Supporting real-time features in a hierarchical component system

Petr Hošek[1,2], Tomáš Pop[1], Tomáš Bureš[1,3], Petr Hnětynka[1], Michal Malohlava[1]

December 2010

Technical report No. 2010/5, December 2010
Version 1.0, December 2010

[1]Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics,
Charles University, Malostranske namesti 25,
Prague 1, 118 00, Czech Republic

[2]Department of Computing
Imperial College London
South Kensington Campus
180 Queen's Gate
London SW7 2AZ

[3]Institute of Computer Science,
Academy of Sciences of the Czech Republic
Pod Vodarenskou vezi 2, Prague 8,
182 07, Czech Republic

Abstract

The SOFA 2 component system allows development of high-integrity real-time embedded systems using the component-based development approach. SOFA 2 employs a hierarchical component model and many advanced features which may be useful in this area of software development. The report offers discussion about necessary changes and features that needs to be incorporated into the SOFA 2 component system. Design and prototype implementation of the extension is also described in order to realise and prove the viability of the proposed concepts. This implementation allows development of high-integrity real-time embedded systems by its decomposition into components which offers strict separation of concerns and higher reuse. The prototype implementation aims to reuse as much as possible of the existing SOFA 2 tools and code. Use cases and comparison with related work is provided to demonstrate usability and features of the prototype implementation.

1 Introduction

With constant evolution of consumer electronics, the complexity of embedded software used by these devices grows exponentially [11]. There are also many other areas requiring the use of embedded real-time systems such as automotive industry, spacecraft engineering, etc. Real-time computing is playing a crucial role as an increasing number of complex systems rely on computer control.

Development of these systems is really challenging especially due to the number of requirements and restrictions that have to be met. These systems are often limited in terms of computational power, available memory and consumed energy. On the other hand, impact of failure in such system can have significant or even critical consequences, usually due to their direct interaction with the physical environment.

Due to aforementioned facts, embedded software development is becoming the significant bottleneck.

Therefore, there are high demands for methods that would ease the development of software for these devices as well as allowing this software to be composed of existing components. As has already been proved in many areas of software development, component-based software development is an approach that simplifies and speeds up the development of software systems due to separation of concerns and emphasis on code reuse which consequently leads to lowering the costs.

Many component systems supporting component-based software development exist today. One of them is *SOFA 2* [2] which is an advanced component system developed at Charles University providing many advanced features such as hierarchical components and transparent distribution. Being direct successor of the SOFA component model, SOFA 2 is completely based on model-driven approach. This allowed to profit from combination of component-based and the model-driven development techniques.

The component-based software development approach has also begun to establish its position in the area of embedded and real-time software development. Because SOFA 2 does not support development of such systems, *SOFA High Integrity* (SOFA HI) research vision emerged. The effort behind this vision is to bring knowledge gained during development of SOFA and SOFA 2 component systems into domain of high-integrity real-time embedded systems development. This requires an adjustments of the SOFA 2 component system to the specifics and requirements of embedded and especially real-time systems development.

The purpose of SOFA HI, introduced in [3], is to bring the knowledge of hierarchical component systems into real-time environment in order to speed up the development and lower the costs of high-integrity systems, especially spacecraft on-board software. This idea originated at SciSys [25] motivated by the European Space Agency [26] SAVOIR[1] initiative. Research vision is therefore to fill the gap between real-time programming and today's software technology.

Ultimately, this work could inquire the means to clarify, simplify and speed up the development of real-time embedded software systems by combining well-known approaches

[1] Space avionics open interface architecture

that are already being used in different areas of software development.

The structure of the report is following. The Section 2 provides detailed analysis of the realisation. The Section 3 describes the high-level design overview and Section 4 provides description of the prototype implementation. The Section 5 evaluates the realisation using sample use cases while Section 6 concludes the report.

2 Real-time systems development

The following section provides overview of process, concepts and techniques used in high-integrity real-time embedded systems design and development. Description of real-time system design process is accompanied with discussion related to real-time properties support, followed by discussion about scheduling, simulation and modelling. Development process of such systems is also discussed, especially their implementation and development environment. The section is concluded with identification of requirements that need to be supported for successful design and development of high-integrity real-time embedded systems.

2.1 Design of real-time systems

Majority of real-time embedded systems is used as control applications. This means that in general, their logic consist of sensory acquisition, control and actuation. The interactions between system and the environment are done by peripheral subsystems. Sensory subsystem acquires information from environment through a number of sensors while actuation subsystem modifies the environment through a number of actuators.

These applications usually require periodic acquisitions of multiple sensors. The actions produced by the actuators strictly depend on the current sensory information. Therefore, various timing constraints are usually imposed on such applications.

Generally used design process [4] of such systems looks as follows:

1. The application is structured in number of concurrent tasks related to activities performed.
2. Proper timing constraints are assigned to individual tasks considering individual task dependencies.
3. Use predictable operating environment that allows to guarantee satisfaction of the specified timing constraints.
4. Schedulability verification is done to guarantee that timing constraints can be satisfied.

Timing constraints are usually denoted as real-time attributes of the real-time application. These attributes and especially their verification as well as adherence is the key part of each real-time system. The real-time embedded systems need to deliver their results on time in order to be correct and the correctness itself is defined by the timing properties.

2.1.1 Schedulability verification

Very important part of real-time systems design is the verification of real-time properties. This usually means the *schedulability* of the whole system. Schedulability can be defined as [4]:

> "A set of tasks is said to be *schedulable* if there exists at least one algorithm that can produce a feasible schedule where a schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints."

Even thought this problem belongs to the most important in the area of real-time software development, it has still not been successfully resolved. However, there are emerging approaches that seems to be feasible.

One such approach is represented by TDL [36], a language conceptually based on the time triggered modeling language Giotto [15], accompanied with simple component model that allows one to specify the timing behaviour of a hard real time applications in a descriptive way separating the timing aspects of such applications from their functionality. Set of tools is also provided for the TDL language that allows to model timing requirements of such applications and verify their schedulability.

Prime importance for the schedulability analysis of hard-real time systems is the knowledge of WCET[2]. Even thought there are tools for WCET analysis, their usability is very limited as they work only for simple programs and their support for different hardware platform is limited due to presence of features that improve processor performance such as caches, branch prediction or pipe-lining.

State-of-the-art WCET analysis tools are AbsInt aiT [34] and Bound-T [35]. However, none of these tools support analysis of a single component as they only support analysis of the whole programs. Therefore it might be challenging to adapt these tools in order to use them for analysis of component-based systems.

2.1.2 Simulation and modelling

Due to issues of the schedulability analysis, simulation is often used as a counterpart or even instead of schedulability analysis in the case of larger real-time embedded systems. Simulation allows to observe and analyze behaviour of real-time systems during its execution. This may be useful in many cases by identifying possible timing constraints violations.

There are many frameworks and tools available that allows simulation of real-time systems.

Many of these frameworks use component abstraction. This means that they require specific implementation of components as an input of simulation. This implementation might not be easily obtained from existing implementation because it has to be done in specific programming language which usually differs from language used for the implementation of real-time embedded system itself. The implementation also needs to use specific programming interface of the simulation framework. This rather limits the use

[2] worst case execution time

of these frameworks, especially in an automated way. Representatives of such simulation frameworks are OMNeT++ [37] and Ptolemy II [38].

Another approach often used in these frameworks is the use of model-driven approach. This means that the application logic is modelled by using means provided by the simulation framework. Downside of this approach is that the resulting model needs to comprise entire application logic and it may be difficult to obtain such model from existing system implementation, especially in the case of large real-time embedded systems. Representatives of these frameworks are SCADE Suite [39] and MATLAB Simulink [40].

2.1.3 Planning and scheduling

Strongly related to schedulability is the approach used for planning and scheduling of real-time tasks. Environment used for real-time system implementation needs to provide scheduling support that allows to guarantee satisfaction of the system tasks specified timing constraints. This is typically achieved by use of scheduler which employs some scheduling algorithm. Overview of scheduling algorithms is provided in [4]. Typically used approach is to rely on scheduler provided by the real-time operating system used. However, there are situations where this approach is insufficient and another needs to be used.

Such situation occurs when scheduling of different resources, especially in the case of distributed systems, is needed. Specialised scheduler frameworks might be needed in these situations. Representative of such framework is FRESCOR [33] project. The main objective of this project is to integrate advanced flexible scheduling techniques directly into an embedded systems design methodology, covering all the levels involved in implementation, from operating systems primitives through middleware up to application level. Downside of the scheduling frameworks is that their implementation is usually non-trivial and rather heavyweight and it therefore brings rather high overhead to the application itself and limit general verifiability of the system schedulability.

Completely different approach to scheduling is off-line scheduling where no scheduler is actually being used and the schedule is incorporated into application itself. This may be very well applicable in the case of many systems when it is possible to generate static global schedule of the whole application. The biggest drawback of this approach appears in situation when pre-emption is needed. This would mean the fragmentation of the application logic which is not always possible.

2.1.4 Computational model

For development of high-integrity real-time embedded systems, restricted computational models are often used to allow these systems to be statically analysable in terms of functionality and schedulability. This means that some features of a programming language and target platform with high overhead and complex semantics are removed for the sake of reliability.

The most often used computational model for development of high-integrity real-time embedded systems is the Ravenscar Computational Model defined by Ravenscar Pro-

file [10]. This profile defines several restrictions for high-integrity hard real-time systems. Despite being originally designed for the Ada programming language, it can be very well used for other programming languages as well. Many different standards used in avionics, spacecraft and military for development of safety-critical systems consider this profile.

Goals of the Ravenscar profile are following:

1. Predictability of memory utilisation.
2. Predictability of timing.
3. Predictability of control and data flow.

Therefore, in order to achieve these goals, the Ravenscar profile forbids several features often used in software systems development. These are in general:

Task types and object declarations other than that at library level are prohibited and therefore there is no hierarchy of tasks.

Dynamic allocation and unchecked deallocation of protected and task objects is not allowed.

Tasks are assumed to be non-terminating, therefore task abortment statements are not allowed.

Tasks can have only static priorities, therefore dynamic priorities are forbidden.

Time can be only used in the context of real-time clock, usage of other time-related functions are not allowed.

Delays have to be always specified as absolute value, therefore there are no relative delays.

There are no user-defined task attributes.

Protected types and object declarations other than at the library level as well as protected types with more than one entry are forbidden.

Protected entries with barriers other than a single boolean variable declared within the same protected type are also not allowed.

Moreover, it may be appropriate, especially in the case of safety-critical systems, to describe the used computational model more formally using some formal specification method such as Alloy [43] or Event-B [44] as these methods provide tools for verification of specification correctness.

2.2 Development of real-time systems

Development of high-integrity real-time embedded systems is affected by the requirements of these systems and their design which has been already discussed.

Using selected computational model, real-time embedded systems are typically implemented directly for selected target platform. These target platforms are often limited in terms of available resources such as memory or computing power. Therefore, it is important to target these limitations during real-time embedded systems development and verify that resulting system meet these limitations.

Development of safety-critical systems usually has to follow different standards so that developed system is compliant with these standards. Moreover, these systems usually need to be certified before use. Number of certifications exist and are required in areas where safety-critical systems are often used. Example of such standards are DO-178B, ARINC 653 or IEC 61508.

2.2.1 Real-time system implementation

Majority of real-time embedded systems uses C or Ada as their implementation language. Despite Ada orientation towards embedded high-integrity application development, C is actually much more widespread in the industrial area. According to many different sources, C has also already started to overtake Ada positions even in the area of military and space research software development where this language once dominated.

Embedded systems are often implemented against the API[3] provided by the underlying real-time operating system allowing to use the tasks, access the shared resources, etc. Problem of these systems is that their interface usually widely differs which limits the portability of the embedded systems implementation. There is a POSIX.1 standard together with POSIX.1b real-time extensions which should act as a unifying interface, but in the reality, it misses some important parts needed for the real-time application and therefore many real-time operating system use their own interface which is usually significantly different from the standard POSIX interface.

2.2.2 Variability of real-time systems

The variability in software systems can be viewed from three points as design time, compile time and run-time variability. These three views are also present in real-time systems in different forms.

The design time and runtime variability is typically represented by *operating modes*. Many embedded real-time systems exhibit variability by different phases of operation and control in their design and implementation. This means that the functionality is different at different system states. Therefore, it is possible to divide system into different operating modes. Typical real-time embedded applications have several operating modes [7]. Each of these modes has different behaviour characterised by the set of functionalities carried out

[3] application programming interface

by different task sets. These modes allow application architecture reconfiguration during the application lifetime as described in [6].

Moreover, embedded real-time systems often require support for run-time variability represented by dynamic reconfiguration and update at run-time. Downside of this features is the amount of introspection and control information needed to implement these variability requirements. This may be a problem when minimal memory footprint of the resulting embedded system is required. Therefore, this information may be optionally included or excluded during compilation phase. This represents the compile time variability.

2.2.3 Distributed and interprocess communication

Support for distributed communication in real-time embedded systems, while not being common, is becoming more and more important with growing amount of interconnected systems used.

Distributed communication allows processes running on same or even different physical nodes to communicate together. This is more difficult in case of real-time systems, especially in the case of remote communication, as it is often problematic to meet real-time properties in such case. Therefore, use of specialised network protocols such as RETHER [16] or RT-EP [17] is required. Some real-time operating systems employ such protocols for different networking technologies and computer buses. Some buses such as CAN bus, or more precisely their communication protocols, may even be specially designed to support real-time communication. Middleware providing real-time communication capabilites for some buses also exists such as COSMIC [18] for CAN bus.

Special case of distributed communication is an interprocess communication. This is often required as many real-time operating systems used for running real-time applications support processes with different timing requirements (such as hard and soft real-time or even non real-time) running together. Therefore, interprocess communication needs to be used when such processes need to communicate with each other.

2.2.4 Development tools and environment

Development tools belong to one of the most important requirement in development of any software systems, including real-time embedded systems. Among common requirements on developments tools belongs support for used programming language or integration with compilation toolchain and other utilities. More sophisticated environment may offer other features such as support for debugging, unit test frameworks, refactoring tools, etc. They may also integrate other embedded tools and languages for modelling, static checking, etc.

Except of these common requirements, there are also requirements specific to development of real-time embedded systems. Embedded systems development requires support for target device platform, often provided by the device manufacturer. Environment for real-time systems development should provide streamline integration with tools that support real-time properties analysis.

