A COMPONENT-ORIENTED FRAMEWORK
FOR SPACECRAFT ON-BOARD SOFTWARE

Marek Prochazka[1], Roger Ward[1], Petr Tuma[2], Petr Hnetynka[2], Jiri Adamek[2]

[1] *SciSys, Clothier Road, Bristol, BS4 5SS, United Kingdom*
Email: {marek.prochazka|roger.ward}@scisys.co.uk
[2] *Department of Software Engineering, Charles University, Malostranske namesti 25,118 00 Prague, Czech Republic*
Email: {petr.tuma|hnetynka|jiri.adamek}@dsrg.mff.cuni.cz

ABSTRACT

This paper presents our vision for architecture and development of spacecraft on-board software composed of well-defined building blocks called software components. This vision is derived from our work in the DisCo and COrDeT studies funded by ESA and fits within the on-board software harmonisation carried out by ESA. The paper is focused on software architecture and corresponding engineering process which would leverage software reuse across different space missions and reduce software development and integration costs.

1   MOTIVATION

Current spacecraft on-board systems become increasingly complex, providing more functionality such as coordination, cooperation, interoperability, autonomy, and improved fault tolerance. Distributed and decentralised architectures are being addressed, based around high-speed networks such as SpaceWire. At the same time, there are cost-saving pressures – optimisation of mission efficiency, use of system resources, software development process.

Standard solutions exist for efficient use of system resources and the required improved functionality. But these are sporadically deployed on different missions with little direct reuse between them. By defining standard software architectures with common building blocks that can be reused between missions, the software development process can be reduced while improving the functionality of the system.

2   COMPONENT-ORIENTED SOFTWARE

In order to address the issues identified above, we argue for:

-   Development of modular architectures fulfilling various mission needs and using a number of well-defined building blocks with well-defined functional interfaces and non-functional properties.

-   Building on-board software systems as assemblies of software components with well-defined and controlled lifecycle, configuration, deployment and interaction with the execution environment.

-   Reusability of sub-assemblies of software units to fulfil requirements definition for different missions.

-   A set of tailorable (at build time) and configurable (at runtime) services which offer a set of well defined primitives.

The goal is to define a software architecture generic enough so that it could be used for building applications in certain space application domain. In our view, such an architecture is reflected by a *component model*. A generic middleware implements this component model. The middleware provides a set of programming and computational abstractions and a set of well-defined services. The middleware must be tailorable and configurable to fulfil needs of both complex on-board applications and simple ones, however it is expected that different implementations will be developed for different system families. Preferably, the middleware itself is composed of components and the desired functionality of the middleware is achieved by composition of components. The applications are also components (from the point of the view of the component model the same type of components as the ones forming the middleware, perhaps with specific functionality or privileges) and they use via required interfaces functionality provided by the middleware.

Middleware services shall be regular components with special interfaces allowing them to be plugged in to the middleware. The set of services shall not be fixed but instead generated based on the application building block requirements reflected by their configuration. A carefully crafted component runtime can be small enough so as not to present a significant addition to the validation requirements. The existence of Real-Time CORBA implementations for mission critical systems illustrates this point.

The following list summarises the benefits of having a component-oriented framework for spacecraft on-board systems:

-   **Reduced development and integration costs**: The framework will allow for software reuse of building blocks across multiple missions and multiple primes. Also, thanks to following the framework, system families will have standardised requirements

which should in turn lead to having standard solutions, tailorable to mission-specific requirements. A component-oriented approach has been successfully applied by Phillips in the Koala component model used in software product lines for consumer electronics. Koala allows late binding of reusable components with no additional overhead [17].

- **Static and dynamic adaptability**: Different system families have different real-time and dependability requirements. We believe that the component-oriented framework could be tailored and configured for different system families. With a very little overhead, the component architecture can also be preserved at runtime. When this is the case, the explicit notion of interfaces makes it possible to detect and intercept component activity, a feature that is important for software updates at runtime (potentially dangerous updates to executing code can be avoided by temporarily suspending activity at component interfaces).

