Breaking down silos with contract based design for industrial software development: illustrated through an aerospace case study
Vijay Singh, Tuur Benoit, Vincent Braibant

To cite this version:
Vijay Singh, Tuur Benoit, Vincent Braibant. Breaking down silos with contract based design for industrial software development: illustrated through an aerospace case study. ERTS 2018, Jan 2018, Toulouse, France. hal-02156100

HAL Id: hal-02156100
https://hal.archives-ouvertes.fr/hal-02156100
Submitted on 14 Jun 2019

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.
Breaking down silos with contract based design for industrial software development: illustrated through an aerospace case study

Vijay B. Singh, Tuur Benoit, Vincent Braibant
Simulation and Test Solutions, Siemens Digital Factory, Product Lifecycle Management

Abstract

Contract based approaches have been studied for quite some time but adoption by the mainstream embedded software community and establishing a proper industrial process and methodology has shown to be slow and mainly limited to the architecture team and is not extended to rest of the software teams. This paper presents an integrated development environment for contract based development of embedded systems that addresses a number of challenges in adopting formal methods and contract based approaches.

Based on an industrial case study of the development of a controller for trajectory control of a UAV, the paper demonstrates several new features such as architectural system modelling, component interfaces with contract annotations and contract based white box analysis and testing, that move the state-of-the-art in model-driven development with contract based design closer towards managing the industrial complexity of software development and synchronizing the software teams across the entire development lifecycle.

The proposed workflow promises a faster development cycle for the different stakeholders of software development: architects, developers and testers. In addition, the methodology promises to provide industrial value in terms of early design error detection, predictable integration of the software, reduced release time and flexible organization level adoption.

Keywords

Contract based design, software development, domain specific language, virtual integration

1. Introduction

The systems development process in the aeronautics domain is an utterly lengthy and costly endeavor. A factor that contributes to this onerous engineering effort is the high amount of integration activities taking place. The growing complexity of aircraft systems, as a result of increasing functionalities as well as the current trend of the Integrated Modular Avionics (IMA) concept, will put more pressure on the integration tasks. This complexity makes the design of these systems subject to design errors and inconsistencies in integration. These are typically hidden in the initial stages of development and only become visible when the system is already well under way. The later these design and specification errors are detected, the more expensive they are to correct.

Therefore, it is very advantageous to ensure integration from the start. The systems engineering methods used today in the aeronautics industry are based on the use of documents (instead of models) for managing the integration activities, despite the fact that information technology has advanced to a point where the technology can easily support fully digital models. In addition, they do not provide a formal process for validation and verification of the designs.

This paper presents a model-based development (MBD) associated with contract based design to efficiently manage the integration activities from the beginning of the project, and enabling synchronization of the different software teams throughout the project, promising faster development and keeping complex software projects on time and within budget. Model-based development is today generally accepted as a key enabler to improve understandability, maintainability, productivity and to the much extent quality of the existing development process and its related artifacts. In model-based software engineering, models are the main artifact of the software development process. Moreover, contract based design is an approach for software development that prescribes formal, precise and
verifiable interface specifications for software components. These specifications are referred to as 'contracts'.

A contract binds two parties to their duties and saves the interest of one party if other attempts to violate that agreement. This is very common framework to go by in business deals, financial investments, government contracts etc. Having a contract avoids confusion and binds the parties involved to the previously defined formal description of duties. In an industrial software development context, a contract is an artifact that states how the interface of a component looks like, what assumptions it makes, and which guarantees it can deliver.

This paper reports on an industrial case study of the development of a controller for trajectory control of a UAV, and how use of a contract based methodology has a strong potential to positively affect the extensibility, testability and verifiability as well as the required effort of virtual and continuous software integration. This is achieved through the modelling of the system architecture using horizontal contracts and the verification of the design with assume guarantee analysis, white box formal verification and testing.

The case study relies on Simcenter Embedded Software Designer, an integrated development environment (IDE) for model based software engineering design built with the open source domain specific language workbench Jetbrains MPS [25] and mbeddr [26]. This IDE features a number of novel capabilities that bring the state-of-the-art in model-driven development with contract based design closer towards managing projects of industrial complexity.

The remainder of the paper presents above elements in more detail. Section 2 introduces a number of references that are related to the work presented in this paper. Section 3 includes an introduction to contract based design and briefly presents the use case. Section 4 explain how contracts can be used in architectural design and section 5 gives more details on contract verification. A short note on the adoption of contracts in continuous software integration is given in section 6. Finally section 7 presents conclusions and future work.