2.2.5 Development methodology

Development methodologies are often used for development of large software systems. System development methodology is a guideline that is used to structure, plan and control the process of development of software system. While many development methodologies exist for development of non real-time systems, that are very few methodologies targeting specifics of real-time embedded systems development [13, 14]. Therefore, general methods are often used even for development of real-time embedded systems even though they are not very suitable for development of these systems.

2.3 Requirements on real-time systems

Development process of real-time embedded systems was presented. This process impose general requirements on embedded and real-time systems [4, 5]:

support for periodic and aperiodic tasks **(R1a)**

support for modeling of real-time attributes **(R1b)**

support for real-time task scheduling at run-time **(R1c)**

scheduling analysis and verification support **(R2)**

support for application operating modes **(R3a)**

small or configurable memory footprint of run-time environment **(R3b)**

support for dynamic reconfiguration at run-time **(R3c)**

platform independent and portable implementation support **(R4)**

support for distributed and interprocess communication **(R5)**

development tools and development environment **(R6a)**

development methodology or guidelines for development process **(R6b)**

These requirements have different significance during real-time systems development. The requirements (R1a), (R1b) and (R1c) are typicall considered as crucial. The requirements (R6a) and (R6b) are important as well as (R2). The requirements (R3a) and (R3b) are considered as regular but they are not neccessarily needed. The requirements (R3c) and (R5) are not needed in many cases and therefore considered as optional.

3 SOFA High Integrity specification

This section provides a specification of the SOFA HI profile and addresses the requirements provided in the previous section. New concept of SOFA 2 profiles, which is the key design prerequisite of SOFA HI implementation, is presented in the beginning of this section. The section then continues with the description of component model extensions as this is the key part of the SOFA 2 component system which represents the high-level view. The current state of the SOFA 2 component system together with a detailed discussion regarding SOFA HI realisation is also provided. Necessary changes are discussed as well as new concepts and features.

3.1 SOFA 2 profiles

SOFA HI should allow development of high-integrity real-time embedded applications by addressing requirements presented in Section 2. Moreover, SOFA HI implementation should be completely based on SOFA 2 component system and its implementation should reuse as much as possible the existing SOFA 2 implementation and tools.

Therefore, many parts of existing SOFA 2 implementation need to be shared with SOFA HI implementation. However, there are still parts specific to each implementation. Because this may be common for other future implementation of SOFA 2 as well, the new concept of *SOFA 2 profile* needs to be defined. This concept is implied by the requirements on SOFA HI implementation. The SOFA 2 profile is a specific implementation of SOFA 2 deployment and runtime environment built on top of common meta-model and development environment. SOFA 2 profiles should share the development environment as well as development tools.

According to this definition, existing implementation of SOFA 2 also forms a profile which is called *SOFA/J* as it is completely based on the top of the Java environment. Similarly to SOFA/J profile, the SOFA HI is also implemented as a profile of SOFA 2 component system. This means that many parts of the system are shared and most of the differences are in application deployment and runtime parts.

3.2 Component model

Being a profile of SOFA 2, SOFA HI is aiming to use its component model. However, the model has to be extended in order to support specification of properties related to real-time nature of the application. Using these extensions, properties of the modelled component-based application such as activity of components, their instantiability or definition of extra-functional properties can be specified. Moreover, there are other requirements that need to be addressed such as support for different programming languages or architectural modes.

3.2.1 Active and passive components

Widely used approach, especially in the domain of real-time component systems, is separation of components to active and passive, in order to address the requirement (R1a) (defined in Section 2). Active component is usually defined as a component which contains its own thread of execution and properties regarding its periodicity, deadline and priority while passive component is a standard unit of composition providing and requiring services. When being called, execution of passive component is carried out in the context of the active component demanding its services.

During system design phase, it is therefore needed to distinguish between active and passive components. What is also worth mentioning is the fact that this distinction has meaning only for the primitive components. Composite components do not have implementation and therefore, they do not have their own thread of execution during runtime. Moreover, in most hierarchical systems, composite components do not even exist during runtime.

Classification of active and passive components is allowed by extension of component model. The `Architecture` meta-class is extended with `active` attribute of boolean type describing whether the component is active or passive. Marking component as active will result in component having its own thread of execution.

3.2.2 Singleton components

Another useful concept in the view of embedded systems is the knowledge about instantiability of the component. There are many components in these systems that realise servicing of a single hardware device. These components are singletons by nature and it is therefore useful to make it possible to mark these components as singletons during system design phase.

Similarly to the previous extension, in order to allow marking component as singleton, the `Architecture` meta-class is also extended with `singleton` attribute of boolean type describing whether more than one instance of component can exist in the system.

3.2.3 Extra-functional properties

To address the requirement (R1b), active components typically also contain properties regarding its periodicity, priority, deadline etc. These have to be defined by a component developer and therefore, concepts of properties already contained in SOFA 2 meta-model can be used to specify these values.

The only drawback of the existing concept is that it cannot enforce definition of common properties such as the ones mentioned regarding to active components. This problem has to be addressed so that existing concept can be used for specification of extra-functional properties of component.

To support specification of extra-functional properties, component model is extended with the new *property set* entity, represented by top-level meta-class `PropertySet`. Property set is a named collection of properties represented by `Property` meta-class.

`PropertySet` meta-class can be then referenced by `Architecture` using `propertySet` attribute enforcing definition of the properties contained inside property set. Properties defined inside property set are merged with properties defined directly on component and their values have to be set during deployment as a part of deployment plan.

The meta-model of property set can be seen in Figure 1.

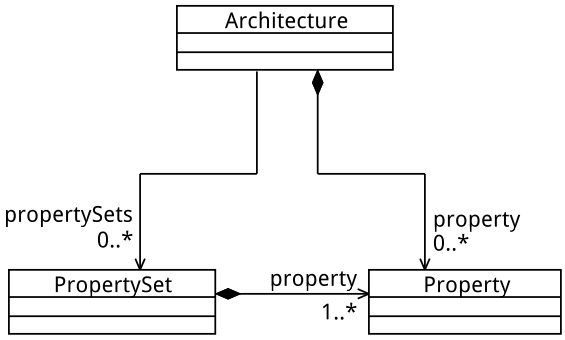


Figure 1: Property set

Extra-functional properties, needed for real-time features support, can be then modelled using predefined property sets such as property set for periodic component, containing properties for periodicity, deadline, priority and etc.

3.2.4 Architectural modes

Architectural modes allow addressing the requirement (R3a) by defining operating modes for each component. The component can describe a set of modes and subcomponents which are active in each of these modes.

From the viewpoint of component system, as the tasks are actually associated with active component, different modes can be described as a set of active components that are active in each mode. Some of the components can be active in different modes with different properties such as their periodicity, priority or deadline. Therefore, it is needed to associate the values of these properties not only with components themselves, but with component's mode as well.

What needs to be considered is that there is no single architectural view of the application in SOFA 2 component model. Each composite component identifies only its subcomponents and their connections. Therefore, architectural modes have to be defined separately for each composite component. This increases overall architecture complexity as well as its re-usability which is the main idea of hierarchical component systems.

The architectural modes are modelled as a part of component definition on different levels. The definition of component is extended with definition of component state modelled by `State` meta-class. States that are visible to other components are defined on component *frame* while states that are visible to component only are defined on component *architecture*. The architecture also contains definition of possible transitions between states modelled by `StateTransition` meta-class which contains references to source and target states

along with the condition under which the transition can be executed. The condition itself is modelled by `StateCondition` meta-class with two subclasses - `ComponentStateCondition` allowing to guard the state of component itself and `SubcomponentStateCondition` allowing to guard the state of component's subcomponent.

The meta-model of architectural modes can be seen in Figure 2.

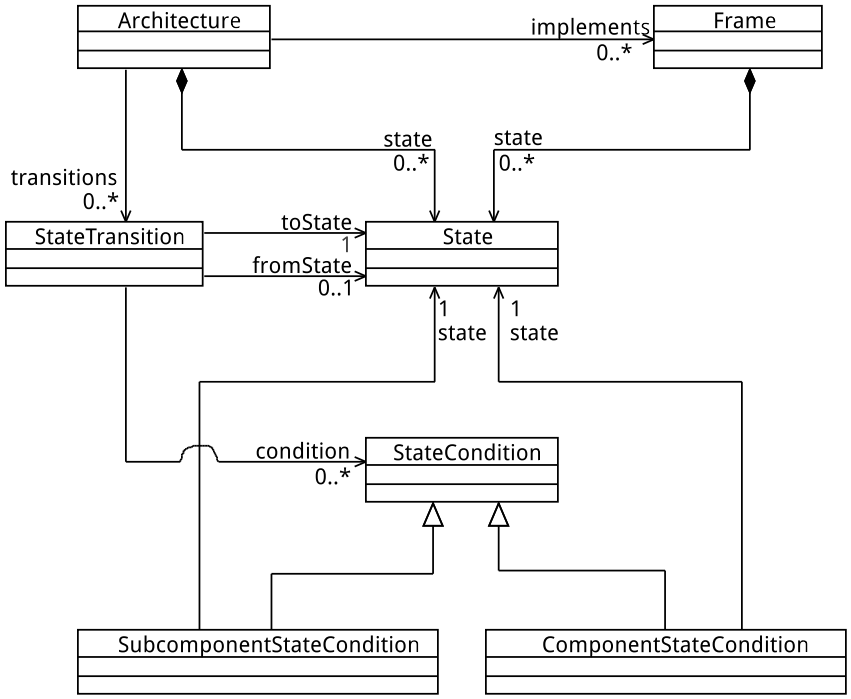


Figure 2: Architectural modes

3.3 Development and design process

Development process of SOFA HI applications is similar to development process of SOFA 2 or more precisely SOFA/J applications. SOFA 2 component model provides guidelines for development of component applications. This address the requirement (R6b).

Development of SOFA HI applications consists of composing already developed components available in repository. During this process, existing interface types, frames and architectures (along with implementation) are reused and new ones are defined.

Development of entirely new component in SOFA HI is following:

1. The interface type of component has to be defined or selected.
2. The frame is created with provided and required interfaces declared with appropriate interface types assigned.

3. The composite or primitive architecture is created (implementing the created frame). Primitive architecture has to be identified as active or passive, their properties needs to be defined. The interfaces has to be implemented within the architecture business code in case of primitive architectures or delegated to subcomponents. Component modes has to be also defined.

4. The assembly description of component application is defined. Architectures of all subcomponents are specified.

During development, all the created fragments of a SOFA HI component need to be stored in the repository. The SOFA 2 repository supports versioning, it is therefore possible to develop different versions of the same component in parallel.

3.3.1 Component development

Component implementation should be independent of any concrete environment. This increases portability of the code and makes it easier to use existing legacy code as a basis for component implementation. Moreover, no predefined code structures should be enforced because this again limit use of existing legacy code. However, this also limits the possibility to use different version of same component in one application as it is not possible to perform renaming of such unstructured code.

Component implementation is required to fulfil the restrictions imposed by computational model. The Ravenscar Computational Model is assumed as the SOFA HI computational model since it is very well suited for development of high-integrity hard real-time systems.

Component implementation itself will be done in C language which was chosen as the actual language out of two possible choices of programming languages for implementation of SOFA HI component. The main reason for this choose is its popularity in comparison with Ada. C is much more widespread among the programmers and companies. Therefore, it has much better support ranging from hardware suppliers to development tools. There is a number of tools for static analysis and verification as well as worst case execution time analysis tools targeting C.

3.3.2 Component portability

While there is no problem with portability of the SOFA/J component implementation due to portability of Java runtime environment, this is a problem for SOFA HI as the components are intended to be implemented in native language. Moreover, the component implementation needs to utilise the application programming interface of the underlying operating system to be able to use the tasks, access the shared resources, etc.

Therefore, to be able to support different operating systems, it is needed to create the operating system abstraction layer which would address the requirement (R4) by introducing the universal abstraction of the operating system application programming interface. Even thought there already exist some abstraction layers such as NASA OSAL [41] or

ASAAC OpenOS [42], these are not well suitable for the purpose of SOFA HI as they do not support all the functionality and platforms needed and their extensibility is limited due to restrictive or incompatible licence.

This abstraction over the underlying operating system API is inspired by the POSIX interface. However, differences and simplifications are included due to support for real-time properties. Most importantly, the abstraction provides interface for scheduling tasks to address the requirement (R1c). An implementation of the abstraction is provided for each supported operating system. The applications themselves are written against this interface. Particular implementation is selected at compile time. Component implementation itself is only allowed to use means provided this abstraction. This guarantees portability of component implementation between different platforms.

3.3.3 Component operating modes

Operating modes or more precisely their switching will be realised by special application service. This service is able to change component properties or even completely stop running component and start component which has been previously stopped. Hierarchical component model is flattened during application deployment and all the component states and transitions are merged together to form single automaton. This automaton is transformed into code representation and is therefore part of the mode reconfiguration service tracking the state of whole application during runtime. State transitions in this automaton trigger application reconfiguration events.

The application service responsible for component state tracking and architecture reconfiguration has to be therefore accessible to all components. This service should provide only interface and general protocol implementation while the concrete logic related to component reconfiguration is generated during application deployment. Very important for the reconfiguration itself is to met the mode change requirements. These are schedulability, periodicity, promptness and consistency. Therefore, it may be convenient to use the mode change protocol such as the ones described in [7, 8, 9] in order to met these requirements and limit the response time of the new mode tasks and therefore the overall time of the whole architecture reconfiguration.

3.3.4 Development tool support

With respect to the requirement (R6a), SOFA HI aims at reuse of the existing tools, as these were already developed to support any SOFA 2 component system implementation. However, because the tools were initially developed for existing SOFA 2 implementation, they have to be extended of other language support which is needed for SOFA HI. These tools already support different languages through different concepts and it should be therefore possible to implement this support without or with only minimal changes to their implementation.

On the other hand, there may be other modification needed in order to support new functionality which will be needed for SOFA HI. Yet it is still important to bear in mind

that these tools should be as much as possible independent of concrete SOFA 2 component system profile and changes to these tools should be therefore carefully considered.

3.4 Deployment process and runtime support

Deployment process of SOFA HI application is tied together with the development process. This process is defined by SOFA 2 component model and SOFA HI deployment environment.

