Undergraduate Topics in Computer Science

Flavio Oquendo
Jair Leite
Thaís Batista

Software Architecture in Action

Designing and Executing Architectural Models with SysADL grounded on the OMG SysML Standard

EXTRAS ONLINE

Springer

Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at http://www.springer.com/series/7592

Flavio Oquendo · Jair Leite
Thaís Batista

Software Architecture in Action

Designing and Executing Architectural Models with SysADL grounded on the OMG SysML Standard

Springer

Flavio Oquendo
IRISA Research Institute
University of South Brittany
Vannes
France

Thaís Batista
Department of Computer Science
Federal University of Rio Grande do Norte
Natal, Rio Grande do Norte
Brazil

Jair Leite
Department of Computer Science
Federal University of Rio Grande do Norte
Natal, Rio Grande do Norte
Brazil

Additional material to this book can be downloaded from http://extras.springer.com.

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-44337-9 ISBN 978-3-319-44339-3 (eBook)
DOI 10.1007/978-3-319-44339-3

Library of Congress Control Number: 2016947944

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

To my beloved parents, my wife Andressa and my son Bruno

—Flavio Oquendo

To my parents Ernani and Marilena and to my daughters Larissa and Cristina

—Jair Leite

To my father, Juarez da Gama Batista
(in memoriam)
To my mother, Lygia Vasconcelos Batista
To my daughters, Larissa and Cristina

—Thais Batista

Preface

The main goal of this book is to cover a wide spectrum of software architecture modeling techniques using viewpoints to describe the structure, the behavior, and the execution of a software architecture. It is a textbook that covers fundamental approaches of software architecture description, reconciling theory and practice with well-established learning outcomes. The book includes additional resources available at www.sysadl.org such as lecture slides that will be helpful to broader learning and an open-source software tool to support practical exercises.

Indeed, in the last two decades, software architecture has become a major discipline in the intersection of Computer Science and Software Engineering in its own (like civil architecture compared to civil engineering). Besides, software architecture plays a key role for enabling the next generation of software-intensive systems.

The recognized importance of software architecture led a number of universities to include regular course on software architecture at both graduate and undergraduate levels. However, there is still a lack of textbooks focusing on software architecture modeling based on explicit learning outcomes. This book aims to cover this gap, presenting a systematic approach supported by a software tool to model software architectures from different viewpoints and execute the resulting model for validation purposes, therefore covering the essence of software architecture design.

This book is designed for teaching software architecture modeling techniques to both graduate and undergraduate students, in order to prepare them to architecting complex software-intensive systems. It is also appealing for practitioners and members of a software development team such as architects, designers, programmers, project managers, since it is structured around practical modeling approaches spanning different roles in software development.

This book defines an architecture description language, named SysADL, as a specialization of the OMG SysML standard to software architecture description. SysADL brings together the expressive power of software architecture description languages (ADLs) for architecture description, with a standard language used by the industry (SysML). SysADL is used in all the chapters of this book.

Vannes, France — Flavio Oquendo
Natal, Brazil — Jair Leite
Natal, Brazil — Thaís Batista

Outline

This book is structured into four parts. Part I covers the fundamentals including the main concepts for modeling software architecture and presents SysADL, derived from the OMG SysML standard. The concepts follow the ISO 42010 reference model. The chapters present the concept of viewpoints and views, and how to describe, using SysADL, the structure, the behavior and the execution of a software architecture.

Part II focuses on how to design a software architecture for achieving quality attributes. Each chapter covers a specific quality attribute and presents well-defined approaches to achieve it.

Part III presents how to apply software architecture style to design architectures that meet the quality attributes. Each chapter covers a specific architectural style.

Part IV presents how to textually represent software architecture models in complement to the visual notation.

Acknowledgments

We would like to acknowledge all those who contributed for our work in this book. Special thanks to Eduardo Alexandre Ferreira Silva for his valuable support in providing many suggestions, reviewing parts of this book, and implementing the SysADL toolset. We also thank Bruno Carlos da Cunha Costa for his contribution in the initial development of the SysADL toolset, and Everton Ranielly de Sousa Cavalcante for reviewing parts of this book.

We would like to thank CNPq—the Brazilian Council for Scientific and Technological Development—for the financial support for our research collaboration that contributed to allow us writing this book.

We would like to thank the staff of Springer for their support, especially Beverley Ford and James Robinson.

Finally, we thank our families and friends for their encouragement and support.

Contents

Part IV Textual Description of Architectures

About the Authors

Flavio Oquendo is Full Professor of Computing at the University of South Brittany, France, and Principal Investigator on Software Architecture at the IRISA Research Institute (UMR CNRS 6074). He was awarded the degrees of Ph.D. and HDR (Research Direction Habilitation) in Computing from the University of Grenoble, France.

He has been working with Software Architecture since 1986. His research interests are centered on formal languages, processes, and tools to support the efficient architecture and engineering of software-intensive systems and their applications in industrial settings.

He was Scientific Director of pioneering European Projects on Software Architecture in cooperation with the industry and founding co-chair of the French and European Conferences on Software Architecture. Also on Software Architecture, he has been involved as Program Chair of the French, European, and IEEE/IFIP conferences and has edited many Special Issues in International Journals.

Jair Leite has been working with Software Design and Architecture since 1998. His research interests are architecture-based development, human computer interaction, design languages. He has been member of several Program Committees of Brazilian and International conferences involving Software Architecture.

Thais Batista is Full Professor of the Informatics Department at the Federal University of Rio Grande do Norte (UFRN), Brazil. She is vice-president of the Brazilian Computer Society (SBC). She has been working with Software Architecture since 2000. Her research interests are architecture-based development, middleware, internet of things, cloud computing, system of systems. She has been involved as Program Committee member of several Brazilian and International conferences involving Software Architecture.

Part I
Fundamentals

Chapter 1
Introduction to Software Architecture

In this chapter, we introduce the concept of software architecture and the SysADL architectural framework for describing, analyzing, and executing software architectures. We present the motivation for defining SysADL and describe the organization of the book for putting software architecture in action with SysADL. We introduce a running example to illustrate software architectures in action along the chapters of this book.

You will learn the following:

- the concept of software architecture;
- the rationale for defining SysADL;
- the approach provided by SysADL to address the description, analysis, and execution of software architectures.

1.1 The Concept of Software Architecture

The concept of architecture is well known for thousands of years. It has been used originally for buildings as "the art or practice of designing and constructing buildings" (Oxford English Dictionary) [1] and evolved to mean nowadays "the complex structure of something," where this "something" could be, for instance, a building, a ship, or a software.

Applying the concept of architecture to software became a main concern to software engineers. In fact, since the 1990s the complexity of software increased in

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_1

a way that mastering this complexity became the central point in software design and more largely in the design of software-intensive systems.

Since 2011 we have an official definition of software architecture. It is given by the ISO/IEC Standard 42010 "Systems and Software Engineering—Architecture description" [2].

> The definition of software architecture is: "*The fundamental properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution*".

To well understand the concept and definition of software architecture, you should think about different interrelated notions, in particular the notions of: (i) *structure* (what is the form of the interrelated elements), (ii) *behavior* (what is the functionality that the interrelated elements provide), (iii) *execution* (how the functionality provided by the interrelated elements is carried out), and (iv) *analysis* (what are the nonfunctional properties that the interrelated elements satisfy).

Structure is the organization of elements and their relationships for executing the functionalities while satisfying the properties of the system. *Behavior* is the detailed specification of activities allocated to the structural elements, which provides the functionality of the architected system. *Execution* is the detailed specification of atomic actions that supports the actual execution of the described behavior. *Properties* complete the description of structure, behavior, and execution with the quality of service. The *nonfunctional properties* need to be analyzed against the architecture description to ensure their satisfaction.

As we can see, the ISO/IEC Standard 42010 definition for architecture description highlights three matters in an architecture:

- the elements;
- the relationships between elements;
- the principles in the design and evolution of these interrelated elements.

This definition also highlights that the architecture comprises the implied properties of these constituents, basically:

- the structure of each element, its behavior, its execution semantics, and its nonfunctional properties;
- the structure of each relationship between elements, its behavior, its execution semantics, and its nonfunctional properties;
- the structure of the interrelated elements, its behavior, its execution semantics, and its nonfunctional properties.

These concerns are embodied in *component* as the software architectural concept of element, and *connector* as the software architectural concept of relationship between elements. The principles in the design and evolution of the architecture will be described through nonfunctional *properties* and encoded in architectural styles.

Software architecture is an essential activity for the development of software-intensive systems enabling to reason about system properties very early in the development lifecycle. The issue is on how to organize a system to, simultaneously:

- provide the required functional services,
- guarantee the required quality of service.

1.2 Language for Modeling Software Architecture

As an engineering artifact, a software architecture must be described. Two lines of work emerged for software architecture description: one based on the definition of new languages for describing the architecture of software-intensive systems—the so-called "Architecture Description Languages (ADLs)"—and another based on the use of general-purpose modeling languages, in particular Unified Modeling Language (UML) [3] and recently Systems Modeling Language (SysML) [4].

Although many ADLs have been proposed since the 1990s, none of them have a broad adoption in the industry [5], even if a few have been adopted in specific domains, for instance, AADL [6] which was developed for the field of avionics and automotive applications.

In 1997, the Unified Modeling Language (UML) emerged as an OMG Standard (UML 1.0) and gained popularity with a high acceptance by the software development community and industry. However, the first versions of UML did not include the support for describing software architectures. While UML 2.0, adopted by the OMG in 2005, has improved the language with some architectural modeling features (i.e., the notion of architectural component and composite structures, however not any of architectural connectors), it is still limited for describing software architectures [7].

In 2007 (10 years after UML), SysML was published as an evolution of UML for systems engineering, and it has been increasingly used by software-intensive systems engineers, inheriting the popularity of UML.

SysML enriches UML with new concepts, diagrams, and it has been widely adopted to describe software-intensive systems. However, in terms of architectural description, SysML inherits the limitations of UML: architectural constructs are basically the same as UML, limited to the notion of component and composite structures.

Let us recall why UML was born in the mid-1990s: in the 1980s and beginning of 1990s, too many modeling languages were proposed by the community, each industry adopting one or several of them, and even some been adopted only internally. The need was to unify all modeling languages around a unique, common and shared language: the Unified Modeling Language. Designed for software modeling, UML has been extended for systems modeling (including software-intensive systems modeling) and issued as SysML.

For ADLs, we are in a similar and even worst situation: along the years more than 120 languages [8] have been proposed by the research community, with only very few of them being adopted in the industry for particular application domains, e.g. AADL in the avionics and automotive industry.

The above-mentioned problems motivated us to define SysADL as a specialization of SysML to the architectural description domain, with the aim of bringing together the expressive power of ADLs for architecture description with a standard language widely accepted by the industry, which itself provides hooks for specialization.

Indeed, as UML, SysML is also a semi-formal modeling language, with semantic gaps. Some of these gaps are the so-called "semantic variation points" which are part of the definition of SysML (as well as of UML) for supporting the specialization of SysML (as well as of UML) for different purposes. The semantic variation points together with the profile mechanism enable specialization of the syntax and semantics of SysML for architecture description.

Therefore, we have designed SysADL as a specialization of SysML (in the sense that all architecture descriptions expressed using SysADL are also valid SysML models), while adding new architectural concepts (completing SysADL in terms of ADL expressiveness) by filling architecture-related semantic gaps of SysML.

The resulting language, named SysADL, reconciles the expressive power of ADLs with the use of a common syntax in line with the SysML standard.

SysADL copes with the architectural concepts defined in the ISO/IEC/IEEE 42010 Standard in terms of multiple viewpoints. According to the Standard, an *architectural viewpoint establishes conventions for the construction, interpretation, and use of architecture views to frame specific systems concerns*.

In addition, SysADL has a rigorous operational semantics, which allows the analysis (in terms of verification of both structure and behavior) and execution (in terms of executable specifications for validation) of the architecture.

1.2.1 Why Conceiving SysADL

SysML is more expressive than UML, however SysML is not an ADL; in the sense that it does not provide the minimal set of architectural concepts of an ADL. SysML is a general-purpose modeling language encompassing some of the architectural concepts (but not all, and not consistently assorted), mixed with design concepts and implementation ones.

The question is therefore: what would be the suitable ADL to specify the architecture of systems? The answer is that the suitable ADL is the one that solves the following trade-off:

- the ADL must satisfy the architecture needs and concepts expressed in the ISO/IEC/IEEE 42010 Standard;

- the ADL must comprise the core architectural concepts according to different viewpoints, in particular the structural and behavioral;
- the ADL should be based on standard notation whenever possible.

This ADL does not exist yet. It is for this reason that we defined SysADL as a specialization of SysML. SysADL was designed to cope with these three key requirements:

- SysADL is based on the standard framework for architecture description, as defined by the ISO/IEC/IEEE 42010 standard;
- SysADL is based on well-established concepts drawn from the R&D on software architecture. It provides a well-proven set of architectural constructs and viewpoints with more expressive power than most of the ADLs proposed in the literature, being able to describe not only static architectures (architectures that not change at runtime), but also dynamic architectures (architectures that may change at runtime);
- SysADL is based on the SysML standard, by being a compliant specialization of SysML.

1.2.2 Introducing SysML for SysADL

SysML was jointly defined by the Object Management Group (OMG) and the International Council on Systems Engineering (INCOSE). It is a general-purpose modeling language for systems engineering, including in particular software-intensive systems engineering. It supports the specification, analysis, design, verification, and validation with a diagrammatic notation that is more expressive and flexible than UML while being smaller.

SysML, by being a generalization of UML with a smaller notation, is easier to learn than UML and, of course, very easy to learn for those who already know UML. As shown in Fig. 1.1, it provides nine kinds of diagram, of which seven are borrowed from UML. Of these seven, four are used as they are (package diagram, use case diagram, sequence diagram, and state machine diagram) and three were extended (block definition diagram generalizes class diagram; internal block diagram generalizes composite structure diagram, and activity diagram was extended with more general features). The two new diagrams provided by SysML are the *requirement diagram* for expressing the definitions and dependencies of requirements and the *parametric diagram* mostly for defining quantitative constraints.

Note that SysML provides a better support for model management than UML in particular regarding viewpoints and views. SysML extends UML with mechanisms that support (even if does not provide) the ISO/IEC/IEEE 42010 Standard.

These significant improvements over UML make SysML a better modeling language for architecture description. But, as discussed, SysML falls short regarding support for architectural concepts.

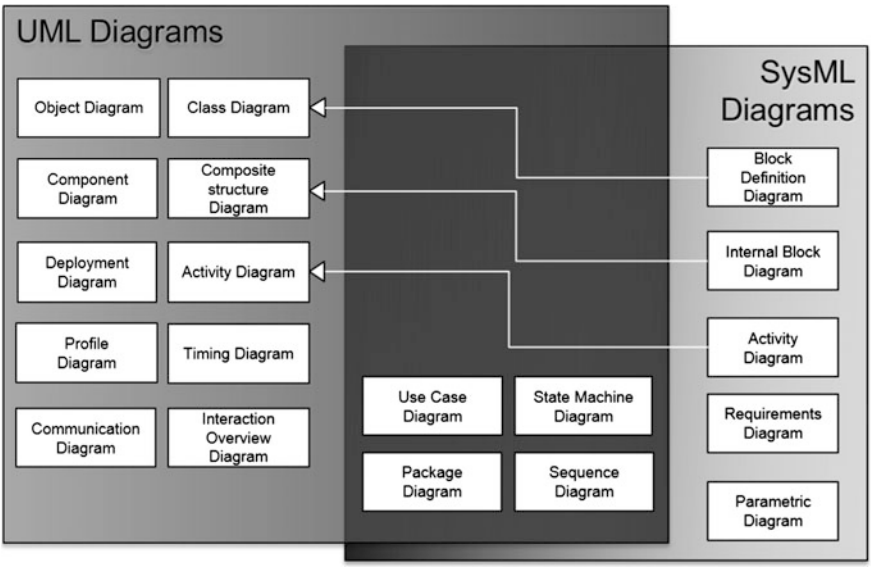


Fig. 1.1 SysML versus UML diagrams

1.2.3 SysADL as a Specialization of SysML for Architecture Modeling

As discussed, SysML provides a suitable foundation for the definition of an ADL. SysADL is thereby defined as a subset of SysML specialized for architecture description and analysis.

As depicted in Fig. 1.2, SysADL reuses the requirement diagram and specializes four diagrams of SysML:

- the block definition diagram (bdd) for modeling the structure of architectural components and connectors;
- the internal block diagram (ibd) for modeling the structure of architectural configurations;
- the activity diagram for modeling the behavior of architectural components and connectors;
- the parametric diagram for expressing qualitative as well as quantitative architectural properties.

In addition, SysADL extends SysML with the OMG Action Language for Foundational, UML (ALF) [9] focusing on architecture-relevant constructs. SysADL provides thereby, a specialization of ALF adapted from UML to SysML for specifying executable models of software architecture. An executable architecture description modeled with SysADL provides an executable specification

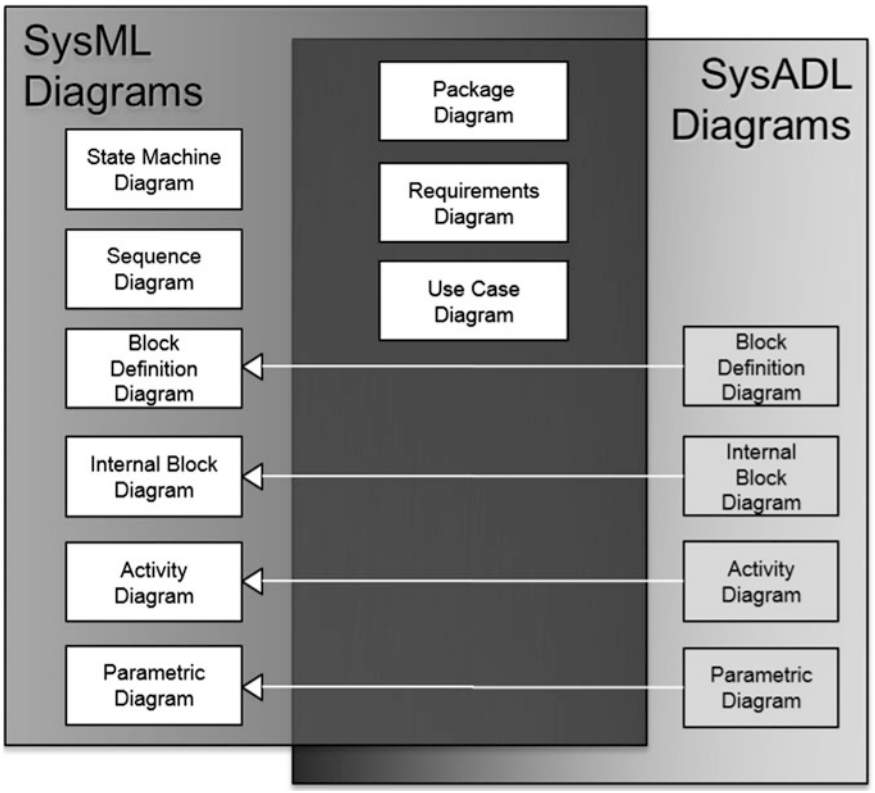


Fig. 1.2 SysADL versus SysML diagrams

detailed enough (but independent of any implementation platform or programming language) that can effectively be run as it was a "program".

In summary regarding architectural modeling, SysADL provides an extremely more powerful language than full SysML.

1.3 Designing Software Architecture with SysADL

Having a language for describing software architecture, both declaratively and prescriptively, and supporting both static and dynamic analyses is a need satisfied by the definition of SysADL, but is not sufficient. The question is therefore how to use SysADL in practice for designing software architecture. This question is answered in the different parts of this book.

1.3.1 Describing Software Architectures

In Part I of the book, we present how to describe a software architecture. First, in Chap. 2, we introduce the concepts of viewpoint and view in the description of software architecture. Then, in Chap. 3 we present how to represent software architecture requirements in SysADL. In this part, we also present in detail how to describe software architecture according to three viewpoints: in Chap. 4, how to describe the software architectures from the structural viewpoint; in Chap. 5, how to describe the software architectures from the behavioral viewpoint; in Chap. 6, how to describe the software architectures from the executable viewpoint. Finally, in Chap. 7, we present the execution semantics of architecture descriptions expressed with SysADL.

1.3.2 Designing Quality-Based Software Architectures

In Part II of the book, we present how to design a software architecture for achieving quality attributes with SysADL. First, in Chap. 8, we introduce the concept of quality attribute and then present three cases of design: in Chap. 9, how to design a software architecture that is easily modifiable (*modifiability* quality attribute); in Chap. 10, how to design a software architecture that is easily scalable (*scalability* quality attribute); and in Chap. 11, how to design a software architecture that is fault tolerant (*fault tolerance* quality attribute).

1.3.3 Designing Style-Based Software Architectures

In Part III of the book, we present how to define software architecture styles in SysADL, which will afterwards be used to describe a software architecture. In Chap. 12, we introduce the concept of *architectural style*. Next, we present four widely used architectural styles: in Chap. 13, we describe how to define and use the *Pipe-Filter* architectural style; in Chap. 14, the *Client–Server* architectural style; in Chap. 15, the *Feedback Control Loop* architectural style; and finally, in Chap. 16, the *Blackboard* architectural style.

1.3.4 Textually Representing Software Architectures

In the last part of the book (Part IV), we present the textual notation to the SysADL constructs represented in the diagrams. In Chap. 17, we present how to textually represent software architecture models in complement to the visual notation.

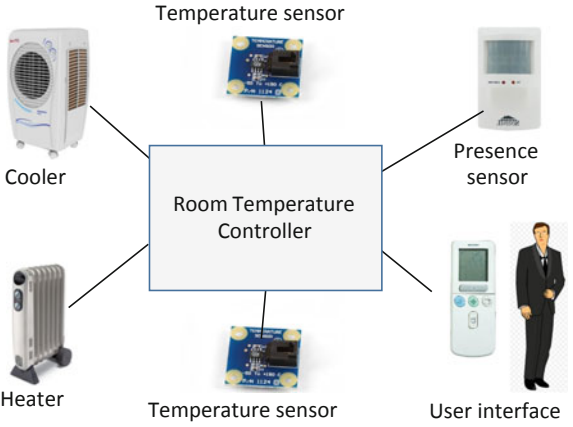


Fig. 1.3 RTC system overview

1.4 Running Case Study to Illustrate Software Architecture

To illustrate the different concepts and constructs of software architecture, including architecture description and analysis, we present hereafter a case study for architecting a *Room Temperature Controller* (*RTC*) system.

This *RTC* system will be architected to control the temperature of a room. As depicted in Fig. 1.3, it has two *temperature sensors* to capture the current temperature in different places of a room. The *central controller* receives the values from the *temperature sensors*, compares them with the desired temperature and turns the *cooler* or the *heater* on or off. Furthermore, the system has a presence sensor to detect if there is someone in the room. In case of presence, the system operates to provide the desired temperature. Otherwise, the system operates to maintain the temperature in 22 °C.