2. Related work

Contract based design methodology has been widely studied. Beneviste et al [1] performed an extensive literature survey on this topic, and provide a treatment where contracts are precisely defined and characterized so that they can be used in model based design methodologies.

Early work by Floyd [5] and Hoare [6], offers a framework for refinement and compositionality, for proving program correctness using pre and post conditions. These ideas were further developed and implemented in several programming language, the Eiffel programming language being the first to add contracts as a core language feature [7]. In most programming language, contracts are implemented as a library, and check the contact pre and post conditions only during execution, examples are: PyContracts (Python) [8], Lib Hoare (Rust) [9], DBC for C (C) [10]. Other programming languages have native support for contracts that can be verified statically: ADA Spark probably most known in the Aerospace industry [11].

In the domain of controller design, Sangiovanni-Vincentelli et al [2] present a design methodology and a few examples whereby contract-based design can be merged with platform-based design. Furthermore, recent work by Wiik and Boström [3] reports on verification of contracts for Matlab programs, but is restricted to a subset of Matlab, suitable for code generation and static determination of matrix sizes. Verification conditions for are generated from the contracts and verified with the Z3 SMT solver. On a similar note, work by Roy and Shankar [4] suggests a methodology for contract based design for Simulink models, by annotating ports and links. Their verifier checks the well formedness of Simulink blocks with respect to these contracts. The SMT solver used is the Yices solver. Mathworks developed Simulink Design Verifier [12], which can statically identify certain design errors in Simulink models. It detects blocks in the model that result in integer overflow, dead logic, array access violations, division by zero, and other requirement violations. Requirements are expressed as special additional blocks in the model. This is in contrast to Simcenter Embedded Software Designer, for which contracts are directly built in every block. Finally, Ansys Scade provides an integration of a formally defined language for discrete control logic (Lustre [13]), in a model based environment [14]. The formal language and graphical environment allow to express and prove requirements or provide counterexamples when the requirements are not met.
Several works assess the challenges in adopting formal methods and contract based approaches based on industrial application cases. Brat et al. describe the experience and lessons learned from their effort of verifying the safety of a flight critical system [23]. The main challenges they identify are rewriting the requirements in a formal way, getting a complete set of requirements and scalability of the methodology. The latter is tackled by the contract system of Simcenter Embedded Software Designer and is described in this paper. Moy et al. [22] discuss experiences with DO-330 [23], the formal methods supplement to DO-178C, at Airbus and Dassault. Furthermore, Woodcock et al. present a survey on the practice and challenges of formal methods in different industries, including a short section on the aircraft industry [21]. Some of the design challenges due to the complexity and heterogeneity of cyber physical systems are highlighted by Nuzzo and Sangiovanni-vincentelli in [17] and they argue how contract based design could tackle integration and specification challenges. Nuzzo et. Al then proceed to show in [18] how a contract based methodology is used for the design of an aircraft electric power system.

Works from other industries confirm the challenges cited by the authors above. Benvenuti et al. [15] focus on heterogeneous models for embedded controllers where the plant, sensors and actuators are described by hybrid systems and give rules on how to compose assumptions and promises for components in cascade and feedback configuration. They apply these rules on a test case, a water-level control problem. Heinz [16] illustrates a number of applications that are interesting from an industrial perspective for which simulation based verification and falsification methods could be used in case of presence of black box components. Westman et al. describe in [19] how to structure requirements using contract theory for automotive applications which follow the ISO 26262 standard. The contracts are based on the contract theory developed in the European SPEEDS project [20].

3. Contract based design

Traditionally engineers design a controller in a simulation tool such as Mathworks Simulink. Different control approaches are explored and optimal parameters are determined once the control structure is selected. Mathematical analysis verifies the properties of the controller, such as stability, bandwidth, margins, etc. In a complex system such as an aircraft, multiple controllers are active at the same moment, controlling different aspects of the system, e.g. longitudinal control, lateral control, engine control, environmental control. As a result, a practical controller always consists of multiple components interacting and communicating with each other.

Contracts are a method for specifying which data is exchanged between these software components and what the behaviour of the component is in terms of assumptions that the component makes and guarantees for what the component is expected to deliver.

Classical models just provide one level of abstractions in form of blocks or sub systems describing only its inputs and outputs. Such a component has parameters and states that affect the transformation of inputs into outputs. Parameters are constants that do not change or evolve during the life time of software execution. States are variables internal to the component and represent the memory of the component. Their values are retained between function invocations.