Deployment of SOFA HI applications is divided into two phases. The first phase of deployment process follows up the development process by defining deployment plan. During the second phase, the application is built and deployed to target platform according to the definition provided by deployment plan.

The first phase of SOFA HI application deployment is following:

1. The deployment plan of application is generated. The values of all properties have to be defined before the plan is deployed into repository. This is particularly important for active components as real-time properties of these components need to be defined.

2. Component connection code is generated and stored after deployment plan is deployed into repository.

Fragments created during this phase are again stored in repository.

During the second phase of deployment, in addition to repository, deployment environment represented by deployment dock and deployment dock registry is also used. This phase is following:

1. Deployment dock obtains the code bundles containing implementation of all interfaces and primitive architectures from repository together with bundles containing generated component connection code.

2. The component application is compiled according to selected target platform and device.

Resulting application binary may be uploaded to the selected device.

3.4.1 Deployment and runtime environment

There are several important differences between deployment and runtime environment of SOFA 2 and SOFA HI. First of all, SOFA HI does not employ any runtime environment so that SOFA HI applications will be completely self-hosted. This is needed as the real-time embedded environment typically offers only limited computing power and memory and it is therefore not possible to use such runtime environment in order to limit the application overhead. On the other hand, as the embedded devices are typically single purposed, it is not even needed.

SOFA HI deployment environment should be aware of different target platforms. Due to its concept of an extension targeted at embedded systems development, it should support

different embedded hardware platforms as this is the common usability requirement in this area of software development. This support should be done in open and extensible way so that it will be possible to eventually add support for new platforms with minimal effort.

During application compilation and deployment phase, other task may also take place as the component application is completed and target platform has been already selected. This is mainly the WCET analysis done automatically for each component on selected target platform, while the calculated values are stored into component model. Using these values, schedulability verification may be accomplished before the application is deployed into selected device to address the requirement (R2).

The environment should also support upload of resulting application directly into the selected hardware device. Moreover, it should be possible to integrate this functionality into existing management tools, ideally with minimal or no changes to their implementation.

3.4.2 Component instantiation and initialisation

Component instantiation is done in static way because selected computational model does not allow dynamic allocation. Therefore, instantiation and initialisation of all components has to be done during system initialisation phase. Component properties values are also being set at this phase, more precisely before component initialisation as these values may be needed during initialisation, especially in the case of real-time properties.

Related to component instantiation and initialisation is support for component and overall application termination. However, there is no need for this support in SOFA HI, or more precisely their tasks, needs to be non-terminating.

3.4.3 Component bindings

When all components are instantiated and initialised, they must be connected according to instructions in deployment plan. Component bindings implementation is generated while deployment plan is being deployed into repository. This implementation may support different communication styles to address the requirement (R5). SOFA/J uses Connector Generator framework for generating component connection implementation. However, this framework is not usable for SOFA HI due to missing support for different languages. Therefore, specialised solution is used for generating component bindings in case of SOFA HI.

3.4.4 Component runtime functionality

Micro-component model may be used to address the requirement (R3c). This is an important part of SOFA 2 component model which provides component control functionality during system runtime. Therefore, it is reasonable to have this micro-component model available also for the SOFA HI. However, additional introspection and control information needed for this functionality represents high overhead. This overhead can be unbearable for many real-time embedded systems with limited resources.

Therefore, it would be ideal to make the use of the micro-component optional in order to address the requirement (R3b). Moreover, the control aspects should be predefined i.e.

user defined aspects should not be allowed in order to limit total overhead and make the resulting system verifiable in the terms of schedulability.

4 Prototype implementation

Following section describes prototype implementation of SOFA HI. This prototype solution implements many of the concepts described in previous sections. Primary goal of this prototype is to verify the viability of the proposal and prepare the ground for future development.

Description of the prototype implementation starts with overview of the global architecture followed by the description of component implementation support. Afterwards, details regarding development and deployment environment implementation are provided together with details related to development tools extensions. This is accompanied with detailed description of prototype usage. Discussion related to prototype limitations is also provided.

4.1 Global architecture

The implementation consists of few distinctive parts - the implementation of different modules and tools extensions. Even though they have been already described from design point of view, implementation details for these parts were not covered in the preceding text. Following subsections therefore give a brief overview of these parts.

The global architecture of SOFA HI runtime and development environment is similar to SOFA 2 development and runtime environment, which is shown in Figure 3, as SOFA HI is a profile of SOFA 2. However there are some minor and major differences. These differences are also covered in following subsections.

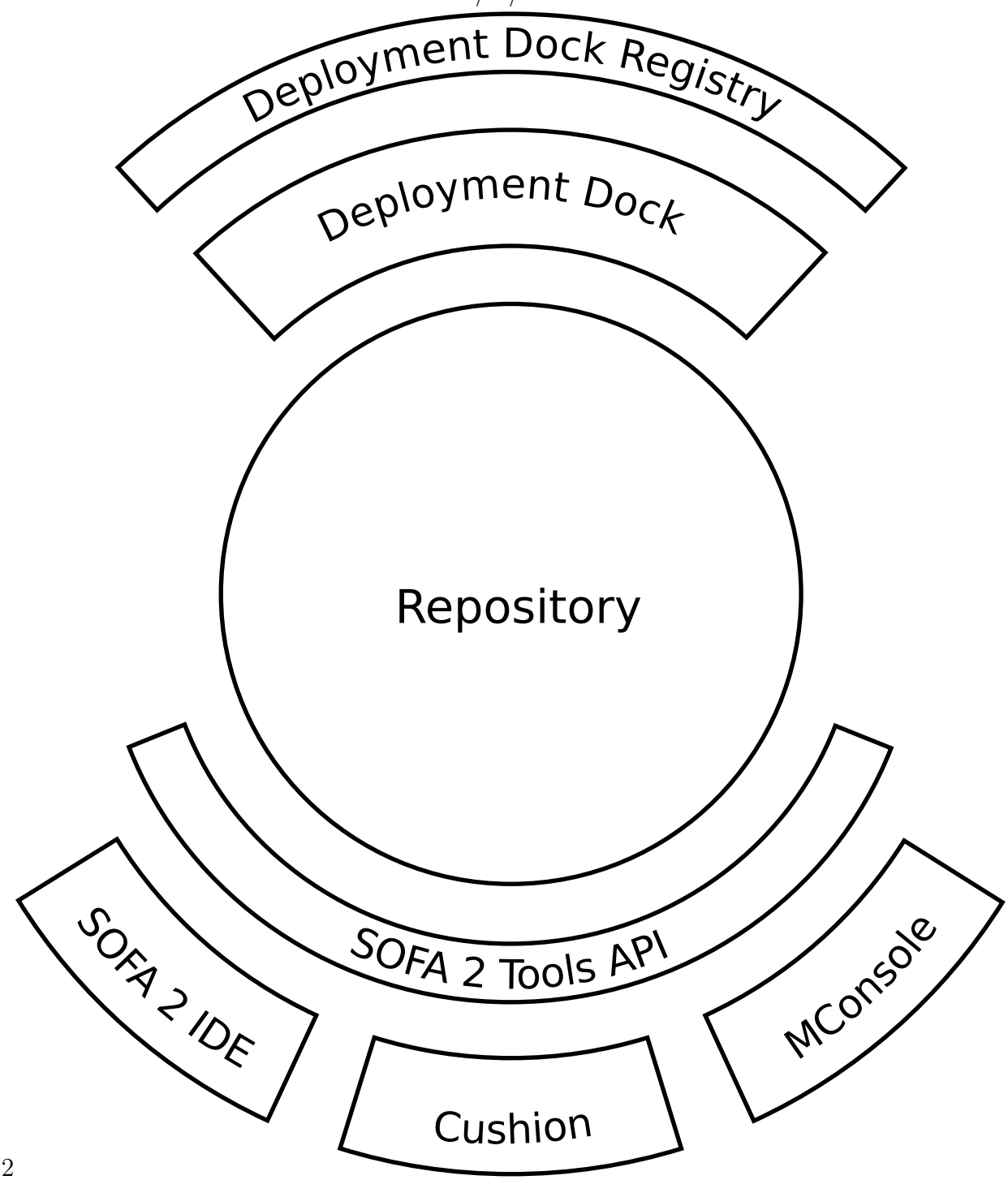
4.2 Component development support

Component implementation in SOFA HI does not require any dependencies except the provided system and component programming interfaces. These are independent of any concrete platform and therefore easily portable. Therefore, component implementation should be completely portable. That is why component implementation itself should not use any specific dependencies as these would limit component implementation portability to different platforms.

4.2.1 System programming interface

The SOFA HI system API provides realisation of the platform abstraction. The implementation of SOFA HI system API is provided for each supported operating system. The applications themselves are written against this interface. Particular implementation of the system API is selected at compile time. Component implementation itself is only allowed to use means provided by system API.

/ 2/



2

Figure 3: SOFA 2 development and runtime environment

The programming interface is available through `sofa-hi-system-api.h` header file. The interface consists of the following parts:

tasks tasks are similar to threads as in POSIX, however their interface is simpler,

mutexes mutexes with optional support for priority inheritance and priority ceiling protocol,

semaphores standard counting semaphores,

queues message queues similar to those in POSIX for communication between tasks,

time interface for obtaining system time.

The implementation is separated to definition of the programming interface itself and its implementation for different operating systems. Prototype implementation support two different systems:

Linux implementation for Linux [51] operating system using standard POSIX interface,

FreeRTOS implementation for real-time operating system FreeRTOS [52] using its native interface.

Due to significant differences in interfaces of both these operating systems, the implementation has to cover these and sometimes implement missing functionality using other means.

4.2.2 Component programming interface

The SOFA HI component API contains set of interfaces and macros intended to be used for component implementation similarly to SOFA/J component programming interface. These macros are independent of the platform used. This programming interface is available through `sofa-hi-component-api.h` header file.

4.3 Development and deployment environment

Development and deployment environment implementation consists of modifications of existing parts as well as new modules which were entirely developed as a part of the prototype implementation.

4.3.1 Environment modularization

Existing implementation of SOFA 2 empowers monolithic design of its development, deployment and run-time environment implementation. This made it easier to create the initial implementation, but it is rather unusable for SOFA HI implementation as this would need to reimplement some parts. Therefore, it is inevitably needed to modularize the existing

SOFA 2 implementation. The ideal solution would be to separate common parts from the parts that are specific for each implementation. This would be also convenient for other SOFA 2 profiles as they would be implemented in the similar way.

Therefore, modularization of existing SOFA 2 implementation is the key prerequisite of SOFA HI profile implementation as this allows to separate parts of the implementation into modules which may be shared by both profiles and creating new modules which are specific for each profile. Even thought the current implementation employs monolithic design, there are still distinctive parts according to which this implementation may be split into modules. These are unsurprisingly similar to the parts of runtime and development environment, this means *repository*, *deployment dock* and *deployment dock registry*.

While some of these modules are completely general and shared by all profiles, such as repository or deployment dock registry, some are profile specific such as deployment dock. However, even there it is possible to separate interface of deployment dock from its implementation, where interface is also part of shared modules while only implementation remain profile specific. This is convenient because management tools can also remain profile independent because they are implemented against deployment dock interface. On the other hand, some parts such as repository are not entirely independent as there is profile specific deployment support implemented inside repository. Therefore, this part has to be separated into profile specific repository extension.

4.3.2 Modification of existing implementation

Following the modularization of existing implementation, which is needed in order to achieve maximal reuse as one of the main requirements, there are also changes necessarily needed to allow SOFA HI implementation. These are related to monolithic design of existing implementation due to the unexpected support for other languages. These are mostly related to the SOFA 2 repository. What is needed is to allow bundles containing implementation in other programming languages because SOFA HI will use native language for component implementation.

Therefore, the most important repository change is related to support of different programming languages as a content of code bundles. The `CodeBundle` meta-class is extended with the notion of programming language used through newly added `language` attribute. `CodeBundleHelper` class is also modified so that code renaming takes place only in the case that code bundle language is Java.

Problem related to existing SOFA 2 repository implementation is also the tight integration of Connector Generator framework during deployment phase which is not usable in the case of SOFA HI. On the other hand, SOFA HI implementation will need its own implementation of the deployment logic. Therefore, it is needed to separate these parts and create new modularized solution.

The implementation of deployment phase during which the connectors were generated located in `XDeploymentPlanImplMethods` class, which was originally hard-coded, now uses service-provider loading facility provided by the Java platform to load implementations of this interface. Using this way, each SOFA 2 profile can provide its own implementation of

the deployment service as a repository extension module without need to modify existing repository implementation.

4.4 Modules

SOFA HI implementation consists of several modules. Some of them are shared with SOFA/J profile while others are completely SOFA HI specific.

Due to growing number of SOFA 2 component system modules, either shared or SOFA/J and SOFA HI profile specific, and it is much more difficult to build the whole system and resolve all dependencies. Therefore, the common *SOFA 2 build system* is introduced. This provides common infrastructure for building modules through common build templates for different module types. Build system itself is based on top of Apache Ant [46] build tool while Apache Ivy [47] is used for dependency management and resolvation.

4.4.1 Intermediate model

SOFA HI intermediate model module contains specialised meta-model that allows representation of SOFA HI application. This is needed, as other SOFA HI modules strongly rely on code generation. SOFA 2 repository meta-model is good for representing models of SOFA 2 applications because it is extremely flexible. However, it is really difficult to use it as an input for Model to Text transformations.

The meta-model is modelled using Eclipse Modelling Framework [27]. The implementation itself is generated using Eclipse EMF code generation facility. The meta-model of SOFA HI intermediate model is shown in Figure 4.

Top level part of the intermediate meta-model is the `System` meta-class which represents the complete SOFA HI application as has been described by the deployment plan. It consists of several components, their instances and bindings between these instances.

Components are represented by the `Component` meta-class. Each component contains arbitrary number of required and provided interfaces represented by the `Interface` meta-class as well as properties represented by the `Property` meta-class which are used for component parametrisation. Every component can also have several bundles which contain source or binary code, these are represented by the `Bundle` meta-class.

Every component can be contained in the system in several instances. Each instance is represented by the `Instance` meta-class. The instance has reference on the component it instantiates and parameters represented by the `Parameter` meta-class which represents concrete value of component property.

