sensors, actuators).
system domain
law
law
application
sysma
opma
resma
API
placom-middleware
sigcom
modcom
platform domain
plamah
opma
resma
replica manag., reliable broadcast
netcom
Inheritance of broadcast resp. consensus properties
Consistent view of platform instance
Consensus concerning resource membership, mapp-allocation, system-function paths, etc.
Reliable Broadcast between modules (e.g. cpm)
Reliable Broadcast between lanes of internally redundant modules (e.g. cpm)

FIG. 10. Layers and their contribution to consistency (taken from [8])

- *module membership management*
  The middleware provides for all modules of the *pfi* (independent of their membership status) *qos* information at the *placom* API (see last section and FIG. 9). Based on this information and the properties of the reliable broadcast between *cpms*, *plamah* establishes consensus between all correct *cpms* concerning the membership of all *pfi*-modules.
- *mapp activation*
  Each *cpm* can load several *mapps*, but only one specific *mapp* is executed per *cpm*. Based on membership consensus, *plamah* provides consensus concerning *mapp* activation between all correct *cpms*.

The relationship between the different software layers with respect to reliable broadcast and the different consensus properties is shown in FIG. 10. The tasks mentioned above establish a consistent operating status in the *pfi*. Based hereon, management decisions are executed concerning global operating modes (e.g. BIT mode) by the operating management (*opma*).

3.4. System management (sysma)

The key tasks of the system management *sysma* are as follows:

- It acts as the interface between the highly generic *plamah* and the highly application specific system aspects.
- It performs the final decisions about long-term membership of redundant aggregates concerning the related system.
- As system-functions are basically run in a semi active way (master/slave/shadow), *sysma* performs the adaption of the slave and shadow replica with respect to the master replica.
- It transforms respectively supplements operating mode commands of *plamah* for the system level.

4. Platform Specialization

With platform-based design approaches, system development basically translates to tailoring the platform components to the system-specific requirements. Consequently, this specialization task represents the pivotal development effort.

4.1. Introduction to Specialization

Resulting from fundamental differences in design, the individual parts of the Flexible Platform software are specialized in three different ways (see FIG. 11):

- model adaption
- model scaling
- parameterization

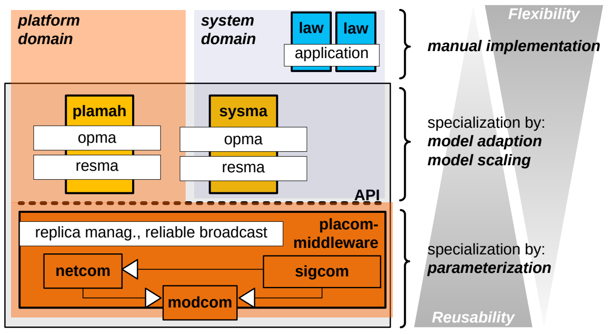

FIG. 11. Specialization types within the Flexible Platform software (as in [8])

Specialization of the model-based *plamah*/*sysma*-part is mainly achieved by model-scaling.

In contrast to this, the *placom*-layer is specialized by generation and composition of configuration data (parameterization). This comprehends in detail:

- Composition of modules.
- Definition of module functionality.
- Scheduling of communication between SW-modules and between HW-modules (for distributed systems).

Generation of the *placom*-layer is a challenging task with respect to parameter-quantity. Instantiation of a large avionics system typically means defining several hundred thousand partly cross-dependable middleware parameters. Manual handling seems unfeasible in practice.

Automation of the middleware-instantiation process offers a solution to this issue.

In order to reduce the effort of defining input data by the systems engineer, the input specification is performed on system level. This was achieved by the development of a tool-suite offering a high level of abstraction for input data.

The tool-suite applies a multi-step refinement process using the input specification to derive the desired output data at source code level. Starting with these highly abstract data, any further instantiation is conducted automatically by algorithmic rules.

4.2. Instantiation Process

The following figure (FIG. 12) is an overview for the following sections, showing the instantiation process, characterized by four individual steps.

4.2.1. Input Specification

The input for the instantiation process is manually created by a systems engineer. For this specification task, a dedicated domain-specific language (DSL) was defined using a meta-modeling approach.