1.5 Summary

In this chapter, you have learnt the following:

- the principles underlying the concept of software architecture;
- the principles underlying the definition of SysADL as an ADL derived from SysML;
- how SysADL will be presented in this book, along the chapters, for designing a software architecture.

Further Reading

1. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6) (2013)
2. Medvidovic, N., et al.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
3. Medvidovic, N. et al.: Modeling Software Architecture in the Unified Modeling Language. ACM Trans. Softw. Eng. Methodol. (TOSEM). **11**(1), 2–57 (2002)

References

1. www.oed.com
2. www.iso-architecture.org/42010/index.html
3. www.uml.org/
4. www.omgsysml.org/
5. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6) (2013)
6. www.aadl.info
7. www.omg.org/spec/UML/2.0
8. Current existing architectural languages, http://www.di.univaq.it/malavolta/al/. Accessed April 2015
9. http://www.omg.org/spec/ALF/

Chapter 2
Viewpoints for Describing Software Architectures

In this chapter we present the architectural framework provided by SysADL. We define software architecture and the fundamental notion of software architecture description according to the ISO/IEC Standard 42010 "Systems and Software Engineering—Architecture description" [1]. We present, in detail, the concepts underlying the architecture description in terms of *viewpoints* and *views*.

You will learn the following:

- the definition of software architecture;
- the concepts of viewpoints and views for describing a software architecture;
- the architectural framework provided by SysADL in terms of concepts, viewpoints, and views.

2.1 The Definition of Software Architecture

The ISO/IEC Standard 42010 "Systems and Software Engineering—Architecture description" defines software architecture as "*The fundamental properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution*".

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_2

As we can see, this definition highlights three matters in an architecture:

(i) the elements;
(ii) the relationships between elements;
(iii) the principles in the design and the evolution of these interrelated elements. This definition also highlights that the architecture comprises the implied properties emerging from these constituents.

2.2 Software Architecture Description

An architecture needs to be described to be used by its different stakeholders. According to the ISO/IEC/IEEE 42010 Standard, an architecture description is: "a work product used to express an architecture."

Let us now put the notion of architecture description in context, before presenting its characteristics. Figure 2.1 depicts this context.

Let us now present this conceptual model of the context of an architecture description:

- An architecture description expresses, at least, one architecture, but not all architectures are described. It means that an architecture description may describe one or many architectures.
- An architecture may be exhibited by none or several systems. It means that an architecture is a notion that may be realized in different concrete systems.
- A system is situated in an environment. It means that the interaction with the environment needs be taken into account in the architecture description. Note that a system is more that the software part of it. Indeed, it is a software-intensive system.

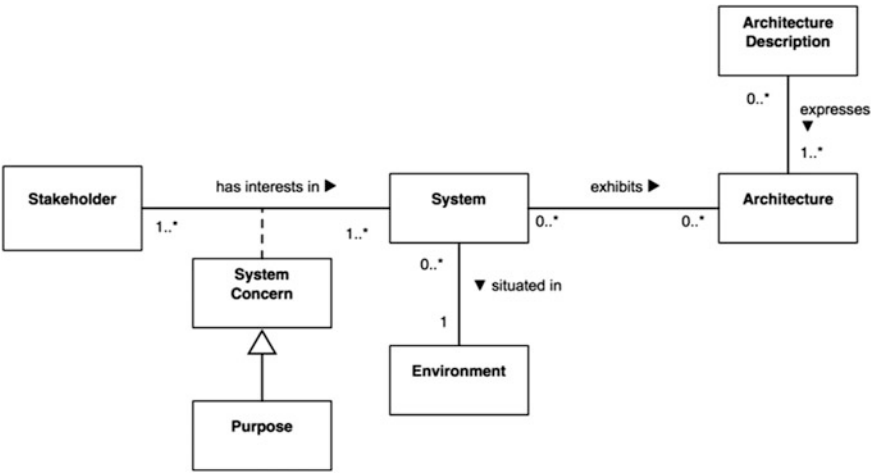


Fig. 2.1 The context of an architecture description [ISO/IEC/IEEE 42010]

Now let us examine the involved stakeholders. They comprise person, group, or organization with an interest in one or several systems. The interests are expressed as *concerns*. Each concern has a *purpose*.

For instance, considering the RTC system, examples of stakeholders are: clients, the architect, and developers. Clients are the users of the system, thereby they are concerned with performance and availability. The associated purpose for availability is to enable the use of the system whenever needed, and the purpose for the performance concern is the effectiveness and efficiency of the system. Developers are mainly concerned with modifiability, in order to facilitate maintenance and also to reduce the costs. All those concerns are important to the architect who is in charge of developing a solution that satisfies client and developers concerns. Figure 2.2 illustrates these stakeholders and concerns.

Let us now explain the elements of an architecture description based on the conceptual model of an architecture descriptions provided by the ISO/IEC/IEEE 42010 Standard, depicted in Fig. 2.3.

According to this conceptual model, an architecture description:

- identifies a system-of-interest (it is the system that, for instance, will be developed based on the described architecture);
- identifies the system stakeholders and their concerns (the stakeholders that will, for instance, use and develop this system);
- defines the architecture viewpoints used to represent the architecture (these viewpoints support the description of architecture in terms of different views);
- supports the correspondences between views in terms of architectural elements, possibly defined by correspondence rules (see Fig. 2.4);
- documents the architectural decisions with their rationale.

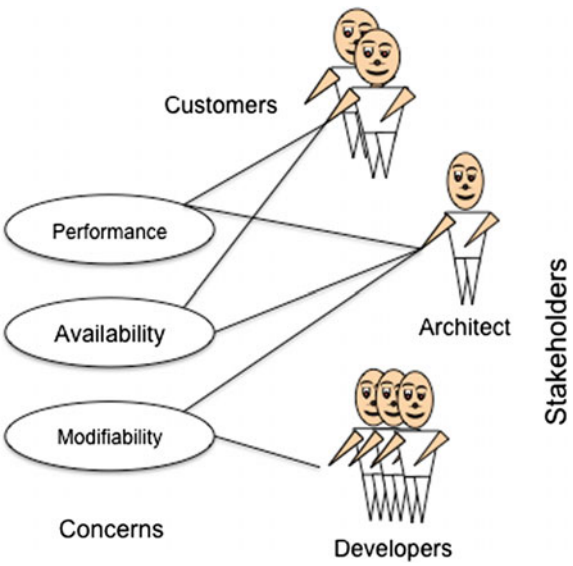


Fig. 2.2 Stakeholders and concerns in the RTC system

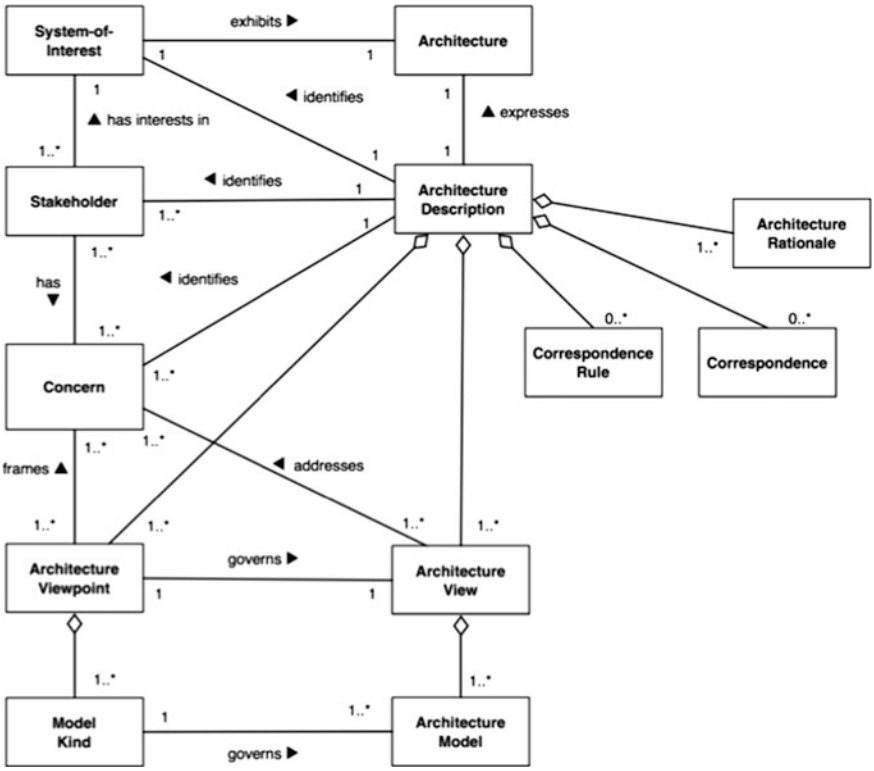


Fig. 2.3 Conceptual model of an architecture description [ISO/IEC/IEEE 42010]

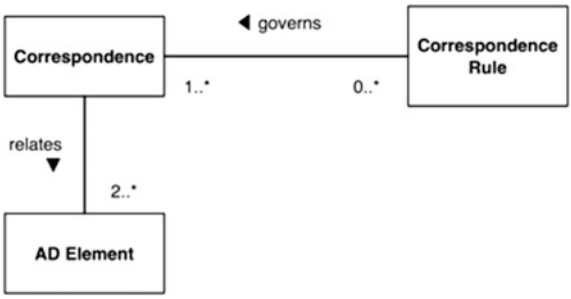


Fig. 2.4 Correspondences relating architectural elements [ISO/IEC/IEEE 42010]

A key concept in architecture description is *viewpoint*. All concerns are mapped onto the viewpoints. Each architectural viewpoint governs one or more architecture *views*. This means that the architecture viewpoint defines the constructs, including rules and conventions, for the definition of the views describing an architecture. An architecture view comprises one or more architecture models. Figure 2.5 illustrates three viewpoints, each one with a view.

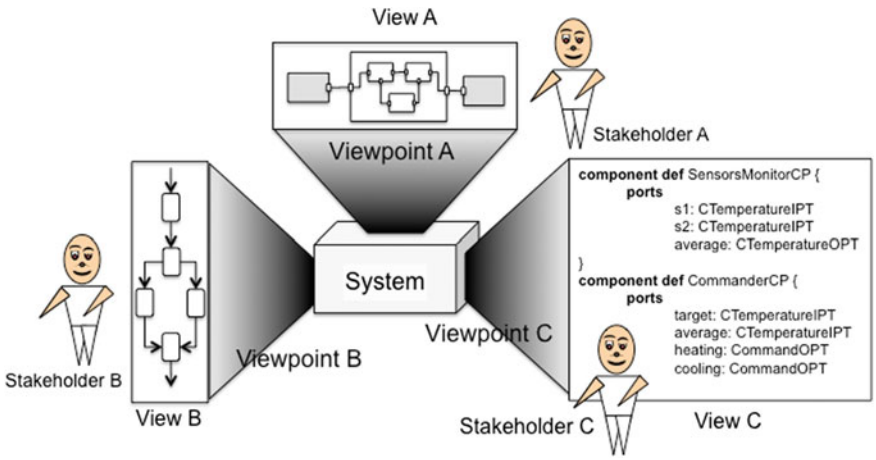


Fig. 2.5 Viewpoints and views

In this context, an architect has the role of describing the architecture of the system-of-interest, considering the environment, the stakeholders, and their concerns. The architect will describe the architecture using different architecture views according to the viewpoints supported by an *architecture framework*. According to the ISO/IEC Standard 42010, an architecture framework provides the "conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders."

SysADL is an architecture framework for software engineers and developers, providing a set of viewpoints to the stakeholders. For each viewpoint, the SysADL framework defines a language with conventions and principles about the definition and use of the architectural elements of the viewpoint.

2.3 Concepts for Describing Software Architecture

The ISO/IEC/IEEE 42010 Standard specifies a conceptual model of architecture description, but it does not define the main building blocks and primary elements of an architecture description. For this purpose, we adopt the well-known abstractions of *components*, *connectors*, and *configuration*. These concepts are independent of any specific notation, concrete syntax, or architecture language.

Figure 2.6 shows a conceptual model relating these three architectural elements. This figure shows that an architecture description element (*AD Element*) may be a component, a connector, or a configuration. We define that a configuration may contain one or more components, and none or several connectors. A connector connects two or more components.

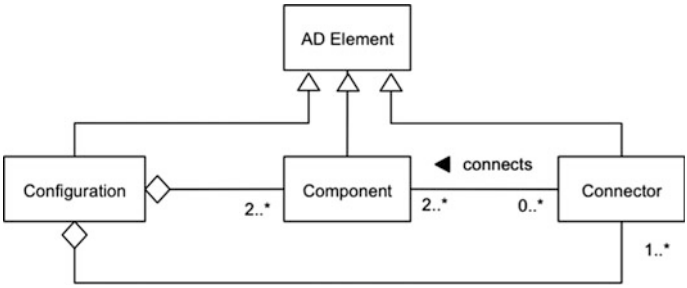


Fig. 2.6 Architectural description elements

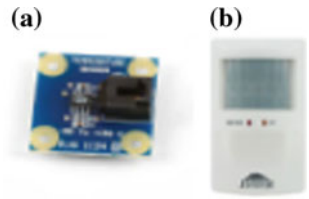


Fig. 2.7 **a** Temperature sensor component; **b** Presence sensor component

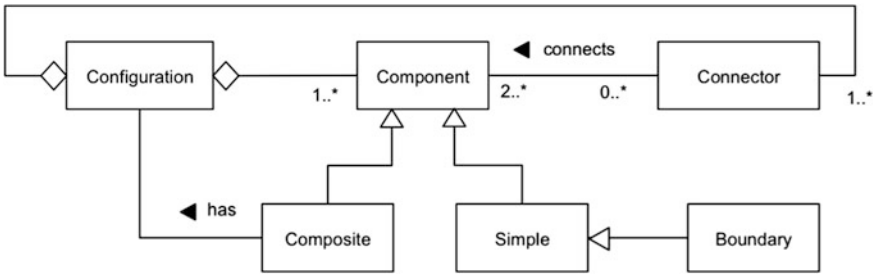


Fig. 2.8 Simple and composite components

Components are architectural elements that provide functionalities in a system. It is the central element, the loci of computation and state. Figure 2.7 illustrates two components: a temperature sensor and a presence sensor.

As depicted in Fig. 2.8, a component can be a simple element providing a simple functionality, or even a composite element representing a system as a whole, which itself is composed of others components too. A *simple component* performs sequential computation using data available in its ports. A *composite component* performs concurrent computation by being composed of components in their internal structure.

A component has clearly defined *ports* that are interaction points between it and the external world, i.e., its environment. Ports specify the data that a component

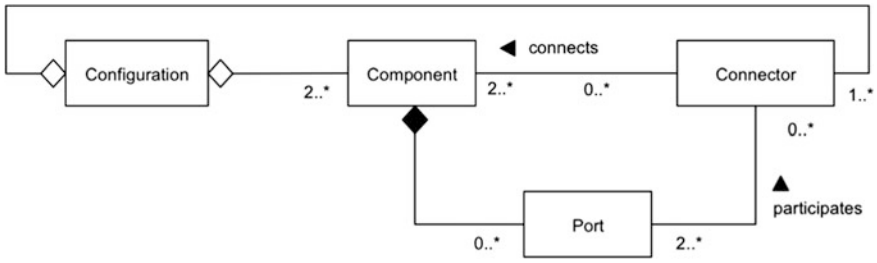


Fig. 2.9 Ports

provides or requires from other components in the architecture. The explicit specification of the data that a component provides and requires is essential to support the component's composability. In fact, the specification of an architecture follows the idea of assembling components together to form a system using information from their interfaces. Figure 2.9 shows that a component has none or several ports. Thus, ports belong to components and participate to connectors (ports do not belong to connectors).

Components can be internally located in a system or in the boundary between the system and the environment. *Boundary components* are thereby placed at the interface with the environment. Internal components are located inside a composite component including the architecture itself, and therefore are not visible outside of it. A component that is located in the boundary represents the system interface with the environment, encapsulating the mechanisms the system uses to interact with the environment. As an example, you can see that the *RTC* System has different boundary components such as a sensor that physically measures the temperature and provides its value in an *out* port. Figure 2.7 illustrates two boundary components.

The second fundamental concept in an architecture description is a *connector*. A connector is an element responsible for mediating the interaction among components, establishing rules that govern those interactions. It provides the glue mechanism to binding components together. Figure 2.10 illustrates a connector. Note that each side of the connector specifies the kind of component's port that can be bound to it.

Connectors define which ports can be connected and how the interactions between connected components take place. Therefore, they are the locus of communication. As sketched in Fig. 2.9 a connector refers to the ports it connects. It connects at least two ports.

As shown in Fig. 2.11, a connector can be a *simple* element connecting two or more ports, or even a *composite* element which itself is composed of others

Fig. 2.10 Connector

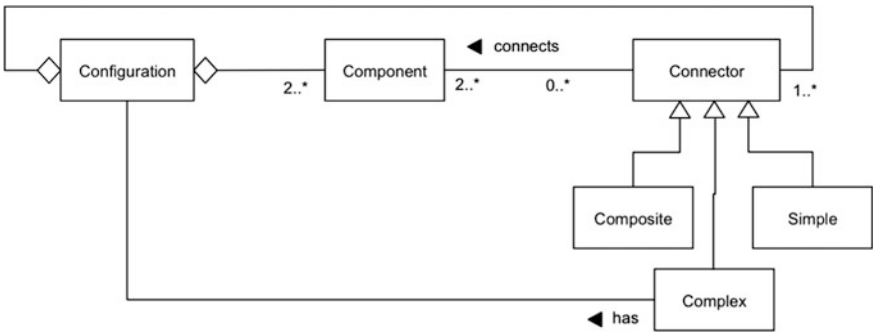
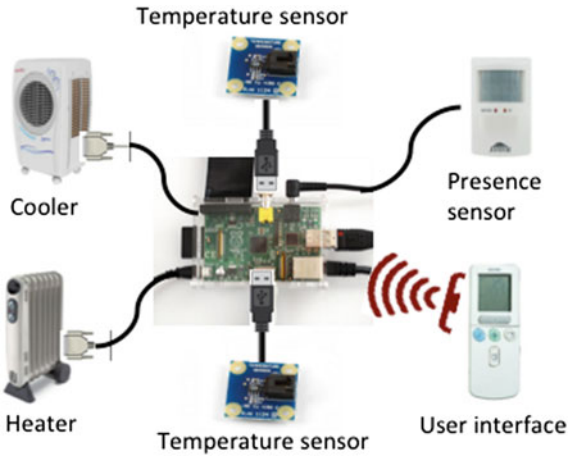


Fig. 2.11 Simple and composite connectors

Fig. 2.12 Configuration of the RTC System



connectors, or a *complex* element that has a configuration. For instance, in the case of the RTC system, a connector (Fig. 2.12) could be used to connect the temperature sensor component with the controller component.

As you can see, components and connectors are complementary concepts and provide a clear separation of concerns between computation and communication.

The third fundamental concept in an architecture description is *configuration*. *Configurations* describe the topology for identifying which components are part of a software architecture and how they are connected together through connectors. In this way, the architecture configuration defines a connected graph of components and connectors that describes the architecture. In the example of the RTC system, as shown in Fig. 2.12, the configuration could define that the controller is connected in a star topology configuration with two temperature sensors, one presence sensor, and two actuators—the cooler and the heater. They use cable connectors to send and receive data to the controller. The user interface is a remote control that sends data via an infrared connector.

These three concepts raise different needs in terms of software architecture description:

- how to describe the ports of components?
- how to describe the ports of connectors?
- how to describe the configuration of components and connectors?
- how to describe the behavior of components?
- how to describe the behavior of connectors?
- how to describe the behavior of the configuration of components and connectors? And also:
- how to validate the designed architecture?
- how to verify the implied properties of the designed architecture?

To address these needs, architecture description languages (ADLs) were defined.

2.4 Architectural Viewpoints and Views

As specified in the ISO/IEC/IEEE 42010 Standard, an architecture is described from different viewpoints in terms of *views* represented by *model diagrams*. We will present now these two notions.

2.4.1 Architectural Viewpoints

Viewpoints realize the stakeholders concerns by means of *architecture views*. A viewpoint defines the kinds of model that govern the diagrams used to represent its corresponding views. Such models are what a stakeholder "sees" when looking at the system from a specific viewpoint. The ISO/IEC/IEEE 42010 Standard also emphasizes that multiple views are essential to cover all the stakeholders' concerns, and to detail the architecture from different perspectives.

Let us make an analogy between software architecture and civil architecture, which also represents architectures from different viewpoints. The well-known structural, electrical, and hydraulic plans are some examples of civil architecture views from the structural, electrical, and hydraulic viewpoints. Each one provides a specific view of a building. The plans are used by the stakeholders—engineer, architect, brick worker—to reason about and govern the building process. They help the stakeholders to understand the building concerns. Similarly, software architecture views show different perspectives of the software-intensive system. They communicate the architecture to the different stakeholders.

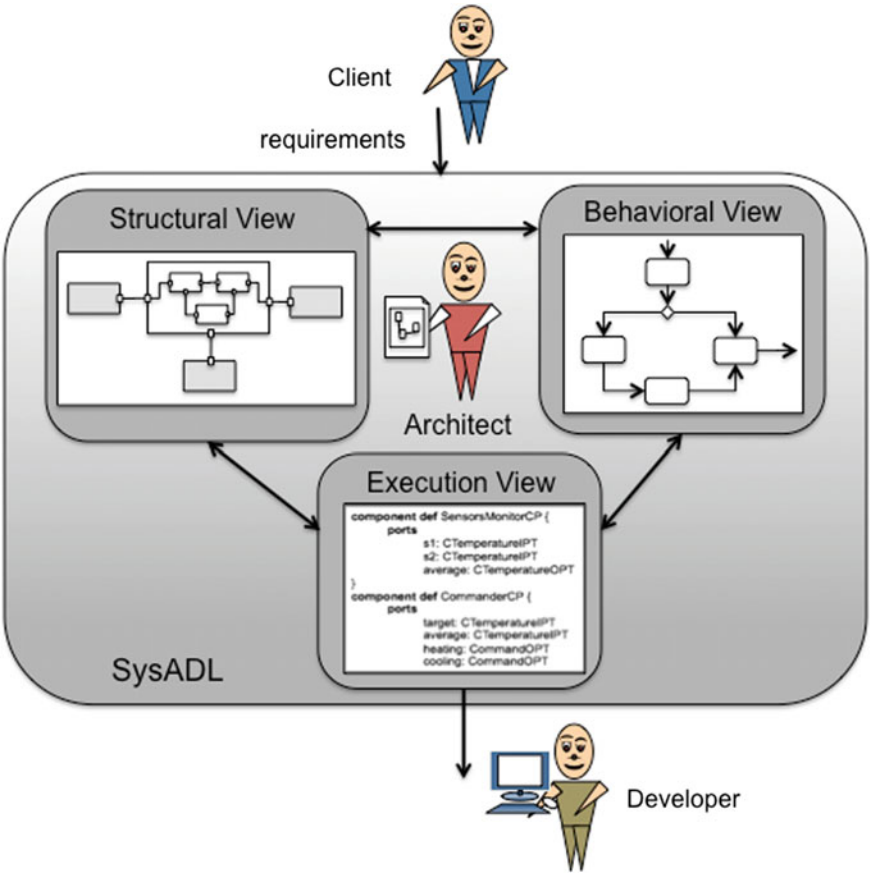


Fig. 2.13 Architectural viewpoints

The main stakeholders interested in the software architecture description are clients, architects, and developers. As illustrated in Fig. 2.13, the client specifies the requirements of the system to be realized by the architecture, and is interested in the executable architecture satisfying the requirements. The architect is concerned with the specification of the architecture, reasoning about it to verify properties and detailing its structure, behavior, and execution. Structure and behavior are declarative specifications that can be mapped to a corresponding executable model, characterizing the executable viewpoint. Finally, the developer is interested in the executable architecture for implementing the system.