The lack of more detailed information means that not much information is available which can be leveraged to drive development, testing, verification and testing activities. As a result, design, development and testing activities become individual activities done in silos which leads to issues later on with respect to change management, traceability and integration. Figure 1 shows an example of such a classical model of a component with three inputs and two outputs.

![Figure 1. Today's model based software engineering and ad-hoc composed tool chains.](image-url)
One way of enriching modeling abstraction is by adding assumptions and guarantees, embracing the full contract based design approach. The assumptions and guarantees can be seen as partial description of the software itself. Assumptions are constraints on the inputs and parameters, and are stated as preconditions, on the other hand guarantees are the expectations on the outputs and are stated as post-conditions. Figure 2 shows the model of Figure 1 enriched with such assumptions and guarantees.

One can also make the distinction between horizontal and vertical contracts. A horizontal contract defines the relation between two or more components at the same level, whereas a vertical contract defines the relation between the properties of a system and those of its implementation.

Using contracts, a system can be developed by composition and refinement, enabling early assessment of safety and reliability, by being able to automatically check software properties like architecture consistency and contract violation. This provides precise interaction among architects, developers and test engineers.

3.1 Use Case: Trajectory controller of a UAV

The use case is a flight controller design for an Unmanned Aerial Vehicle (UAV). The UAV has a wingspan of 6 m and a maximum take-off weight of 80 kg. The controller performs longitudinal control. Different control approaches are investigated: a Linear Quadratic Regulator (LQR) and a Linear Quadratic Integrator (LQI). The controller software is developed using a model based approach in Simcenter Embedded Software Designer, while the plant model is modelled in Mathworks Simulink. Unit tests, formal verification and closed loop simulation tests are used to verify the controller design in Simcenter Embedded Software Designer.

4. Contract creation and architecture design

4.1 Contract definition

A contract is defined as part of a software interface which is typically an open system, i.e. it contains some inputs that are provided by the external environment or other interfaces in the system, and generates some output. A parameter of the software interface can also become part of the contract. The collection of input data space is considered as the input environment of the software interface. Adding an input contract is the act of adding assumptions on that input environment and an output contract is defining the guarantees on the output ports.

For example, in Figure 3, the contract of a component called speed_controller is given. The component implements the function that controls the longitudinal speed of the aircraft. To this end, the component has two inputs: speed and speed_wanted, both inputs of type double and with unit m/s (mps). The component computes the desired control signal, speed_control, which is a signal of type double with unit V. The input_contract states that the speed shall be larger than 0 m/s. Under this assumption, the component guarantees that the speed_control signal will be smaller than 5 V. An additional post condition, dependent_contract, captures a relationship between an input and output. It can be read as: ‘If speed goes negative, speed_control shall always be greater than zero’.

The above contracts correspond to invariants that the system should always uphold. The requirement that the speed_control signal should be between 0 and 5 V, could have been derived from the control actuator specifications for example. Note that, if the input_contract assumptions are violated, this
component can still be executed, however it cannot guarantee that the post conditions will be met, that is, the speed_control signal cannot be guaranteed to be smaller than 5 V for example.

Having contractual relationships of each component like this, enables to assess the complex safety and reliability properties of a system that is composed of these components. To this end horizontal and vertical contract verification activities are performed. This is explained in the following sections.

```
exported blockinterface speed_controller [double/m/s/ -> speed]
contract [pre(0) input_contract: speed >= 0 m/s;
          post(1) output_contract: speed_control < 5 V;
          post(2) dependent_contract: (speed < 0 m/s) -> (speed_control > 0 V)].
```

Figure 3. Contract definition for a software component in Simcenter Embedded Software Designer

### 4.2 Horizontal contracts in context of software architecture

A software system pre-dominantly has three main elements, i.e. components, ports and connections. A component can be an isolated unit or it can be a subsystem that consists of a number of nested components. The components themselves have input and output ports which can be associated with contracts.

Ports at the system architecture layer itself correspond to system in/out ports. The system input contract assumes the combination of all assumptions stated for each component and subsystem it contains. On the other hand, the guarantees offered by system output contract is the combination of the guarantees of constituent component contracts.

By connecting input and output ports, relations among subsystems at the same level of design hierarchy are established. Connections are simply used as a data/signal flow medium and they do not have any intelligence on their own. However, when considered in a contract environment, the connections bind the input and output contracts of the components it connects, hence establishing the horizontal contracts at that system level.

Horizontal contracts can be mathematically analyzed to check data flow consistency of the system architecture. This virtual software system integration scenario leads to a software architecture that is consistent. This is further explained in section 5.1.