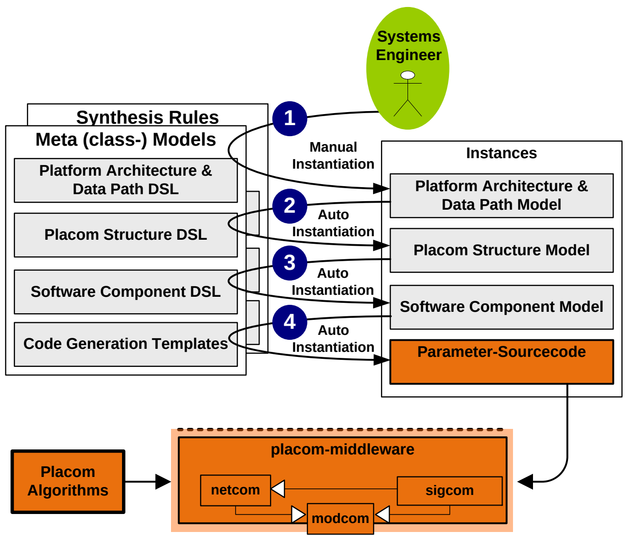

FIG. 12. Instantiation of the *placom*-middleware (as in [8])

Domain-specific means that the language provides description elements well-known to the systems engineer. Hence, the abstraction level focuses on a system-engineering perspective, not a software-engineering view. Basically, this DSL provides means to describe the following aspects:

- The structure of the system instance hardware (modules, sensors, actuators, networks).
- Aggregates can be split into or composed of so called granulates, which are the smallest units managed separately by the platform management.
- All sensor and actuator interfaces, all API of all modules (*iom*, *acm*, *cpm*).
- Virtual links.
- The allocation of interfaces, laws, part of management mechanisms onto the hardware modules respectively structure.

The following figure (FIG. 13) depicts as an example, a simple input specification being progressively transformed to parameter source code as described in the further instantiation process steps.

4.2.2. First Auto-Instantiation Step

The first auto-instantiation step has the manually created specification as its input. Basically, the synthesis rule set of this first stage augments this input with structural knowledge of the *placom*-internal architecture. This interim transformation process generates artifacts with an instance-wide scope.

These artifacts belong to a dedicated DSL which is designed for describing *placom*-internal processing as a sequence of functional black-boxes comprising:

- *Sigcom* "segments" distributed on the modules
- Interconnecting *netcom* "segments"

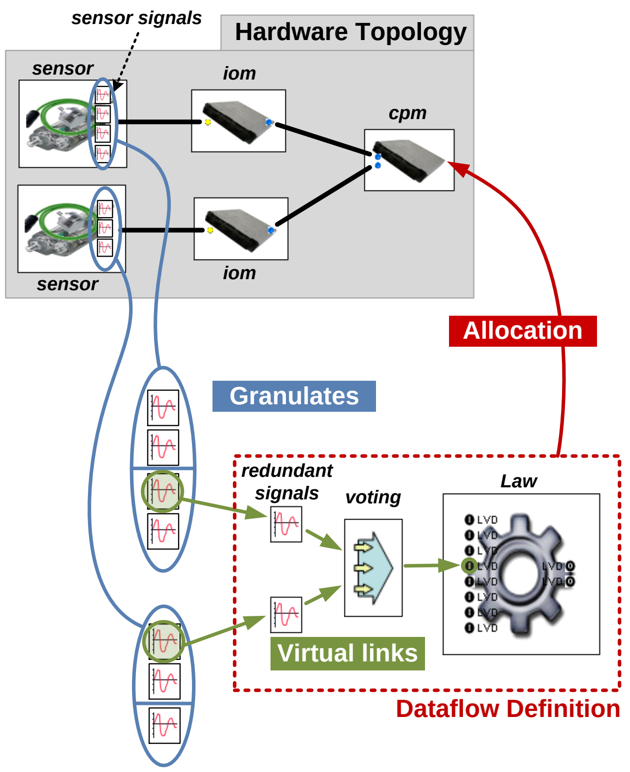

FIG. 13. Systems engineer inputs to the tool-suite for a very simplified system example

The artifacts also contain:

- All inter-module communication messages
- All sensor / actuator messages

One example is the automatic insertion of *netcom*-functionality, whenever a system-function path crosses from one module to another (FIG. 14).