- **Model-based design**: We argue for the component-oriented framework being used as a target for Model-Driven Architecture (MDA) techniques. A mission requirements analysis will produce a tailoring of the framework and architectural design described by a set of models (Simulink or a similar tool for the AOCS, UML or its profile for generic software, and domain-specific languages for specialised domain-specific software). This should make the validation phase based not only on testing but also on correctness-by-design techniques. This approach also allows for automation – the model-to-code transformation is validated and could be repeated multiple times. From the validation point of view, the introduction of components also decreases complexity. Having to explicitly define (provided and required) interfaces, component developers are less likely to introduce inadvertent dependencies between components. Explicitly defined dependencies such as communication also make it possible to construct runtimes that forbid undefined dependencies, which in turn means that validation does not have to cover such dependencies.

- **Verification**: To ensure dependability of the spacecraft software, several formal verification methods could be used. If components are utilised, the size of the software that can be efficiently verified is much bigger than in the non-component case. The reason is that the complexity of the verification problems decreases exponentially if the software under verification is divided into several smaller subsystems communicating in a well defined manner. In the case of component software, such a division is done implicitly by design. Three

verification methods are usually used – Software Verification Facility (SVF, an exhaustive simulation including the on-board software, on-board computer, all the hardware of the spacecraft and the environment conditions [10]), PolySpace checker [16] (verification of the on-board software source-code using abstract interpretation [8]) and the Open Loop Test (first stage of testing an algorithm implementation in software). In all these cases, the existing verification tools have to be adapted to perform the per-component analysis. Such a tool adaptation results in two benefits: better performance (and therefore applicability of the methods to much more complex systems) and an option of the incremental verification (e.g. in the case of considering a component update at runtime).

3   RELATED WORK

While the ASSERT project [2] was investigating simultaneously several technological paths (e.g. architecture, modelling, process) under several application perspectives and with certain technological background, the COrDeT project [7] focuses on a domain engineering approach, addresses core space systems of satellites and exploration probes, supports the harmonisation trend in Europe and details the notion of software components for space on-board systems.

The objectives of the DisCo project [9] were twofold: on the first hand the study shall result in the availability of a middleware product specific to the space domain; on the other hand a reference application fitting the needs of a complex payload requiring distributed processing was developed and used to evaluate and refine the space-oriented middleware. The DisCo Space-Oriented Middleware (SOM) provides a simple component-oriented application framework which allows the SOM to control the application lifecycle.

The DisCo approach followed by COrDeT is similar to the Automotive Open System Architecture (AUTOSAR), with standardised interaction between components, standardised component implementations and well-defined functional interfaces (provided, required) [3].

SOFA 2 is a component system employing hierarchically composed components [14, 4]. The main features offered by SOFA are as follows: (1) the component model is defined by means of its meta-model; (2) it allows for a dynamic reconfiguration of component architecture and for accessing components under the Service-Oriented Architecture (SOA) concepts; (3) via connectors, it supports not only plain method invocation, but various communication styles; (4) it introduces aspects to components and uses them to clearly separate the control (non-functional) part of a component from its functional part and to make the

control part extensible; (5) it uses formal specification for describing components behaviour. In addition, SOFA 2 is not only a tool for modelling components, but it provides a complete framework supporting all stages of application lifecycle from architecture modelling through design, development, verification and validation to execution in its runtime environment.

Both ComFoRT [5] and CRE [6] projects focus on verification of component-oriented software. In both cases the behaviour of a component is described via its implementation in a common programming language (C for ComFoRT, Java for CRE) while the interactions between components are specified via a high-level specification language (FSP for ComFoRT, behaviour protocols for CRE).

4 THE SOFA HI COMPONENT MODEL

In this section we introduce a component model which represents a generic software architecture for building applications composed of a set of building blocks (called components). After carrying out a survey of current component models both in academia and industry (e.g. Koala [17], AUTOSAR [3], CCM [12], SOFA 2 [14], Fractal [15]), we have decided to adopt the existing SOFA 2 component model. The justification of this is as follows:

- SOFA 2 provides a state-of-the-art component model, offering most features available in the domain of component-oriented software architectures.

- SOFA has been around for more than 10 years and evolved significantly towards a realistic design and runtime platform, being a result of tens of man-years of research and over one hundred publications.

- We especially appreciate the fact that SOFA 2 keeps the component architecture entities at runtime and provides an execution environment.