Component instances are connected using bindings which are represented by the `Binding` meta-class. Each binding contains references to instances of components which it connects, as well as references to their provided and required interfaces which it is connected to.

The meta-model is accompanied with transformation which allows converting SOFA 2 repository model into SOFA HI intermediate model. Therefore, each other SOFA HI module can rely on this functionality and use solely this meta-model as an input instead of

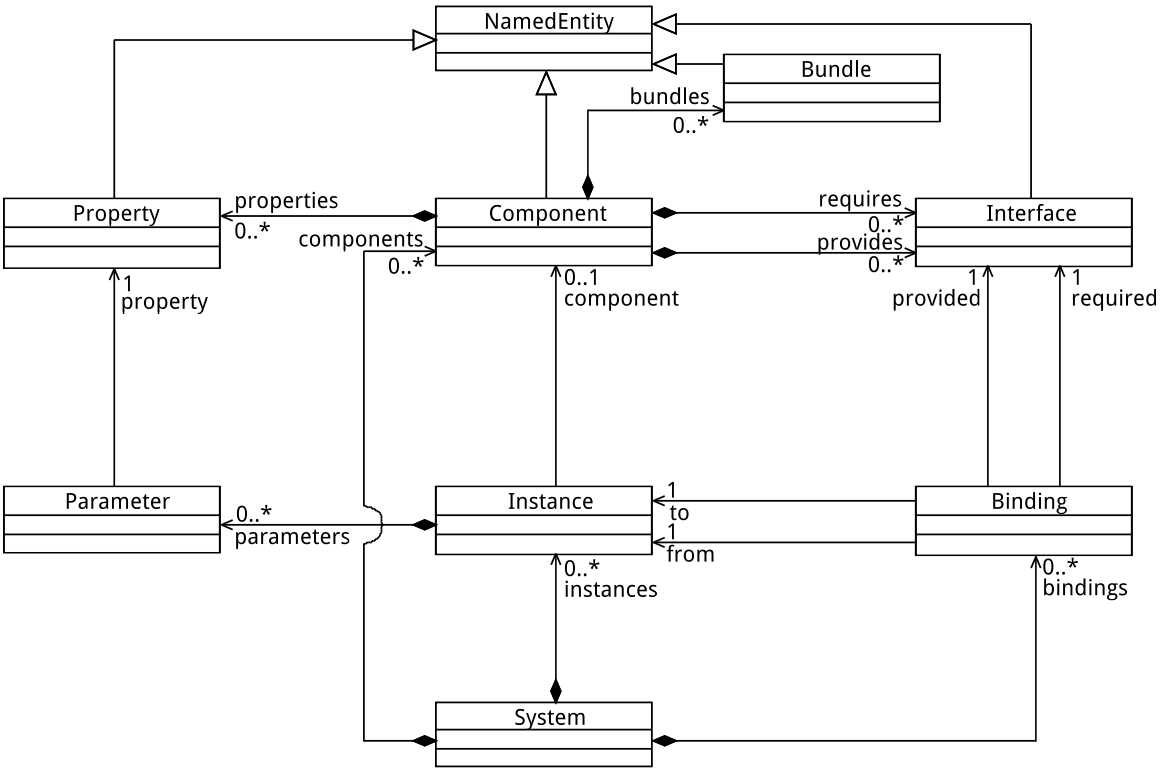


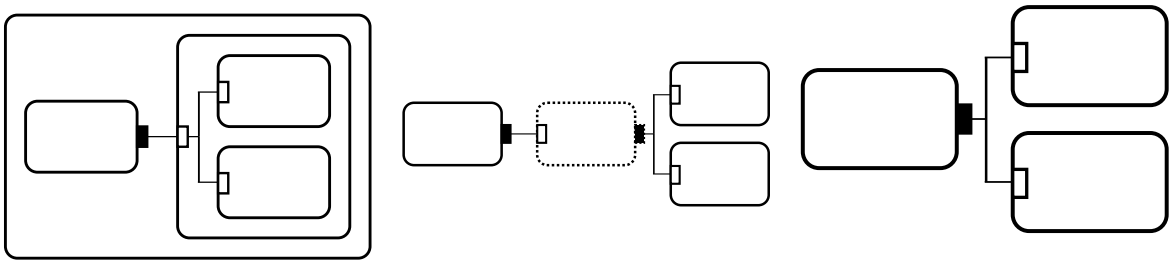
Figure 4: SOFA HI intermediate meta-model

SOFA 2 repository meta-model. The transformation is implemented using Eclipse Model To Model Operational QVT engine [28]. The transformation itself is not straightforward because the SOFA 2 repository model employs hierarchical component model while SOFA HI intermediate model employs flat model. Therefore, the transformation is taken in two phases.

An example of the transformation can be seen in Figure 5. The original architecture which serves as the transformation input is displayed in Figure 5a.

During the first phase, all the application components, both primitive and composite ones, are transformed into system components including their interfaces and code bundles. For the composite components, all their interfaces are duplicated with opposite roles (if not originally present). This is needed because all the system components are subsequently binded according to the connections described inside architecture of composite components. Many advanced features of QVTO language such as ability to attach temporary properties to meta-classes is used during first phase as this information is needed during the second phase. Resulting architecture after executing first phase can be seen in Figure 5b.

Purpose of the second phase is to remove all the system components corresponding to composite components inside original application architecture. This is achieved by passing through all of these composite components inside intermediate model and reconnecting all the components connected to their corresponding interfaces which has been generated



(a) Original architecture (b) Result of the first phase (c) Result of the second phase

Figure 5: Example of model refinement

during first phase. These components and interfaces are easily identified using attached temporary properties. Finally, ale the composite components are removed from intermediate model. Resulting architecture after executing second phase is displayed in Figure 5c.

4.4.2 Repository extension

The SOFA HI repository module is an extension of SOFA 2 repository providing SOFA HI related logic. This is mainly the logic related to application deployment. This module provides facility to generate component connections according to description in application architecture, system initialisation and bootstrap code. Therefore, a code generation facility is needed for implementation of this extension. The Acceleo framework [29] is used for this purpose.

Because the Model to Text language is not Turing complete and lacks some of the advanced features which are provided by Query/View/Transformation language, it is not possible to easily generate component connections code directly from the SOFA 2 repository model employing hierarchical model. That is why the SOFA HI intermediate model is used as an input for the template generating component connections code, transformed from SOFA 2 repository model using provided transformation. Moreover, the code for component initialisation is also generated. Generated code uses C programming language same as the implementation of components themselves.

Generated code is packed to be single code bundle and uploaded to the SOFA 2 repository as a deployment plan dependency.

Repository extension also contains definition of available communication styles and allows generating connections for each of the predefined style.

4.4.3 Deployment dock

The SOFA HI deployment dock module is the main part of SOFA HI environment. The main difference from existing SOFA/J deployment dock implementation is that the SOFA HI deployment dock does not serve as a runtime environment which is not present in SOFA HI. It rather serves as a compilation and deployment service. Yet, it still implements

the same interface which allows developers and administrators to use the same management tools together with SOFA HI deployment dock to deploy SOFA HI applications.

Central part of SOFA HI deployment dock module is the new implementation of `DeploymentDock` interface represented by the `DeploymentDockImpl` class. This class also implements the `DeploymentDockRegistryClient` interface in order to allow registration of this deployment dock inside SOFA 2 dock registry. This is not necessarily needed as there is no support for distributed applications in SOFA HI prototype implementation. However, it is convenient because management tools such as SOFA 2 MConsole use deployment dock registry for locating running docks.

The `DeploymentDockImpl` class contains mainly the logic needed for downloading all the application code bundles as described inside deployment plan and application architecture. After this has been done, necessary files has to be generated in order to compile the application. Because this is specific to different target platforms and devices, there is a concept of *deployment platform* and *deployment profile*. Each deployment platform supports single target platform while deployment profile supports single target device or set of devices. By choosing specific platform and profile, user or developer selects target platform on which the application will run. Moreover, after successful compilation of application, it is needed to deploy this application into selected device. This is time when *deployment uploader* concept comes in play. Each uploader allows to upload resulting binary into specified location using its own way.

Each of these concept is implemented in open extensible way to allow easily adding new platforms, profiles and uploaders so that other developers and users could implement support for their own target platforms and devices.

The SOFA HI application is compiled using GNU Make [48] and GNU Compiler Collection [49]. The *makefile* is generated using Acceleo framework [29] by the `DeploymentCompiler` class. This makefile includes platform and device specific compilation logic contained in separate files which are generated by the selected deployment platform and deployment profile. The class `DeploymentCompiler` also launches the compilation of the resulting application.

Deployment platform is represented by the implementation of `DeploymentPlatform` interface. Deployment platform should prepare target platform related files needed for compilation of the application related to selected platform. Prototype implementation contains these deployment platform implementations:

Linux platform supporting Linux operating system,

FreeRTOS platform supporting FreeRTOS operating system.

Goal of the deployment platform is to prepare the partial makefile which contains logic related to the selected target platform. Therefore, it uses Acceleo framework to generate necessary the makefile which is then used by the master makefile to compile the application. Each platform is registered in `platforms.xml` file which is as a part of the deployment dock implementation. New platform can be easily added by editing this file. The structure of this file is described by the `platforms.xsd` schema.

Similarly to deployment platform, deployment profile is basically the implementation of `DeploymentProfile` interface. The purpose of deployment profile is to prepare all the files needed for compilation of the application related to selected target device or environment. Prototype implementation contains few deployment profile implementations:

IA-32 profile targeted at IA-32 architecture,

LPC214x profile targeted at LPC2141/42/44/46/48 microcontrollers based on ARM7 architecture.

Both these profiles again generate the partial makefile which contains compilation logic related to selected device or environment. They both use Acceleo framework for this task. Profiles themselves are registered in `profiles.xml` file which is as a part of the deployment dock implementation, it is therefore easy to add new profiles by editing this file. The structure of this file is described by the `profiles.xsd` schema.

Each deployment platform also defines a set of supported deployment profiles because not each profile is supported by each platform. SOFA HI intermediate model is used as an input of deployment platform and profile due to similar reasons as for SOFA HI repository module implementation.

After the application is compiled, it needs to be uploaded to a specific location which is achieved by the deployment uploader represented by `DeploymentUploader` interface. As for deployment profiles, the prototype implementation also contains several deployment uploaders implementations:

Local this uploader copies the binary file into specified location on local filesystem,

SecureShell uploader which copies the binary file to a given location using SSH,

LPC21ISP this uploader use LPC21ISP utility in order to upload binary file into selected hardware device.

Uploaders are registered in `uploaders.xml` file contained in the deployment dock implementation. Therefore, it is easy to add new uploaders by editing this file. The structure of this file is described by the `uploaders.xsd` schema.

4.4.4 Code processor

The SOFA HI code processor module is an extension of SOFA 2 tools API which adds facility needed for C language support. The extension adds ability for code processing and code generation into SOFA 2 tools API. This consists of C code processor which takes care of packing of the C source code into code bundles and C code generator which allows for generating of the component and interface skeletons. Component and interface skeletons are generated according to description of application stored in repository.

During application compilation and code bundle upload, SOFA 2 tools API use code processor represented by the `CodeProcessor` interface for all the operations. Therefore,

it is easy to add support for new language simply by implementing this interface which is also the case of SOFA HI code processor module or more precisely the `CCodeProcessor` class representing the code processor with C language support.

Support for generation of skeletons of interfaces and components is achieved by using code generators represented by `CodeGenerator` interface in similar way to code processor functionality. Therefore, SOFA HI code processor module also provides implementation of this interface represented by `CCodeGenerator`. This class allows to generate skeletons using C language. The Acceleo framework is again used for generating the code.

4.4.5 Bootstrap

The SOFA HI bootstrap module contains entities necessarily needed for SOFA HI development and deployment environment proper functionality. Similarly to SOFA/J, the bootstrap also contains definition of predefined control interfaces, micro-components and aspect used in case that micro-component model is a part of the SOFA HI component application. For prototype implementation, this means mainly the `sofa.properties.Periodic` property set which allows to mark component as periodic; it contains three properties - `priority`, `period` and `deadline`.

What is specific to SOFA HI is that the SOFA HI system API and component API also part of the bootstrap module converted to standalone code bundles. They are being uploaded to the repository as well. Moreover, each SOFA HI code bundle has implicit dependency on these code bundles. This is convenient because during application compilation prepare phase when all code bundles are being retrieved, these code bundles are downloaded as well.

Content of SOFA HI bootstrap is automatically uploaded into repository when SOFA HI distribution is being built.

4.4.6 Library

The SOFA HI library module contains set of predefined interfaces and components for the concrete hardware peripherals. These can be used for development of the new applications out of predefined components. This follows the basic concept of building systems out of pre-existing components. Because this module is implemented as a regular SOFA HI application, it is therefore easy to add new components to support other hardware peripherals.

Prototype implementation contains several predefined components and their fragments. There are interfaces `sofa.library.display.Write`, `sofa.library.input.Press` and `sofa.library.output.Toggle`. These are provided by frames `sofa.library.display.AlphanumericDisplay`, `sofa.library.input.ButtonSet` and `sofa.library.output.DiodeSet`. The frames are implemented by architectures `sofa.library.display.CM1624` for Data Image CM1624 alphanumeric display, `sofa.library.input.ETT10PINP` for ETT 10P/INP hardware input board equipped with button set as well as `sofa.library.output.ETT10POUT` for ETT 10P/OUTP hardware

output board equipped with LED[4] set.

Similarly to SOFA HI bootstrap, SOFA HI library is automatically uploaded into repository during SOFA HI distribution preparation.

4.5 Tool support extension

SOFA 2 development tools presented by Cushion, SOFA 2 IDE and MConsole need to be slightly extended to accompany new features and newly added extensions needed for SOFA HI application development. This is mainly improved support for different programming languages and functionality that allows to generate interface and component skeletons. This functionality is useful for SOFA/J as well, but it is essential for SOFA HI as the component implementation using C language is not as straightforward as in Java due to the absence of object oriented programming concepts such as classes and interfaces which easily maps to SOFA 2 entities.

4.5.1 SOFA 2 tools API

SOFA 2 tools API is the common programming interface which acts as the basis for all other tools. The most important part it offers is the workspace abstraction which allows to map SOFA entities to files and directories. On top of this, it provides complete set of actions for development and management tasks.