According to the concerns depicted, these three architectural viewpoints are essential to communicate the architecture to the involved stakeholders.

2.4.2 Architectural Views

As discussed, an ADL must support different viewpoints. SysADL defines three architectural viewpoints that are essential to communicate the software architecture to the involved stakeholders:

(i) the structural viewpoint;
(ii) the behavioral viewpoint;
(iii) the executable viewpoint.

Each viewpoint provides the constructs for describing different views of the architecture from these viewpoints. For instance, Fig. 2.14 shows two viewpoints (style level viewpoint and instance level viewpoint) and an architecture description with three views (one for each of the three viewpoints).

The structural view describes the main building blocks of the architecture from a conceptual point of view: components, connectors, and configurations. It makes use of two kinds of diagrams: the block definition diagram (*bdd*) and the internal block diagram (*ibd*). The *bdd* is used to define the components and connectors of the architecture. The *ibd* goes a step further, showing how components and connectors bind each other defining the configuration of the architecture or of the internal structure of composite components and connectors.

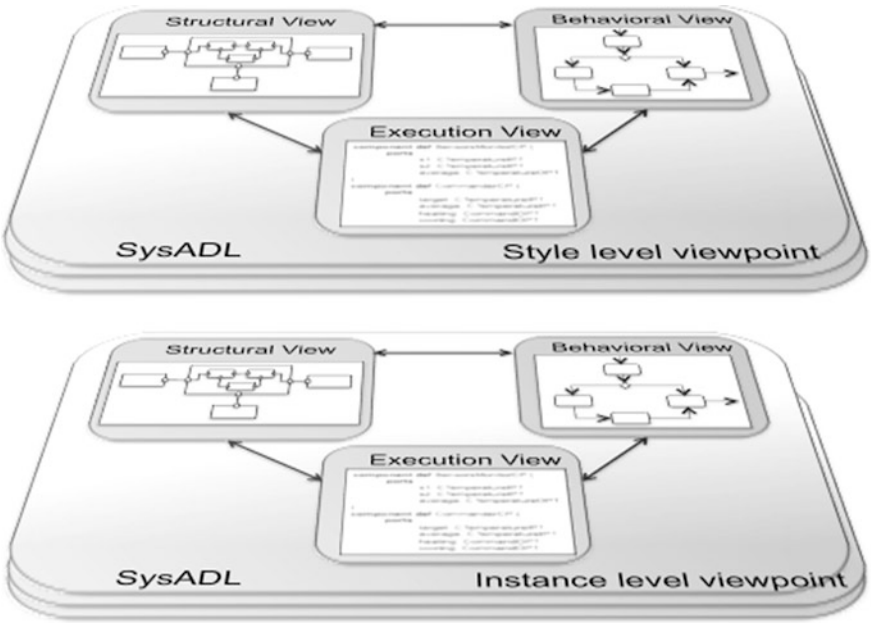


Fig. 2.14 Architectural views from viewpoints

The behavioral view is concerned with the behavior of the architecture, specifying sequential and concurrent activities by making use of the activity diagram (*act*). The behavioral description also includes the specification of protocols supporting the interaction between components through connectors and the specification of actions. In the latter case, parametric diagrams are used (*par*).

The executable view describes the execution details of the architecture. It represents the code view. This viewpoint provides a superset of the OMG Action Language for Foundational UML (ALF) as part of SysADL. The use of ALF allows for a direct mapping onto the programming language level.

Note that, in addition to views, we have two abstraction levels: style level and instance level.

The *style level* is where the architecture style is specified, defining the types of components and connectors that compose the architecture, as well as the set of constraints on how they are combined. The style is used to derive instances of the architecture, following the types and constraints that are defined. For instance, the client–server style defines server and client as component types, a server–client connector type to connect servers and clients, and constraints on systems composition specifying the kinds of allowed compositions. For example, clients must not communicate among them, clients communicate only with servers, servers may not initiate the communication with a client, and may allow no more than 10 clients to be simultaneously connected to them.

The *instance level* is where the instances of the style are defined. For each style, several instances can be defined. An instance is specified by instantiating the elements that compose the style, and by satisfying the constraints. For the client–server style, a possible simple instantiation can be a Web server connected to different clients using one connector for linking each client to the server. Another instantiation can be a banking server with a dedicated connector for each client.

2.5 Summary

In this chapter, you have learnt the following:

- how to define a software architecture;
- what are the underlying concepts needed for describing a software architecture: viewpoints and views;
- what is the architectural framework provided by SysADL in terms of viewpoints and abstraction levels.

Further Reading

1. Bass, L., Clements, P., Kazman, R.: Software Architectures in Practice, 2nd edn. Addison Wesley, Reading (2003)
2. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)
3. Rozanski, N., Woods, E.: Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives. Addison-Wesley (2012)

Reference

1. www.iso-architecture.org/42010/index.html

Chapter 3
Eliciting Requirements of Software Architectures

In this chapter, we present the SysADL constructs for expressing requirements and decisions related to software architectures. We explain the concepts of *requirements* (the needs from stakeholders) on one hand, and *decisions* (the choices of the architect on how requirements will be satisfied by the architecture) on the other. We present, in detail, each of the requirement constructs and illustrate their use by applying them to our running example.

You will learn the following:

- the SysADL constructs for expressing requirements;
- the underlying concepts needed for expressing dependencies between requirements;
- the SysADL constructs for documenting architectural decisions.

3.1 Introduction

We introduced, in Chap. 1, the notion of software architecture and saw that an architecture provides the means to satisfy the needs of stakeholders. But, how to describe these needs? The answer is: *requirements specification*.

In SysADL, a requirement specification identifies a documented need that can be a required capability or a required quality of a system-of-interest. According to the concerns, it designates a requirement elicited from a customer, a provider, or other stakeholder.

SysADL provides requirement constructs and a requirements diagram for documenting the requirements of a system. In SysADL, its use is mainly to clearly

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_3

relate the architecture decisions with the requirements they have the responsibility to satisfy.

Regarding requirements, the questions are as follows:

- what are the concepts needed for specifying the requirements of a system and its related architectural decisions?
- which are the SysADL constructs provided by the requirements diagram?
- how to apply these constructs for specifying requirements and architectural decisions to satisfy the requirements?

We will address each one of these questions in the sequel.

3.2 The Concept of Requirement

A ***requirement*** specifies a capability or a quality to be satisfied. In terms of architectural requirements, they may be related to the structure, the behavior or the properties of a software architecture must (in the case of mandatory requirements) or should (in the case of desirable, but not mandatory requirements) satisfy. In this perspective, a requirement may specify a service that a system must perform or a quality of service a system must achieve. In the former case, it refers to a ***functional requirement*** and in the latter case to an ***extra-functional*** one (often also called as ***nonfunctional requirement***).

As an example, you can elicit the following functional requirement for the *RTC* System from the concern of both the costumer and provider: the *RTC* system must be capable of maintaining the temperature in the room. As example of extra-functional requirement, we could have: the *RTC* system must consume at least 20 % of less energy than a manual system.

3.3 Requirement Constructs

In SysADL, we apply the ***requirement construct*** to specify a requirement and the ***rationale construct*** to document a rationale for that requirement. A requirement is a documented need related to a concern of a stakeholder and the rationale is the reason for that need.

Requirement. A requirement is directly specified using the "requirement" stereotype with a name and two tags, as shown in Fig. 3.1: id gives the unique identification of the requirement and *text* gives the expression of the requirement in a natural, semiformal, or formal language. We will use English as the language for specifying requirements from stakeholders. As a convention, we use the suffix FR for functional requirements, and NFR for nonfunctional requirements.

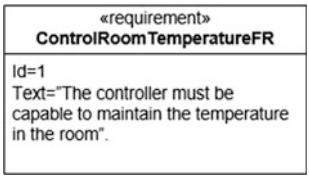


Fig. 3.1 Example of requirement

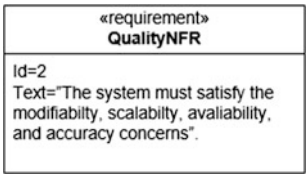


Fig. 3.2 Example of nonfunctional requirement

For instance, in architectural terms, we could have the following functional requirement: the controller component must be capable to maintain the temperature set by the user in the room, shown in Fig. 3.1, defined by the *ControlRoomTemperatureFR* requirement.

In terms of nonfunctional requirements, as sketched in Fig. 3.2, we could define that the system must satisfy some quality concerns such as modifiability, scalability, availability, and accuracy.

Rationale. A requirement should be justified. A rationale documents the justification for requirements, as well as, for decisions. The "rationale" stereotype makes it possible to attach a rationale to a requirement or to a dependence related to a requirement or to an architectural decision. For example, a rationale can be attached to a requirement to justify the decision. Figure 3.3 shows the notation for specifying the rationale to the *VerifyPresenceFR* requirement.

In addition to these stereotypes, we add in SysADL two sub-stereotypes of "requirement" for expressing "functional" and "extra-functional" requirements.

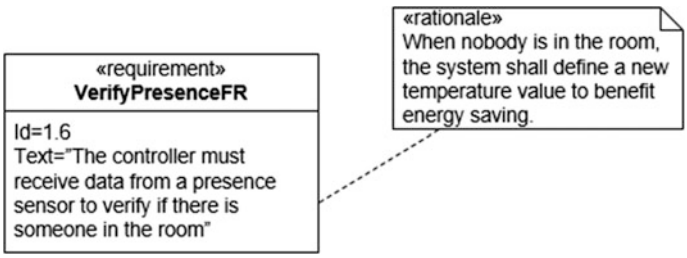


Fig. 3.3 Rationale

3.4 Dependencies Between Requirements

In SysADL, the following dependencies may be expressed between requirements: containment, derivation, satisfaction, verification.

Containment. A requirement may be simple or composite, where in the later case it contains sub-requirements. The containment relationship allows decomposing a requirement into simpler requirements. Figure 3.4 shows a containment between requirements: *RTCRequirements* is decomposed into the *ControlRoom TemperatureFR* functional requirement and the *QualityNFR* nonfunctional requirement.

Figure 3.5 shows the notation for specifying a containment relationship among requirements. The *ControlRoomTemperatureFR* requirement is decomposed into three sub-requirements: id 1.1 *MonitorTemperatureFR*, id 1.2 *DefineTemperatureFR*, and id 1.3 *CommandHeaterAndCooler*.

Figure 3.6 shows the notation for specifying a containment relationship among nonfunctional requirements. The *QualityNFR* requirement is decomposed into three sub-requirements: id 2.1 *ModifiabilityNFR*, id 2.2 *ScalabilityNFR*, id 2.3 *AvailabilityNFR, and AccuracyNFR*.

Derivation. A requirement may be primitive or derived from another. The derivation dependence enables a requirement to be derived from another requirement. Figure 3.7 shows the notation for specifying a derivation relationship between requirements: requirement *VerifyPresenceFR* is derived from *DefineRequirementFR*. It means that requirement *VerifyPresenceFR* implies *DefineRequirementFR*, therefore, if *VerifyPresenceFR* is satisfied, *DefineRequirementFR* is, by consequence, satisfied too. In the case of several derived requirements, all of them need to be satisfied.

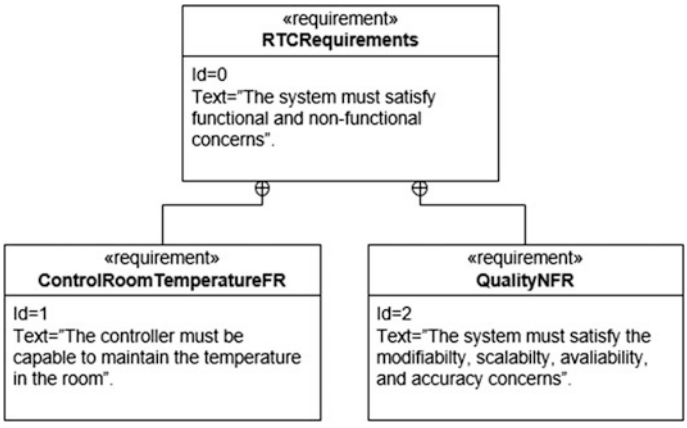


Fig. 3.4 Containment between requirements

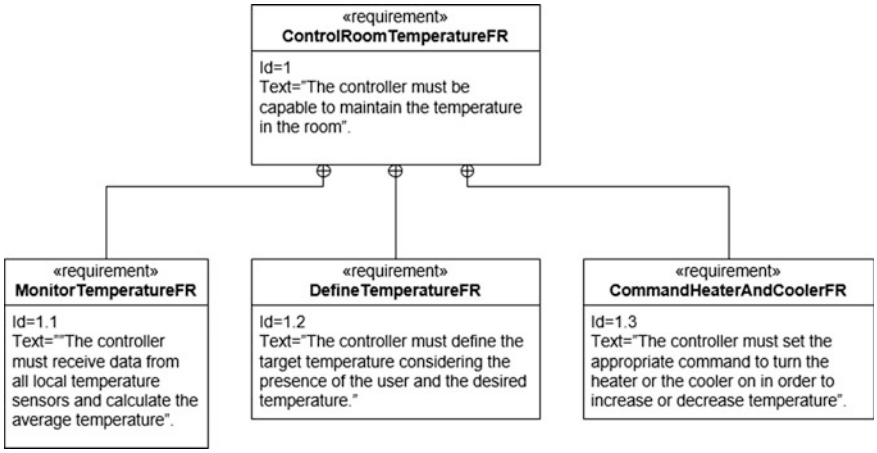


Fig. 3.5 Containment among functional requirements

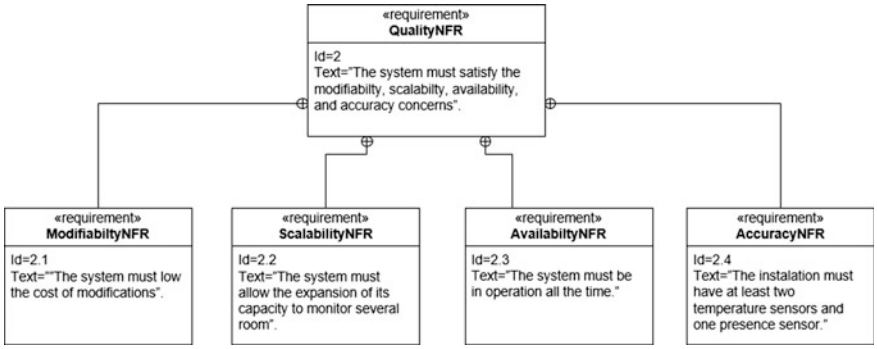


Fig. 3.6 Containment among nonfunctional requirements

Fig. 3.7 Derivation between requirements

Satisfaction. A requirement is satisfied by an architectural decision. The *satisfy* dependence describes how an architectural element satisfies one or more requirements. Figure 3.8 shows the notation for specifying the satisfaction of a requirement: *the PresenceSensorCP* component satisfies the *VerifyPresenceFR* functional

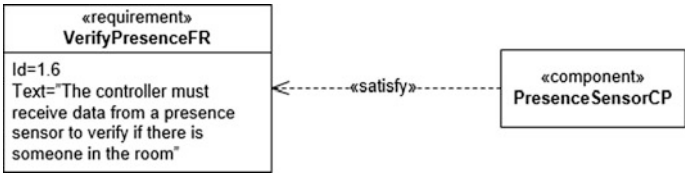


Fig. 3.8 Satisfaction between a requirement and an architectural element

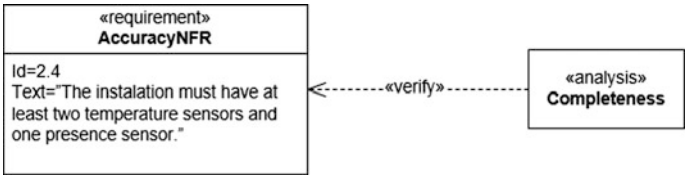


Fig. 3.9 Verification relationship

requirement. This means that the architectural decision is to give to the identified architectural element—the presence sensor—the responsibility to satisfy that requirement—to verify the presence of somebody in the room.

Verification. A requirement needs to be verified, where this verification may be carried out by different means, e.g., it could via a test case or by model checking. An "analysis" stereotype or another stereotype representing a verification technique can be used to identify verification methods, e.g., inspection, demonstration, or formal analysis. Figure 3.9 shows the notation for specifying the verification technique for a requirement: *Completeness* is used to verify *AccuracyNFR*, a nonfunctional requirement.

3.5 Requirements Diagram

The requirements specification as a whole is described using one or several requirements diagram. It presents the set of interrelated requirements using the requirement and rationale constructs and the different dependencies. Architectural decisions are represented in the requirements diagram by associating to the specified requirements, the architectural elements that satisfy these requirements.

As shown in the figure below, the header of a requirements diagram, *req*, is marked [*Requirements*] followed by the name of the diagram. The suffix *FRD* is a convention to state that this is a specification of functional requirement diagrams. Figure 3.10 shows an example of a requirements diagram.

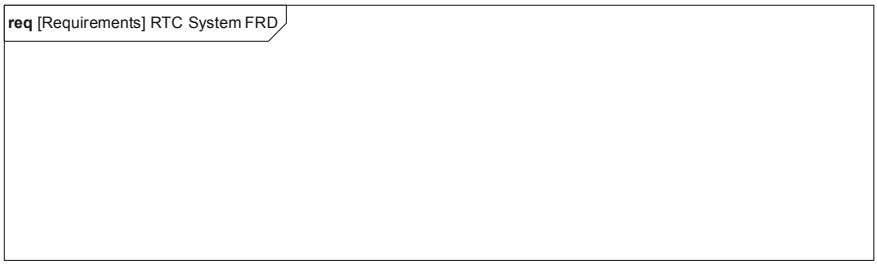


Fig. 3.10 Requirements diagram

3.6 Applying Requirement Constructs

As an example, we define the requirements diagram for the *Room Temperature Controller (RTC)* system, introduced in Chap. 2, sketched in Fig. 3.11. The *RTC* system is designed to control the temperature of a room. As depicted, it has two temperature sensors to capture the current temperature in different places of a room. The central controller receives the values from the temperature sensors, compares them with the desired temperature and turns the cooler or the heater on or off. The system has a presence sensor to detect if there is someone in the room. In the case of presence, the system operates to provide the desired temperature. Otherwise, the system operates to maintain the temperature at 22 °C.

Figure 3.12 shows the resulting requirements diagram of our running example, i.e., the RTC system. The overall functional requirement of the RTC system is *ControlRoomTemperatureFR*. This requirement is decomposed into three sub-requirements: id 1.1 *MonitorTemperatureFR*, id 1.2 *DefineTemperatureFR*, and id 1.3 *CommandHeaterAndCooler*.

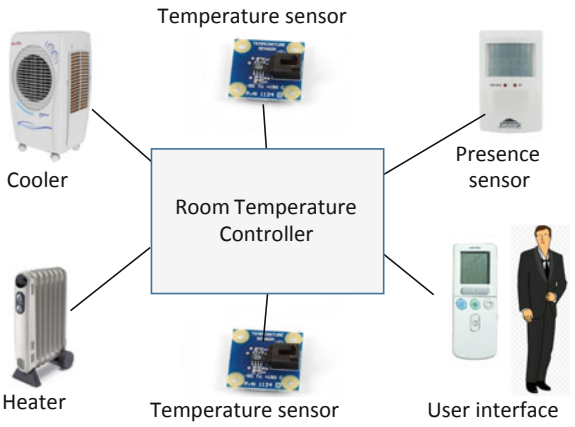


Fig. 3.11 *RTC* system overview

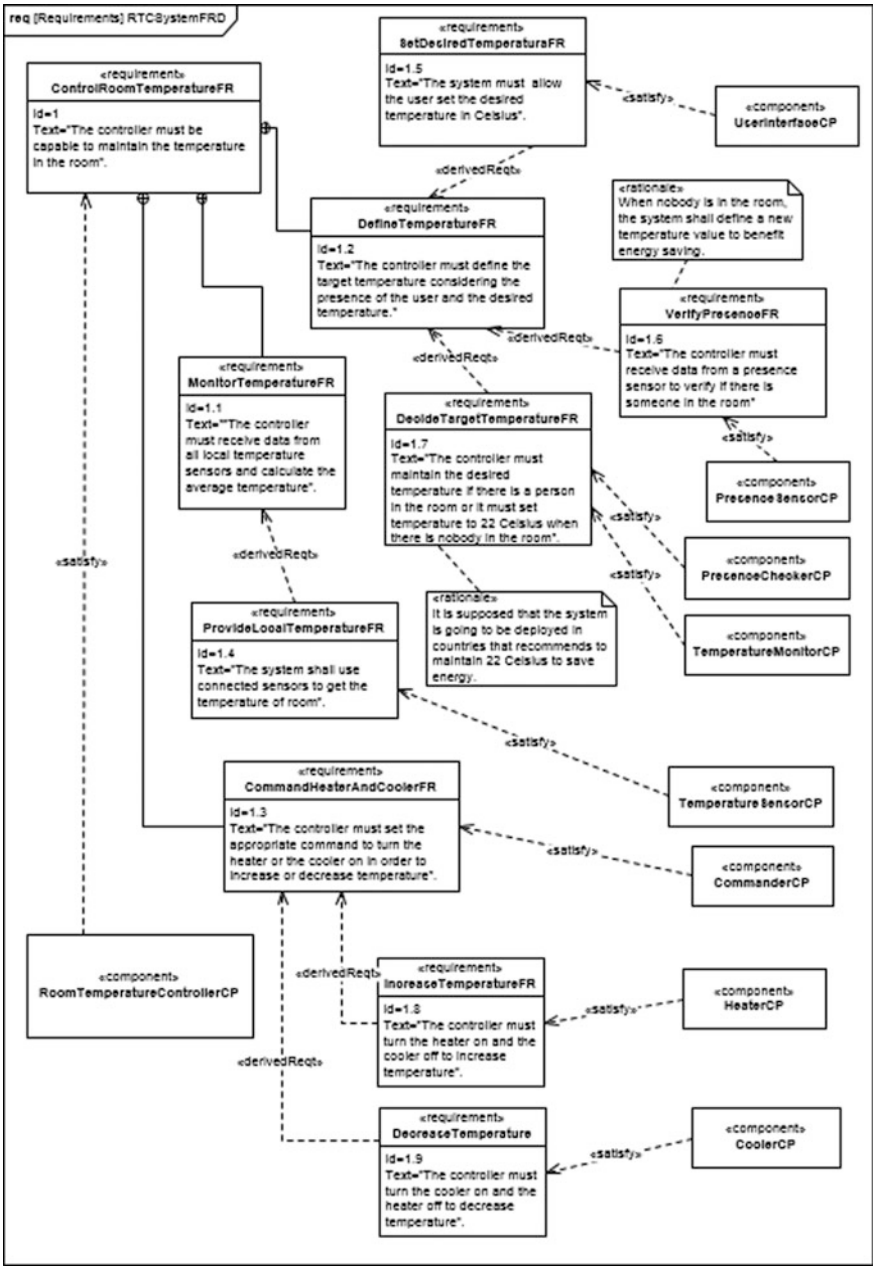


Fig. 3.12 Requirement diagram of the *RTC* system

Some of those requirements have derived requirements. The requirement id 1.1 *MonitorTemperatureFR* has one derived requirement: id 1.4 *ProvideLocal TemperatureFR*.