- SOFA 2 uses formal specification for describing and reasoning about behaviour of simple components as well as complex component-based architectures. This is foreseen to become important when considering semantics of components during e.g. component update.

- As a proof of the concept, prototypes of SOFA have been implemented in Java and C. Also, supporting tools have been developed or are under development.

- SOFA 2 is an open architecture.

We call our component model SOFA HI (where HI stands for High-Integrity) to keep the distinction between both the original SOFA 2 component model and also its implementation, as SOFA HI targets mission-critical spacecraft on-board applications with emphasis on timing and memory constraints. However, key features of SOFA HI remain compatible with SOFA 2.

4.1 Metamodel

SOFA HI provides a state-of-the-art hierarchical component model. Components are software building blocks characterised by their *provided* and *required* *interfaces*. Components could be nested into one another, hence applications correspond to hierarchies of software components bound via their interfaces.

In the following subsections we will describe elements of the SOFA HI component model, which is defined using the metamodel shown in Fig. 1.

4.1.1 Common Elements

The SOFA HI component model defines several basic elements that are used across the entities of the metamodel. *NamedEntity* is a class used as an ancestor class for any class which has a textual string name as an attribute. Version is an entity used to version other entities. For the sake of brevity we do not introduce any versioning system in this document, so we only assume that there is the *Version* class whose instances could be used to denote different versions of interfaces and components. For this purpose the SOFA 2 metamodel also defines the *VersionedEntity* class which is used as an ancestor for all versioned classes in the metamodel.

Note that due to space limitations the common elements are not shown in Fig. 1.

4.1.2 Interface

The *Interface* class represents an abstract interface. It has a name (it inherits from *NamedEntity*) and *isCollection* attribute which denotes whether the interface is a collection interface or not (see explanation later). Each interface represents a type, which is defined by the *InterfaceType* class. *InterfaceType* defines interface type using a *signature*, which is a non-empty set of function signatures written in the Interface Definition Language (IDL). Since interface types are immutable, the *InterfaceType* class is versioned, which is achieved by being inherited from the *VersionedEntity* class. The *isMandatory* attribute of the *Interface* class is used to distinguish whether an interface is a mandatory part of a component or not. The meaning of this will be in detail explained when talking about component implementations (Section 4.1.4).

Another attribute of the Interface class – *isCollection* – denotes whether an interface is a collection or not. Collection interfaces are used to represent building block entry points which are typically

used by multiple users at the same time and where each of the users keeps its own unique identifier (e.g. session id) which allows distinguishing between individual connections.

The *CommunicationStyle* attribute represents supported communication styles of the interface. The most common communication style is local function call, but more communication styles are possible (remote call, message passing, etc.) The attribute value is a list of string values representing communication styles supported. At runtime the value of the *CommunicationStyle* attribute plays a crucial role during the generation of runtime entities representing bindings between interfaces (connectors).

4.1.3   Component Type

The key entity in the SOFA HI component model is the *ComponentType* class which represents a black-box view of a software building block. *ComponentType* is characterised by a set of interfaces it provides and a set of interfaces it requires. Either of the two sets could be empty.[1] Provided and required interfaces are represented by *ProvidedInterface* and *RequiredInterface* classes which are associated with the *ComponentType* class.

4.1.4   Component Implementation

The *Component* class represents an implementation of a component type. A single component could implement multiple component types, that is, it has to implement all the provided interfaces and it requires for its function the sum of all the required interfaces. A component type could be of course implemented by multiple components.

In SOFA HI we distinguish between *primitive components* which correspond to a piece of code written in a programming language and *composite components* which are composed of a set of subcomponents. That is why the *Component* class is associated with the *Subcomponent* class, with zero-to-many cardinality. A

---

[1]   It certainly makes sense to have a component with only provided interfaces. Such component implements all functionality on its own and does not need to call any functions implemented by other components. It also makes sense to have a component with only required interfaces, i.e. a component which does not provide any functionality through its provided interface. Such a component could either provide its functionality performed by accessing a hardware device not reflected by the component type, or it could be a "top-level" component which only requires other components to be connected via their provisions (e.g. main control loop). However, it may appear to have not much sense to have a component with neither provisions nor requirements. Such a component would not be able to be connected to any other component. However, such a subcomponent can perform some processing or interactions with hardware (e.g. a load simulator does not provide or require anything, yet its job is purely to consume the CPU).

component with zero subcomponents is a primitive component.