Although the implementation itself is already language independent, there are still some modifications needed to fully support C programming language as an component implementation language which is needed for SOFA HI. The most important is an extension of the workspace abstraction in order to contain information about default programming language used for all entities contained inside workspace. The workspace implementation represented by the `XMLConfiguration` class is therefore extended to contain current workspace language as its attribute. The support for different languages represented by the concept of code processor has been extended by introduction of `CodeProcessorFactory` class. This class represents factory that is used to obtain code processor according to workspace language. This factory internally again use the service-provider loading facility to obtain the implementation of `CodeProcessor` interface.

Another functionality which is added to SOFA 2 tools API is the code generation support which allows to generate skeleton of interfaces and components. Similar approach as in case of code processor is used for implementation of the code generation support. The code generator itself is represented by the `CodeGenerator` interface and `CodeGeneratorFactory` class is used to obtain its implementation using the service-provider loading facility. This facility is accompanied with action which makes it available to other tools built on top of this module. The action is represented by `Generate` class which allows to run the generation for given entities.

[4] light-emitting diode

4.5.2 SOFA 2 ADL

SOFA 2 ADL[5] is a language based on XML used for description of single top level entity. This language is used by SOFA 2 tools API to represent SOFA 2 entities and subsequently by all the tools built on top of SOFA 2 tools API. Graphical Eclipse-based editors for this language also exist. This language is described using XML Schema for the purpose of SOFA 2 tools API implementation as well as Eclipse Modelling Framework meta-model for the purpose of Eclipse-based graphical development tools. Relation between SOFA 2 ADL and SOFA 2 meta-model is described using XSLT as well as QVTO language.

Definition of SOFA 2 ADL language is extended in several ways similarly to the extensions of SOFA 2 meta-model. The `architecture` element is extended with `active` and `singleton` attributes. New `properties-set` is defined as a collection of `property` elements together with `property-reference` element which allows to reference existing property set from architecture using `set` attribute.

Similarly, the SOFA 2 ADL meta-model is extended to comprehend the same information. Therefore, `Architecture` meta-class is extended with `active` and `singleton` attributes. `PropertySet` meta-class is defined, containing set of `Property` meta-class instances through `property` attribute and `PropertySetReference` meta-class that allows to reference existing property sets throught `propertySet` attribute which has been added to `Architecture` meta-class.

Following the extensions of SOFA 2 ADL language, both XSLT and QVTO transformations are extended together with implementation of SOFA 2 tools API contained in `AdlCreatingAction` and `AdlReadingAction` classes. Graphical Eclipse-based SOFA 2 ADL editors contained in `org.objectweb.dsrg.sofa.adl.editor` plug-in are extended to support these extensions as well.

4.5.3 Cushion

The Cushion command line development tool is extended of two new commands in order to support new functionality of SOFA 2 tools API which Cushion implementation completely relies on.

Such new functionality is the possibility to define workspace programming language. Therefore, a command that allows to initialise new workspace and optionally specify its language is added. The *init* command represented by the `InitAction` class allows to initialise new workspace with specified programming language given as optional parameter. This class internally uses the new functionality introduced in `XMLConfiguration` class which is part of SOFA 2 tools API.

Another command allowing to generate skeletons of components and interfaces is also needed in order to support this facility provided by SOFA 2 tools API inside Cushion. The *generate* command is represented by the `GenerateAction` class and allows to generate skeletons of implementation for interfaces and components. Name of the entity has to be

[5] architecture description language

passed as command parameter. This command delegates the call to the new `Generate` class provided again by the SOFA 2 tools API.

4.5.4 SOFA 2 IDE

SOFA 2 IDE needs to be extended with support for the C language which is the primary programming language of SOFA HI applications, to fully support development of SOFA HI applications inside SOFA 2 IDE. Support for various languages inside SOFA 2 IDE is done through the concept of SOFA 2 runtime platforms. Therefore, C runtime platform has to be implemented in the similar way to the existing Java runtime platform.

Eclipse IDE, on top of which the SOFA 2 IDE is built, fully supports C programming language through the Eclipse C/C++ Development Tooling. Therefore, C runtime platform for SOFA 2 IDE is completely based on top on top of this tooling. The whole implementation is contained in two plug-ins.

The `org.objectweb.dsrg.sofa.eclipse.cdt` plug-in contains the logic of C runtime platform itself. This is represented mainly by the `SOFA2CRuntimePlatform` class accompanied by two classes `SOFA2CCodeProcessor` and `SOFA2CCodeGenerator` adapting existing classes from SOFA 2 tools API.

The `org.objectweb.dsrg.sofa.eclipse.cdt.ui` plug-in extends C runtime platform with user interface logic represented by the `SOFA2CRuntimeUIPlatform`. This plug-in also contains extension of SOFA 2 navigator supporting elements of Eclipse C/C++ Development Tooling common navigator framework extensions.

The workspace language is automatically set according to runtime platform selected by the `SOFA2ProjectConfiguration` class contained in `org.objectweb.dsrg.sofa.eclipse` plug-in which adapts the default `Configuration` interface implementation or more precisely its `XMLConfiguration` implementation which newly supports the notion of workspace language.

Also, similarly to Cushion, support for code generation facility has to be integrated into SOFA 2 IDE. This support is implemented on several places. First of all, interface type and architecture ADL editors contained in the `org.objectweb.dsrg.sofa.adl.editor` plug-in had to be extended to allow user to generate code skeletons. The code generator invocation is done through `org.objectweb.dsrg.sofa.adl.editor.implementationGenerator` extension point which requires implementation of `ISOFA2ADLImplementationGenerator` interface.

The `org.objectweb.dsrg.sofa.adl.editor.implementationGenerator` extension point is registered by the `org.objectweb.dsrg.sofa.eclipse.repository.ui` plug-in which contains the implementation of required interface represented by the `SOFA2ImplementationGenerator` class. Finally, the invocation of the generation action represented by the `Generate` class provided by SOFA 2 tools API is done by `SOFA2GenerateCommand` class contained in `org.objectweb.dsrg.sofa.eclipse.repository` plug-in.

4.5.5 SOFA 2 MConsole

SOFA 2 MConsole is a management tool and it is therefore completely independent of application implementation language. No changes are therefore needed in order to support SOFA HI. This also applies in the case of property sets, because the properties contained in property sets are being merged and their values specified directly in the deployment plan.

4.6 Usage

Even thought the SOFA HI shares many parts with existing SOFA/J implementation, together with the development tools, its usage is not same nor similar as many concepts used in SOFA HI are specific only for this implementation. Therefore, it is important to be aware of these differences to be able to develop component applications using SOFA HI.

4.6.1 Building

All modules of SOFA HI prototype implementation are integrated into SOFA 2 build system which has been described in Section 4.4. This allows to easily resolve all the dependencies and build the implementation of all modules. Similarly to SOFA/J profile, SOFA HI prototype implementation also allows to prepare SOFA HI distribution. This consists of several steps:

0. All the SOFA HI modules need to be built and uploaded into repository. This task is not taken as a part of distribution assemblance and needs to be done separately.

1. Each SOFA HI module is resolved from repository together with shared SOFA 2 modules and all the third party dependencies.

2. Shell scripts for running SOFA HI development and deployment environment together with configuration file templates are processed and copied into distribution.

3. Empty repository is started and content of bootstrap module is uploaded.

4. Content of library module is uploaded into running repository together with example application.

After all these steps are done, complete SOFA HI development and deployment environment is prepared to use.

4.6.2 Development

Development of SOFA HI components and component-based applications is similar to development of components using SOFA/J as have been described in official SOFA 2 user's and programmer's guide [1]. However, there are some differences related to the nature of

SOFA HI, this means mainly the C programming language used for SOFA HI application development.

SOFA HI component implementation does not enforce any predefined structure, whole implementation is based on usage of C macro definitions. These macros have to follow strict naming conventions as the code component initialisation and connection, generated during deployment, relies on these conventions. However, most of these macros are not required. Each macro is prefixed by the name of component, this is defined on component architecture through *implementation* attribute.

Component macro naming conventions

`<NAME>_TYPE` for defining component data type (required).

`<NAME>_TYPE_INIT` for component type initialisation.

`<NAME>_SINGLETON` for defining component singleton instance (required if singleton).

`<NAME>_SET_<interface>(self, reference)` for setting component interface reference.

`<NAME>_GET_<interface>(self)` for getting component interface reference (required).

`<NAME>_PROPERTY_<property>(self, value)` for setting component property value.

`<NAME>_INIT(self)` for initialisation of the component instance.

`<NAME>_START(self)` for starting the component task.

`<NAME>_STOP(self)` for stopping the component task.

Moreover, it is required that the name of the header file containing these definitions is same as the name of the type containing component implementation specified by the *implementation* attribute of architecture.

This is also true in the case of interface type, where name of file has to be same as the name of type containing interface definition which is described by the *signature* attribute of interface type.

Even thought there is no structure required to follow, recommended structure still exists as this is used in skeletons which can be automatically generated. This structure looks as follows for interface types

```
#ifndef COMPONENT_ITF_H_
#define COMPONENT_ITF_H_

typedef struct {
```

```
  /* interface operations */
  void (*op)(void *this, int data);
} component_itf;

#endif /* COMPONENT_ITF_H_ */
```

For components, this structure looks as follows

```
#ifndef COMPONENT_H_
#define COMPONENT_H_

#include "component_itf.h"

#include <sofa-hi-system-api.h>
#include <sofa-hi-component-api.h>

typedef struct {
  int value;
} component_data;

typedef struct {
  /* interfaces */
  component_itf provided;
  component_itf *required;

  /* properties */
  int property;

  /* attributes */
  component_data data;
  sofa_task_t task;
} component;

#define COMPONENT_TYPE component
#define COMPONENT_INIT { \
  .provided = { \
    .operation = component_operation \
  }, \
  .data = { \
    .value = 0 \
  } \
}
#define COMPONENT_DATA(self) (self)->data
```

```
#define COMPONENT_GET_provided(self) &(self)->provided
#define COMPONENT_SET_required(self, reference)
   (self)->required = (reference)

#define COMPONENT_PROPERTY_property(self, value)
   (self)->property = (value)

#define COMPONENT_START component_start
#define COMPONENT_STOP component_stop

void component_start(component *self);
void component_stop(component *self);

void component_operation(void *this, int data);

#endif /* COMPONENT_H_ */
```

4.6.3 Deployment

Deployment of SOFA HI applications is same as in SOFA/J from the perspective of deployment process. Most notable difference comes from the fact that the primitive component implementation is not compiled while being uploaded to repository in case of SOFA HI. This is not possible as the target platform is not known at this time. Therefore, no compilation errors can be found during component upload or deployment phase. This is different from SOFA/J and requires much more deliberation from component developers during component implementation in order to prevent such compilation errors.

4.6.4 Launching

Launch of SOFA HI application, despite its name, does not mean actual execution of resulting application as this is in many cases not possible. The launch in case of SOFA HI means to compile the application for selected platform using selected deployment profile and upload the resulting application binary in selected device using deployment uploader.

Before the application can be launched, repository and dock registry has to be running. This can be achieved by running each part separately using provided `sofa-repository.(sh|bat)` and `sofa-registry.(sh|bat)` scripts or at once by running complete SOFAnode using provided `sofa-node.(sh|bat)` script.

Now, deployment dock has to be launched using provided script with the following parameters

```
sofa-dock.(sh|bat) <name> -platform <platform> -profile
   <profile> -uploader <uploader> [arguments]
```

When run without these parameters, list of available platforms, profiles and uploaders is displayed together with usage description. Each deployment dock is limited to single platform, profile and uploader. However, number of deployment docks is virtually unlimited. Optional arguments are passed to deployment uploader in case they are specified.

After deployment dock has been started, it is possible to launch the application using provided script with the following parameters

```
sofa-launch.(sh|bat) <application> [tag|-v version]
```

When successfully compiled, resulting application binary will be uploaded using selected uploader, it is possible to optionally specify destination as this parameter will be passed to uploader and used if it is supported. Compilation errors will be shown in case there were found any during compilation.

4.7 Prototype limitations

At this place, it is worth mentioning that prototype implementation does not aim to be the complete SOFA HI implementation. Rather it aims to provide implementation of basic features which allows to evaluate usability of the SOFA HI profile. Therefore, there are features described in previous sections but not presented in the actual prototype implementation.

The most significant is the absence of schedulability verification represented by the requirement (R2). This is mainly due to non existence of usable solution as the approaches described in Section 2.1.1 are not available or their implementation does not exist.

Another missing part is the support for operating modes represented by the requirement (R3a). This would require support for mode automaton generation as well as mode change protocol mentioned in Section 3.3.3 which would again require a lot of effort to implement, especially optimisations needed in order to limit the architecture reconfiguration overhead.

Micro-component model addressing the requirement (R3b) is also not supported in the prototype as its implementation will require C language parser implementation with support for AST transformations. This requires a lot of effort as there is no existing technology that provides functionality for these AST transformations. However, as the micro-component model is not yet fully supported by all the development tools, current absence of this functionality is not so critical.

Prototype implementation also does not support distributed communication represented by the requirement (R5), because this would require great effort to either create new solution for generating connectors for distributed communication or to modify connector generator currently used in SOFA 2.

5 Evaluation and use cases

The following section contains evaluation of the SOFA HI prototype implementation. For this purpose, demonstration of the features described in previous sections is given using two

use cases. Development of a smaller sample application is described as well as description of larger real-life application which has been designed using SOFA 2 component model. These both applications can be used to evaluate the features of prototype implementation from different point of view. Smaller sample application shows how can use of the SOFA HI profile simplify development of modular embedded real-time application while larger real-life application shows how the SOFA HI profile ease the process of designing of large embedded real-time system. These demonstrations are accompanied with comparison to standard development as a part of the evaluation.

5.1 Hardware and operating system

For the purpose of demonstration, the CP-JR ARM7 USB LPC2148 EXP board 6 powered with NXP LPC 2148 processor which has 32-bit ARM7 architecture will be used. This board will be extended with Data Image CM1624 alphanumeric display which will be used for displaying application output and button set for simple user input.

Figure 6: CP-JR ARM7 USB LPC2148 EXP board

This board has been selected mainly because it is intended for educational purposes and therefore offers quality documentation as well as technical support. Moreover, the processor used inside this board is widely used in various industrial applications.