The requirement id 1.2 *DefineTemperatureFR* has three derived requirements: id 1.5 *SetDesiredTemperatureFR*, specifying that the user can set the temperature he/she wants; id 1.6 *VerifyPresenceFR* specifying that the presence of somebody in the room must be checked; and id 1.7 *DecideTargetTemperature* to establish the value of the temperature in case there is no one in the room. *All* of them need to be satisfied in order to imply the satisfaction of the requirement id 1.2 *DefineTemperatureFR*.

Regarding the requirement id 1.3 *CommandHeaterAndCooler*, it has two derived requirements: id 1.8 *IncreaseTemperatureFR* and id 1.9 *DecreaseTemperature*.

Once the requirements specified, the architect should reason about how to satisfy the requirements in terms of architectural decisions.

In this case, the architectural decisions are

- Component *PresenceSensorCP* will satisfy requirement id 1.6 *VerifyPresence FR*. It means that a sensor for detecting the presence of the user in the room will be part of the architecture of the *RTC* system. Note that this is an architectural decision, as this requirement could be solved in different ways, e.g., instead of having a presence sensor, the decision could be to use the agenda of the user for determining her/his presence.
- Component *UserInterfaceCP* will satisfy requirement id 1.5 *SetDesired TemperatureFR*. It means that the architectural decision is to provide a user interface to the *RTC* system enabling the user to set the temperature that is to be maintained.
- Component *PresenceCheckerCP* will satisfy requirement id 1.2 *Define TemperatureFR*. Note that in this case, the architectural decision is to encapsulate the components *UserInterfaceCP* and *PresenceCheckerCP* as subcomponents of the component *PresenceCheckerCP* that satisfies the base requirement of the two derivations.
- Component *TemperatureSensorCP* will satisfy requirement id 1.4 *ProvideLocal TemperatureFR*. It means that the architectural decision is to provide a temperature sensor to measure the temperature in the room.
- Component *TemperatureMonitorCP* will satisfy requirement id 1.7 *Decide TargetTemperatureFR* and therefore it will also satisfy id 1.1 *MonitorTemperatureFR*. The architectural decision is to provide a component to monitor deviations of the desired temperature set by the user, having *TemperatureSensorCP* as a sub-component.
- Component *HeaterCP* will satisfy requirement id 1.8 *IncreaseTemperatureFR*. The architectural decision is to command a heater to heat the room.

- Component *CoolerCP* will satisfy requirement id 1.9 *DecreaseTemperatureFR*. The architectural decision is to command a cooler to cool the room.
- Component *CommandCP* will satisfy requirement id 1.3 *CommandHeaterAnd CoolerFR*. The architectural decision is to encapsulate the components *HeaterCP* and *CoolerCP* as subcomponents of the component *CommandCP* that satisfies the base requirement of the two derivations.

These architectural decisions pave the way to design the software architecture of the *RTC* system that we will see in the next chapters.

3.7 Summary

In this chapter, you have learned

- how to apply the SysADL constructs for expressing requirements;
- how to express the dependencies between requirements;
- how to document architectural decisions.

Further Reading

1. Friedenthal, S., Moore, A.: A Practical Guide to SysML: The Systems Modeling Language, 3rd edn. The MK/OMG Press (2014)
2. Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications (2009)
3. Pohl, K.: Requirements Engineering: Fundamentals, Principles, and Techniques. Springer (2010)

Chapter 4
Specifying the Structure of Software Architectures

In this chapter, we present the structural viewpoint provided by SysADL. We explain the SysADL constructs that enable the description of structural views. We also present, in details, the concepts underlying each of these constructs and how each one is applied to specify the different views comprising the architectural structure. Finally, we illustrate each definition and how to use it with our running example.

You will learn:

- the architectural constructs to express structure: *components*, *connectors*, and *configurations*;
- the auxiliary concepts: *port* for defining interfaces of components and connectors, and *value type* for data manipulation; and
- the diagrams to represent these constructs.

4.1 Introduction

In Chap. 2, we presented the notion of viewpoint and introduced the three viewpoints provided by SysADL:

- the structural viewpoint,
- the behavioral viewpoint, and
- the executable viewpoint.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_4

The ***structural viewpoint*** enables the description of the different views of the structure of a software architecture. The structure of a software architecture refers to the way in which the architectural elements are organized to achieve the required functionalities and system qualities. These structural elements are:

- components expressing the locus of computation,
- connectors expressing the locus of communication, and
- their composition in configurations.

From the structural viewpoint, the questions are:

- what are the concepts required for specifying the structure of a software architecture?
- what are the SysADL constructs provided by the structural viewpoint?
- how to apply these constructs for describing different views of the architectural structure?
- how the diagrams are used to express these views?

We will address each one of these questions in the sequel.

4.2 Structural Viewpoint and Views

4.2.1 Structural Viewpoint

The structural viewpoint provides the constructs to describe the different views of the structure of a software architecture. SysADL structural viewpoint defines the following structural constructs: *components*, *connectors*, and *configurations*. It also defines auxiliary concepts for the structural description: *ports* and *value types*. In Chap. 1, we introduced the notion of components, connectors, and configurations. In this chapter, we discuss how to define their structure.

Each of these architectural elements is itself structured. The structure of a component is defined in terms of the specification of its *ports*. As components, connectors also have ports. The structure of an architectural configuration describes how components are linked together via connectors for supporting the required system qualities. For instance, in the architecture description of the *RTC* system (our running example), the structural viewpoint provides the SysADL constructs to declare components, connectors, and configurations.

As you saw in the first chapter, the *RTC* System (sketched in Fig. 4.1) has two temperature sensors to capture the current temperature in different places of a room. A controller receives the values from the temperature sensors, compares them with the desired temperature and turns the cooler or the heater on or off. The system also has a presence sensor to detect if there is someone in the room. In case of presence, the system operates to provide the desired temperature. Otherwise, the system operates to maintain 22 °C.

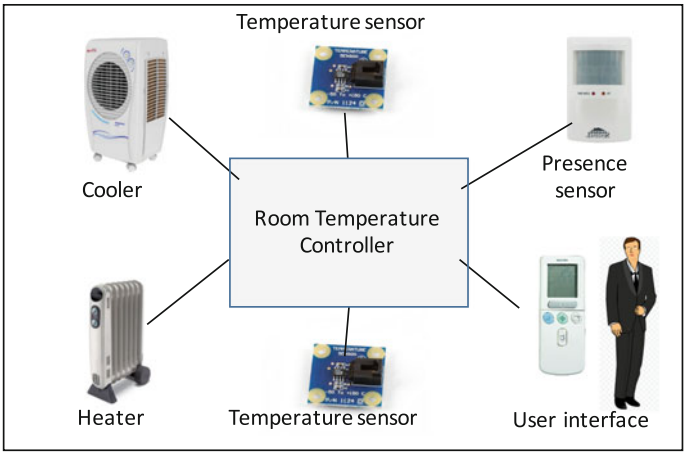


Fig. 4.1 *RTC* system overview

By analyzing the structure of this system, we can identify different components: two temperature sensors, a presence sensor, a user interface, a cooler, a heater, and a controller. We can also identify that connectors are needed for linking the different sensors and the user interface to the controller and the controller to the cooler and the heater. The configuration identifies the topology of these components and their binding connectors. In this case, it is a star topology with the controller in the middle.

With respect to data, components process data and connectors transmit data among components.

The structural description of the software architecture includes the specification of the definition and use of components, connectors, and configurations and data handled by them. The *definition* of an element is the specification of its type. The *use* of an element is an instantiation of the type. All elements need to be defined before their use.

The different SysADL constructs are applied in the definition of structural views by using block definition diagrams (bdd) and internal block diagrams (ibd).

4.2.2 Structural Views

Using the SysADL constructs of the structural viewpoint, we can define different structural views. A *structural view* shows a subset of the structural description of a software architecture. One or several views describe the definitions of components, connectors, and configurations (with associated ports and data) that are used in the definition of the architectural structure. One or several views describe how the defined components, connectors, and configurations are used and composed to

define the architectural structure to achieve the required system functionalities and quality properties.

For instance, in the specification of the architectural structure of the *RTC* System, we specify each component (the different sensors, the controller, and the actuators) and each connector linking a sensor to the controller, and the controller to an actuator. Each component and connector is defined in terms of *ports*, which are defined in terms of *data flows*. The configuration is defined by linking components via the proper connectors to allow the controller to receive data from sensors for monitoring the temperature in the room, as well as to allow the controller to transmit commands to the cooler and heater in order to achieve the desired temperature.

Views are created to organize the structural description of the architecture using block definition diagrams (bdd) and internal block diagrams (ibd) according to the concerns of different stakeholders. In the case of the RTC System, a view could show the sensor definitions, another view could show the controller definition, and another view could show the actuator definitions. Other views could show the definitions of port types and value types. Yet another view could show the definition of configurations.

4.3 Structural Constructs

4.3.1 Component

In SysADL, we apply the component construct to specify the structure of components in terms of ports. A component can have input ports (called *in* ports) to obtain required data, and output ports (called *out* ports) to produce provided data. A component performs services by consuming data in its *in* ports and providing the results in its *out* ports. A component can have several *in* and *out* ports.

Definition notation. We should first define component types and then use components of the defined type.

A component definition is represented using a rectangle, with a stereotype «component» and its name. As a convention, we use *CP* as a suffix to the name of a component type. Figure 4.2 shows the definition of the *SensorsMonitorCP* component.

We represent a component located in the boundary using the same notation of the other components but filled in gray color (as shown in Fig. 4.3).

Fig. 4.2 Component definition of *SensorsMonitorCP*

Fig. 4.3 Boundary component definition

Fig. 4.4 Use of the *TemperatureSensorCP*

The use of a component has the same visual notation, but it has a name followed by a colon and the type name to identify the instance. Figure 4.4 illustrates two uses of the *TemperatureSensorCP*.

4.3.2 Ports

A port is an interaction point between a component and other architectural elements. It represents how the data flow to a component (*in* ports) from another component (*out* ports) or vice versa.

We use a value type to specify the data that flows through a port. Note that ports do not provide operations, but only data flows. A port can be composed by other ports. In that case, it is a called a *composite port*. An example of port is the one that provides the value of the temperature by a sensor component.

Definition notation. A port type is defined using a «port» stereotype with its name and its flow properties. The flow property definition includes the value type to specify the kind of data that flows through the port and the direction (in or out) of the flow.

A port is represented as a part of a component using a small square with its name, a colon, and the name of the port type. The data flow is represented using an arrow inside the port square indicating its direction.

We show in Fig. 4.5a the definition of the *FTemperatureOPT* port type, an out port that provides temperature values in Fahrenheit. In Fig. 4.5b we show the use of this port in the definition of the *TemperatureSensorCP* component type. As a convention, we name port types using the following suffixes: *OPT* to output ports and *IPT* to input ports.

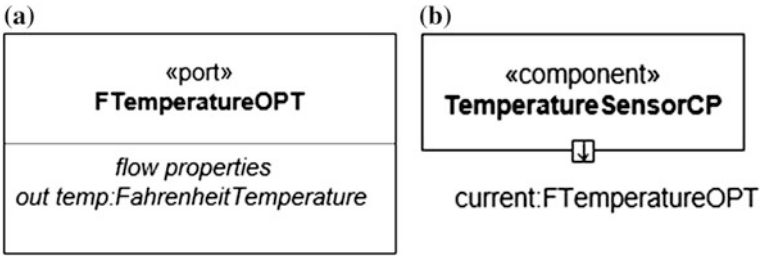


Fig. 4.5 **a** Port type definition. **b** Port use in a component definition

The definition of a port type should include its flow properties in a specific compartment to specify its direction: in for information flowing into the port or out for information flowing out of the port. A flow property should have a name and a value type. We can use a primitive type, i.e. *Boolean*, *Real*, *Integer*, or *String*, or a user-defined value type.

In Fig. 4.6, we have two examples of port definitions. The *CTemperatureOPT* and *CPTemperatureIPT* are both port types with data of the same value type flowing in opposite directions.

Use notation. We should first define port types and then create ports of the defined type. A port is represented in a component as a small square in its border as shown

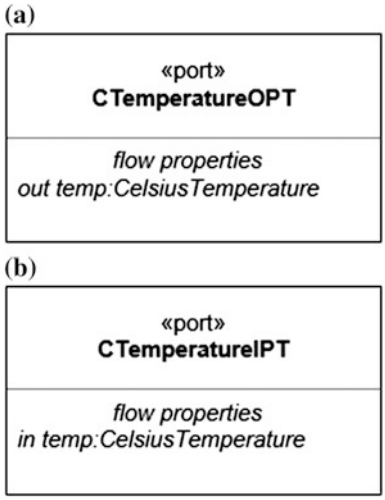


Fig. 4.6 **a** Port type of out direction. **b** Port type of in direction

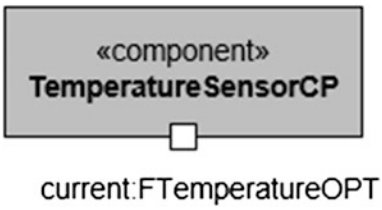


Fig. 4.7 Port *current* used in the definition of *TemperatureSensorCP*

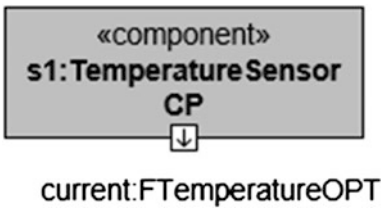


Fig. 4.8 Out port *current* used in the definition of *TemperatureSensorCP*

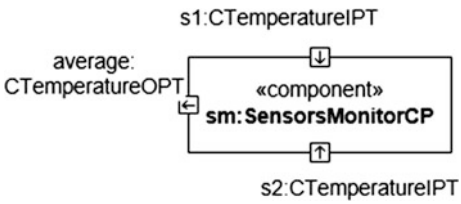


Fig. 4.9 Component with three ports of different port types

in Fig. 4.7. The temperature sensor provides temperature values in Fahrenheit in its *current* port (which is an instance of the *FTemperatureOPT* port type).

We use a port in a component to specify what type of data is required (in) or provided (out) by the component. *In* ports require data flowing into the component. *Out* ports provide data flowing out of the component. We can represent the data flow in SysADL using an arrow inside the port square indicating its direction. For instance, a temperature sensor has an out port providing temperature values flowing out of the *current* port as shown now in Fig. 4.8.

A component may have several ports of different types. Figure 4.9 illustrates a component with three ports (*s1*, *s2*, and *average*) of two different types, i.e., *CTemperatureOPT* and *CtemperatureIPT*.

A composite port may contain several ports of different types in its structure. Composite ports are ports themselves and therefore are used in the same way as any simple port. As a convention, we use a *CPT* suffix to the name of a composite port.

We define a composite port type by specifying it as a composition of ports. We need to specify the flow properties type of the internal ports. As in our running example we do not have composite ports, let us use another one: a client–server system, in which the client requests services from the server and receive the answer.

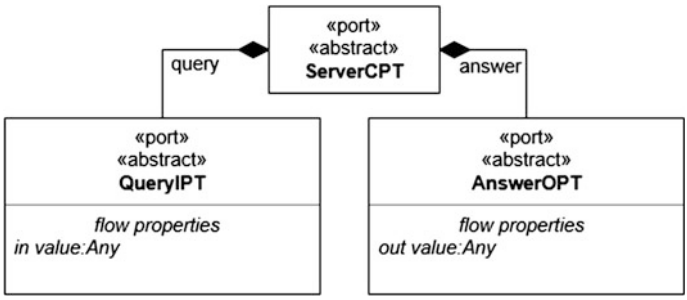


Fig. 4.10 Definition of a composite port and internal ports

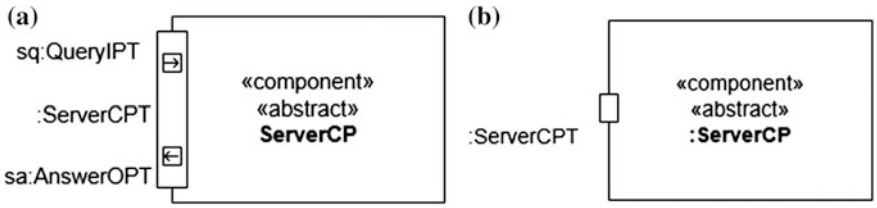


Fig. 4.11 **a** Composite port and its internal ports. **b** Composite port as a whole

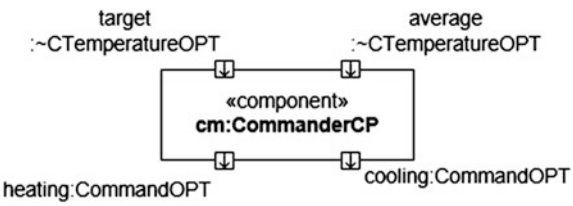
In this example, as shown in Fig. 4.10, the *ServerCPT* is defined as a composite port. The *ServerCPT* port type is defined as a composition of two simple ports (*QueryIPT* and *AnswerOPT*), where each port can receive and send values of type *Any*.

In Fig. 4.11a, we show the use of this defined composite port in the definition of the *ServerCP* component type. Figure 4.11b shows the use of this port in a component use.

A *conjugate port* is a port of the same type of an originally defined port, but with a flow in opposite direction. We use a conjugate port to simplify the definition of ports since we do not need to create new port definitions for expressing ports of opposite directions. Conjugate ports are indicated by a tilde '∼' before the type name. *Composite ports* can also be conjugated.

We illustrate, in Fig. 4.12, the notation of conjugated ports. In the *cm* component, we use two ports, *target* and *average*, both of the type of the conjugate

Fig. 4.12 Definition of ports with conjugates



CTemperatureOPT. This means that they are the reverse of the defined out port *CTemperatureOPT*, i.e., the equivalent of *CTemperatureIPT* previously shown in Fig. 4.6.

4.3.3 Value Types

Using value types, we can define the different data types to be used in the architecture description. In SysADL, there are primitive data types (*Real*, *Integer*, *Boolean*, *String*), scalar data types (enumerations), and user-defined value types. Figure 4.13 shows a definition of a user-defined data type (a), and an enumeration (b). *Commands* are structured data type composed of two uses of the Command enumeration (b) that has the values *On* and *Off*.

Note that user-defined value types may have an intensity, a dimension, and a unit. When defined with no dimension and unit, it is simply declared as a data type with the «dataType» stereotype. The dimension is a measurable extent, e.g., temperature, force, power, mass. In turn, a unit refers to the measurement of a dimension, e.g., the unit for temperature could be Fahrenheit, for power could be Watts.

We show in Fig. 4.14 temperature as an example of a value type that we can define to represent the value of temperature (dimension) in Fahrenheit (unit) or Celsius (unit).

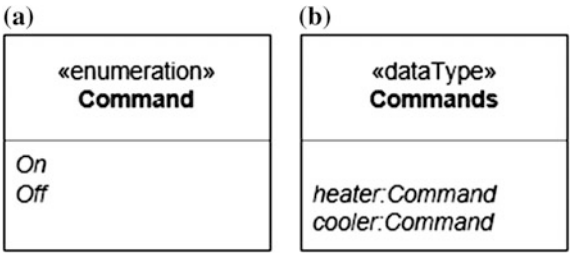


Fig. 4.13 **a** Pure data types. **b** Enumeration

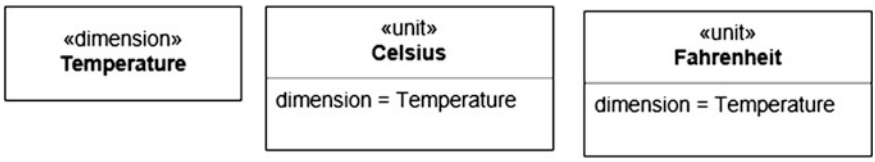


Fig. 4.14 Defining dimension and unit for value types

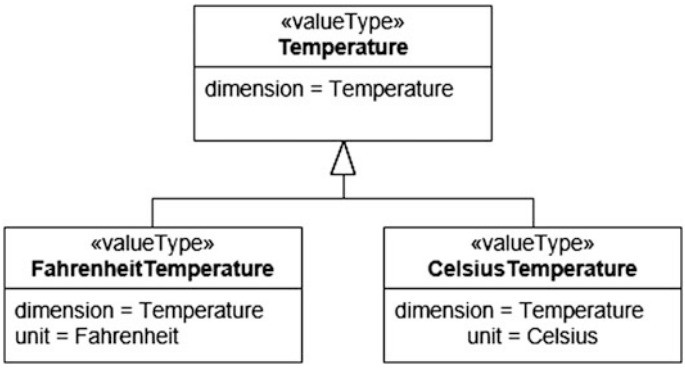


Fig. 4.15 Value type

Note that we define dimensions and units using the «dimension» and «unit» stereotypes, respectively. The definition of a unit requires the associated dimension.
Use notation. We can use value types in the definition of flow properties of ports and connectors. For example, in the definition of the *FTemperatureOPT* port type previously shown in Fig. 4.5a, a value type is used in the out port that provides temperature values in Fahrenheit (the *FahrenheitTemperature* value type, shown in Fig. 4.14).

To define the type of data we use the «valueType» stereotype. The example in Fig. 4.15 shows a definition of a *Temperature* supertype and two subtypes of specific units types to the same dimension: Celsius and Fahrenheit.

4.3.4 Components with Ports Typed by Value Types

As we saw, components have ports where these ports are typed by flows of defined value types. To illustrate, let us define the component types identified in the case study for the *Room Temperature Controller* (*RTC*) System, sketched in Fig. 4.1. Recall that the RTC system will be architected to control the temperature of a room. As depicted, it has two temperature sensors to capture the current temperature in different places of a room. The central controller receives the values from the temperature sensors, compares them with the desired temperature and turns the cooler or the heater on or off. The system has a presence sensor to detect if there is someone in the room. In case of presence, the system operates to provide the desired temperature. Otherwise, the system operates to maintain 22 °C.