Since subcomponents could also be composite components, we can see a component as a hierarchy of components with multiple levels of nesting.

4.1.5   Interface Compatibility

In order to be bounded to one another, two interfaces must be *compatible*. The compatibility is defined as follows:

-    Two interfaces can be bound to one another only if they have at least one supported communication style in common.[2]

-    Two interfaces can be bound by a *connection* if the provided interface contains all operations listed in the required interface. That is to say that there could be some extra operations in the provided interface, which are not required.

-    Two interfaces can be bound by a *delegation* if the subcomponent's provided interface contains at least all operations listed in the parent's component provided interface. That is to say that there could be some extra operations in the subcomponent's provided interface, which are not used by the parent component.

-    Two interfaces can be bound by a *subsumption* if the parent component's required interface contains at least all operations listed in the subcomponent required interface. That is to say that there could be some extra operations in the parent component's required interface, which are not used by the subcomponent.

4.2   Runtime Support

4.2.1   Controllers

Every component has its *control part* which does not contribute to the functionality of the component, but it rather manages some of its non-functional properties and functions required by the SOFA HI application framework. The control part is composed of a set of *controllers* represented by the *Controller* class.

---

[2]   Optionally, even interfaces with completely different communication styles could be bounded one to another. The compatibility of communication styles is then resolved as late as during the assembly phase. The connector generator can resolve, whether it can create a connector for the binding (e.g. a connector can allow a binding of a local function call to a remote function call by implementing a CORBA stub).
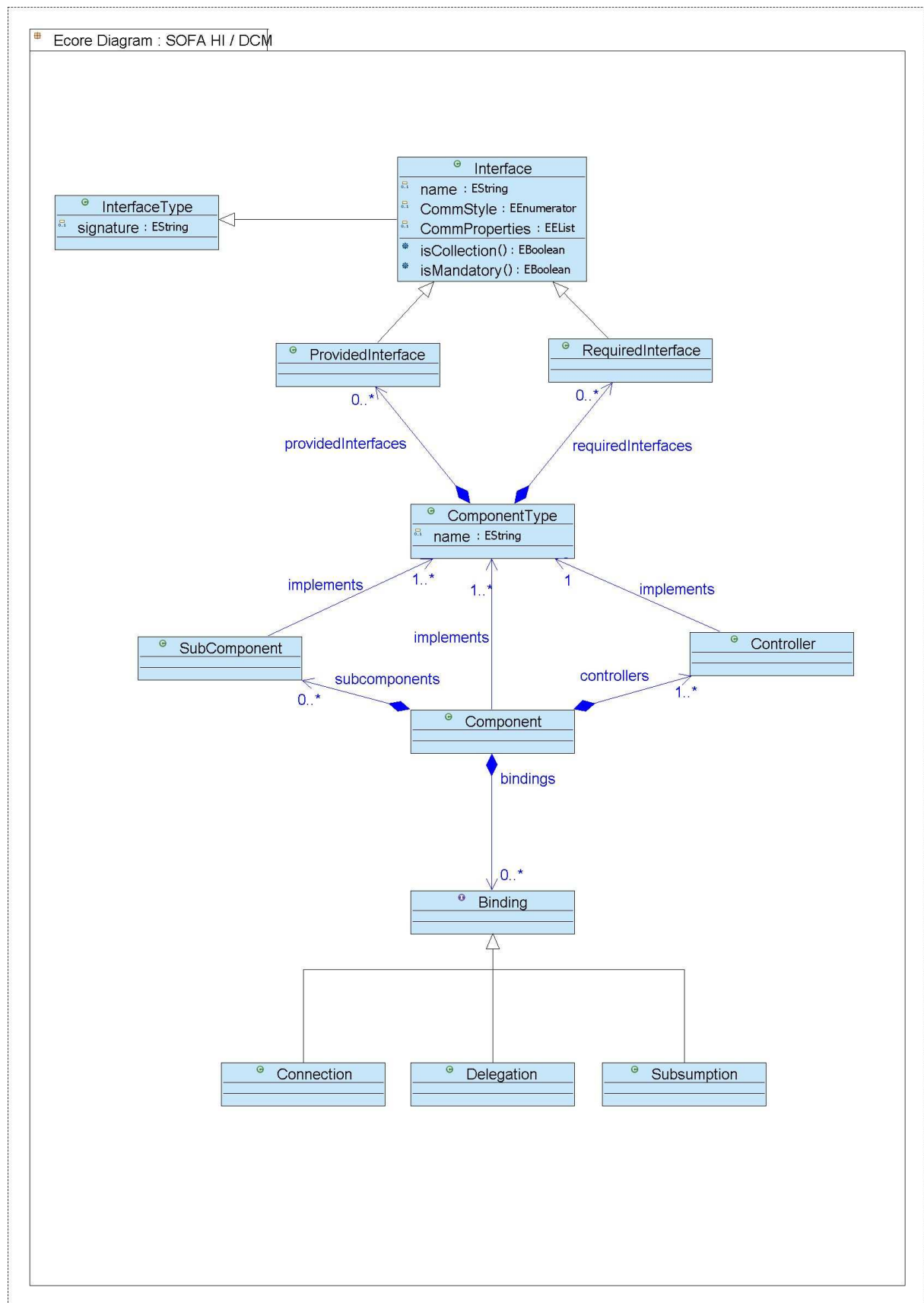
Figure 1. The SOFA HI metamodel

