# A Step-wise Approach for Integrating QoS throughout Software Development

Stéphanie Gatti, Emilie Balland, Charles Consel

# A Step-wise Approach for Integrating QoS throughout Software Development

Stéphanie Gatti and Emilie Balland and Charles Consel

Thales Airborne Systems / University of Bordeaux / INRIA, France
first-name.last-name@inria.fr

**Abstract.** When developing real-time systems such as avionics software, it is critical to ensure the performance of these systems. In general, deterministic Quality of Service (QoS) is guaranteed by the execution platform, independently of a particular application. For example, in the avionics domain, the ARINC 664 standard defines a data network that provides deterministic QoS guarantees. However, this strategy falls short of addressing how the QoS requirements of an application get transformed through all development phases and artifacts. Existing approaches provide support for QoS concerns that only cover part of the development process, preventing traceability.
In this paper, we propose a declarative approach for specifying QoS requirements that covers the complete software development process, from the requirements analysis to the deployment. This step-wise approach is dedicated to control-loop systems such as avionics software. The domain-specific trait of this approach enables the stakeholders to be guided and ensures QoS requirements traceability via a tool-based methodology.

**Keywords:** Quality of Service, Domain-Specific Design Language, Tool-Based Development Methodology, Generative Programming.

## 1 Introduction

Non-functional requirements are used to express the *quality* to be expected from a system. For real-time systems such as avionics, it is critical to guarantee this quality, in particular time-related performance properties. For example, the avionics standard ARINC 653 defines a Real-Time Operating System (RTOS) providing deterministic scheduling [3] and thus ensuring execution fairness between applications. Another example is the ARINC 664 that defines Avionics Full DupleX switched Ethernet (AFDX), a network providing deterministic Quality of Service (QoS) for data communication [4]. In this domain, deterministic QoS is generally ensured at the execution platform level (*e.g.,* operating systems, distributed systems technologies, hardware specificities), independently of a particular application. When addressing the QoS requirements of a given application, these platform-specific guarantees are not sufficient.

There exist numerous specification languages to declare QoS requirements at the architectural level [1]. Initially, these languages were mostly contemplative. Several recent approaches also provide support to manage specific aspects (*e.g.,* coherence checking [10], prediction [20], monitoring [17]). These approaches are generally dedicated to a particular development stage, leading to a loss of traceability (*i.e.,* the ability to trace all the requirements throughout the development process). In the avionics certification processes [11,12,5], traceability is mandatory for both functional and non-functional requirements. The functional traceability is usually ensured by systematic development methodologies such as the V-model that guides stakeholders from the requirements analysis to the system deployment. Similarly, QoS should be fully integrated into the development process as it is a crosscutting concern [19].

In this paper, we propose a step-wise QoS approach integrated through all development phases and development artifacts. This approach is dedicated to control-loop systems. Control-loop systems are systems that sense the external environment, compute data, and eventually control the environment accordingly. This kind of systems can be found in a range of domains, including avionics, robotics, and pervasive computing. For example, in the avionics domain, a flight management application is a control-loop system that (1) senses the environment for location and other navigation information, (2) computes the trajectory and (3) modifies the wings configuration accordingly. The contributions of this paper can be summarized as follows.

*A step-wise QoS approach dedicated to control-loop systems.* We propose a step-wise approach that systematically processes QoS requirements throughout software development. This integrated approach is dedicated to control-loop systems, allowing to rely on a particular architectural pattern and thus enhancing the design and programming support level for non-functional aspects. In this paper, we focus on time-related performance but the approach could be generalized to other non-functional properties (*e.g.,* CPU or memory consumption).

*Requirements Traceability.* In the avionics domain, the traceability of both functional and non-functional requirements is critical [11]. In our approach, the traceability is ensured by the systematic propagation of constraints derived from the QoS declarations and applied to each development step.

*A tool-based methodology.* Our approach has been integrated into DiaSuite, a tool-based development methodology dedicated to control-loop systems [8]. DiaSuite is based on a dedicated design language that we have enriched with time-related performance properties. This non-functional extension has been used to offer verification and programming support at each development stage.

*Experiments in the avionics domain.* Our approach has been applied to the development of various avionics applications, including a flight management system and a collision avoidance system. These experiments have demonstrated that our step-wise approach can effectively guide the avionics certification process.