We will define all components and ports of the *RTC* System in block definition diagrams (*bdd*).

The *RTC* System has the following boundary component types with their respective ports:

- A temperature sensor component type, named *TemperatureSensorCP*, to provide the current temperature value in the room, is shown in Fig. 4.16a. It has an out port—*current*—that provides the temperature value. The port type is *FTemperatureOPT*.
- A presence sensor component type, named *PresenceSensorCP*, to detect if someone is in the room, is shown in Fig. 4.16b. It has an out port—*detected*—that provides a value indicating a presence in the room. The port type is *PresenceOPT*.
- A user interface component type, named *UserInterfaceCP*, to allow the user to set the desired temperature value, is shown in Fig. 4.16c. It has an out port—*desired*—that sends the desired temperature value. The port type is *FTemperatureOPT*.
- A heater component type, named *HeaterCP*, to heat the room what will increase the room temperature, shown in Fig. 4.16d. It has an in port—*controller*—that receives a command to the component. The port type is *CommandIPT*.
- A cooler component type, named *CoolerCP*, to cool the room what will decrease the room temperature, is shown in Fig. 4.16e. Similarly, it has an *in* port—*controller*—that receives a command to the component. The port type is *CommandIPT*.

In addition, the *RTC* system has the following internal composite type:

- A room temperature controller component type, named *RoomTemperature ControllerCP*, to control the temperature by receiving temperature values and information about the presence of the user and by commanding the heater and cooler appropriately, is shown in Fig. 4.17. The controller has the following

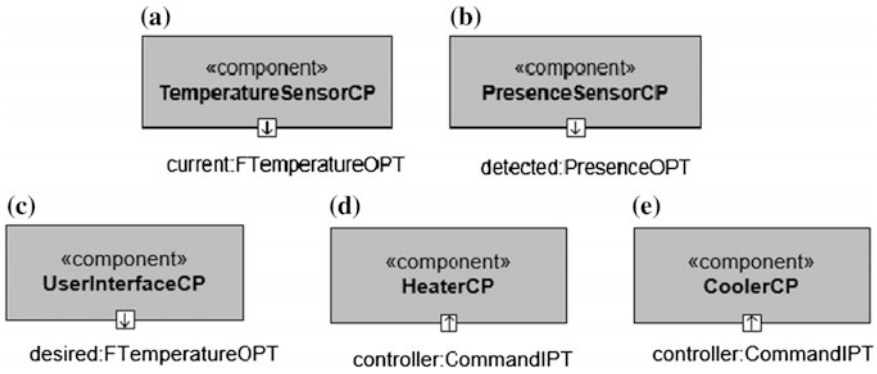


Fig. 4.16 Boundary component types of the RTC architecture

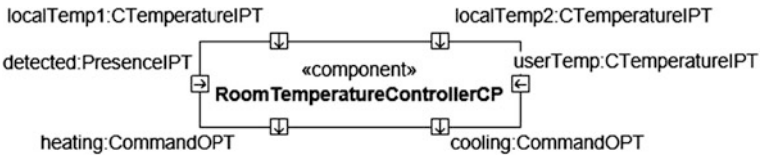


Fig. 4.17 Internal component type of the RTC architecture

ports: (i) two ports to receive a Celsius temperature value; (ii) a port to the user interface to receive the desired temperature value; (iii) a port to receive the information about the presence of a person; (iv) two ports to provide the commands to the heater and the cooler.

4.3.5 Connector

A connector refers to software element that represents the interaction between components. A connector supports communication and coordination between components. It binds ports of the connected components, allowing data to flow between them, possibly having been processed during the transmission. It has a behavior. An example is a connector that links the temperature port of a sensor to a temperature port of a controller for passing the sensed value and transforming that value from °F to °C.

Definition notation. A connector type is defined as an association block with a «connector» stereotype followed by its name and two port types specifying participants. The association specifies that it can connect any component that has ports of the same type of the participant ports. The association label indicates the type of the information that flows through the connector from one port to another.

Figure 4.18a shows the definition of the *CTemperatureCN* connector type (as a convention, we use a *CN* suffix to the name of the connector type), showing the participants and the type of information (*Temperature*) flowing from the left to the right port. Both ports have flow properties of *CelsiusTemperature* type that is a kind of *Temperature*. The use of the connector is illustrated in Fig. 4.18b where an *uc* connector of the *CTemperatureCN* type connects the *c* and *ui* components, using their *source* and *destination* ports.

Note that in this example, the *CTemperatureCN* connector type defines the association between two ports types: *CTemperatureOPT*, represented by its conjugated ∼*CTemperatureOPT* and *CTemperatureIPT*, represented by its conjugated ∼*CTemperatureIPT*.

The type of data flowing from the first port to the second is *Temperature*. Both ports have flow properties of type *CelsiusTemperature* that is a value type of *Temperature*.

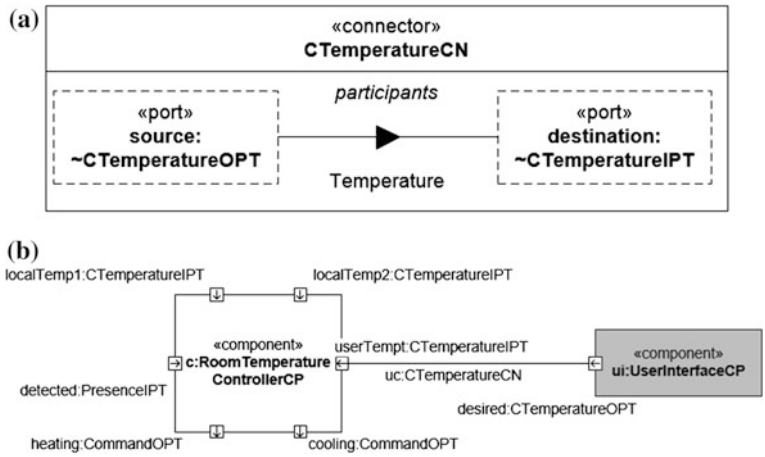


Fig. 4.18 **a** Connector definition. **b** Connector use

Use notation. We can use a connector to represent the communication between the components, as shown in Fig. 4.19a. When we use a connector we should provide its name and the name of its connector type. Thereby, we should first define connector types and then create connectors of the defined type.

In that case, shown in Fig. 4.19b, we have two components named *iu* and *c* with a *uc* connector of the *CTemperatureCN* type which connects the components *ui* and *c*, using theirs ports *desired* and *userTemp.* The data flows from the *ui* to the *c* component.

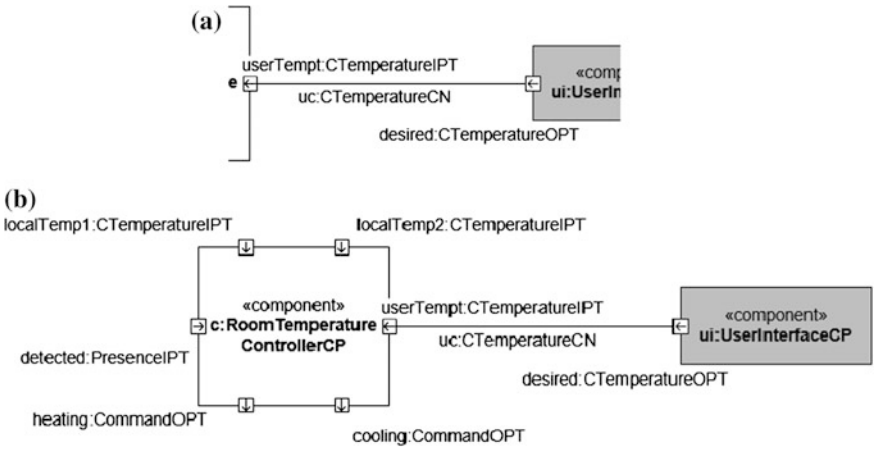


Fig. 4.19 Use of a connector type of the *RTC* architecture

Connectors may be simple (as expressed above) or composite. A composite connector is used to connect components that have composite ports.

The connectors of composite connect must be defined and needs to be compatible with ports of a composite port they connect with. Before using a composite connector, we need to define it by specifying its constituent connectors. We define each connector and use a composition to specify the composite connector, as shown in Fig. 4.20. The *ClientServerCN* connector type is defined as a composition of two connectors: *ClientServerQueryCN* and *ClientServerAnswerCN*.

Figure 4.21 shows an example of composite connector—*ClientServerCN*—that connects the composite ports—*ClientCPT* and *ServerCPT*.

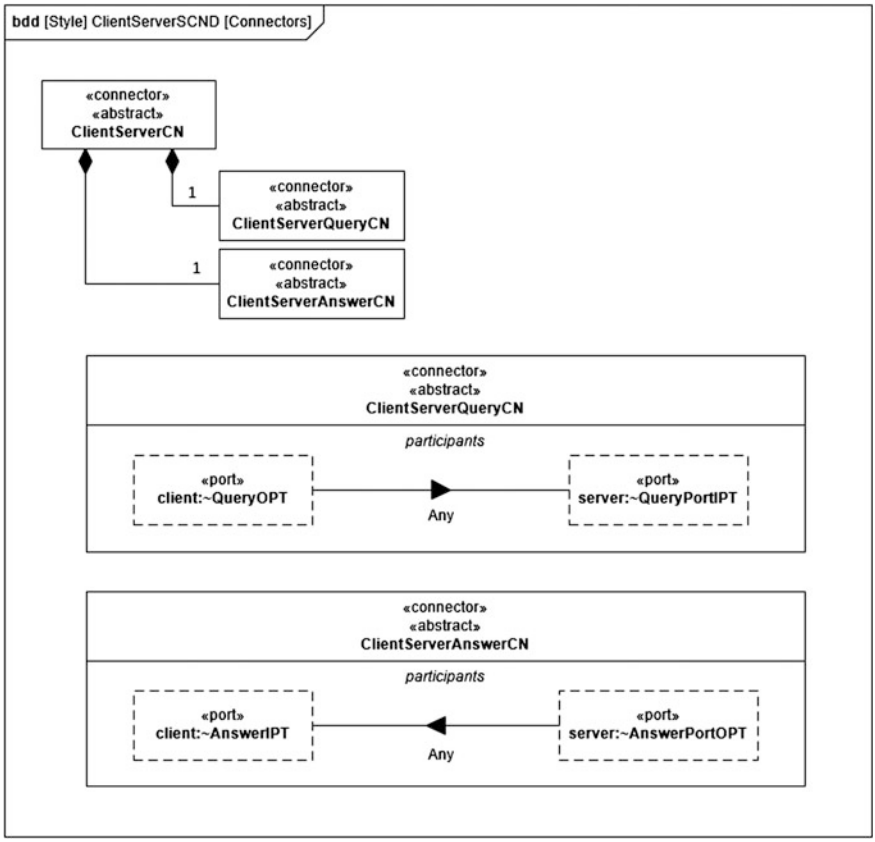


Fig. 4.20 Definition of a composite connector

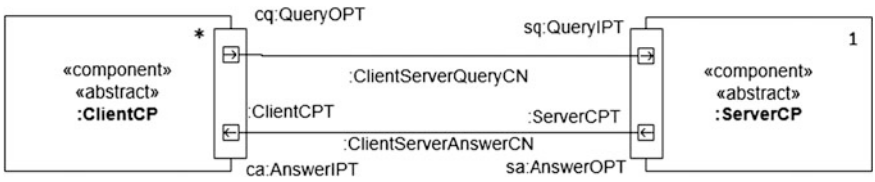


Fig. 4.21 Use of a composite connector

4.3.6 Configuration

A configuration is an architectural concept that refers to the structural organization of the architecture. It describes how components are connected via connectors. A configuration can represent the structure of a composite component or the overall software architecture of a system. It uses the defined components and connectors.

In Fig. 4.22 we show the definition of a configuration for our *RTC* example. The main component in the *RTC* System is rtc:*RoomTemperatureControllerCP*. It is connected to two *TemperatureSensorCP* components (s1 and s2) that inform the temperature in Fahrenheit values via their *current:FTemperatureOPT* out ports. The *s3:PresenceSensorCP* informs a *Boolean* value in the *detected:PresenceOPT*

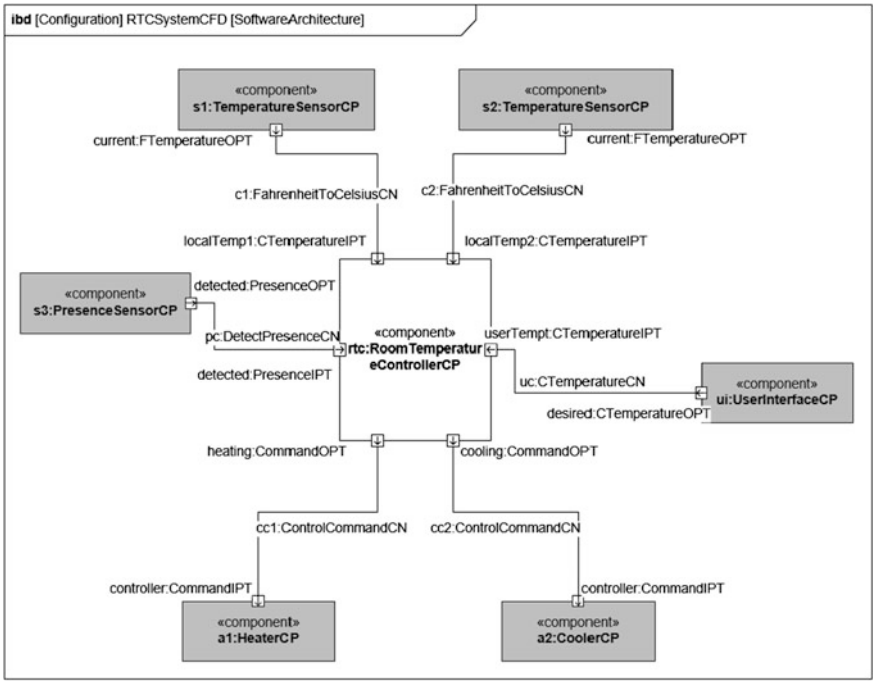


Fig. 4.22 The software architecture configuration of *RTC* System

port. *ui:UserInterfaceCP* is a component that informs the user-desired temperature value in Celsius. The *a2:CoolerCP* and *a1:HeaterCP* components receive the command values in their respective ports. All elements in this configuration are instances of the previously defined elements. Six component types are used: *PresenceSensorCP, TemperatureSensorCP, HeaterCP, CoolerCP, UserInterface CP*, and *RoomTemperatureControllerCP*. Moreover, the components connected to *RoomTemperatureControllerCP* are boundary components.

Figure 4.22 also illustrates the connectors that link each component to the central controller. Most connectors are simple links that transmit the data from an out port to an in port. The exception in this case is the *FahrenheitToCelsiusCN* connector that has the responsibility of converting values from Fahrenheit to Celsius units.

4.3.7 Composite Components

We use a composite component in the same way of a component. We should provide its name and the name of its previously defined composite component type. For instance, Fig. 4.23 shows the *rtc* composite component of the *RTC* system which is of the *RTCRoomTemperatureControllerCP* composite component type.

Figure 4.24 shows the definition of the *RoomTemperatureControllerCP* composite component. It has three internal components, *pc*, *sm*, and *cm* represented in the composition relationship. The names used in the compositions are the same names of the components used in the configuration.

Let us now show its internal structure in terms of a configuration. We can define the configuration of a composite component using an internal block diagram (*ibd*) as shown in Fig. 4.25. We use three components: *pc*, *cm* and *sm*. Connector *target* connects *pc* to *cm* and connector *average* connects *sm* to *cm.*

Note that the ports of the internal components are not visible to the external world. When a port needs to be visible, we link it to a proxy port via a binding connector. Therefore, a proxy port specifies the features of the internal ports that are visible through external connectors. We use a binding connector to link an internal port to a proxy port.

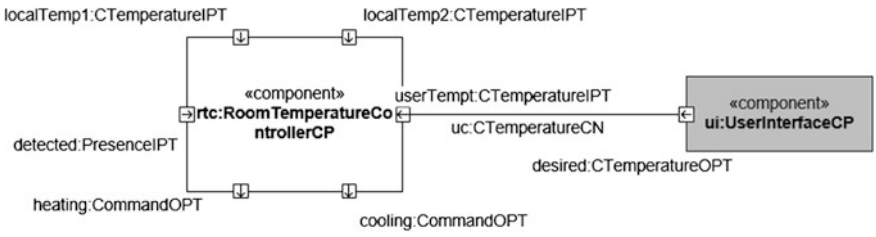


Fig. 4.23 Use of composite component *rtc* and simple component *ui*

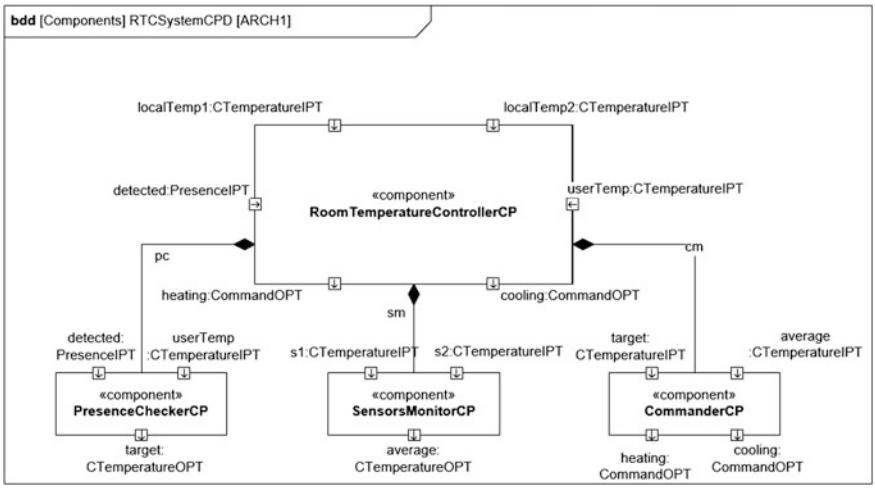


Fig. 4.24 Definition of composite component *rtc*

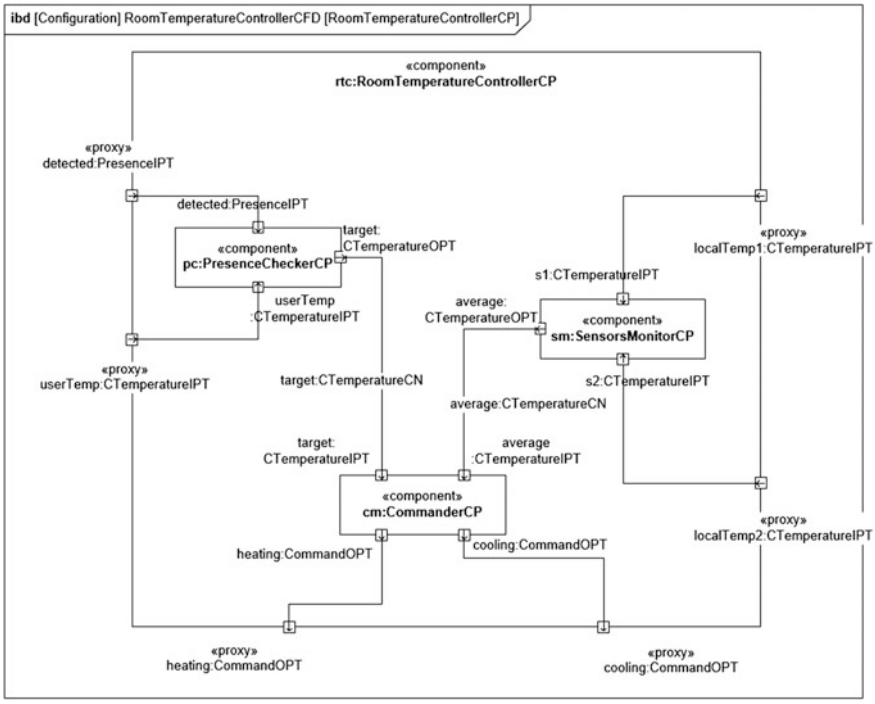


Fig. 4.25 Internal structure of the composite component *rtc*

4.4 Diagrams for Structural Views

Structural views are organized in terms of diagrams. The structural definitions of components and connectors are expressed using block definition diagrams (*bdd*). Configurations are defined with internal block diagrams (*ibd*). The block definition diagram is applied to specify the types of components and connectors as well as the types of data and ports. The internal block diagram is applied to specify the configuration in terms of components and connectors and how they are combined together. Note that while architecture types are defined using *bdd* and architecture configurations are represented using *ibd*.

All component and connectors as well as port and data types must be defined in a *bdd*. In turn, an *ibd* defines a configuration of component and connectors. It contains instances of the defined architectural elements linked together to express the software structure.

4.4.1 Block Definition Diagrams

We define architecture types using block definition diagrams (*bdd*). We define a *bdd* specifying its header and its contents, as shown in Fig. 4.26. As a convention, the following elements are used in the header:

(i) *bdd*—indicates that the diagram is a block definition diagram;
(ii) [*Model Element Type*]—an optional item indicating the element type specified in the diagram (in Fig. 4.26 we use [Components] to indicate we are defining the components of the system);
(iii) *Diagram name*—used to uniquely identify the diagram; and
(iv) [*Element*]—used to identify the architecture or a specific element (in Fig. 4.26 [ARCH1] is used to refer to the name of an architecture.

In the content, we declare the architectural definitions.
Another convention is the suffixes to diagram names. We defined:

(i) *VLD* for value type diagrams;
(ii) *DUD* for dimension and unit diagrams;
(iii) *CPD* for component type diagrams;
(iv) *CND* for connector type diagrams; and
(v) *PTD* for port type diagrams.

It is important to note that we need to first define all elements before using it in the architectural definition.

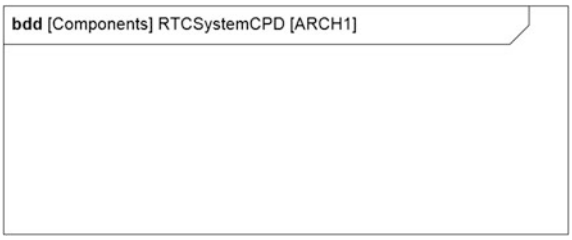


Fig. 4.26 Block definition diagrams

4.4.2 Internal Block Diagrams

We define architecture configurations using internal block diagrams (*ibd*). The *ibd* specifies instances of the defined architectural elements linked together to express the architectural structure. We define an *ibd* by specifying its header and its contents. As a convention, shown in Fig. 4.27, in the header we use the following elements:

(i) *ibd*—the diagram type;
(ii) [*Configuration*]—to inform that the diagram represents a configuration;
(iii) the *diagram name* with *CFD* as suffix to the name of the configuration; and
(iv) [*Element*]—the name of the configuration depicted by the diagram regarding a composite component or the software architecture.

In Fig. 4.28 we show an *ibd* with the definition of a configuration used in our running example. Note that composite components have an internal structure represented by a configuration, therefore expressed by an *ibd*.