Technically, controllers behave like primitive components with an empty control part and pre-defined interface. The set of component's controllers is extensible, depending on a particular configuration. However, some controllers are mandatory, such as the `LifecycleController` which is in charge of controlling the component lifecycle, and `BindingController` which is in charge of connections with other components.

Thanks to controllers, the runtime execution framework (i.e. the SOFA HI middleware) does not see only the functional interfaces of components, but also controller interfaces which it uses to control the component behaviour (lifecycle, connections, updates, etc.) at runtime (see Fig. 2 as an illustration).

Controllers are in fact aspects which are applied to components as late as at the assembly time. However, not every component is ready for any controller. The conclusions of [11] show that certain features (such as transactions) are not easily added to software components in the form of aspects, especially without prior knowledge of this during the component development. We follow the Jironde concept introduced in [13] where component controllers (a.k.a. aspects) declare a set of requirements for component in order to allow them to be aspectised.

4.2.2 Connectors

Connectors are runtime entities which represent bindings. Connectors are generated automatically from the component description and configuration properties specified in the *deployment descriptor*. The generation is performed at the *deployment time*.

The selection of connector types to be generated is limited by the communication styles which the component supports. Remember that the communication style is an attribute of the `Interface` class, hence an interface providing an interface type defines which communication style is supported by the component. This might seem as an unnecessary limitation, but we believe that in practice the component must be aware whether e.g. it is called locally or remotely (i.e. it must follow a particular RPC-like framework). Note that the compatibility of interfaces is checked at the `Interface` level, not at the `InterfaceType` one.
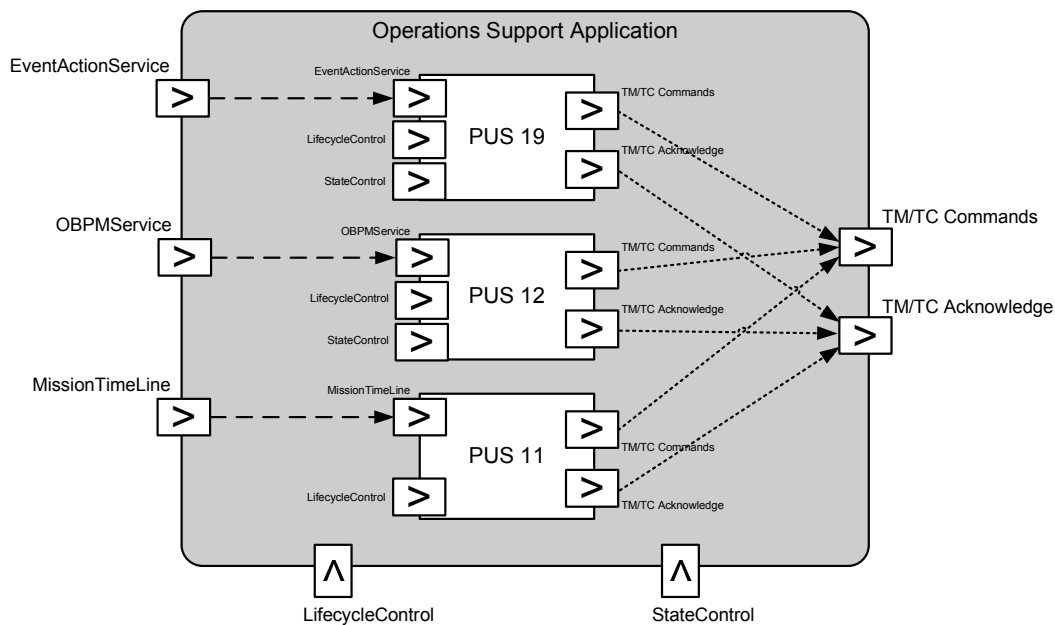