Out of all operating systems available for this board, FreeRTOS has been chosen as the example of real-time operating system. This has been done due to various reasons. The most important one is that FreeRTOS, yet its implementation is very simple, has very good support for different hardware. This real-time kernel is specifically designed for embedded processors and as such has enjoyed considerable popularity growth during its development.

FreeRTOS has also very non-restrictive licence model. While the kernel itself is completely open-source, application side code which simply uses FreeRTOS features can still remain closed source.

5.1.1 Real-time embedded systems programming

During high-integrity real-time embedded systems implementation, developers have to be aware of certain limitations. These limitations have been already discussed in Section 2.1.4.

The most important limitations are the following ones:

Dynamic memory allocation is not allowed.

No tasks can be created after system is initialised.

All tasks have to be non-terminating.

Usage of time-related functions is not allowed.

Most of these limitations are already enforced by system programming interface provided by SOFA HI. However, developers should be aware of these limitations even during system design phase.

5.2 Development of sample application

This subsection shows the development of sample application. The objective is to demonstrate the development process of a small embedded real-time application using the SOFA HI prototype implementation and SOFA 2 development and management tools.

SOFA Watch is a small application consisting of four components, three primitive ones and one composite. The architecture of the application is shown in Figure 7. Active components are marked by black triangle. Purpose of this application is to provide functionality of a stopwatch. This means it should be able to measure the amount of time elapsed from particular time when activated. Measurement can be started or stopped and the measured value can be reset. Timing functions are controlled by buttons connected to demonstration hardware board. Measured value should be displayed using hardware display connected to hardware board as well.

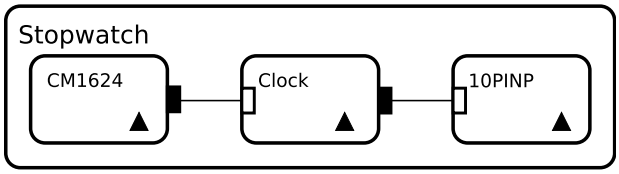


Figure 7: Sample application architecture

Guideline to implementation of sample application using command line tools is provided in Appendix A. Appendix B provides guideline to implementation of sample application using graphical tools. Goal of this guidelines is to create the implementation of this sample application using SOFA HI and to deploy this application into demonstration hardware board.

This application uses components provided by the SOFA HI library for reading input from button set and to write output to display. This clearly shows one of the biggest advantages brought by the component-based software development paradigm which is simple code reuse. Application can be simply assembled from existing parts without their modification. Moreover, due to strict use of explicit interfaces for component communication, it

is easy to change the component implementation in case that underlying hardware component is changed without need to modify application logic. While this is common in object oriented programming, it is not yet common in the world of real-time embedded systems development.

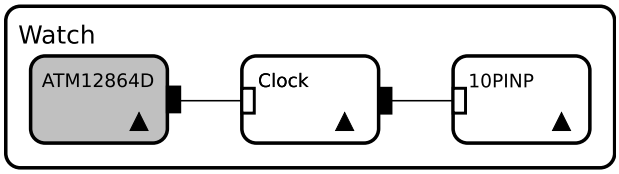


Figure 8: Sample application architecture after modification

This feature can be simply demonstrated by modification of the application architecture in the way shown in Figure 8. This modification only requires to change the top-level composite component, therefore no code modification is needed. After application redeployment and most importantly replacement of the Data Image CM1624 alphanumeric display to ATM12864D graphical display on the demonstration hardware board (this component is not a part of SOFA HI library), application works as before with completely different hardware component. This clearly shows that component-based development approach may be useful in the real-time embedded systems development.

5.3 Real-life application

This use case is aiming at converting existing non real-time non component-based embedded application to SOFA HI in order to show the process of designing of large embedded real-time application and benefits it brings. Moreover, goal of this application is to demonstrate features of SOFA 2 component system or more precisely its SOFA HI profile.

SOFA Robot is a slightly simplified version of software used by the robot developed at Charles University for the Eurobot competition. This has been slightly modified by removing the unnecessary features as these are not needed for the purpose of this use case. The purpose of this application is to control the move of an autonomous robot in the open environment which contains obstacles that prohibit robot in move. Therefore, the control system has to effectively evade this obstacles. Moreover, the robot is equipped with skimmer for collecting various items layout in the environment.

The robot hardware is based on standard personal computer components. Computational unit is VIA EPIA (IA-32) processor. Robot is composed of various hardware modules connected via I^2C bus. These modules consists of HBmotor boards, MCP23016 board, SRF02 distance sensor and CMPS03 compass.

The robot software is a classic control system. Therefore, it reads data from various sensors, processes them using robot logic and then interact with environment through various actuators.

Proposed SOFA Robot architecture can be seen in Figure 9.

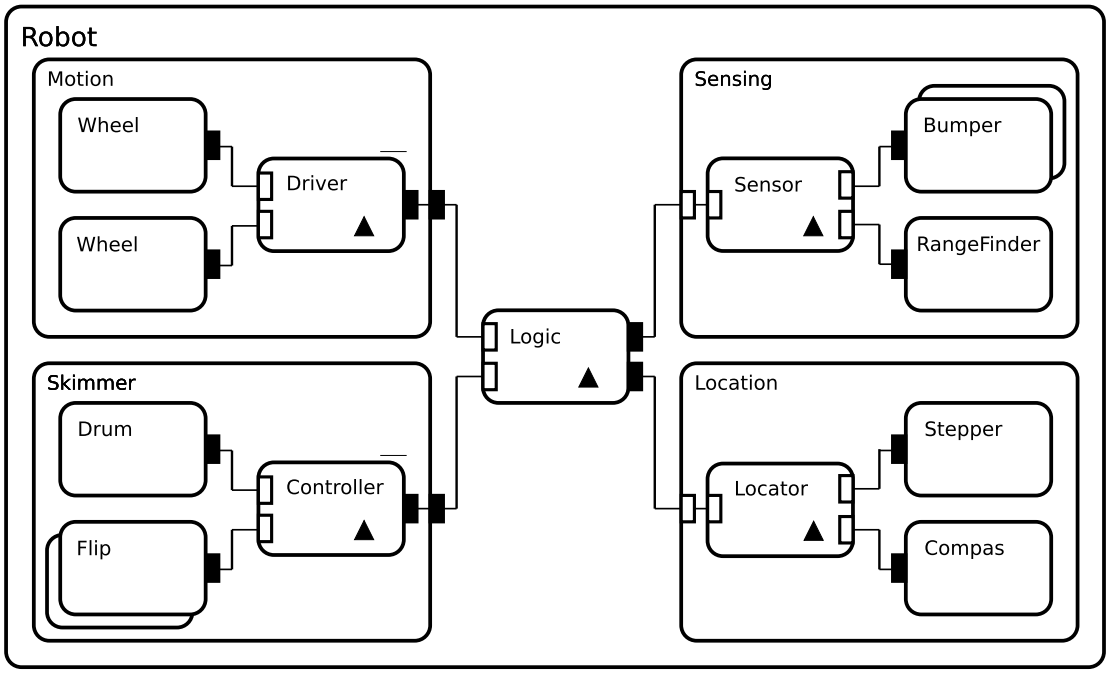


Figure 9: SOFA Robot architecture

The `Robot` architecture comprises of five composite components and eighteen primitive components. These are clearly separated to three different categories. Sensors are represented by the `Sensing` and `Location` hierarchical components. Actuators are represented by the `Motion` and `Skimmer` hierarchical components. The logic itself is contained in component `Logic`.

Each composite component is composed of several primitive components providing interactions with robot hardware and components containing control and processing logic. This clearly shows the biggest advantage of hierarchical component systems the ability to compose the system from atomic parts with strict separation of concerns. Another big advantage over non-component based development is the high code reuse, in this application presented mainly by the primitive components used for controlling different hardware parts which can be used in many instances, one for each hardware part, which is typical in many embedded systems.

Moreover, the robot logic also consists of different operating modes. Each mode corresponds to different robot behaviour.

Robot logic has several operating modes which can be seen in Figure 10.

5.4 Benefits

The two presented application clearly show the benefits of component-based software engineering in this case presented by SOFA 2 component system. These are mainly the clear separation of concerns, strong encapsulation and high code reuse. These properties are

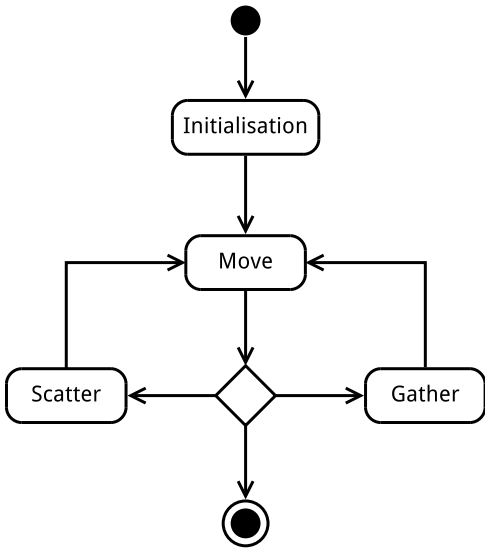


Figure 10: SOFA Robot states

considered as extremely important in all areas of software development including real-time embedded software systems development as they can result in shorter development time and reduced cost.

6 Conclusion

The SOFA HI shows that usage of hierarchical component systems in the area of high-integrity real-time embedded systems development could be beneficial. The advantages of component-based software engineering concepts are beneficial in this area of software development as the currently used development methods do not address many of the requirements that are important in this area of software development.

The prototype implementation has proved that it is really possible to extend existing SOFA 2 component system implementation to support development of high-integrity real-time embedded systems while retaining the most parts of existing development environment as well as development and management tools.

A Development of sample application using Cushion

This tutorial provides a step by step walk-through of the development of SOFA HI sample application described in Section 5.2 using Cushion command line development and management tool. Development tools and environment need to be properly set up in order to follow this tutorial. The instalation instructions for development tools and environment may be found in SOFA 2 users' guide [1]. SOFA HI environment installation instructions are same as installation instructions for the SOFA 2 environment described in this guide.

Correctly installed Cushion alongside the SOFA HI environment is needed in order to develop the application according to the steps provided.

1. Run the SOFA HI repository (or complete SOFAnode) by calling

```
sofa-repository.(sh|bat)
```

2. Create workspace directory and change to this directory, then initialise new workspace with C as its default language by calling

```
cushion.(sh|bat) init -l C
```

3. First of all, create frame of the Controller component by calling

```
cushion.(sh|bat) new frame initial
   stopwatch.frame.Controller
```

   This command creates new frame entity and also generates an ADL file, which has to be filled in. The ADL file named `adl.xml` is created in `stopwatch.frame.Controller` directory and should be completed to look as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<frame name="stopwatch.frame.Controller">
  <requires comm-style="method_invocation" itf-type=
     "sofatype://sofa.library.display.Write?tag=current"
     name="write" />
  <provides comm-style="method_invocation" itf-type=
     "sofatype://sofa.library.input.Press?tag=current"
     name="press" />
</frame>
```

   This frame provides and also requires single interface. Both of them are contained in SOFA HI library of predefined interfaces and components.

4. Create a frame for top-level composite Stopwatch component by calling

```
cushion.(sh|bat) new frame initial
   stopwatch.frame.Stopwatch
```

   The corresponding ADL of this component should look as follows

```
<?xml version="1.0"?>
<frame name="stopwatch.frame.Stopwatch" />
```

5. Create an architecture for the Controller component by calling

```
cushion.(sh|bat) new architecture initial
   stopwatch.arch.Controller
```

This component is primitive and its architecture is therefore empty, the ADL file should look as follows

```
<?xml version="1.0"?>
<architecture
   frame="sofatype://stopwatch.frame.Controller"
   name="stopwatch.arch.Controller" impl="controller"
   active="true">
  <property-reference set=
     "sofatype://sofa.properties.Periodic?tag=current" />
</architecture>
```

The `impl` attribute defines name of the type which implements the primitive architecture. The component is also marked as active using `active` attribute and will be used as periodic, therefore it is referencing `sofa.properties.Periodic` property set.

6. Create an architecture for the top-level Stopwatch component by calling

```
cushion.(sh|bat) new architecture initial
   stopwatch.arch.Stopwatch
```

The ADL of this architecture should look as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture name="stopwatch.arch.Stopwatch"
   frame="sofatype://stopwatch.frame.Stopwatch">
  <sub-comp name="display" frame="sofatype://sofa.␣
     library.display.AlphanumericDisplay?tag=current"
     arch="sofatype://sofa.library.display.CM1624␣
     ?tag=current" />
  <sub-comp name="buttons" frame="sofatype://sofa.␣
     library.input.ButtonSet?tag=current"
     arch="sofatype://sofa.library.input.ETT10PINP␣
     ?tag=current" />
  <sub-comp name="controller"
     frame="sofatype://stopwatch.frame.Controller"
     arch="sofatype://stopwatch.arch.Controller" />
  <connection>
    <endpoint itf="write" sub-comp="display" />
    <endpoint itf="write" sub-comp="controller" />
  </connection>
  <connection>
    <endpoint itf="press" sub-comp="buttons" />
    <endpoint itf="press" sub-comp="controller" />
```

```
  </connection>
</architecture>
```

The architecture defines three subcomponents *display*, *buttons* and *controller*. The first two are existing components from SOFA HI library, the third one is the component defined in the previous steps. The architecture also defines connections between these components.