As an example, we show in Fig. 4.28 the configuration of the *RoomTemperatureControllerCF* composite component. It contains three components and two connectors presented in the *ibd*. The *RoomTemperatureControllerCF* composite component contains three components and two connectors: *pc*—a *PresenceCheckerCP* component type; *sm*—a *SensorsMonitorCP* component type;

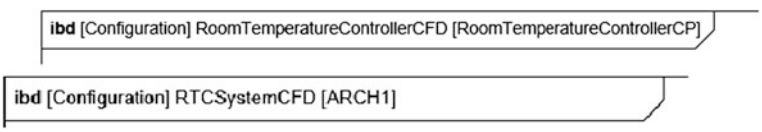


Fig. 4.27 Header of internal block diagram

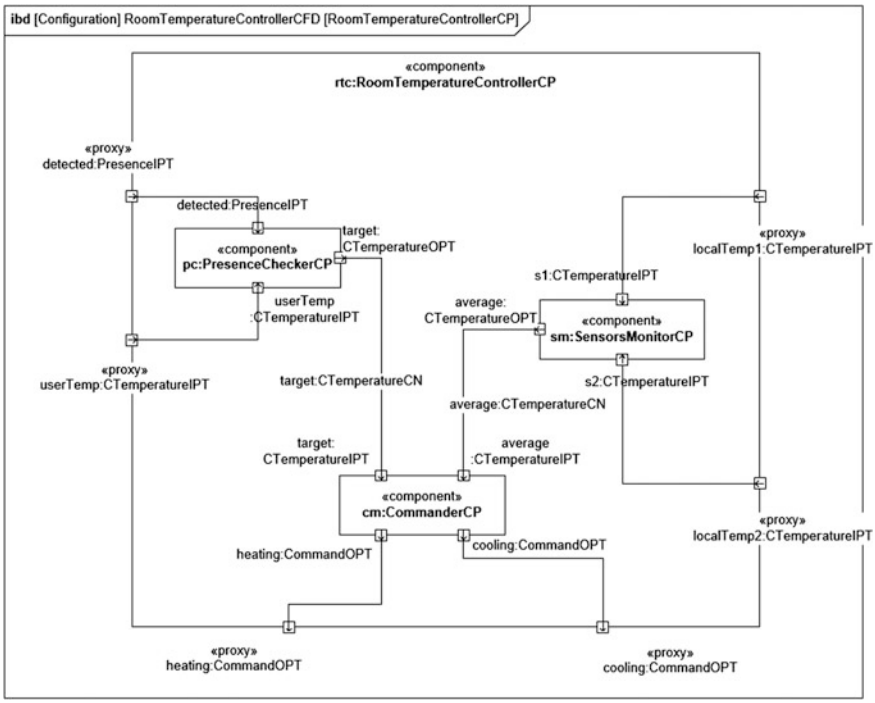


Fig. 4.28 Internal block diagram

cm—a *CommanderCP* component type; and *target* and *average* connectors: both of *CTemperatureCN* connector type.

4.5 Describing the Architecture from the Structural Viewpoint

Let us now apply the concepts and constructs we presented in this chapter to describe the structure of the software architecture of the *RTC* system, our running example.

We will define a diagram for each type of element: (i) a *bdd* for value types, (ii) a *bdd* for port types, (iii) a *bdd* for component types, (iv) a *bdd* for connector types.

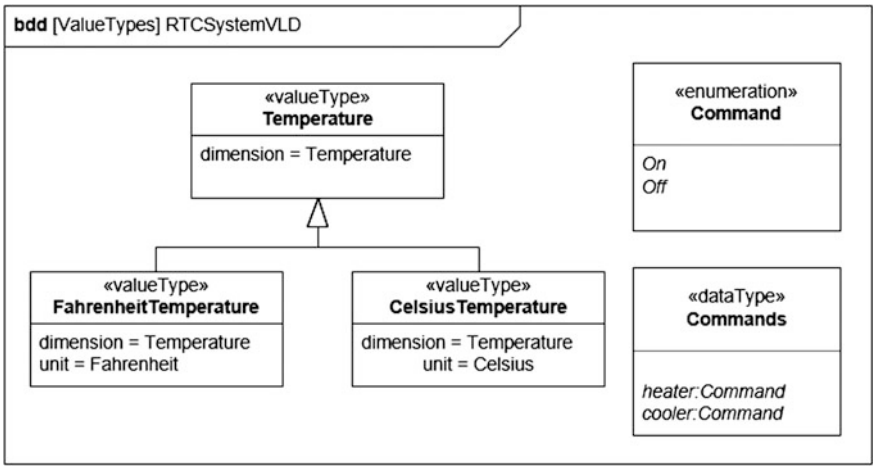


Fig. 4.29 Value types of the *RTC* architecture description

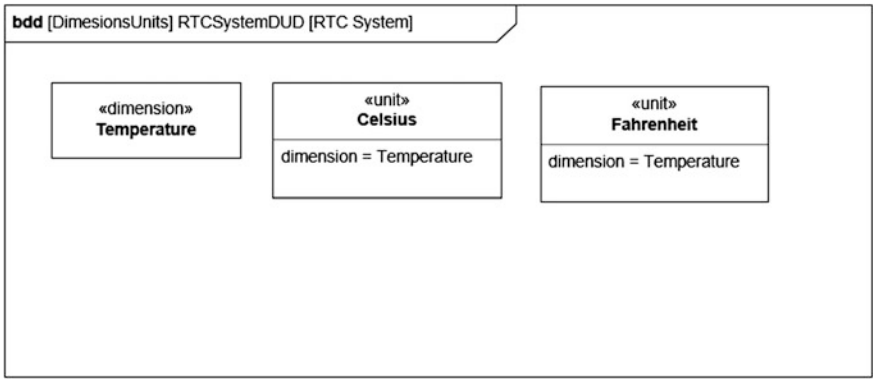


Fig. 4.30 Dimension and units for value types of the *RTC* architecture description

In Fig. 4.29, we define the following value types for the *RTC* System:

- the *Temperature* value type with two specializations: *FahrenheitTemperature* and *CelsiusTemperature.*

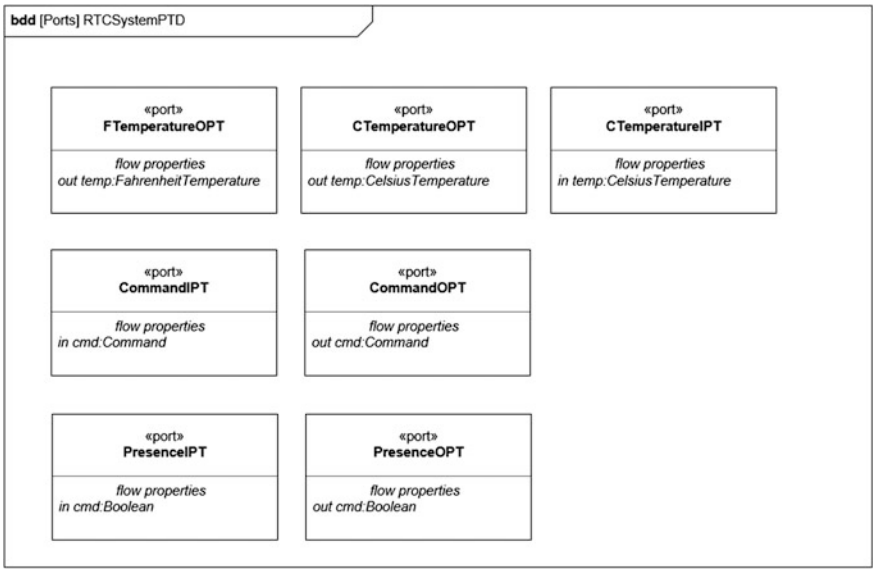


Fig. 4.31 Port types of the *RTC* architecture description

Also in Fig. 4.29, we include data types definitions and enumerations definitions:

- the *Command* value type is an enumeration of *On* and *Off* (*Command* defines a data type that is an enumeration with two value types).

These definitions are based on the following dimension type for the *RTC* System, shown in Fig. 4.30:

- the *Temperature* dimension type.

And two units associated to the *Temperature* dimension:

- *Celsius* and *Fahrenheit*.

In Fig. 4.31 the *bdd* of the port types are defined. We define seven port types that use the value types defined, i.e. *CelsiusTemperature* and *FahrenheitTemperature*, the *Command* enumeration and a *Boolean* primitive type.

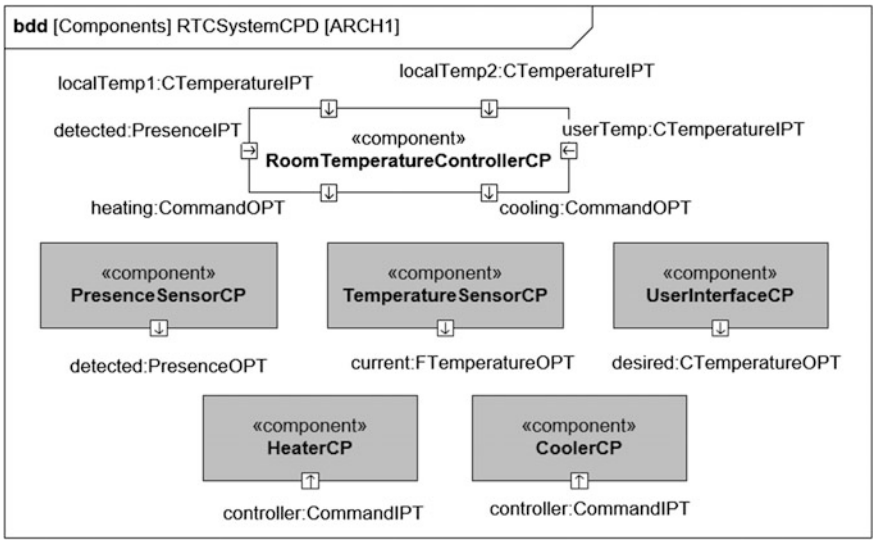


Fig. 4.32 Component types of the *RTC* architecture description

The three port types to provide temperature information are:

- *FTemperatureOPT* is an out port type to provide temperature in Fahrenheit;
- *CTemperatureOPT* is an out port type to provide temperature in Celsius;
- *CTemperatureIPT* is an in port type that require temperature in Fahrenheit.

The port types are:

- the *CommandIPT* and *CommandOPT* port types to, respectively, require and provide *On* and *Off* information;
- the *PresenceIPT* and *PresenceOPT* port types to, respectively, require and provide *Boolean* data.

We show now, in Fig. 4.32, the *bdd* of the component types and their ports. We define six component types: *PresenceSensorCP*, *TemperatureSensorCP*, *HeaterCP*, *CoolerCP*, *UserInterfaceCP*, and *RoomTemperatureControllerCP*.

The *RoomTemperatureControllerCP* component type has six ports. All ports have name and a previously defined port type.

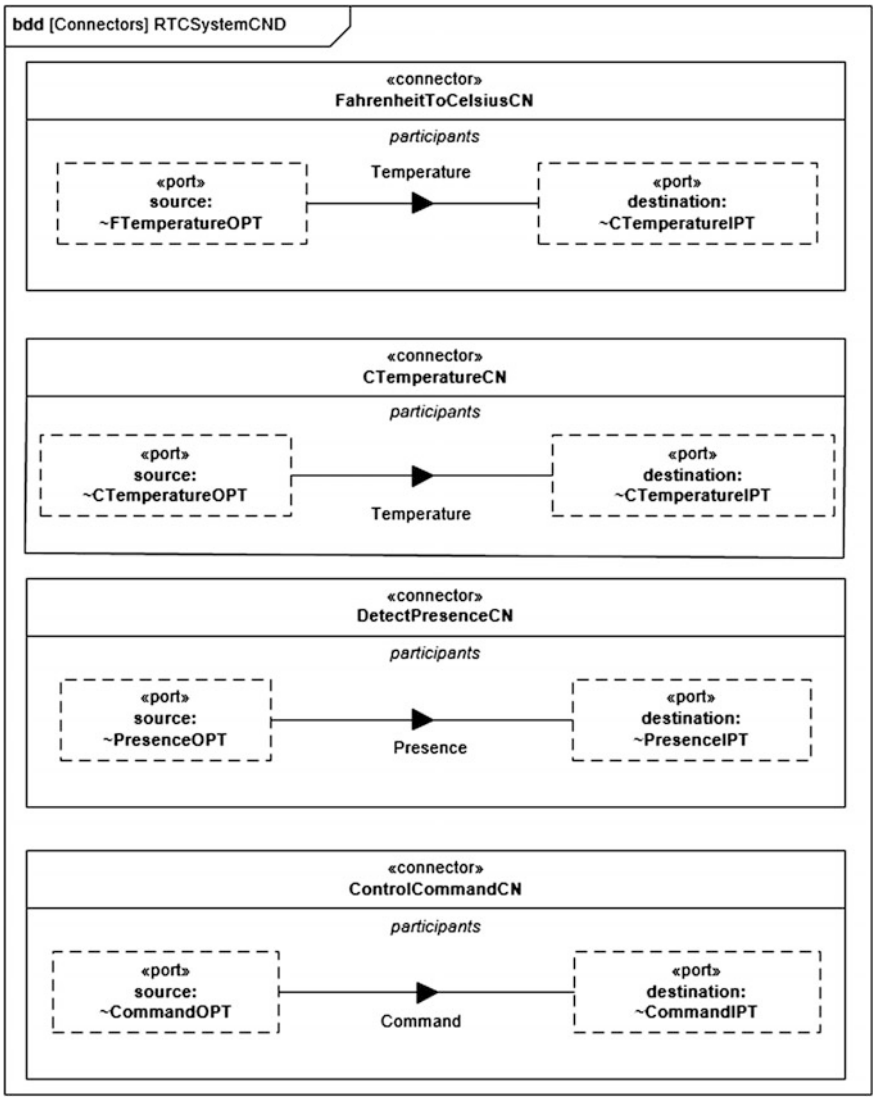


Fig. 4.33 Connector types of the *RTC* architecture description

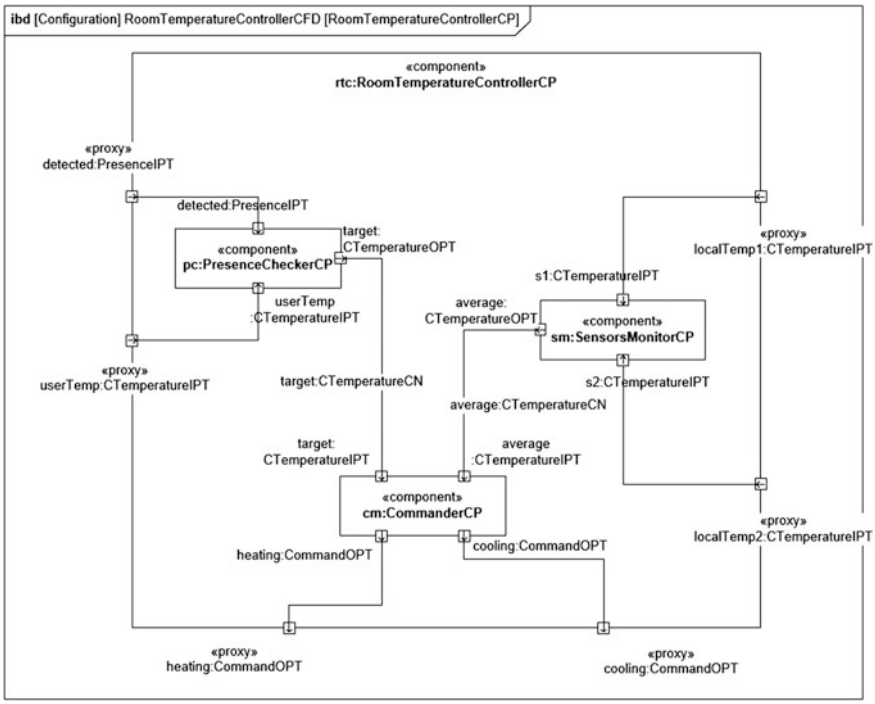


Fig. 4.34 Configuration of composite component *rtc*

We define, in Fig. 4.33, the *bdd* of the connector types. We define four connector types: *FahrenheitToCelsiusCN*, *DetectPresenceCN*, *CTemperatureCN*, and *ControlCommandCN*. All connector types in this case have two ports as participants. Note that all ports and the value type that flows in the connection were previously defined. Note also that the *FahrenheitToCelsiusCN* connector type has the responsibility of converting the data type from Fahrenheit to Celsius.

To finish, let us show, in Fig. 4.34, the internal structure of the composite component *rtc* and then, in Fig. 4.36, the configuration of the whole software architecture. The *rtc* of *RoomTemperatureControllerCF* composite component type includes, as explained, three components and two connectors: *pc*—a *PresenceCheckerCP* component type; *sm*—a *SensorsMonitorCP* component type; *cm*—a *CommanderCP* component type; *target* and *average* connectors: both of *CTemperatureCN* connector type.

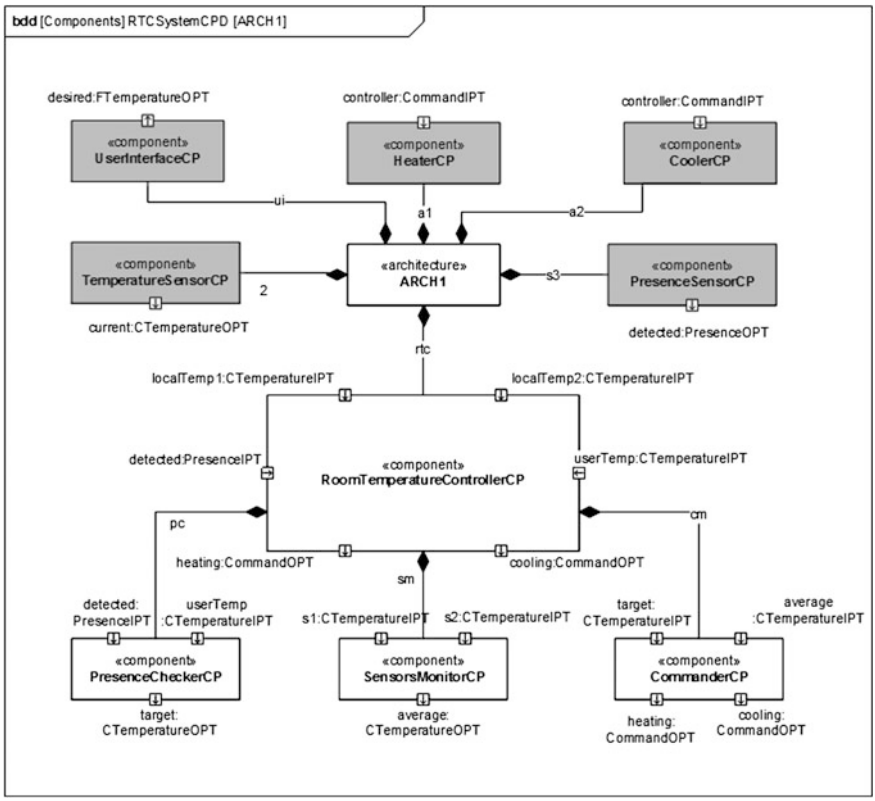


Fig. 4.35 The component types for the configuration of *RTC* System

Finally, we define the structure of the whole architecture. Six component types are used in this definition: *PresenceSensorCP, TemperatureSensorCP, HeaterCP, CoolerCP, UserInterfaceCP*, and *RoomTemperatureControllerCP*. These six component types are defined as shown in Fig. 4.35.

Using these component types, we define the whole configuration of the software architecture of the *RTC* system, shown in Fig. 4.36.

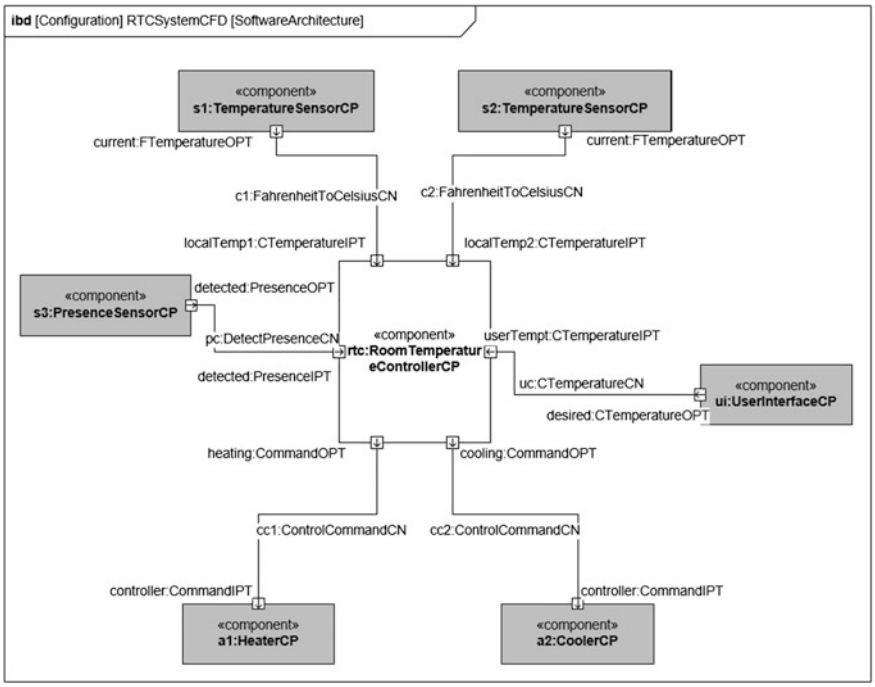


Fig. 4.36 The software architecture configuration of *RTC* System

4.6 Summary

In this chapter, you learned how to:

- define and use the main structural constructs of a software architecture: components, connectors, and configurations;
- define and use the auxiliary constructs applied to define the structural concepts: ports, value types, dimensions, and units;
- organize these architectural definitions in the form of block definition diagrams and internal block diagrams specifying different structural views of the software architecture;
- apply the SysADL notation to express the different constructs provided by the structural viewpoint.

Further Reading

1. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)
2. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley Professional, Boston (1999)
3. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice, 1st edn. Wiley, New York (2010)

Chapter 5
Specifying Behavior of Software Architectures

In this chapter, we present the behavioral viewpoint provided by SysADL. We explain the SysADL constructs that enable the description of behavioral views. We present in details the concepts underlying each of these constructs and how each one is applied to specify the different views comprising the architectural behavior. We illustrate each definition and use of behavioral elements with our running example.

You will learn:

- the architectural constructs to express behavior: activities, actions, interactions;
- the auxiliary constructs to define activities and actions: equations for defining actions and protocols controlling interactions;
- the SysADL notation to represent these constructs in terms of diagrams.

5.1 Introduction

In the previous chapter, we presented the SysADL constructs provided by the structural viewpoint. Indeed, we saw that the structure of a software architecture is defined in terms of configurations of components that interact through connectors. Nevertheless, identifying which are the components and connectors is not enough to define a software architecture. Definitely, just defining which are the components and connectors tells nothing about how components behave, how connectors behave, and how these interacting behaviors imply the system functionality.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_5

The questions that we will address are now:

- what are the concepts required for specifying the behavior of a software architecture?
- which are the SysADL constructs provided by the behavioral viewpoint?
- how to apply these constructs for describing different views of the architectural behavior?
- how diagrams are used to express these views?