Figure 2. The Operations Support Application component with controller interfaces

4.2.3 Dynamic Reconfigurations

By *dynamic reconfiguration* of an application we mean a modification of the application architecture at runtime. There are many modifications of the architecture possible, e.g. adding or removing a binding, adding or removing a component, etc. At present we only consider

dynamic component updates, i.e. replacing of a component with another one. On those SOFA HI imposes the following requirements:

- The new component has at least all provided interfaces as the original component, but it can have more. In case there are new provided interfaces,

there are nevertheless no additional bindings added automatically, connecting these with the component siblings or its parent component.[3]

- The new component has at most all required interfaces as the original component, but it can have less. In case a required interface is not present in the updated component, all its original bindings get removed during the component update.

The second requirement allows replacing components with updated ones which have fewer requirements imposed on other components. Dynamic updates are relatively easy to handle comparing to arbitrary modifications of the component hierarchy, as the update is only visible for the component being updated and remains transparent for the rest of the application.

4.2.4 Deployment

SOFA HI components are deployed into the SOFA HI deployment environment called *container*[4] which offers components a set of pre-configured services and also uses component controllers to manage their lifecycle, updates, etc. A container is a single unit of deployment of the SOFA HI middleware which manages distribution over a set of processing modules (nodes). In case of a uniprocessor application, there is only a single container and the distribution feature is not used (the SOFA HI execution environment is tailored accordingly).

In addition to distribution the SOFA HI container provides a set of services which are available for the applications. A *Component Repository* is available which allows uploading new interfaces and components as well as downloading new versions of components when a component update is performed.

An application can span multiple containers as component bindings are at runtime represented by connectors. These can realise the component interactions modelled as bindings using e.g. remote operation invocations, in addition to other possible communication styles, such as local function call or passing messages via message queues.

4.3 Software Development Lifecycle in SOFA HI

Let us recall all the phases of the software development lifecycle considered in SOFA HI. The phases are as follows:

1. **Design**: In this phase components and applications are designed by means of defining interfaces, and component types.

2. **Implementation**: Components which implement one or more component types are implemented. Implementation of composed components might involve reuse of existing components.

3. **Assembly**: This phase involves selection of appropriate controllers and connector generation. After this phase all "binaries" are ready for deployment and execution.

4. **Deployment**: In this phase the components are deployed into containers located on target processing modules, eventually to the Component Repository.

5. **Execution**: After a component is deployed into one or more target containers, it can be executed as a part of a software application.

It is worth noting that SOFA HI follows incremental design, so that development phases of individual components do not necessarily have to be synchronised (for instance, while one component could be already in the execution phase, another only becomes designed and is implemented, assembled, deployed and eventually executed later).

SOFA HI provides primarily a *composition view* of a software system. If combined with a model-driven engineering approach, other sorts of views could be used to create a set of models describing the system. Model transformation and/or code generation techniques could be used to generate component types, and implementations, controllers, connectors and deployment descriptor compliant with the SOFA HI component model. Some of the alternative views shall reflect the system's *dynamic architecture* i.e. reasoning about activities (composed of tasks and mapped to chains of component operations), synchronisation, execution times, deadlines, etc.