Figure 4 shows the system architecture of the longitudinal flight controller system, as created in Simcenter Embedded Software Designer. The figure shows that this system consists of several components: a speed controller component, an aircraft angle controller component, a sensor fusion / interpreter component, etc. A plant block is included which models that system to be controlled.

The components each have one or more ports. The black lines represent the connections between the ports of two components. In addition, one can see on the figure that the system itself has input ports and output ports. Two input ports: angle_wanted and speed_wanted, and five output ports: state, speed, speed_control, angle_control and angle_attack. These ports are visualized as blocks with an arrow indicating the flow of the signal.

### 4.3 Vertical contracts for software implementation and test orchestration

So far horizontal contracts ensure consistency at the architectural design level. However, to ensure consistency among the different design levels, from architecture to development and test, one needs vertical contracts.

In our view, a component contract acts as a vertical contract when the component is taken individually and the contract is used to verify and test whether the implementation satisfies the requirements given by the contract. Individual component contracts can be published after successful architecture analysis and can then be assigned to software engineers responsible for its implementation. Use of vertical contracts also provides a way to specify sub-contracts between OEMs, suppliers, and outsourcing companies. At any point in time, the contract ensures that the implementation provided by supplier will satisfy the requirements that are described by the contract such that it can be easily integrated with rest of the system.
Once software engineers extend the interface with their implementation, contracts at first provide key design requirements for the developers and later binds the developers to produce an implementation that fulfills the contract. To this end, they make use of unit tests and formal methods.

The process of implementing a component and verifying it using unit tests and formal methods is illustrated in Figure 5. The figure shows a contract without implementation (top left) that is subsequently implemented with C-code (bottom left). Next, the implementation is verified with a bounded model checker (top right) that does a number of robustness checks, such as out of bound indexing, overflow/underflow and division by zero. In addition, the model checker verifies whether the implementation fulfills the contract. If it detects a violation, the analyzer will report a diagnostic, including a counter example. Furthermore, certification authorities require additional testing, even when formal methods are used. To this end, a number of unit tests check the outcome of the component for a set of input values. They are specified in a table format (bottom right).

Based on the component interfaces and design contracts, a user can build an extensible library of components which can be shared and accessed throughout the organization. The user can also extend one form of component interface to meet their project needs and adapt the design contracts accordingly.

Design contract creation is especially useful from architect perspective; nonetheless developers can also add local contracts to support their work. Contracts added by developers do not affect the overall system architecture. Contracts specified in Simcenter Embedded Software Designer also extends to developers who are working with an external IDE.
5. Contract verification methods

5.1 Verifying architecture consistency using horizontal contracts

Section 4.2 explained horizontal contracts in an architectural context and how they apply to overall software design assurance. Finding integration related problems early in the development lifecycle is very important when it comes to integrating software coming from multiple sources.

Assume-guarantee analysis (A/G analysis) can be used to perform and verify a virtual integration scenario before giving the design to a development team. A/G analysis essentially verifies that the connections in a composite block are consistent. It is a compatibility check between the components connected in an architecture. It assumes that all the constituent components of a composite block satisfy their post-conditions, when their preconditions are satisfied. Based on this assumption, it verifies that the preconditions of a component are satisfied through the post conditions of the components to which it is connected or the system inputs. To this end, the design team makes sure that all the pre conditions are satisfied by the post conditions of preceding components and there is no inconsistency in data flow.

Figure 6 illustrates the A/G analysis of part of the flight controller architecture. The figure consists of three parts. On the left, the definition of the A/G analysis is shown. In the middle, one can see the architecture, and the right side shows the result of the A/G analysis. In this case all preconditions have been successfully verified to be fulfilled. That means that the composition of the components as designed is consistent with the expected inputs and outputs of each component. A/G analysis in Simcenter Embedded Software Designer works on top of formal method engine CBMC [28].

5.2 Verifying implementations using vertical contracts

Contracts or data constraints specifications not only work as guidelines for development and test engineers but they also increase the testability of the overall system and its components by acting as a high-level executable functional test specification. Test activities can be front-loaded, early in the development lifecycle, by checking these functional tests every time the model or code is executed. Such a process can be regarded as test-driven development which is bound by contracts.

White box analysis is a method of verifying an implementation against its contract by statically analyzing the code. Since it is a white box analysis method, the source code must be available. Figure 7 shows such a white box analysis for the sensor interpreter of the flight controller. The analysis definition (left) is created to analyze the implementation (center), resulting in the contract verification results (right). Note that contract checks can be made optional or mandatory by turning the post condition check on or off. It can be seen that 3 out of 6 contracts specifications are violated. The IDE provides a quick debugging functionality taking the user to the relevant location in model and code respectively after a double click on the failed case. In addition, unit tests can be used for verification of the implementation as well. This was illustrated in Figure 5.
Figure 8 shows the configuration of contracts in the build environment for testing and production code generation. Contracts (left) are basically converted as set of if-else statements (right) in generated C code. In production project where the contracts are not required, code generation for contract can be turned off by making changes to configuration items (center).