## 2 Background & Working Example

This section presents a working example used throughout this paper. This presentation is done in the context of the DiaSuite development methodology [8]. We choose a control-loop system from the avionics domain: a simplified version of an aircraft guidance application, controlling the trajectory of an aircraft by correcting the configurations of ailerons.

### 2.1 Overview of the DiaSuite Approach

The DiaSuite approach is a tool-based methodology dedicated to control-loop systems. DiaSuite provides support for each development stage (from design to deployment) as depicted in Figure 1.

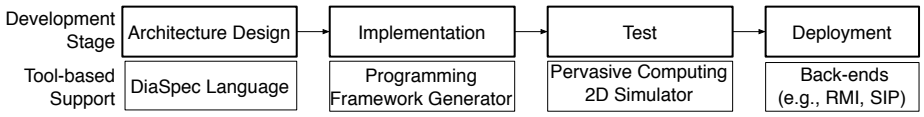


Fig. 1: The DiaSuite tool-based development process

During the design stage, the DiaSpec language allows to design an application using an architectural pattern dedicated to control-loop systems. This specific architectural pattern comprises four layers of components: (1) *sensors* obtain raw data from the environment; (2) *contexts* process data and provide high-level information; (3) *controllers* use this information to control actuators; (4) *actuators* impact the environment. The sensors and actuators are the two facets of *entities* corresponding to devices, whether hardware or software, deployed in an environment.

This specification guides the developer throughout the development process. The DiaSpec compiler generates a Java programming framework dedicated to the application. This framework precisely guides the programmer during the implementation stage by providing high-level operations for entity discovery and component interactions. Based on these declarations, a simulator dedicated to pervasive computing environments is used to test and simulate the system. Then, the DiaSuite back-ends enable the deployment of an application by targeting a specific distributed systems technology such as RMI, SIP or Web Services.

### 2.2 Aircraft Guidance Application

The aircraft guidance application uses two sensors for computing the actual aircraft trajectory: the inertial reference unit, providing the localization, and the air data unit, supplying such measurements as the airspeed and the angle of attack. The synchronization of both information sources allows to compute

the actual aircraft trajectory. This trajectory is then compared to the flight plan entered by the pilot and used for controlling and correcting ailerons by the automatic pilot, if necessary.

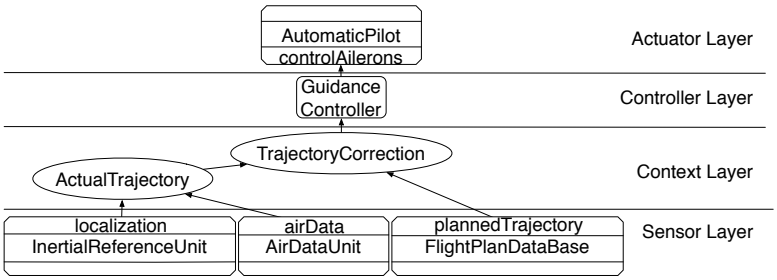


Fig. 2: A data-flow view of the aircraft guidance application

Following the DiaSuite development methodology, the first step identifies the devices involved in the aircraft guidance application using a domain-specific taxonomy of entities, as can be found in the aeronautics literature. In this example, we have identified four entities: the inertial reference unit, the air data unit, the flight plan database and the automatic pilot. The second step of the methodology consists of designing the application using DiaSpec. The system description is illustrated in Figure 2, making explicit the four component layers of DiaSpec.

In the example, the `InertialReferenceUnit` sensor provides the current localization of the aircraft. The `AirDataUnit` sensor supplies several air data such as the airspeed and the angle of attack. All these data are sent to the `ActualTrajectory` context that is responsible for computing the current trajectory of the aircraft. This information is then sent to the `TrajectoryCorrection` context component. When receiving a new trajectory, the `TrajectoryCorrection` component gets the planned trajectory (from the flight plan initially entered by the pilot) from the `FlightPlanDatabase` component. By comparing these information sources, it computes trajectory corrections that are sent to the `GuidanceController` component, responsible for controlling ailerons through the `AutomaticPilot` actuator.

The avionics certification process requires this trajectory readjustment to be time-bounded. In the next section, we show how the DiaSuite approach, enriched with time-related properties, can guide the development of such critical applications.

## 3 QoS throughout Software Development

This section presents how QoS requirements can be systematically processed throughout software development.