4.4 Formal Verification

In order to support automatic verification of component behaviour, a behaviour specification could be associated with each component type in SOFA HI (the specification language is based on *behavior protocols* [1]). There are two main verification steps: (1) behaviour compatibility among all components in the system is verified (only the behaviour specifications are needed here), and (2) for each primitive component, the compliance of the component implementation with the behaviour specification is checked; this way, the whole system is verified. The properties to verify include e.g. deadlock freedom, correctness of dynamic update, or compliance with a specific communication protocol. The information from the alternative design views of the

---

[3]   As the component is considered a black box, its internal structure does not affect its view from the outside. Hence the new provided interfaces are perhaps within the component connected to new subcomponents, which is perfectly legal in the dynamic update scenario.

[4]   SOFA 2 also uses the term *deployment dock*.

system may be used either for optimisation or to extend the class of the verified properties (e.g. to real-time behaviour).

4.5 Computational Model

Computational model for a software system defines abstract system entities representing computations, data exchange between computations, and their temporal and concurrency properties. Based on the computational model, space and temporal properties of a piece of software could be derived. A particular computational model is reflected by the programming model and requirements imposed on the underlying operating system, communication protocols and wires, hardware access protocols, etc.

The SOFA HI component model neither imposes any requirements on the computational model, nor is it affected by which computational model is used by the applications. However, the implementation of the SOFA HI middleware will follow and enforce a particular computational model.

In the present practice, all software applications are compliant with a computational model defined for the whole software system. The goal of the COrDeT study is to reuse software components across different systems, and it must be considered whether we want to impose a single computational model on all components. The implication for SOFA HI is that the compatibility is not at the component model level, but rather at the level of its concrete implementation.

5 CONCLUSIONS

This paper presents our vision for architecture and development of spacecraft on-board software composed of components. The paper is focused on software architecture and corresponding engineering, verification and validation process, which would in our opinion leverage software reuse across different space missions and reduce software development and integration costs.

REFERENCES

1. Adamek, J., Hnetynka, P. (2008). *Perspectives in Component-based Software Engineering*, Proceedings of SEESE 2008, Leipzig, Germany, ACM Press.

2. ASSERT, Automated Proof-based System and Software Engineering for Real-Time Systems (2004-2007). http://www.assert-project.net.

3. Automotive Open System Architecture (AUTOSAR), http://www.autosar.org.

4. Bures, T., Hnetynka, P., Plasil, F. (2006). *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*, Proceedings of 4th ACIS International Conference on Software Engineering Research, Management and Applications (SERA), Seattle, USA, IEEE CS.

5. Component Formal Reasoning Technology (ComFoRT), Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/pacc/comfort.html.

6. Component Reliability Extensions for Fractal Component Model (CRE), Academy of Sciences of the Czech Republic, France Telecom, http://kraken.cs.cas.cz/ft/public/public_index.phtml.

7. COrDeT, Component-Oriented Development Techniques (2007-2008). ESA AO/1-5237/06/NL/JD.

8. Cousot, P., Cousot, R. (2001). *Verification of Embedded Software: Problems and Perspectives*, Proceedings of the First International Workshop on Embedded Software, LNCS, Volume 2211.

9. DisCo, Compact Computer Core Software Architecture for Onboard Scalable Super DSP System (2005-2007). ESA AO/1-4552/04/NL/JA.

10. Eickhoff, J., Falke, A., Röser, H.-P. (2006). *Model-based design and verification—State of the art from Galileo constellation down to small university satellites*, Acta Astronautica, Volume 61, Issues 1-6, June-August 2007, Pages 383-390. Bringing Space Closer to People, Selected Proceedings of the 57th IAF Congress, Spain, Valencia.

11. Kienzle, J., Guearraoui, R. (2002). *AOP: Does it Make Sense? The Case of Concurrency and Failures*, Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02), pp. 37-61, Malaga, Spain.

12. Object Management Group (2006). *CORBA Component Model Specification*, *Version 4.0*, formal/06-04-01.

13. Prochazka, M. (2003). *Jironde: A Flexible Framework for Making Components Transactional*, Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2003), Paris, France.

14. SOFA 2, http://sofa.objectweb.org/.

15. The Fractal Project, http://fractal.objectweb.org/.

16. The PolySpace Checker, http://www.mathworks.com/products/polyspace/index.html.

17. van Ommering, R., van der Linden, F., Kramer, J., Magee, J. (2000). *The Koala Component Model for Consumer Electronics Software*, IEEE Computer, Volume 33, Issue 3.