Each of these questions will be tackled in this chapter.

5.2 Behavioral Viewpoint and Views

5.2.1 Behavioral Viewpoint

As you know, SysADL comprises three viewpoints: (i) the structural viewpoint, (ii) the behavioral viewpoint, and (iii) the executable viewpoint. The behavioral viewpoint enables the description of the different views of the behavior of a software architecture. The behavior of a software architecture refers to the way in which the structural elements perform activities and interactions to achieve the required system functionality. These structural elements are components and connectors and their composition in configurations.

The behavior of a component defines its functional behavior while the behavior of a connector defines the interaction behavior regulating communication between components. The behavior of a port of a component or connector defines the protocol that governs the dataflow interaction from or to that port. The behavior of a configuration emerges from the composition of the individual behaviors of components and connectors.

For instance, in the architecture description of the *RTC* System (our running example), the behavioral viewpoint provides the SysADL constructs to describe how each of the sensors behaves for monitoring the temperature, how the presence sensor behaves, and how both influence the behavior of the controller for commanding the heater and cooler actuators. The behavioral viewpoint also provides the SysADL constructs for expressing the behavior of connectors: how the sensors transmit the room temperature to the controller, how the controller connects to the presence sensor, and how it transmits commands to the heater and cooler actuators.

SysADL provides its behavioral viewpoint by defining the following behavioral constructs: *activity, action, interaction, equations, and protocol*. An *activity* is expressed by a sequential control flow of actions. *Actions* may be internal actions (called *simply actions*) or interactions (expressing *send and receive of data*). *Equations* define the semantics of actions by expressing their pre- and post-conditions. A *protocol* is expressed by a sequential control flow of interactions. The behavior of a component or connector is expressed by an activity, while protocols describe the behavior of their ports.

The behavioral specification includes both the definition and the use of how structural elements perform activities to consume and produce data:

- the activities and actions of components and connectors;
- the protocol of ports of components and connectors;
- the equations for defining the semantics of actions.

These different SysADL constructs are applied in the definition of behavioral views by using the activity and parametric diagrams.

5.2.2 Behavioral Views

Using the SysADL constructs of the behavioral viewpoint, we can define different behavioral views. A *behavioral view* shows a subset of the behavior description of a software architecture. One or several views describe the definitions of activities, actions, equations, and protocols that are used in the definition of the architectural behavior. One or several views describe how the defined activities, actions, equations, and protocols are used and composed to define the architectural behavior, which achieves the required system functionality.

For instance, in the *RTC* System, a view could show all the sensor definitions, another view could show the controller definition and yet another view could show the actuator definitions. Views will be created according to the concerns of stakeholders. In the specification of the architectural behavior, we specify the activity of each component (the different sensors, the controller, and the actuators) and each connector linking a sensor to the controller, and the controller to each actuator. Each activity is defined in terms of actions and control and data flows that govern their executions.

5.3 Behavioral Constructs

5.3.1 Activity

In SysADL, we apply the *activity* construct to specify the behavior of components or connectors and to specify the behavior of ports as protocols (activities comprising only interactions). An activity depicts the behavior of an architectural element by expressing:

- its *parameters*: how the element consumes and produces data through pins;
- its *body*: the basic actions that execute the behavior and the control and data flows between actions.

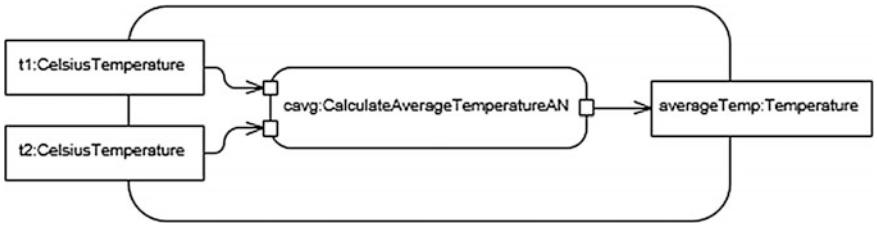


Fig. 5.1 Definition of an activity

The definition of an activity, as illustrated in Fig. 5.1, includes thereby its *pins (t1, t2, averageTemp)*, *actions (CalculateAverageTemperatureAN)* and *data*, and *control flows*.

Pins are representations of parameters and specify a stream of data. *In* pins represents input parameters, while *out* pins represents output parameters. *Flows* represent the control and data flows in activities. Flow arrows should link in and out pins appropriately, as they represent how the data and control flows concurrently progresses from an action to another. *Actions* are simple behaviors that execute from begin to end receiving parameters and returning a result.

The behavior specified by an activity indicates a data flow behavior. It means that an activity starts when all *in* pins receive data. The data flows and the control flows are then carried out to call its actions. When an action ends, the result flows to the *out* pin and the activity waits for its consumption. After the consumption, the activity is ready to receive new data. When all in pins receive data, the execution starts again, sequentially. Figure 5.2 illustrates the sequence of steps involved in the behavior of an activity. Figure 5.2a illustrates the activity with no data in its pins. Figure 5.2b shows that the activity received data in the *t2* pin. Figure 5.2c shows that the activity received data in the *t1* pin. After receiving data in all *in* pins, the action starts—Fig. 5.2d—and when it finishes, the *averageTemp* out pin receives data—Fig. 5.2e. After that, the activity returns to the initial state of waiting data—Fig. 5.2f. Note that the execution of an activity is triggered when all input pins receive data. After the consumption of the data available *in* input pins, the activity is ready to receive new data to execute again. The execution of the activities may result on sending possibly transformed data through output pins.

For instance, in the architecture description of the behavior of the *RTC* System, we can define an activity to calculate the average of temperatures coming from sensors. It has as input the sensed temperatures and output the resulting average. In the case of two temperature sensors, each time the two values of temperature are available in the in pins, the activity is executed to provide in its out pin the resulting average. This sequential behavior will continue to execute when new two values of temperature are available in the inputs pins and so on.

In SysADL, there are complementary notations for use and definition. We need to define activity types before using them to instantiate activities.

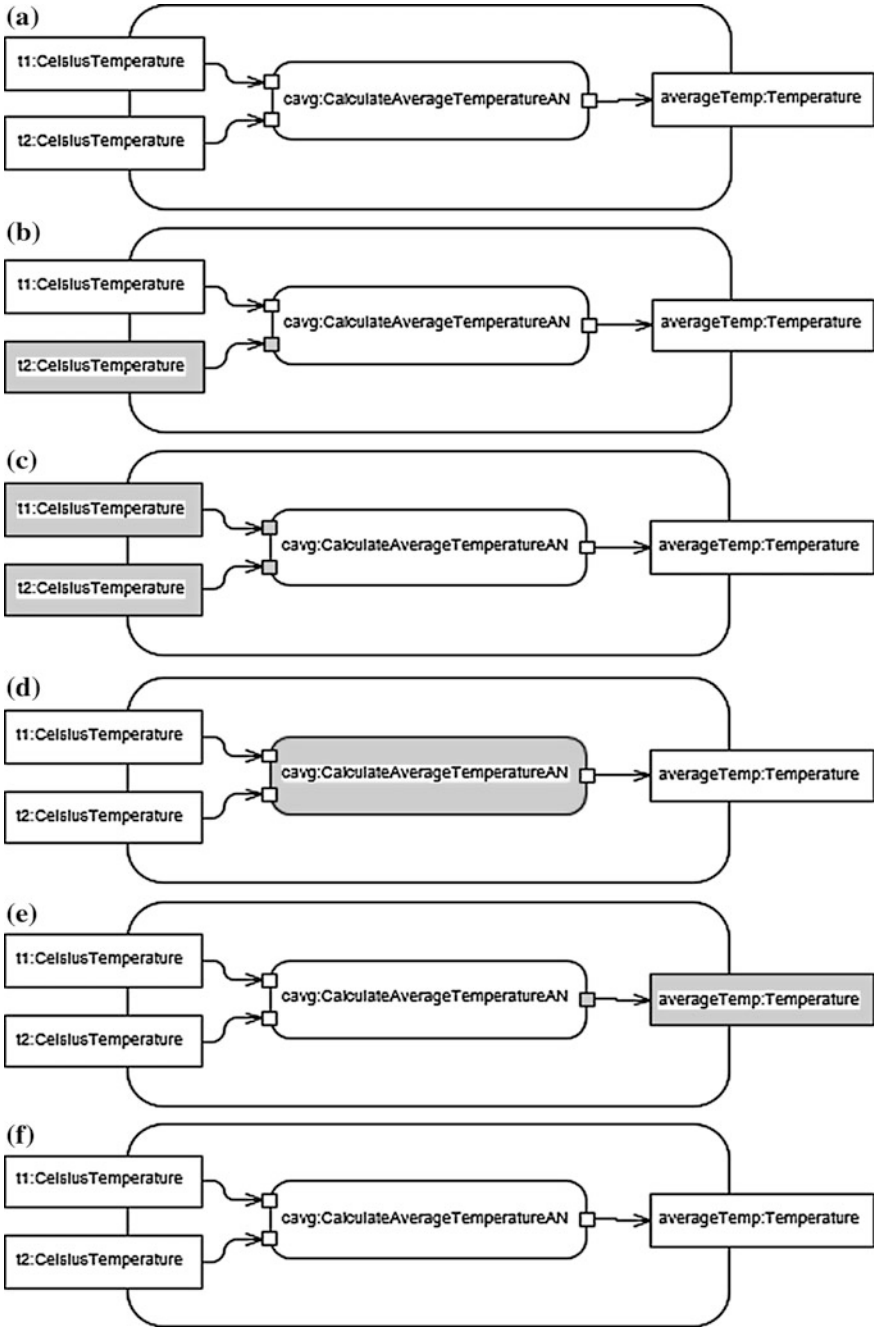


Fig. 5.2 An illustration of the behavior of an activity

Definition notation. An activity type is specified by its parameters and its body. The notation to define the activity is using a rectangle with compartments. The first compartment contains the «activity» stereotype with the name of the activity and another compartment contains the parameters of the activity. The notation for activity body definition is a rounded rectangle. The pins represent the parameters (data streams). Inside the rectangle there are actions, and the control and data flows.

Flows represent the control or data flow in the activity body. Flow arrows represent how control change from an action call to another. Flow arrows should link in and out pins appropriately.

To illustrate, we show, in Fig. 5.3, the notation to define the *Calculate AverageTemperatureAC* activity type. Figure 5.3a shows the definition of the parameters, and Fig. 5.3b illustrates the activity body with two in pins, *t1* and *t2,* an out pin, *averageTemp*, and the *cavg* call action. The two *in* pins and the *out* pins corresponds to the input and output parameters. There is only one action in this activity: *cavg* of *CalculateAverageTempAN* action type. The action has the same parameters that are directly connected to the respective pins by the data flows.

Use notation. The use notation for an activity is a rounded rectangle, with its name followed by a colon, the activity type name, and its parameters shown as pins. A pin of an activity represents a parameter that may be an input or an output parameter, i.e., an in or an out data flow to or from an activity.

A pin is mapped to a port in the component that embodies the activity. The data and value types must be compatible with the type specified in the flow properties of a port definition.

To illustrate the use notation, we show in Fig. 5.4 the use of an activity—*cavg* of *CalculateAverageTemperatureAC* type with two input parameters (*t1* and *t2*) and one output parameter—*averageTemp*. A parameter declaration has a name and a value type.

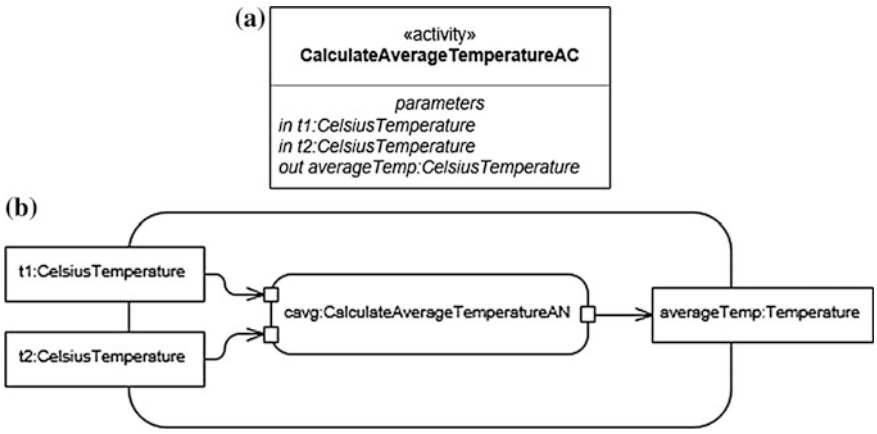


Fig. 5.3 Notation to represent the definition of activity: **a** parameters, **b** body

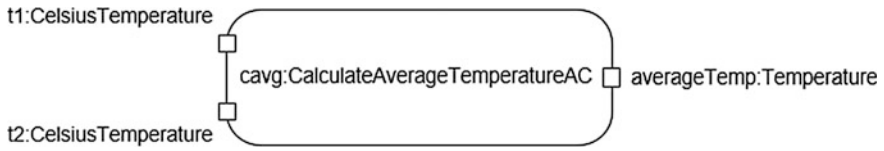


Fig. 5.4 Notation to represent the use of an activity

5.3.2 Action

An action refers to a simple, atomic behavior that receives input parameters, computes them, and provides a result to the output parameter. An action executes atomically when all in pins receive data and ends when it puts data in the out pins.

It executes from begin to end without interruption.

Thereby, when an action consumes data from the in pin, it is executed and when the results are sent to the out pins, it finishes. An action has therefore a start-stop semantics that is different from the semantics of the activities:

- when an action provides its results in the out pin it stops;
- when an activity provides its results in the out pins, it waits again new data in the in pins for a new sequential execution.

An example of an action is a function to calculate the average using two input parameters and providing the result in the output parameters.

Actions may be:

- internal actions, which are SysADL built-in actions or user-defined actions;
- interactions, which are formed by one of the two SysADL built-in interactions, one for sending data and the other for receiving data.

An action is defined by its parameters, its pre-conditions expressed in terms of input parameters, and its post-conditions expressed in terms of input and output parameters. Pre- and post-conditions are expressed using equations. An example of a post-condition is the equation expressing that the average between two temperatures is the half of their sum. A pre-condition is a proof obligation that must be satisfied by the action invoker. If the pre-condition is not verified it means that no guarantee can be provided regarding its result or even the termination of the action. We can use an action to specify a call to a simple behavior. It is only used inside an activity.

Definition notation. Before using an action it is necessary to define its type by specifying its name and the pre- and post-conditions that are expressed using equations.

We define an action type using the «action» stereotype with two compartments: (i) parameters to define parameters names and value types; and (ii) equational constraints to define pre- and post-conditions.

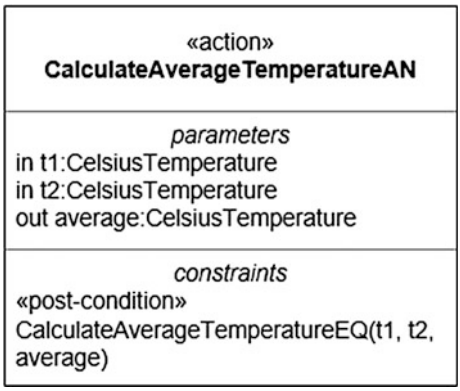


Fig. 5.5 Notation to represent the definition of an action

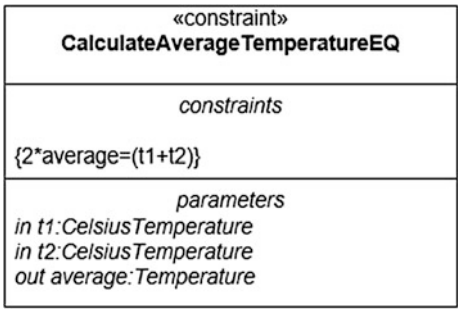


Fig. 5.6 Notation to represent the definition of an equation

In Fig. 5.5, we show the definition of the *CalculateAverageTemperatureAN* action type (as a convention, we use an *AN* suffix to the name of an action type in our running example) with three parameters—*t1, t2, averageTemp*—and the post-condition equation (a constraint).

In Fig. 5.6, we show the *CalculateAverageTemperatureEQ* equation specifying that *averageTemp* is the sum of *t1* and *t2* divided by 2.

Use notation. The SysADL notation to represent the use of an action is a rounded rectangle with small squares in its border to represent its parameters. An example is the *cavg* action of the *CalculateAverageTemperatureAN* type in Fig. 5.3b.

Note that activities and actions are similar, but complementary concepts (understanding their complementarity is very important to understand the behavioral viewpoint). An activity directly depicts the behavior of a software architecture component specifying the flow of data from its in ports to its out ports and the actions that process that data. Thus, the overall behavior of an activity is determined by the combination of actions. Each action in particular is constrained by equations that specify its semantics. Actions perform the basic computations of the input parameters to provide a result that satisfy the defined constraints and activities perform long-standing computations by calling actions.

5.3.3 Equations in the Action Definition

An equation is used to specify the constraints that govern the execution of actions. It defines a logical expression using the parameters. An output parameter is related to the input parameters through conditions. SysADL uses OCL (part of UML adapted to SysML) to express equation constraints. We rely on the ALF syntax when using OCL expressions.

Before using an equation it is necessary to define it. The SysADL notation to define equations types uses a constraint block. It has a «constraint» stereotype, the name of the equation type, and two compartments. The first compartment contains the equation to specify the "pre-conditions" and "post-conditions." The second one contains the parameters used in the equation.

In Fig. 5.6, we show the definition of the *CalculateAverageTemperatureEQ* equation (as a convention, we use an *EQ* suffix to the name of an equation in our running example) expressing the formula to calculate the average of the two temperatures. This equation type is used to describe the constraints that apply to the *CalculateAverageTemperatureAN* action, which is the only action used in the *CalculateAverageTemperatureAC* activity. This activity describes the behavior of the *SensorMonitorCP* component.

5.3.4 Relating Definition of Activities and Actions

Since actions are part of activities, we can represent this relationship using the composition association in their definition. Note that cardinality is a constraint that defines how many times an action is called inside an activity. By default, the cardinality is 1.

In Fig. 5.7 we illustrate the relationship between the *CalculateAverage TemperatureAN* action and the *CalculateAverageTemperatureAC* activity with cardinality 1. Thereby, this action can called only once in the activity body.

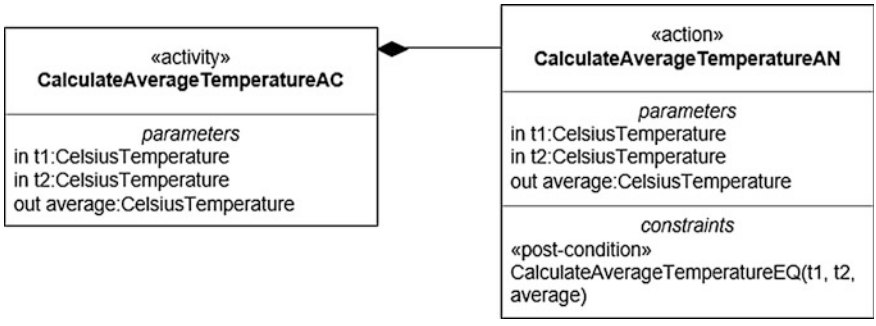


Fig. 5.7 Activities are composed of actions

5.3.5 *Relating Components and Connectors with Activities and Actions*

The examples in the preceding sections illustrate the relationships among the architectural elements. Considering component, we can state that:

- each component has an activity to express its behavior;
- each activity must have one or more actions to describe the behavior details;
- each action must have equations describing the constraints governing its atomic behavior.

Considering connectors:

- each connector has an activity to express its behavior;
- the activity of a connector may have none, one, or more actions to describe the behavior details;
- each action must have equations describing its pre- and post-conditions.

5.4 Diagrams for Behavioral Views

Behavioral views are organized in terms of diagrams. The behavioral definitions, i.e., definitions of activities and related actions, are expressed using block definition diagrams (*bdd*), activity diagrams (*act*), and parametric diagrams (*par*).

The block definition diagram is applied to specify the interface of activities and actions.

The activity diagram is applied to specify the internal behavior of components, connectors, and configurations as well as the protocols of ports in terms of actions, interactions, data, and control flows.

The parametric diagram is applied to specify the behavior of actions in terms of equations that define the pre- and post-conditions.

5.4.1 *Block Definition Diagrams*

Activities and actions definitions are described in block definition diagrams. Since actions are part of activities, we represent this relationship using composition.

To illustrate, we show, in Fig. 5.8, the *bdd* that describes the definition of activities and actions in the architecture description of the *RTC* System. This figure shows an interesting feature of SysADL. It illustrates the definition of the behavior of the *CommanderCP* component with the *DecideCommandAC* activity that is composed of three different actions: *CommandHeaterAN, CommandCollerAN,* and *CompareTemperatureAN.*

It is worth to note that Fig. 5.8 depicts the *bdd* for defining the behavior of a component. Remember that components are first-class citizens in SysADL and have behaviors like connectors, however with different architectural roles, the components as the locus of computation and the connectors as the locus of communication.

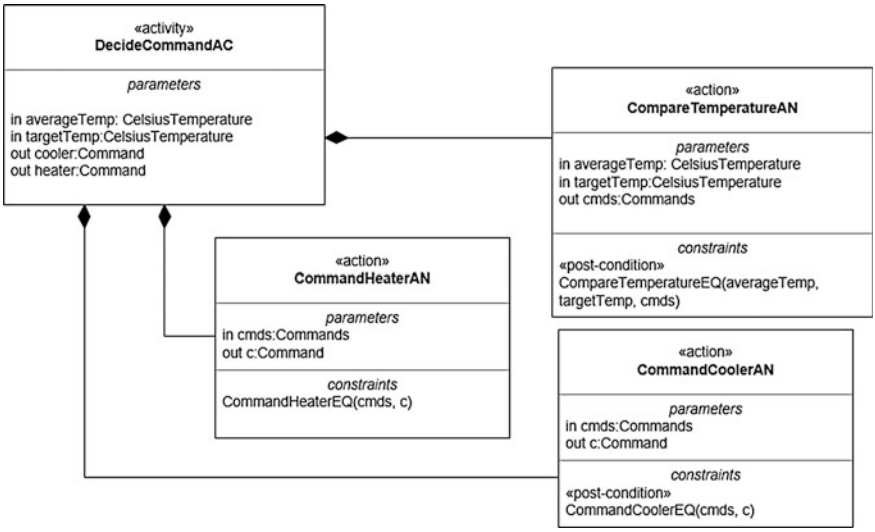


Fig. 5.8 The bdd (partial) with activities and actions types definitions

5.4.2 *Activity Diagrams*

An activity diagram is used for the description of the body of an activity. The activity diagram has a header informing the kind of the diagram (*act*), the indication it is representing a behavior, the name of the diagram and the name of the element (component or connector). The contents represent the body of the behavior with the pins representing the parameters and the data and control flows specifying the chaining of action calls.