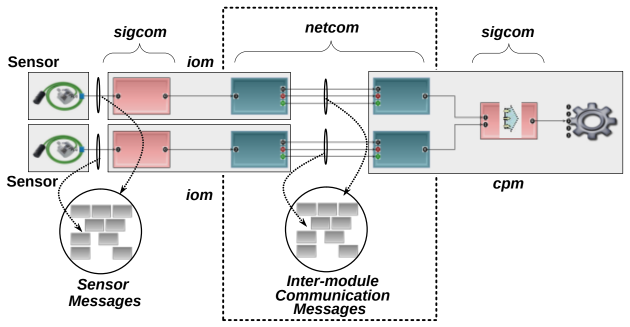

FIG. 14. Automatic *placom* structure instance generation (according to example in FIG. 13)

4.2.3. Second Auto-Instantiation Step

The second auto-instantiation step comprises rules which transform the black-boxes created in the preceding step. The black-boxes are decomposed to software components and they in turn are specialized by defining their parameterization. Some parameters are mapped unaltered from the initial input, but the majority is deduced from user specifications or architectural relations. The results of the transformation are stored as an instance of a third DSL (FIG. 15):

- SW-components and parameterization for *modcom*, *sigcom*, *netcom*
- Derived parameter data (e.g. memory layout)

This language formally specifies the degrees of freedom for each software component with a per-lane scope.

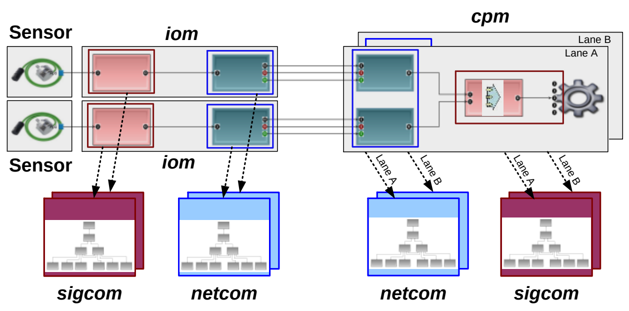

FIG. 15. Automatic software component instance generation (example from FIG. 13)

4.2.4. Code Generation

The last transformation step translates instances of the software component DSL into source code representing the component parameterization (FIG. 16). Practically, this is a one-to-one conversion which merely alters their representation.

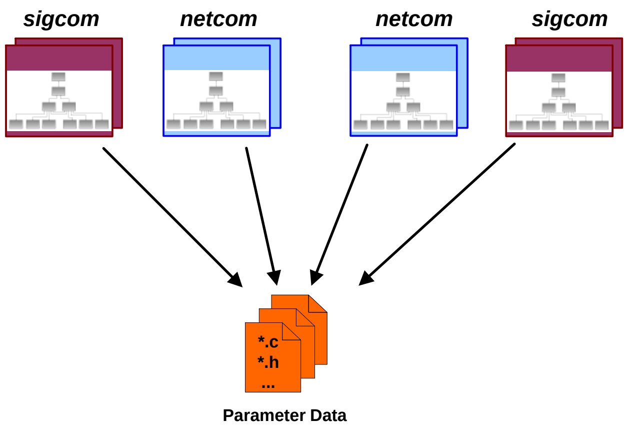

FIG. 16. Automatic parameter data generation (example)

5. Flexibility Frame

Originally, the usage domain of the Flexible Platform covers fixed-wing aircraft applications. In order to make the Flexible Platform applicable to rotorcraft applications, the usage domain has to be extended appropriately. The major extensions are driven by:

FC-Laws

- Basic flight control laws mainly comprising basic stabilization, axes decoupling and partly “command & hold” functionality.
- Enhanced flight control laws including full “command & hold” functionality.
- Conventional autopilot modes as well as specific SAR modes (e.g. hover, ground speed mode).

The frame of FC-laws fixes the set of sensors to be applied to a flight control system as well.

Actuator arrangement

Each of the actuators – three for the main rotor and one for the tail rotor – is a hydraulic actuator controlled by means of four direct drive motors in “all active mode”.

Safety and Dispatchability

- P(loss of basic flight control) < 10^{-9} per flight hour
- P(loss of enhanced flight control) < $10^{-5}..10^{-9}$ per flight hour
- System extensions shall be possible in order to allow dispatch in case of any single fault in the electronics.
- Robustness against generic faults shall be based on dissimilarity.

System Aspects

The system architecture can be considered to be covered widely by the fixed-wing aircraft usage domain of the Flexible Platform – except the actuation area. This specific area will be considered in more detail in the following chapter.