### 3.1 Requirements Analysis and Functional Specification

In software development methodologies, the requirements analysis stage identifies the users' needs. Then, the functional specification stage identifies the main functionalities to be fulfilled by the application to satisfy the users' requirements. In the avionics domain, each of these functionalities is generally associated to a *functional chain* [26], representing a chain of computations, from sensors to actuators.

The aircraft guidance system has a unique functional chain, whose execution should take less than 3 seconds, according to our expert at Thales Airborne Systems. In avionics, such time constraints, directly associated to a specific functional chain, is referred to as Worst Case Execution Time (WCET)[1]. If the design process involves refinement steps such as the identification of functional chain segments, the WCETs can be further refined. For example, we can identify a functional chain segment corresponding to the computation of the actual trajectory (from the sensors feeding the `ActualTrajectory` context). This chain segment can be reused in other applications, *e.g.,* displaying the actual trajectory on the navigation display unit. According to our expert, this chain segment must not take more than 2 seconds to execute.

### 3.2 Architecture Design

During the architecture design stage, the functional chains are decomposed into connected components. The architect can then refine the WCET of the functional chain on time-related constraints at the component level.

In the DiaSuite architectural pattern, the data flow between two components can be realized using two interaction modes: by pulling data (one-to-one synchronous interaction mode with a return value) or by pushing data to event subscribers (asynchronous publish/subscribe interaction mode). Pull interactions are typically addressed by a *response time* requirement as is done in Web Services [17]. Push interactions raise a need to synchronize two or more input events of a component. This need is addressed by introducing a *freshness* requirement on the input event values. This requirement is in the spirit of synchronization skew in the multimedia domain [18]. In addition to the freshness constraint, we define the *bounded synchronization-time* constraint that authorizes desynchronization during a bounded time, avoiding diverging synchronization strategies.

Figure 3 shows the QoS contracts associated to each component in the flight guidance application. The WCETs associated to the functional chain of the aircraft guidance and to the trajectory computing chain segment are mapped to the `GuidanceController` and `ActualTrajectory` components, respectively. The WCET of the functional chain of the aircraft guidance is refined into (1) a freshness constraint of 1 second between `localization` and `airData` with an equal bounded synchronization-time constraint (since the WCET is not compatible with a longer desynchronization time); and (2) a response time of 100

[1] This usage of WCET is only loosely related to the notion of WCET as documented in the literature for hard real-time systems.

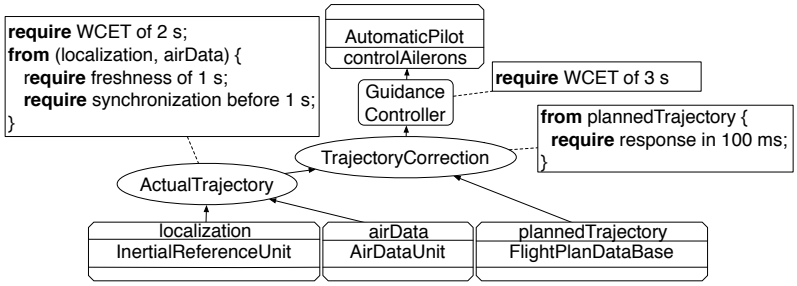


Fig. 3: Architecture of the working example, enriched with QoS contracts

milliseconds of `FlightPlanDataBase`. The WCET associated to the functional chain of the aircraft guidance is translated into a QoS contract, attached to `GuidanceController` as controllers are generally dedicated to a given functional chain.

The QoS contracts are introduced as an extension of DiaSpec. Figure 4 shows an extract from the DiaSpec specification.

```
context ActualTrajectory as Trajectory {
   source localization from InertialReferenceUnit;
   source airData from AirDataUnit;
   qos {
     from (localization, airData) {
         require freshness of 1 s;
         require synchronization before 1 s;
     }
   }
}
```

Fig. 4: DiaSpec declaration of the `ActualTrajectory` component

The `ActualTrajectory` component is declared with the `context` keyword, and returns values of type `Trajectory`. This component processes two sources of information: `localization` and `airData`. These sources are declared using the `source` keyword that takes a source name and a class of entities. Then, the QoS contract declared using the `qos` keyword defines freshness and synchronization constraints between `localization` and `airData`. This domain-specific approach guides the stakeholders when adding QoS requirements by automatically enforcing the conformance between the QoS contracts and the functional constraints.