5.3 Verifying integration using closed loop simulation

Once the individual software units are implemented, in Simcenter Embedded Software designer or as an external C-code file, they can be brought back to Simcenter Embedded Software Designer. These concrete implementations are then mapped to their respective interfaces. What was once a system of empty architectural components before (Figure 4), is now a functionally working system with concrete implemented components (Figure 9). Note the difference between the two figures: in Figure 4 the blocks are shown with a dashed border indicating that no implementation is associated with the component yet, while in Figure 9 the blocks have a solid border, indicating that this component is mapped to a concrete implementation.

The integrated system can be simulated. To this end it is coupled to a plant model. Simcenter Embedded Software Designer currently supports plant models designed in Simcenter AMESim and Mathworks Simulink. The closed loop virtual validation of system can be launched directly from Simcenter Embedded Software Designer, which then runs the software in co-simulation environment [29]. Through co-simulation two models – here: the aircraft plant model modelled in Mathworks Simulink, and the controller software as exported from Siemens Embedded Software Designer – are coupled and simulated together. After the simulation, any violation of a contract during the simulation is reported back in the IDE (Figure 10). Furthermore quantities that are of interest to the user can be plotted (Figure 11).

Figure 8. Code generation configuration for contracts

Figure 9. Mapped interface-to-implementation view of system model

Figure 10. Failed contract messages reported after closed loop simulation with plant model
6. **Contracts as enabler for more effective continuous software integration**

Continuous Integration is considered an important form of software testing used to ensure changing code and updates integrate smoothly and give software engineers a way to ensure stability and safety. It is a software development practice where members of a team integrate their work frequently — leading to multiple integrations per day. In current software engineering practice, although this approach is very desired for handling bigger projects, at the same time it can also be time consuming for developers to get feedback on whether their code ‘passes’ the verification suite, before the code is committed.

In this paper, it was established that a contract by itself acts as a functional test case, which means any behavioral implementation which does not meet the contract specification fails the basic test case. Therefore contract based design can also support a more efficient continuous software integration practice by making sure that a developer at any time commits code which is correct at least by contract design. Contracts in this regard provide developers insights on what is required from a system design perspective and basically off-loads the developer to commit their code everytime hoping it will go through. And in case of a commit, stakeholders immediately know the cause of integration failure owing to contract failure. In that sense, a contract based development approach gives a whole new gamut to how the software development is being done in distributed teams and it greatly reduces the integration burden of project management.

7. **Conclusion**

This paper presents an integrated development environment, Siemens Embedded Software Designer, with several novel features that move the state-of-the-art in model-driven development with contract based design closer towards managing the industrial complexity of software development and synchronizing the software teams across the entire development lifecycle. This is achieved through the modelling of the system architecture using horizontal contracts and the verification of the design with assume guarantee analysis, white box formal verification and testing.

Based on a case study of a flight controller of a UAV, this paper demonstrates how to address a number of challenges – such as proper formulation of the requirements and scalability of the method – in adopting formal methods and contract based approaches.

- Contracts are used to enrich the flight controller components with assumptions and guarantees which can be seen as partial description of the software component behavior.
- Consistency at the architectural design level is assured by the so-called horizontal contracts. Assume-guarantee analysis (A/G analysis) can be used to perform and verify a virtual integration scenario before giving the design to a development team.
- To ensure consistency among the different design levels, from architecture to development and test, vertical contracts are employed. Unit tests and white box static analysis is used to verify implementations against its contract.
- Finally, closed loop simulation with a plant model in the loop is used to verify that the integration activities were executed correctly.

Experience with this case study and other projects also reveal that introducing a contract-based design approach into an organization may be difficult at first but might be alleviated by incremental introduction of contract. Future work includes expanding the contract methodology to hardware specification in order to be able to properly do software and hardware co-development.
8. Acknowledgments

The research of T. Benoit is funded by a grant from the Flanders Innovation & Entrepreneurship Agency.

9. References

[26] mbeddr project http://mbeddr.com/
[28] CBMC http://www.cprover.org/cbmc/ software developed by Daniel Kroening, ETH Zurich and Edmund Clarke, Computer Science Department, Carnegie Mellon University