7. Commit all changes to the repository by calling

```
cushion.(sh|bat) commit
```

8. Create the code for the Controller architecture by calling

```
cushion.(sh|bat) generate stopwatch.arch.Controller
```

The architecture skeleton is generated into four files. The files named `controller.generated.h` and `controller.generated.c` does not need to be modified in most cases. The `controller.h` file contains definition of component data while `controller.c` file contains implementation of the component methods. The `controller.h` should be modified to look as follows

```
#ifndef CONTROLLER_H_
#define CONTROLLER_H_

typedef struct {
  int running;
  unsigned long value;
} controller_data;

#define CONTROLLER_data { \
    .running = 0, \
    .value = 0 \
  }

#define CONTROLLER_press { \
    .pressed = controller_pressed \
  }

void controller_pressed(void *this, int state);

#include "controller.generated.h"

#endif /* CONTROLLER_H_ */
```

The `controller_data` structure contains component private data. This is used by the component logic, which is contained in `controller.c` file, this should be modified to look as follows

```
#include "controller.h"

#define START_STOP_BUTTON 0
#define RESET_BUTTON 1

void *controller_task(void *this) {
  DECLARE_SELF(this, controller);
  controller_data *data = CONTROLLER_DATA(self);

  static unsigned int hour, minute, second, millisecond;
  static char text[16];

  if (data->running) {
    data->value += self->period;
  }

  hour = data->value / 3600000;
  minute = (data->value / 60000) % 60;
  second = (data->value / 1000) % 60;
  millisecond = (data->value % 1000) / 100;

  snprintf(text, 16, "%02u:%02u:%02u.%1u", hour, minute,
     second, millisecond);
  CALL(self->write, write, 3, 0, text);

  return NULL;
}

void controller_pressed(void *this, int state) {
  DECLARE_SELF(INTERFACE_ENTRY(this, controller, press),
     controller);
  controller_data *data = CONTROLLER_DATA(self);

  switch (state) {
  case START_STOP_BUTTON:
    data->running = !data->running;
    break;
  case RESET_BUTTON:
    data->value = 0;
```

```
    data->running = 0;
    break;
  }
}
```

9. Pack and upload the code bundles of the interface type and both architectures by calling

```
cushion.(sh|bat) compile
cushion.(sh|bat) upload
```

10. Create an assembly for the application by calling

```
cushion.(sh|bat) assembly initial
   stopwatch.assm.Stopwatch stopwatch.arch.Stopwatch
```

The assembly ADL will be generated and does not require any changes and can be committed directly by calling

```
cushion.(sh|bat) commit stopwatch.assm.Stopwatch
```

11. Create a deployment plan by calling

```
cushion.(sh|bat) deplplan initial
   stopwatch.deplplan.Stopwatch stopwatch.assm.Stopwatch
```

The generated ADL file should be modified to look as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<depl-plan name="stopwatch.deplplan.Stopwatch"
   node="nodeLPC2148">
  <depl-subc name="display" node="nodeLPC2148">
    <depl-prop-value name="priority">2</depl-prop-value>
    <depl-prop-value name="period">50</depl-prop-value>
    <depl-prop-value name="deadline">50</depl-prop-value>
  </depl-subc>
  <depl-subc name="buttons" node="nodeLPC2148">
    <depl-prop-value name="priority">3</depl-prop-value>
    <depl-prop-value name="period">10</depl-prop-value>
    <depl-prop-value name="deadline">10</depl-prop-value>
    <depl-prop-value name="gpio">16</depl-prop-value>
  </depl-subc>
  <depl-subc name="controller" node="nodeLPC2148">
    <depl-prop-value name="priority">1</depl-prop-value>
    <depl-prop-value name="period">100</depl-prop-value>
    <depl-prop-value name="deadline">100</depl-prop-value>
```

```
  </depl-subc>
</depl-plan>
```

The periods are selected according to Rate Monotic scheduling so that task with the shortest period has the highest priority.

12. Deploy the application using the deployment plan by calling

```
cushion.(sh|bat) deploy stopwatch.deplplan.Stopwatch
```

13. Launch the deployment dock registry (if you did not started complete SOFAnode in the beginning) by calling

```
sofa-dockregistry.(sh|bat)
```

14. Run the deployment dock with correct settings by calling

```
sofa-dock.(sh|bat) nodeLPC2148 -platform FreeRTOS
   -profile LPC214x -uploader LPC21ISP
```

This will start the new deployment dock and will register it to the previously started dock registry. Deployment dock is parametrised by its name, runtime platform, hardware profile and target uploader.

15. Launch the SOFA 2 application by calling

```
sofa-launch.(sh|bat) stopwatch.deplplan.Stopwatch -v
   36c8c4964e0ec82dc9e126276dc18166ed699b91
```

This call will actually compile and upload application into target device. Follow the onscreen instructions printed by the deployment dock.

B Development of sample application using SOFA 2 IDE and MConsole

This tutorial provides a step by step walk-through of the development of SOFA HI sample application described in Section 5.2 using SOFA 2 IDE and MConsole graphical tools. Development tools and environment need to be properly set up in order to follow this tutorial. The instalation instructions for development tools and environment may be found in SOFA 2 users' guide [1]. SOFA HI environment installation instructions are same as installation instructions for the SOFA 2 environment described in this guide.

Correctly installed SOFA 2 IDE and MConsole alongside the SOFA HI environment is needed in order to develop the application according to the steps provided.

1. Open the SOFA 2 perspective in the running Eclipse IDE and create the new project by selecting the menu item *File ¿ New ¿ Project...* to open the *New Project* wizard.

2. Select the *SOFA 2 Project* then click *Next* to open the *New SOFA 2 Project* page. On this page, type `stopwatch` in the *Project name* field and select *SOFA 2 C Runtime Platform* in the *Use project specific* field as can be seen in Figure 11.

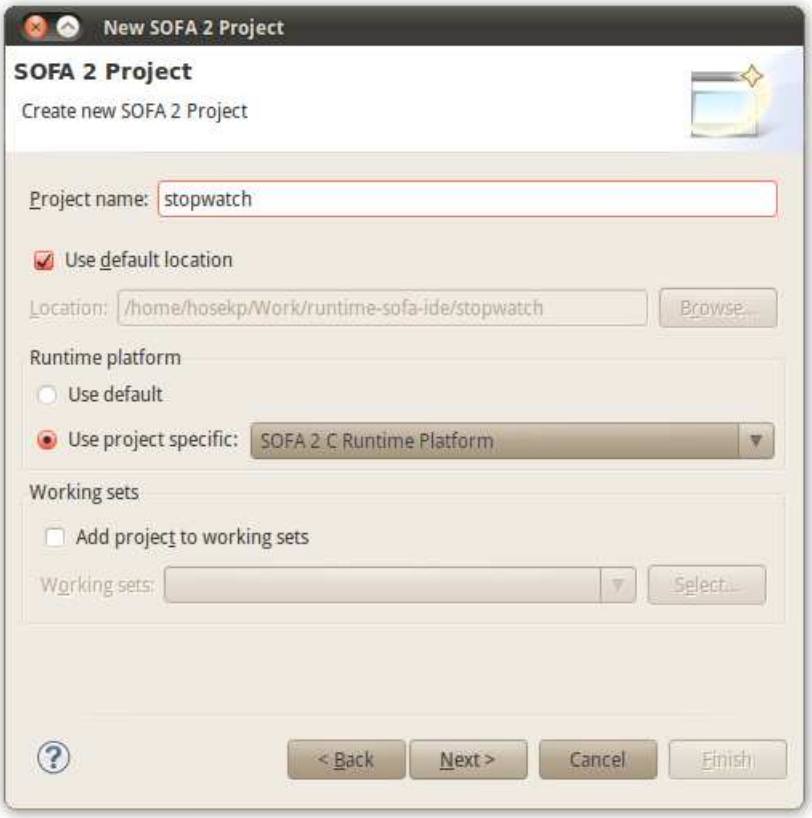


Figure 11: Creating new SOFA HI project

3. Then click *Next* to move on to the next page. There open the drop down field *Host* and select `http://localhost:8173/SofaServlet` (this is the default repository url). Click *Finish* to finish and close the new project wizard.

4. Create the SOFA 2 frame again by selecting the `stopwatch` project in SOFA 2 navigator and selecting *New ¿ SOFA 2 Frame* from the project's context menu. Make sure that `stopwatch` appears in the *Project* field. Type `stopwatch.frame.Controller` in the *Name* field. Then click *Finish*.

5. Select the new `stopwatch.frame.Controller` entity in the SOFA 2 Navigator and open the `adl.xml` file. The SOFA 2 ADL Form Editor opens. There select the *Provides* editor tab.

6. On the *Requires* editor tab, click *Add...* to add the new frame interface.

7. Type `press` into *Name* field of SOFA 2 Interface dialog. Select `method_invocation` in the *Communication Style* field. Click the *Browse* button in the *Interface Type* field and select `sofa.library.input.Press` in the interface type selection that appears and confirm the selection by *OK*.

8. Click *Finish* to confirm the frame interface dialog, which should look like as what can be seen in Figure 12 and select the *Requires* editor tab.

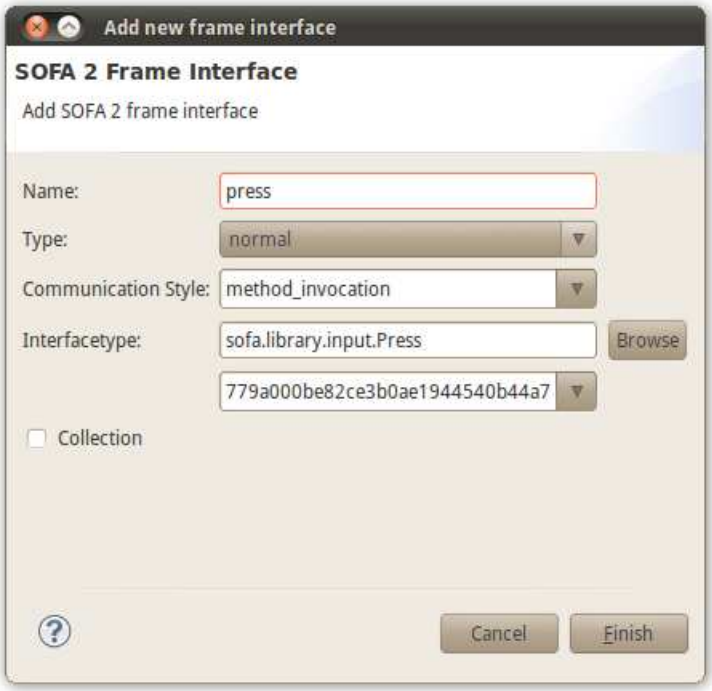


Figure 12: Creating new SOFA HI frame

9. On the *Requires* editor tab, click again *Add...* to add the new frame interface. There add interface named `write` with the `method_invocation` communication style and `sofa.library.display.Write` interface type (as in previous case). Click *Finish* to confirm the interface dialog and save the frame `adl.xml` file.

10. Create one more frame named `stopwatch.frame.Stopwatch` without any provided or required interfaces.

11. Now, create SOFA 2 primitive architecture by selecting the `stopwatch` project in SOFA 2 navigator and selecting *New ¿ SOFA 2 Architecture* from the project's context menu. Make sure that `stopwatch` appears in the *Project* field. Type `stopwatch.arch.Controller` in the *Name* field and confirm wizard by clicking on *Finish*.

12. Open the `adl.xml` file of newly created architecture in the SOFA 2 ADL Form Editor. Fill the `controller` into *Implementation* field. Click the *Browse* button in the *Frame* field and select `stopwatch.frame.Controller` frame using the opened frame selection dialog.

13. Continue with architecture creation. Check the architecture *Active* field. Click *Add...* in the *Property set* field and select the `sofa.properties.Periodic` property set using the selection dialog. The resulting architecture should look like as can be seen in Figure 13. Save the architecture `adl.xml` file and close the editor.

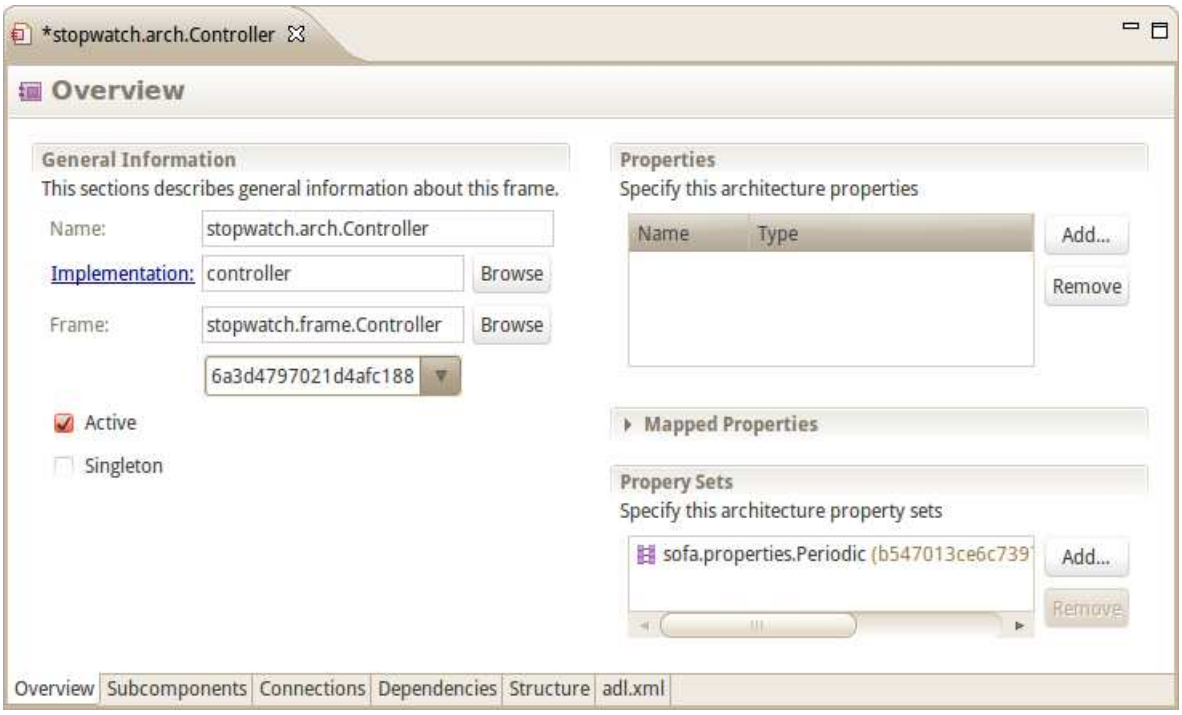


Figure 13: Creating new SOFA HI architecture

14. Create SOFA 2 composite architecture again by using the SOFA 2 Architecture wizard with the `stopwatch.arch.Stopwatch` name.

15. Open the `adl.xml` file of the created composite architecture and select *Subcomponents* editor tab.

16. Click *Add..* to add the new architecture subcomponent. In the SOFA 2 Subcomponent dialog, type `controller` into *Name* field.

17. Click the *Browse* button in the *Architecture* field and select the `stopwatch.arch.Controller` architecture using the architecture selection dialog. Following the same pattern, click *Browse* in the *Frame* field and select

`stopwatch.frame.Controller` frame. Confirm the subcomponent dialog, which should look like as in Figure 14 using *Finish*.

Figure 14: Creating new SOFA HI architecture

18. Add another subcomponent named `write` implementing `sofa.library.display.AlphanumericDisplay` frame and `sofa.library.display.CM1624` architecture.

19. Moreover, add one more subcomponent named `press` implementing `sofa.library.input.ButtonSet` frame and `sofa.library.input.ETT10PINP` architecture.

20. Select the *Connections* editor tab and click *Add...* to add the new architecture connection. Then, select *New ¿ Endpoint* from the connection's context menu. Select `controller` into *Subcomponent* field and type `write` into *Interface* field.

21. Add one more endpoint with `display` subcomponent and `write` interface.

22. Moreover, add another architecture connection with two endpoints. One with `controller` subcomponent and `press` interface, second with `buttons` subcomponent and `press` interface. Resulting architecture should look like as in Figure 15. Save the architecture `adl.xml` file and close the editor.

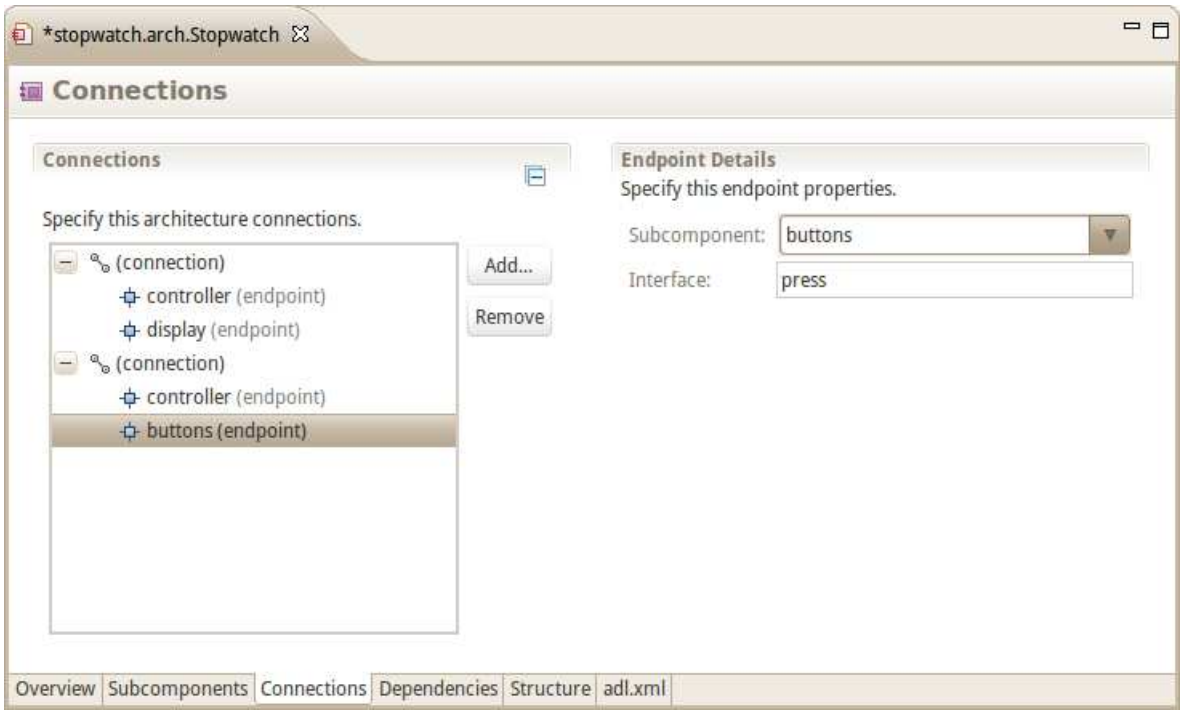


Figure 15: Creating new SOFA HI architecture

23. Commit all changes by selecting the `stopwatch` project in the SOFA 2 Navigator and selecting *Team ¿ Commit*. Select all the entities in the commit dialog, which can be seen in Figure 16 and click *Finish* to commit all local changes to a remote repository location.

24. Now, open again the `stopwatch.arch.Controller` architecture `adl.xml` file and click on the *Implementation* label in the SOFA 2 ADL Form Editor. The skeleton of component implementation is generated into architecture folder. Click *Refresh* on `stopwatch.arch.Controller` architecture in the SOFA 2 Navigator to see the `controller.h`, `controller.c`, `controller.generated.h` and `controller.generated.c` files.

25. Open the `controller.h` file and modify it by using C/C++ editor to look as follows