### 3.3 Implementation

The DiaSuite approach includes a compiler that generates a dedicated programming framework from a DiaSpec description. Our approach enriches this process

by generating runtime-monitoring support from QoS declarations. At the implementation level, QoS requirements on components become runtime verifications that rely on the communication methods of the generated programming framework. Monitoring mechanisms are encapsulated into component containers that ensure that the response time and the freshness requirements are respected. The approach based on containers allows a separation of concerns between functional and non-functional requirements because a container is only in charge of intercepting calls for monitoring requirements, and forwarding the calls to the functional component. If a QoS contract is violated, the container throws specific exceptions `ResponseTimeException` or `SynchronizationException`. The treatment of such exceptions is left to the developer. It may involve any number of actions, including logging or reconfiguration [17]. DiaSuite provides declarative support at the architectural level to design exceptional treatments [21], preventing the application to be bloated and entangled with error-handling code.

The code corresponding to the response time requirement is straightforward. It is based on a timer that calculates the elapsed time between the request and the response. The more elaborate part concerns the synchronization defined by the automaton depicted in Figure 5. Suppose we want to synchronize `data1` and `data2` values. When receiving the first data (`S2` and `S4` states), the container activates the `t` and `t'` timers for measuring respectively synchronization time and freshness (`t,t':=0`). While synchronization time is not reached (`t<=ts`), the container waits for fresh data. If the other data is received before the freshness time has elapsed, the container pushes both data to the functional component (`S5` final state). Otherwise, if the freshness is not respected (`t'>tf`) ,the data is rejected. This is not considered as an error state since we authorize desynchronization for a finite period. Thereby the container waits for new values (`S3` state). If the synchronization time has elapsed (`t>ts`), the synchronization is aborted and a `SynchronizationException` is thrown (error state).

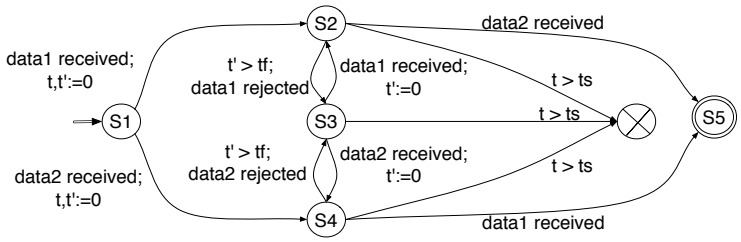


Fig. 5: Synchronization automaton

### 3.4 Deployment

Our approach offers support for predicting the performance of an application by injecting deployment parameters, such as distributed systems technologies,

platform and hardware characteristics. By taking advantage of their QoS characteristics (*e.g.,* the guaranteed deterministic timing of the AFDX network [4]), it is possible to refine the time-related requirements generated from the QoS declarations, and thus to compare several deployment configurations. In particular, it allows technologies to be selected according to their time-related properties.

This prediction tool takes numerical constraints generated from the QoS declarations as input. Then, an external constraint solver [9] checks whether a configuration respects the WCET of the functional chain and predicts constraints to the other architectural elements. In the next section, we detail how these numerical constraints are generated and propagated throughout the software development process.

## 4 QoS Requirements Traceability

The requirements traceability is the guarantee for each requirement to be traced back to its origin (*i.e.,* a QoS declaration), by following its propagation in the software development process. By generating numerical constraints from the QoS declarations, our step-wise approach allows the traceability of QoS requirements during software development. This approach is summarized in Figure 6.

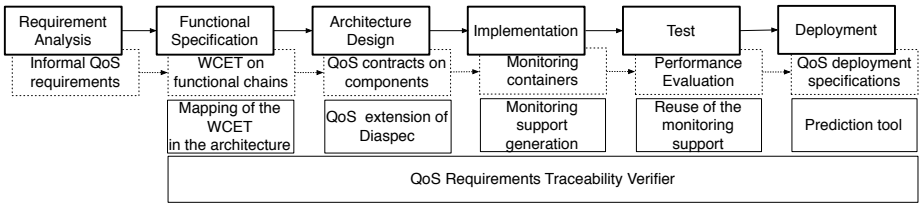


Fig. 6: Development process of Figure 1 extended with QoS concerns

At each development step, QoS declarations are translated into numerical constraints that are fed to the verifier of requirements traceability. This verifier propagates these numerical constraints between the development stages, and checks whether no new constraint invalidates constraints from preceding stages. In this section, we detail how these numerical constraints are generated and propagated.