6. Helicopter FBW Demonstrator

6.1. System Structure

A laboratory helicopter FBW demonstrator (as described in detail in [10]) is realized as an instance of the Flexible Platform. Compared with the exemplary platform instance shown in FIG. 6, an additional flexray bus (side(G)) as well as additional *ioms* are added (FIG. 17).

All actuators of the main rotor are controlled by an actuator electronics consisting of a quad-duplex *acm* arrangement. The four *acms* communicate with each other via a separate x-*acm* communication bus[11]. The actuator of the tail rotor is controlled by another quad-duplex *acm* arrangement.

6.2. Specific operating Aspects

This chapter shall provide insight into selected platform management aspects as applied in the demonstrator.

6.2.1. Sensor Management

In the demonstrator, sensor data as well as data from other systems are handled by the *ioms*. Relating to *sigcom* functionality as in section 3.2, this comprises data acquisition, unification of data

[11] The inter-*acm* communication is a separate flexray bus. For real applications this approach will not meet the safety requirements. Its substitution by another communication means will only marginally affect the platform management mechanisms.

representation and monitoring of non-redundant data.

Concerning Voting/Monitoring of redundant data, the main share is performed by the *cpms*, covering the sensor domain, and a smaller share by *acms*, covering actuator-specific data.

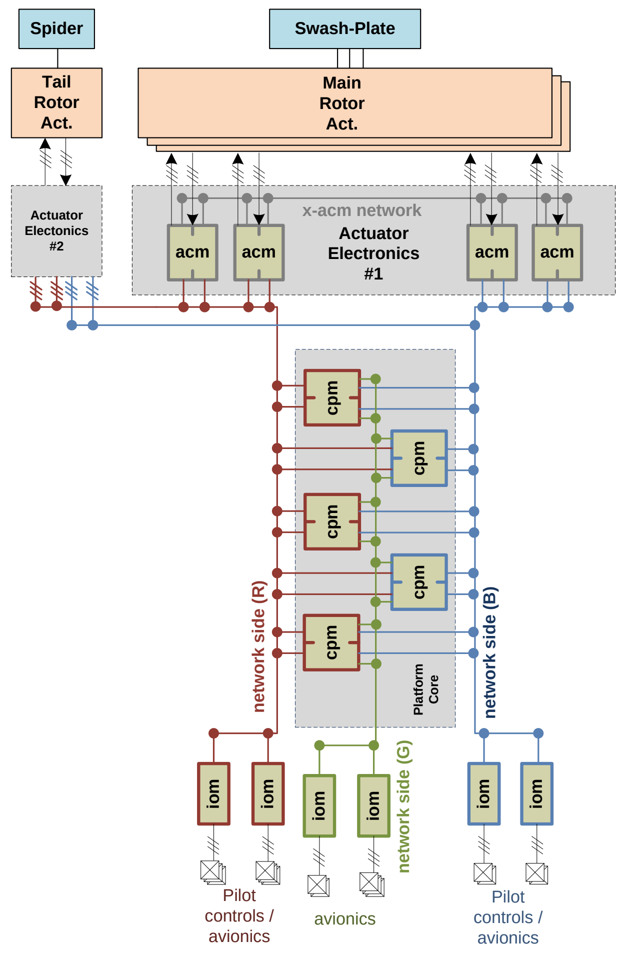

FIG. 17. Platform instance of laboratory demonstrator [10].

6.2.2. *pfi*-Core Management

As described in chapter 2.3 the control of *cpm*-replica is based on a master/slave/shadow philosophy.

FIG. 18 shows a reconfiguration of *cpms* with respect to *cpm*-faults, applying the reconfiguration rules of chapter 2.3. FIG. 19 shows a scenario with respect to an inconsistent failure, such that the commands of *cpm(m)* are not received correctly any more by some *acms* of one actuation core but *cpm(sl)* is still operating without any failure in its system-function-path.

Once this failure scenario happens, the overall qos of *cpm(m)* is reduced resulting in $qos(sfp)_{master} < qos(sfp)_{slave}$.

Consequently, the reconfiguration according to FIG. 19 restores the maximum possible *qos* in the system.