```
#ifndef CONTROLLER_H_
#define CONTROLLER_H_

typedef struct {
  int running;
  unsigned long value;
} controller_data;
```

Figure 16: Committing SOFA HI entities

```
#define CONTROLLER_data { \
    .running = 0, \
    .value = 0 \
  }

#define CONTROLLER_press { \
    .pressed = controller_pressed \
  }

void controller_pressed(void *this, int state);

#include "controller.generated.h"

#endif /* CONTROLLER_H_ */
```

26. Similarly, open and edit the `controller.c` file to look like follows

```
#include "controller.h"

#define START_STOP_BUTTON 0
#define RESET_BUTTON 1
```

```
void *controller_task(void *this) {
  DECLARE_SELF(this, controller);
  controller_data *data = CONTROLLER_DATA(self);

  static unsigned int hour, minute, second, millisecond;
  static char text[16];

  if (data->running) {
    data->value += self->period;
  }

  hour = data->value / 3600000;
  minute = (data->value / 60000) % 60;
  second = (data->value / 1000) % 60;
  millisecond = (data->value % 1000) / 100;

  snprintf(text, 16, "%02u:%02u:%02u.%1u", hour, minute,
     second, millisecond);
  CALL(self->write, write, 3, 0, text);

  return NULL;
}

void controller_pressed(void *this, int state) {
  DECLARE_SELF(INTERFACE_ENTRY(this, controller, press),
     controller);
  controller_data *data = CONTROLLER_DATA(self);

  switch (state) {
  case START_STOP_BUTTON:
    data->running = !data->running;
    break;
  case RESET_BUTTON:
    data->value = 0;
    data->running = 0;
    break;
  }
}
```

27. Commit the changes by using the commit dialog and select the *Upload bundles* option to upload newly created component implementation into repository.

28. Create new SOFA 2 assembly by selecting *New ¿ SOFA 2 Assembly* from the

project's context menu. Make sure that `stopwatch` appears in the *Project* field. Type `stopwatch.assm.Stopwatch` into *Name* field and click the *Browse* button in the *Architecture* field. Using the entity selection dialog that appears, select the `stopwatch.arch.Stopwatch` architecture. Finish the wizard which should look like as what can be seen in Figure 17 by clicking *Finish*.

Figure 17: Creating new SOFA HI assembly

29. Commit the created assembly using the commit dialog as in previous cases.

30. Switch to MConsole perspective in the running Eclipse IDE. The running SOFAnode should be added to workbench by selecting the menu item *File ¿ New ¿ SOFAnode* to open the *SOFA 2 SOFAnode* wizard. Type `localhost` in the *Name* and select *Manually configure* as the *SOFAnode Type*. Click *Next* to move to the second page. There leave default values and confirm the wizard by clicking *Finish*.

31. Create new deployment plan by selecting *New ¿ SOFA 2 Deployment Plan* from the SOFAnode context menu in the MConsole Navigator.

32. Type `stopwatch.deplplan.Stopwatch` into *Name* field. Click the *Browse* button in the *Assembly* field. Select `stopwatch.assm.Stopwatch` assembly in the selection dialog that appears and click *Next* in the deployment plan wizard.

33. Fill `nodeLPC2148` to *Node* field for all components. Fill in the property values for all subcomponents as follows. The `display` subcomponent should have `priority`

2 and `period` 50, `deadline` can be left to 50. The `buttons` subcomponent should have `priority` 3, `period` 10 and `deadline` can be again left to 10. Finally, The `controller` subcomponent should have `priority` 1, `period` 100 and `deadline` 100. The resulting dialog can be seen in Figure 18.

Figure 18: Creating new SOFA HI deployment plan

34. Click *Finish* to save and deploy the deployment plan.

35. Newly created deployment plan can be now launched by selecting it in MConsole Navigator and clicking *Run As ¿ SOFA 2* from the context menu. This compiles and deploys the application into selected device by using the running dock.

References

[1] SOFA 2 Component System documentation: *User's and programmer's guide*, http://sofa.ow2.org/docs/.

[2] Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: *Balancing Advanced Features in a Hierarchical Component Model*, Proceedings of SERA 2006, IEEE, August 2006.

[3] Prochazka, M., Ward, R., Tuma, P., Hnetynka, P., Adamek, J.: *A Component-Oriented Framework for Spacecraft On-Board Software*, Proceedings of DASIA 2008,

DAta Systems In Aerospace, Palma de Mallorca, European Space Agency Report Nr. SP-665, ISBN 978-92-9221-229-2, May 2008.

[4] Buttazzo, G. C.: *Hard Real-Time Computing Systems: Predictable Scheduling, Algorithms and Applications*, Springer, 2005.

[5] Siewert, S.: *Real-Time Embedded Components and Systems*, Cengage Learning, 2006.

[6] Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: *Modes for Software Architectures*, EWSA 2006, December 2006.

[7] Real, J., Crespo A.: *Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal*, Kluwer Academic Publishers, 2004.

[8] Bertrand, D., Deplanche, A.-M., Faucou, S., Roux, O. H.: *A study of the AADL mode change protocol*, Proceedings of ICECCS 2008, April 2008.

[9] Holenderski, M., Bril, R. J., Lukkien, J. J.: *Swift mode changes in memory constrained real-time systems*, Proceedings of ICCES 2009, August 2009.

[10] Burns, A.: *The Ravenscar Profile*, ACM SIGAda Ada Letters, 1999.

[11] Borde, E., Haik, G., Watine, V., Pautet, L.: *Really Hard Time developing Hard Real Time*, Proceedings of CAR 2007.

[12] Bures, T.: *Generating Connectors for Homogeneous and Heterogeneous Deployment*, Ph.D. Thesis, September 2006.

[13] Eriksson, C., Maki-Turja, J., Post, K., Gustafsson, M., Gustafsson, J., Sandstrom, K., Brorsson, E.: *An overview of RealTimeTalk, a design framework for real-time systems*, Academic Press, 1996.

[14] Calvez, J. P.: *Embedded Real-Time Systems: A Specification and Design Methodology*, John Wiley and Sons, 1993.

[15] Henzinger, T. A., Horowitz, B., Kirsch, C.: Giotto: *A Time-Triggered Language for Embedded Programming*, Proceedings of the IEEE, January 2003.

[16] Venkatramani, C., Chiueh, C.: *Design, implementation, and evaluation of a software-based real-time Ethernet protocol*, October 1995.

[17] Martnez, J. M., Harbour, M. G., Gutirrez, J. J.: RT-EP: *Real-Time Ethernet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel*, Proceedings of RTLIA 2003, 2003.

[18] Kaiser, J., Brudna, C., Mitidieri, C.: COSMIC: *A real-time event-based middleware for the CAN-bus*, Journal of Systems and Software, July 2005.

[19] Vergnaud, T. and Pautet, L. and Kordon, F.: *Using the AADL to Describe Distributed Applications from Middleware to Software Components*, 2005.

[20] Object Management Group: *CORBA Component Model Specification*, 2006.

[21] Bures, T. and Carlson, J. and Crnkovic, I. and Sentilles, S. and Vulgarakis, A.: *ProCom - the Progress Component Model Reference Manual*, version 1.0, June 2008.

[22] Fassino, J.-P. and Stefani, J.-B. and Lawall, J. and Muller, G.: *Think: A Software Framework for Component-based Operating System Kernels*, In Proceedings of the 2002 USENIX Annual Technical Conference, June 2002.

[23] Bruneton, E. and Coupaye, T. and Stefani, J.-B.: *The Fractal Component Model*, http://fractal.ow2.org/specification/.

[24] Nierstrasz, O. and Arévalo, G. and Ducasse, S. and Wuyts, R. and Black, A. P. and Müller, P. O. and Zeidler, C. and Genssler, T. and Born, R.: *A Component Model for Field Devices*, Proceedings of the IFIP/ACM Working Conference on Component Deployment, 2002.

[25] SciSys, http://www.scisys.co.uk/

[26] European Space Agency, http://www.esa.int/

[27] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[28] Eclipse Model To Model, http://www.eclipse.org/m2m/

[29] Eclipse Model To Text, http://www.eclipse.org/modeling/m2t/

[30] OMG MetaObject Facility 2.0, http://www.omg.org/spec/MOF/2.0/

[31] OMG MOF 2.0 Query/View/Transformation, http://www.omg.org/spec/QVT/

[32] OMG MOF 2.0 Model To Text, http://www.omg.org/spec/MOFM2T/1.0/

[33] Framework for Real-time Embedded Systems based on COntRacts, http://www.frescor.org/

[34] AbsInt aiT Worst-Case Execution Time Analyzers, http://www.absint.com/ait/

[35] Bound-T time and stack analyser, http://www.tidorum.fi/bound-t/

[36] Timing Definition Language, http://www.preetec.com/index.php?id=3

[37] OMNeT++, http://www.omnetpp.org/

[38] Ptolemy II, http://ptolemy.berkeley.edu/ptolemyII/

[39] SCADE Suite, http://www.esterel-technologies.com/products/scade-suite/

[40] MATLAB Simulink, http://www.mathworks.com/products/simulink/

[41] NASA OSAL, http://opensource.gsfc.nasa.gov/projects/osal/

[42] ASAAC OpenOS, http://sourceforge.net/projects/asaac-openos/

[43] Alloy, http://alloy.mit.edu/community/

[44] Event-B, http://www.event-b.org/

[45] The Cheddar project, http://beru.univ-brest.fr/~singhoff/cheddar/

[46] Apache Ant, http://ant.apache.org/

[47] Apache Ivy, http://ant.apache.org/ivy/

[48] GNU Make, http://www.gnu.org/software/make/

[49] GNU Compiler Collection, http://gcc.gnu.org/

[50] devkitPro, http://www.devkitpro.org/

[51] Linux, http://www.linux.org/

[52] FreeRTOS, http://www.freertos.org/