### 4.1 From Functional Specification to Architecture Design

From functional specification to architecture design, requirements on functional chains (or chain segments) are refined into requirements on components. By generating numerical constraints, it is possible to ensure that the refinement does not invalidate requirements from the previous stage. Doing so amounts to checking whether the constraint system is still satisfied. The generation of

these numerical constraints is inspired by the Defour *et al.*'s work [10] and relies on the DiaSuite architectural pattern. It consists of automatically translating the QoS contracts presented in Section 3 into numerical equations specifying time relationships between the components. For example, let us detail how these equations are generated for the aircraft guidance application.

*WCET on functional chains and chain segments.* The functional chain that controls the trajectory has a WCET of 3 seconds, represented by the contract attached to `GuidanceController`. This leads to the following numerical constraints:

```
T_wcet_GuidanceController <= 3;
T_wcet_GuidanceController =
     T_provide_GuidanceController +
     T_com(GuidanceController,AutomaticPilot);
```

The first equation represents the WCET associated to `GuidanceController`. The second equation refines this functional chain into a sequence of two functions: one for computing orders (the `GuidanceController` chain segment), and one for communicating these orders to the automatic pilot. Thus, the global time is the sum of the `T_provide_GuidanceController` time and the communication time between `GuidanceController` and `AutomaticPilot` (denoted by the `T_com` function). The `T_provide_GuidanceController` time corresponds to the chain segment between the moment when the `InertialReferenceUnit` or `AirDataUnit` sensor sends a value and the moment when `GuidanceController` issues orders to `AutomaticPilot`.

Similarly, the `T_provide_GuidanceController` time can also be refined into the time of `GuidanceController` to compute orders, the communication time between `TrajectoryCorrection` and `GuidanceController`, and the global time associated to the chain segment of `TrajectoryCorrection`:

```
T_provide_GuidanceController =
    T_provide_TrajectoryCorrection +
    T_com(TrajectoryCorrection,GuidanceController) +
    T_compute_GuidanceController;
```

*Response Time.* The time associated to `TrajectoryCorrection` can be refined with respect to its relation with `ActualTrajectory` and `FlightPlanDataBase`:

```
T_provide_TrajectoryCorrection =
    T_provide_ActualTrajectory +
    T_com(ActualTrajectory,TrajectoryCorrection) +
    2 * T_com(FlightPlanDataBase,TrajectoryCorrection)+
    T_provide_FlightPlanDataBase +
    T_compute_TrajectoryCorrection;
```

Between the `ActualTrajectory` and `TrajectoryCorrection` contexts, the communication mode is of type publish/subscribe and thus can be decomposed

into `T_provide_ActualTrajectory` corresponding to the chain segment for computing the trajectory and `T_com(ActualTrajectory,TrajectoryCorrection)` corresponding to the communication time between these two contexts. Because the `plannedTrajectory` source of the `FlightPlanDataBase` is accessed by pulling the value in a synchronous manner, it is decomposed into the time to compute the data (`T_provide_FlightPlanDataBase`) and the communication round-trip (`2 * T_com(FlightPlanDataBase,TrajectoryCorrection)`) between the two components, assuming the size of the request and the response fit within a MTU (Maximum Transmission Unit).

*Freshness and Bounded Synchronization Time.* The QoS contract associated to `ActualTrajectory` specifies freshness and bounded synchronization time between `InertialReferenceUnit` and `AirDataUnit`. In the worst case, the synchronization takes the sum of the bounded synchronization time and the maximum time for receiving an event from `InertialReferenceUnit` or `AirDataUnit`. This leads to the generation of the following constraints:

```
T_provide_ActualTrajectory <= 2;
T_synchronization <= 1;
T_provide_ActualTrajectory =
 max(
  T_provide_airData + T_com(AirDataUnit,ActualTrajectory),
  T_provide_localization + T_com(InertialReferenceUnit,ActualTrajectory)) +
 T_synchronization +
 T_compute_ActualTrajectory;
```

The refinement of the numerical constraints and the checking of coherence at each step ensures the coherence between all non-functional requirements. Each numerical constraint is defined using Prolog IV [9], a constraint logic programming language over real numbers, coupled with a real interval arithmetic solver for checking the coherence of each refinement step.