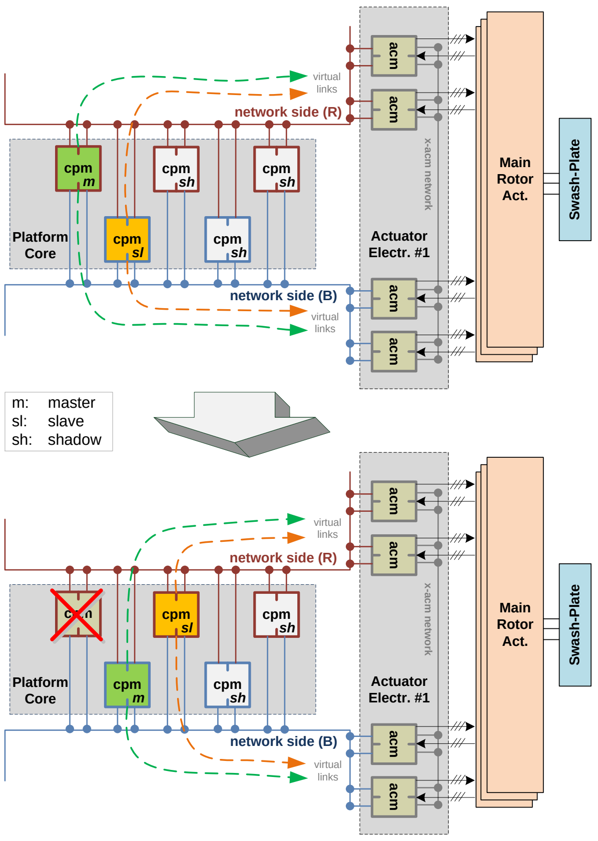

FIG. 18. *Cpm*-reconfiguration with respect to *cpm*-faults

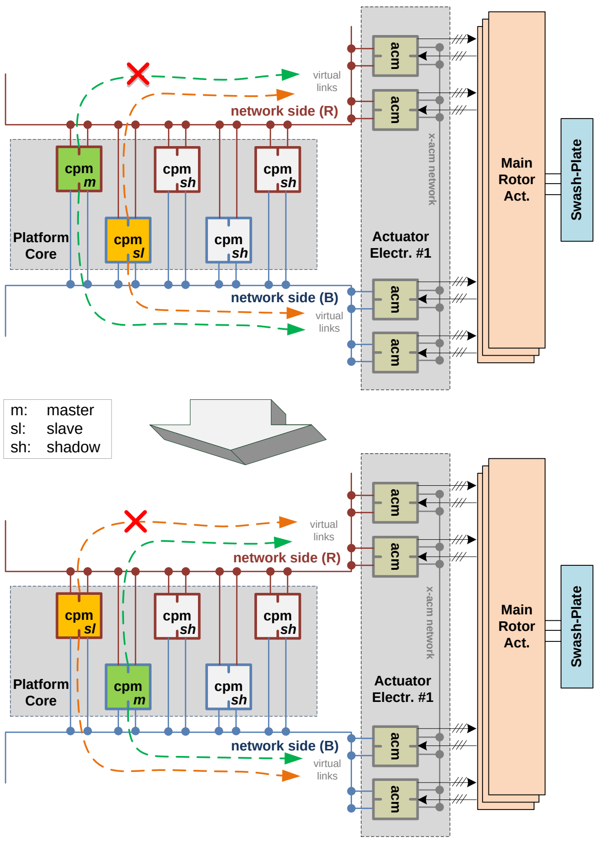

FIG. 19. *Cpm*-reconfiguration with respect to performance degradation in the active system-function-path

6.2.3. Operation of quad-duplex *acms*

Contrarily to *cpms*, replica control of the quad-duplex *acms* is performed “all active”. In order to allow for efficient *acm* monitoring and to prevent any force-fighting in the actuator, the four redundant *acm* actuator commands have to agree to a very high degree. In order not to overburden the x-*acm* communication or the cpu-performance in the *acms* the following strategy has been selected:

- The control loop is split into a high frequency inner loop and a low frequency outer loop.
- All *acms* are synchronized via the x-*acm* communication bus.
- With respect to outer loops, consensus mechanisms are implemented in the *acms* such that exact agreement of reference and command values is ensured between the lanes of each *acm* and between the *acms* even in case of byzantine (inconsistent) failures between *acms* or inconsistent failures between *cpm(m)* and *acms*, respectively.
- With respect to inner loops, x-*acm* data are exchanged at outer loop frequency as well. Consequently, additional x-*acm* mechanisms are implemented to ensure precise (not exact) agreement of output commands generated by the inner loops.
- In case of an *acm*-fault, the affected *acm* passivates itself. Basically, the passivation of an *acm* can be initiated by itself or by the majority of the other (not passive) *acms*.
- In case of an inconsistent failure between *cpm(m)* and the *acms*, all *acms* continue operation but the system might react with a reconfiguration of the master status in the *pfi*-core (see FIG. 19).