### 4.2 From Architecture Design to Implementation

Monitoring support is generated from the non-functional specifications. Each non-functional container is in charge of monitoring the time-related constraints associated to a given functional component. Since QoS declarations at the design level are used to generate the monitoring support, the traceability is automatically ensured between the design and implementation stages. Moreover, as this support is embedded into the programming framework, it is completely transparent to the developer. Doing so prevents the developer from introducing errors in the monitoring code. Specifically the DiaSuite exception mechanism allows to separate the detection mechanism that is generated from the treatment code that is implemented by the developer.

### 4.3 From Implementation to Deployment

During the deployment stage, the prediction tool is based on the numerical constraints generated from the QoS-extended DiaSpec specification, ensuring the

traceability of the requirements. In the avionics domain, the execution platforms offer deterministic QoS characteristics (*e.g.,* the deterministic timing of the AFDX network). Such information allows to refine the time-related constraints generated from a QoS declaration and to compare the performance of several deployment configurations.

In the generated constraints, there are several numerical variables that depend on the deployment. The `T_com_<component_name>` variables depend on the communication mode. If the application is deployed on a non-distributed platform, the communication time can be considered as null between applicative components, simplifying the numerical constraints. If the application executes on a distributed platform, the communication time between the applicative components depends on the distributed systems technologies. In avionics, the most commonly used network is the AFDX. The communication constraints can be refined according to the AFDX bandwidth and the associated Bandwidth Allocation Gap (BAG). Concerning the communication between applicative components and devices, different sort of communication technologies can be used, such as a serial link (*e.g.,* ARINC429, RS422) that leads to different communication times. The `T_compute_<component_name>` variables depend on the complexity of the algorithm and the execution platform (*e.g.,* CPU frequency and memory access time). Finally, the `T_provide_<data_sensed>` variables depend on sensor technologies (*e.g.,* mechanical or LASER probes for air data).

For example, assume we enrich the system with a new functionality for displaying the trajectory on the navigation display. We want to reuse the chain segment computing the actual trajectory but with stronger QoS requirements, leading to the constraint `T_provide_ActualTrajectory <= 0.8`. From all these constraints, the prediction tool infers the following value range for the Air Data Unit: `0 <= T_provide_airData <= 0.8`. This constraint is propagated to the stakeholders in charge of selecting the technologies for the execution platform. In this situation, they will choose LASER probes whose performance is conform with this constraint.

## 5 Towards Certification of Avionics Systems

Our approach has been applied to the design of several avionics applications, including the flight management system, the aircraft guidance system, and the traffic collision avoidance system. These experiments have shown that the DiaSuite methodology is well-suited for the development of avionics control-loop systems. In this section, we discuss how our integrated QoS approach guides the avionics certification process.

In avionics, aircrafts have to respect the Certification Specification (CS) standard to obtain the airworthiness certificate. CS proposes the classification of the failure conditions: conditions impacting the aircraft and/or its occupants. They depend on the flight phases (*e.g.,* landing) and the environmental conditions. Failure conditions are associated with a Design Assurance Level, indicating their degree of criticality and safety objectives. To reach safety objectives, aeronau-

tical standards specify constraints on the development process (*e.g.,* distribute the development stages across several teams) and the testing process (*e.g.,* structural and black-boxes tests). These constraints cover all the levels of a system, including the software and hardware layers. Both functional and non-functional requirements [22] have to be guaranteed, including performance concerns.

Modern avionics platforms such as the Integrated Modular Avionics (IMA) allow to host several functions on the same platform [25]. The IMA approach introduces different stakeholders in the development process. The *system integrator* is the leading authority (generally, the airframer). The role of this stakeholder is to integrate all IMA systems: the software applications, provided by the *function suppliers*, and the hardware systems, provided by the *platform suppliers*. The integrator and suppliers play different roles in the certification process [12]. The system integrator specifies general constraints about all the systems at the aircraft level. For example, this stakeholder defines the Worst Case Communication Time (WCCT) of the network, ensured by the AFDX technology [4], and the WCET of the applications. These WCET requirements are passed on to the function suppliers. To fulfill them, in turn, function suppliers produce specific requirements for platform suppliers regarding issues such as time slots, CPU power, memory capacity and throughput. To do so, the platform suppliers have to provide the function suppliers with all the characteristics of this platform, including WCET for core software services (*e.g.,* drivers and health monitoring).

From the high-level WCET constraints delivered by the system integrator, our approach guides the function suppliers in systematically specifying and refining all the non-functional requirements during the development of the application. Function suppliers can also use the generated monitoring support for validating the performance of their application. Furthermore, the prediction tool can be helpful in determining an interval of timing on each deployment technologies and passing these constraints on to the platform suppliers. The platform suppliers are now responsible for giving applications the means to access input data from the sources, and send output data to the actuators. Regarding constraints issued by all the function suppliers, the platform supplier can use the prediction tool for proposing a platform configuration that matches all their needs. Finally, the system integrator can use the generated monitoring support for both controlling the applications in flight, and performing aircraft maintenance. In doing so, information about the non-functional behavior of the application can be logged in flight and the pilot can be alerted in case of unexpected behaviors. The monitoring support can also be used on-ground for correcting errors. As this support is entirely generated, it is sufficient to test and certify the generator for guaranteeing the correctness of all future generated monitoring containers.

Requirements traceability is key to obtain avionics certification. Because the role separation proposed by the IMA platforms leads to the collaboration of several companies, requirements traceability has become significantly more challenging. The tool-based approach proposed in this paper can facilitate this certification process by offering support for propagating automatically QoS requirements between the stakeholders.

## 6 Related Work

Among existing QoS specifications languages, some of them focus on performance properties and already offer design and programming support. For example, in the spirit of our approach, Defour *et al.* generate numerical constraints from time-based requirements specified with the QoSCL language [10]. These constraints are not only used to check the requirements compatibility according to the architecture, but also to predict the QoS from the number of instances of each component. AlTurki *et al.* propose a real-time rewriting model backing a timing specification language [2]. It allows them to verify various real-time properties using the Maude rewriting engine. Krogmann *et al.* have set up a quantitative performance prediction tool into the Palladio Component Model [20], allowing architects to choose between different architectural designs. They define many types of performance requirements that would be interesting for us to take into account for critical systems (*e.g.,* CPU). Bertolino *et al.* [6] also propose another performance-based prediction approach that focuses on the assembling of existing components. In contrast to our work, the above approaches are general-purpose and are limited to the design phases of the development process.

Other approaches are domain-specific, for example dedicated to the specification of real-time systems. As a result, they can offer better design and programming support. For example, Fredriksson *et al.* propose a framework for leveraging non-functional requirements (*e.g.,* time and memory consumption) to build control systems components [15] and thus to predict the functional/non-functional behavior of the composed system. Doose *et al.* formalize real-time systems as a set of functionalities linked within timed communications and then verify the time-coherency of the whole system using model-checking techniques [13]. Carcenac *et al.* also validate real-time systems according to the incremental specification of non-functional requirements [7]. Yet, these approaches only focus on the validation of the systems.

In contrast, the approach of Robert *et al.* enables generating monitoring support from non-functional requirements, represented as exceptional transitions in timed-automata [24]. In the same spirit, Duclos *et al.* [14] have proposed to specify QoS requirements as aspects in the architectural models for providing separation of concerns, monitoring these requirements at runtime. The approach of Genssler *et al.* enables generating scheduling support based on QoS declarations [16].

The above QoS approaches are dedicated to real-time systems and offer support at design time (*e.g.,* prediction) and/or at runtime (*e.g.,* monitoring). However, they are mostly dedicated to specific development stages and do not consider the traceability of non-functional concerns through the software development process.

To conclude, specifying non-functional requirements only at the architecture level is not sufficient. As observed by Koziolek *et al.* [19], it is crucial to clearly identify the stakeholders and the workflow between the functional development and the non-functional layer. Towards this end, we propose a unified approach that integrates QoS into the complete development process.

## 7 Conclusion

In this paper we have presented a step-wise approach integrating QoS concerns through all phases of software development. This approach dedicated to control-loop systems extends the DiaSuite tool-based methodology by offering support for specifying, validating and monitoring time-related requirements. We have shown that this domain-specific approach allows to guide the stakeholders in systematically refining non-functional requirements and ensures requirements traceability by generating numerical constraints. We have illustrated our approach in the avionics domain where such QoS requirements are critical.