7. Conclusion

The approach of the Flexible Platform is based on the powerful middleware, allowing specialization by parameterization, and the model-based upper management layers, allowing specialization by model scaling. In particular, the specialization of the comprehensive middleware has revealed to be an extraordinary complex task. Automation of the middleware-instantiation process offers a solution to this issue. This is achieved by the development of a tool-suite offering a high level of abstraction for input data at system level. The tool-suite applies a multi-step refinement process using the high level input specification to derive the desired specialization output data at source code level. This is done for the middleware as well as the OS and all SW-drivers of the complete platform instance, i.e. all *cpms*, *ioms* and *acms*.

Through years of research, the Flexible Platform approach has been applied to different demonstrators of fixed-wing aircraft applications (laboratory and inflight demonstrators) and even automotive applications (x-by-wire systems implemented in prototype cars and trucks tested on test circuits). All these demonstrators show a degree of complexity close to real product applications. Thereby, it has proven that the instantiation of a new platform management instance can be achieved to a high degree simply by specialization, i.e. in a very efficient way.

The paper extends the fixed-wing aircraft usage domain of the Flexible Platform to a rotorcraft usage domain. This required one-time modifications of the upper management layers, the middleware and the tool-suite. In spite of this particular additional one-time effort, the approach of the Flexible Platform has reconfirmed its efficiency in installing a new FBW system.

Abbreviations

acm – actuator control module

API – application programming interface

BIT – built-in test

cpm – core processing module

FBW – fly-by-wire

HMI – human machine interface

i/o – input/output

iom – input/output module

mapp – mega applicaton

opma – operation management

OS – operating system

pf – platform

pfc – primary flight control

pfi – platform instance

placom – platform communication layer

plamah – high-level platform management part

QoS – quality of service

sfp - system function path

SW – software

sysma – system management

References

[1] C. Watkins, “Integrated Modular Avionics: Managing the Allocation of Shared Intersystem Resources“, in 25th Digital Avionics System Conference, 2006 IEEE/AIAA, 2006, pp. 1-12.

[2] C. Watkins and R. Walter, “Transitioning from federated avionics architecture to Integrated Modular Avionics”, in Digital Avionics Systems Conference, 2007.DASC ’07. IEEE/AIAA 26th, 2007, pp. 2.A.1-1.

[3] V. Hadzilacos and S. Toueg, “A Modular Approach to Fault-Tolerant Broadcasts and Related Problems”, Ithaca, NY, USA, 1994.

[4] J. Rushby, “Modular Certification”, SRI International, Menlo Park, CA, Tech. Rep. NASA/CR-2002-212130, Dec. 2002.

[5] J. Lewis and L. Rierson, “Certification concerns with integrated modular avionics (IMA) projects”, in 22nd Digital Avionics Systems Conference, DASC ’03, 2003.

[6] G. Bartley and B. Lingberg, “Certification concerns of Integrated Modular Avionics (IMA) systems”, in Digital Avionics Systems Conference, DASC ’08, IEEE/AIAA 27th, 2008.

[7] A. Wilson and T. Preyssler, “Incremental certification and Integrated Modular Avionics”, in Digital Avionics Systems Conference, DASC ’08, IEEE/AIAA 27th, 2008.

[8] R. Reichel, S. Görke, F. Cake, S. Polenz, R. Riebeling, „Flexible Avionics Platform“, in *Proceedings of the 61. German Aerospace Congress (DGLR)*, Berlin, 10.-12. September 2012.

[9] G. Coulouris, J. Dollimore, T. Kindberg, G. Blair, “Distributed Systems - Concepts and Design”, Addison-Wesley, 5th Edition, 2012.

[10] N. Bickel, J. P. Klaubert, M. Hammerlindl, S. Korn, R. Reichel, R. Riebeling, “Design Validation of a new Generic Fly-By-X Flight Control System for Helicopters”, in *Proceedings of the European Rotorcraft Forum*, Moscow, 03.-06. September 2013.

Acknowledgments

This research was funded with support of the German Federal Ministry of Economics and Technology.

Gefördert durch:

Bundesministerium
für Wirtschaft
und Technologie

aufgrund eines Beschlusses
des Deutschen Bundestages

Copyright Statement