We are currently working on a deeper evaluation of this approach with the development of an autopilot application coupled to the FlightGear simulator [23]. This work will allow to leverage DiaSpec's architectural support of error handling [21] for treating the violation of a QoS contract at runtime. In particular, we plan to show how the error handling support provided by DiaSpec can be used to implement logging and reconfiguration treatments. This evaluation would also help in refining the specification language (*e.g.,* time constraints depending on input parameters). Future work concerns the integration of this methodology into the avionics certification process. In particular, we will need to certify our tools and their associated development approach.

## References

1. J. Ø. Aagedal. *Quality of service support in development of distributed systems.* PhD thesis, University of Oslo, 2001.
2. M. AlTurki, D. Dhurjati, D. Yu, A. Chander, and H. Inamura. Formal specification and analysis of timing properties in software systems. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*, pages 262–277. Springer, 2009.
3. ARINC 653, system partitioning and scheduling (Aeronautical Radio, Inc.), 2003.
4. ARINC 664, AFDX: Avionics Full DupleX switched ethernet (Aeronautical Radio, Inc.), 2005.
5. ARP4754, certification considerations for highly-integrated or complex aircraft systems (SAE), 1996.
6. A. Bertolino and R. Mirandola. CB-SPE tool: putting component-based performance engineering into practice. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, pages 233–248. Springer, 2004.
7. F. Carcenac and F. Boniol. A formal framework for verifying distributed embedded systems based on abstraction methods. *International Journal on Software Tools for Technology Transfer*, 8(6):471–484, 2006.
8. D. Cassou, B. Bertran, N. Loriant, and C. Consel. A generative programming approach to developing pervasive computing systems. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 137–146. ACM, 2009.
9. A. Colmerauer. *Specifications of Prolog IV*, 1996.
10. O. Defour, J.-M. Jézéquel, and N. Plouzeau. Extra-functional contract support in components. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, pages 217–232. Springer, 2004.

11. DO-178B, software considerations in airborne systems and equipment certification (RTCA, Inc.), 1992.
12. DO-297, Integrated Modular Avionics (IMA) development guidance and certification considerations (RTCA, Inc.), 2005.
13. D. Doose and Z. Mammeri. Polyhedra-based approach for incremental validation of real-time systems. In *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pages 184–193. Springer, 2005.
14. F. Duclos, J. Estublier, and P. Morat. Describing and using non-functional aspects in component-based applications. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 65–75. ACM, 2002.
15. J. Fredriksson, M. Tivoli, and I. Crnkovic. A component-based development framework for supporting functional and non-functional analysis in control system design. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 368–371. ACM, 2005.
16. T. Genssler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. O. Müller, and C. Stich. Components for embedded software: the PECOS approach. In *Proceedings of the Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 19–26. ACM, 2002.
17. R. B. Halima, K. Drira, and M. Jmaiel. A QoS-oriented reconfigurable middleware for self-healing web services. In *Proceedings of the 6th IEEE International Conference on Web Services*, pages 104–111. IEEE, 2008.
18. S. Jha and A. Seneviratne. Synchronization skew: a QoS measurement study. In *Proceedings of the Conference on Local Computer Networks*, pages 77–78, 1999.
19. H. Koziolek and J. Happe. A QoS driven development process model for component-based software systems. In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, pages 336–343. Springer, 2006.
20. K. Krogmann, C. M. Schweda, S. Buckl, M. Kuperberg, A. Martens, and F. Matthes. Improved feedback for architectural performance prediction using software cartography visualizations. In *Proceedings of the 5th International Conference on the Quality of Software Architectures*, volume 5581 of *LNCS*, pages 52–69. Springer, 2009.
21. J. Mercadal, Q. Enard, C. Consel, and N. Loriant. A domain-specific approach to architecturing error handling in pervasive computing. In *Proceedings of the 25th International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 2010.
22. M. Paulitsch, H. Ruess, and M. Sorea. Non-functional avionics requirements. In *Proceedings of the 3rd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 369–384. Springer, 2009.
23. A. R. Perry. The FlightGear flight simulator. In *Proceedings of the USENIX Annual Technical Conference*, 2004.
24. T. Robert, J.-C. Fabre, and M. Roy. On-line monitoring of real time applications for early error detection. In *Proceedings of the 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 24–31. IEEE, 2008.
25. C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to Integrated Modular Avionics. In *Proceedings of the 26th IEEE/AIAA Digital Avionics Systems Conference*, page 2. IEEE, 2007.
26. J. Windsor and K. Hjortnaes. Time and space partitioning in spacecraft avionics. In *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology*, pages 13–20. IEEE, 2009.