Figure 5.9 illustrates the activity diagram representing the behavior of a connector named *FahrenheitToCelsiusCN* that connects the *s1* and *s2* components to *rtc* and needs to convert the temperature value in Fahrenheit provided by the *current* ports in those components to a Celsius value that is required by the *localTemp1* and *localTemp2* ports in *rtc*.

Observing Fig. 5.9, the *FahrenheitToCelsiusAC* activity diagram specifies the behavior of *FahrenheitToCelsiusCN*. It defines an input parameter represented as the *t* pin, an output parameter represented as the *c* pin, and the *ftc* action.

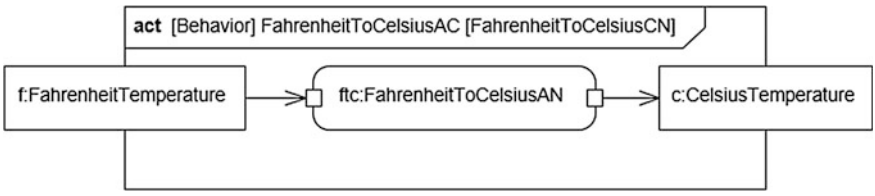


Fig. 5.9 The behavior of the *FahrenheitToCelsiusCN* connector

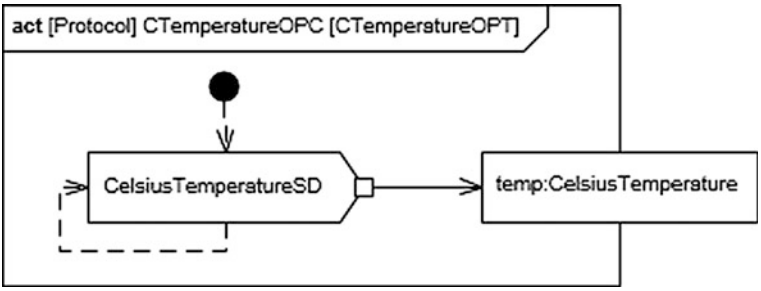


Fig. 5.10 The protocol of the *CTemperatureOPT* port

SysADL also uses activity diagrams to specify protocols of ports. The activity diagram header should inform that the diagram represents a protocol, the name of the activity, and the name of the port type whose protocol we are describing. The diagram contents represent the body with the pins representing the data stream that flows through the port, and the send or receive actions that specify if the port is sending or receiving data.

Figure 5.10 shows an activity diagram for describing the protocol of the *CTemperatureOPT* port. It defines a sending action, named *CelsiusTemperatureSD* with a pin. It sends a value *temp* of the *CelsiusTemperature* type to the out port. A control flow from *CelsiusTemperatureSD* to itself represents that after sending a value, it returns to send a next value.

In Fig. 5.11, we illustrate the activity diagram of a more complex behavior corresponding to the definitions of Fig. 5.8. Figure 5.12 shows the steps carrying out the behavior of the *DecideComandAC* activity. This activity decides the command (*on* or *off*) for both the *heater* and the *cooler*. The defined activity receives the current temperature and the desired target temperature (Fig. 5.12b).

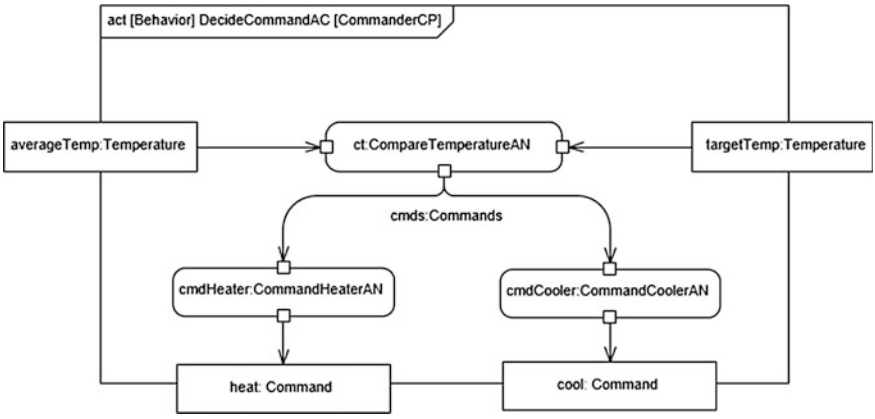


Fig. 5.11 The behavior of the *DecideCommandAC* activity

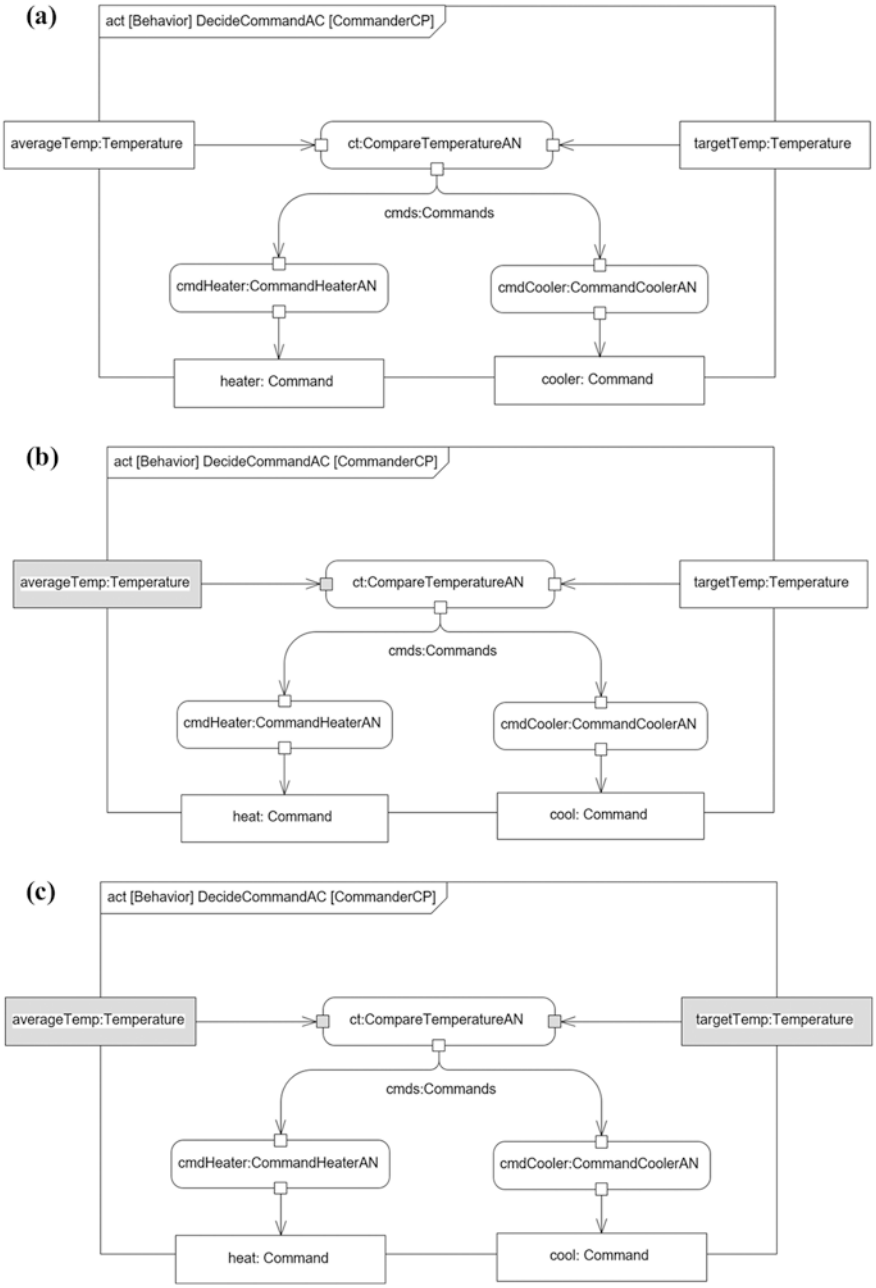


Fig. 5.12 The steps carrying out the behavior of the *DecideCommandAC* activity

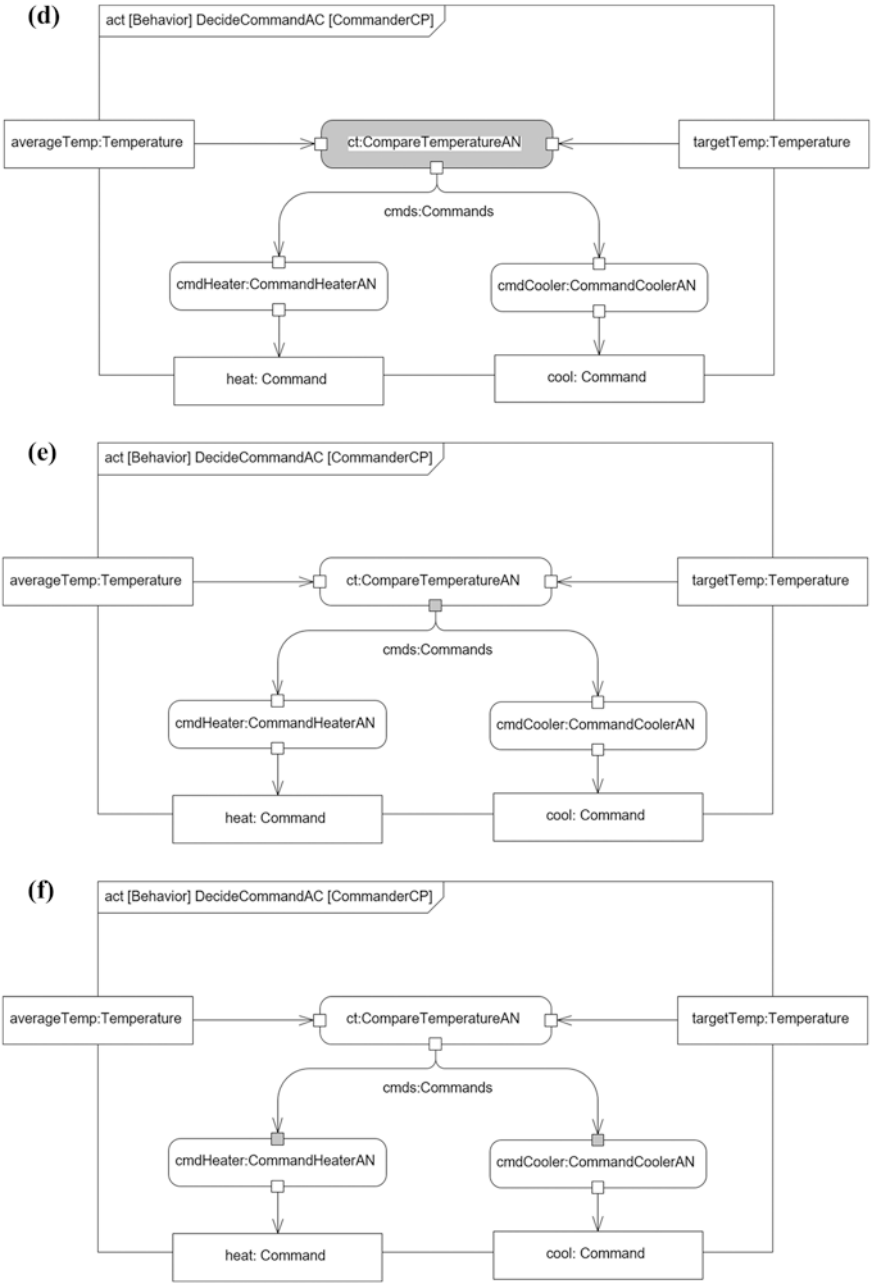


Fig. 5.12 (continued)

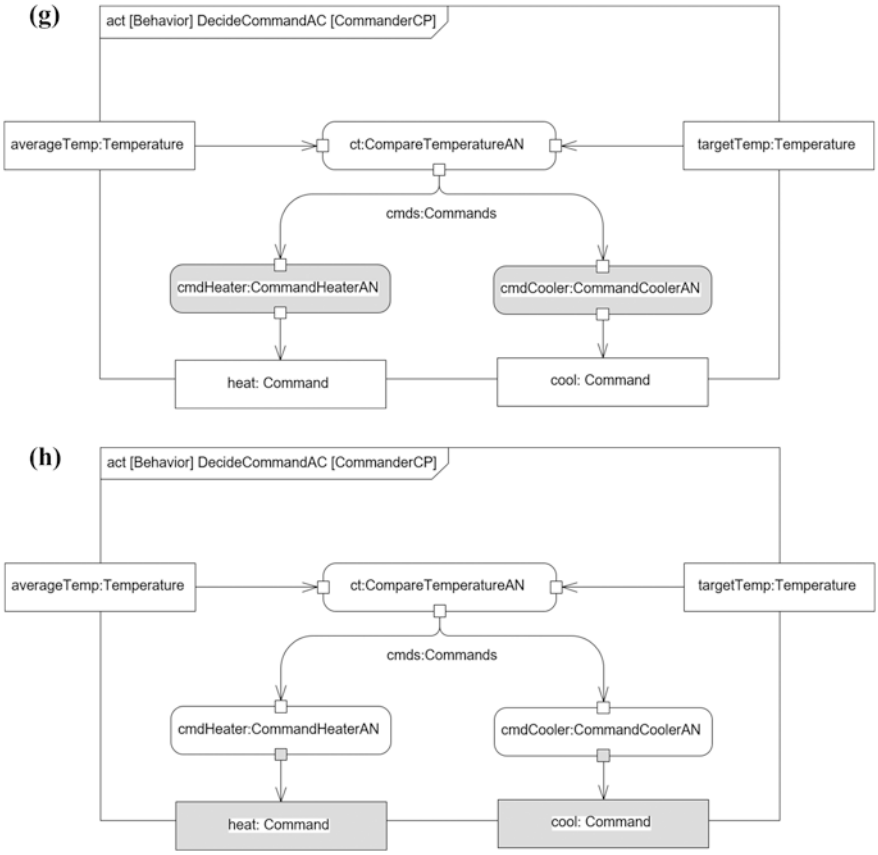


Fig. 5.12 (continued)

Note that the values can arrive at different time, so the activity starts (starting the *CompareTemperatureAN* action) only when both are received. Based on the comparison of the two temperatures (Fig. 5.12c), the *CompareTemperatureAN* action decides the output, which is a *Commands* instance (*cmds*) composed of the *cooler* and *heater* attributes. According to the comparison result, the value of each attribute can be set *on* or *off* (the possible values of the *Command* enumeration). If the average temperature is lower than the target temperature, then it sets *on* to the *heater*, and *off* to the *cooler*. Otherwise, it sets *off* to the *heater*, and *on* to the *cooler*. The *CommandHeaterAN* and the *CommandCollerAN* actions receive the data (Fig. 5.12d), observe the received values (*on* or *off*) and apply the proper action according to the values (Fig. 5.12f). Since both actions are concurrent, the outputs are not necessarily provided at the same time.

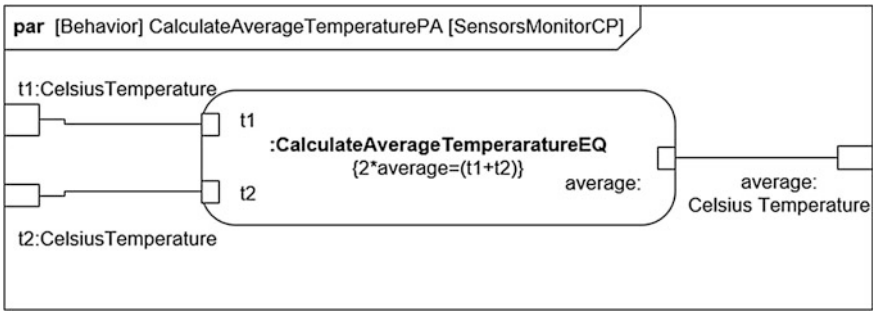


Fig. 5.13 The parametric diagram for defining action semantics

5.4.3 Parametric Diagrams

The parametric diagram is used to describe the use of an equation that specifies the behavior of an action in terms of pre- and post-conditions. It associates the parameters of an action to an equation that defines how the output parameters depend on the input parameters. The header of a parametric diagram describes the kind, name, and associated elements, and its contents represent the use of an equation to specify the action. The equation needs to be defined in a *bdd* before its use in a parametric diagram.

We show, in Fig. 5.13, the *CalculateAverageTemperaturePA* parametric diagram that describes the constraints to the behavior of the *SensorMonitorCP* component. Next section gives more details about this relationship.

5.5 Relating Structural and Behavioral Viewpoints

In SysADL, the behavioral viewpoint is allocated to the structural viewpoint. For each defined component, connector, and port, there is an activity specifying its behavior. The behavior of the configuration of a composite component or the overall software architecture can be derived from the specification of theirs constituent elements.

We show, in Fig. 5.14, an example of a component type, *SensorsMonitorCP*, and its allocated behavior.

Parametric and activities diagrams provide complementary views of the behavioral viewpoint, as illustrated in Fig. 5.13. The equation in the parametric diagram is directly linked to the action in an activity diagram. The input parameter of the parametric diagram is directly related to the input parameters of the activity diagram, and as a convention they have the same name. Both of them represent the type of data that can flow in the correspondent ports. Figure 5.14 also shows the component and its behavior described in activity and parametric diagrams.

The allocation of the behavioral viewpoint on the structural one is sketched in Fig. 5.15.

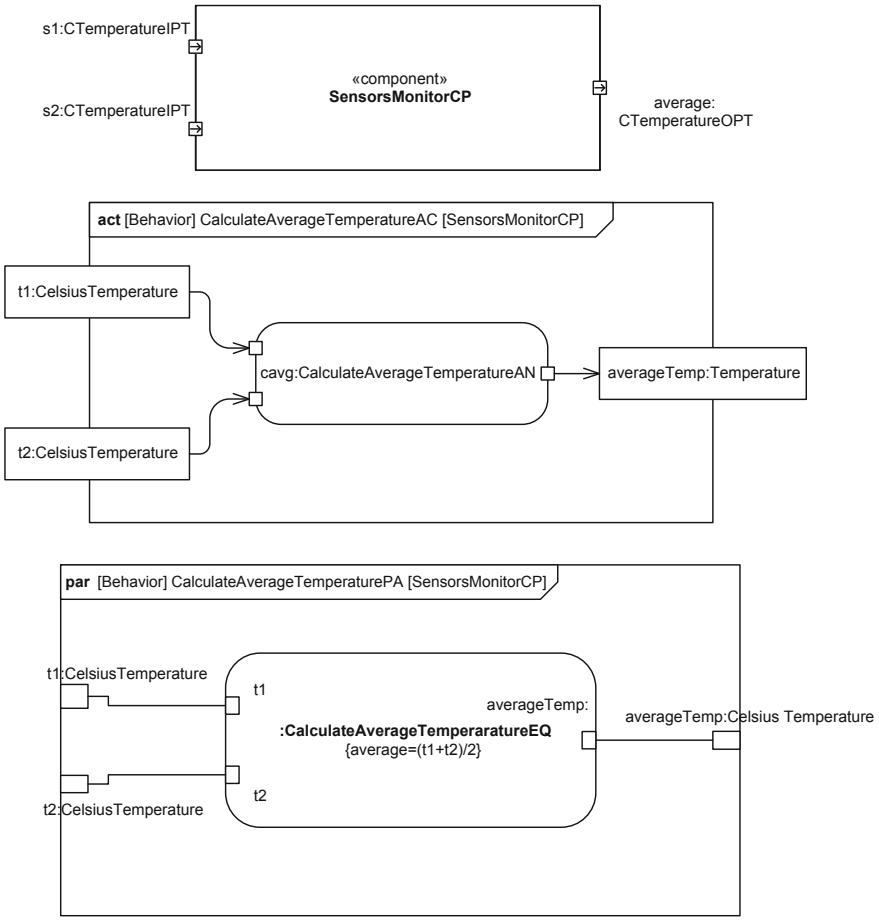


Fig. 5.14 Corresponding definitions from the structural and behavioral viewpoints

5.6 Describing the Architecture from the Behavioral Viewpoint

Let us now apply the concepts and constructs you learned to describe the behavior of the software architecture of the *RTC* system, our running example.

For each of the component types we defined, in block definition diagrams (bdd) from the structural viewpoint (presented in the previous chapter, recalled hereafter in Fig. 5.16), we will define the protocol of each port and, afterwards, the activity of components and connectors of the defined types from the behavioral viewpoint.

As shown in Fig. 5.16, the *RTC* System has a *RoomTemperatureControlerCP* composite component. Its behavior is specified by describing the behavior of the

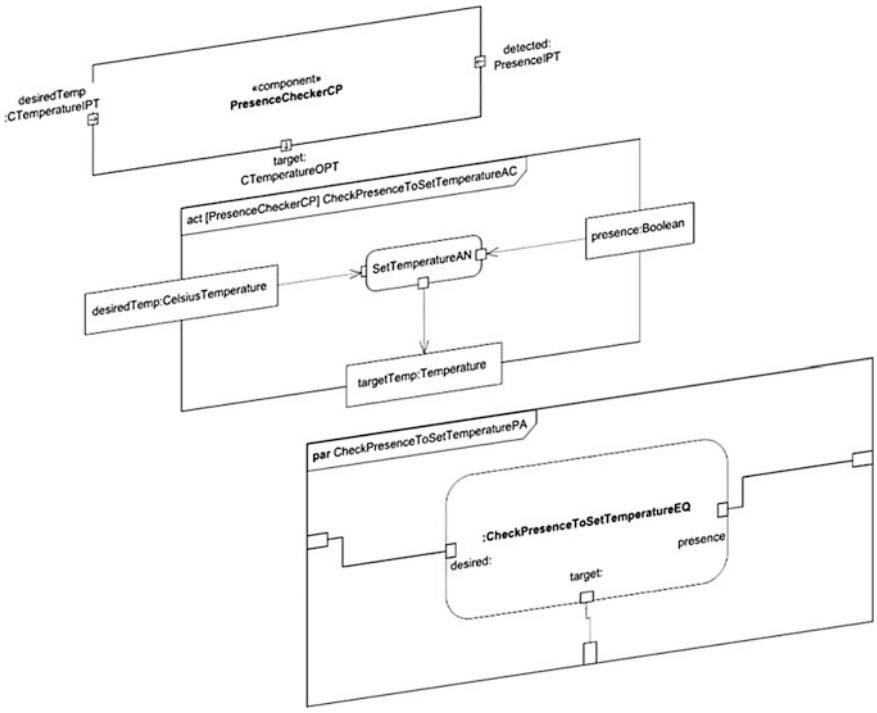


Fig. 5.15 Allocation of the behavioral viewpoint on the structural one

internal components. It has three internal components as we can see in the same figure.

For defining a protocol to be allocated to a port type, we need to define an activity diagram that specifies the control flow of input/output interactions associated to the port activity. These input/output interactions are used to receive or send data that flow through the port.

We show in Fig. 5.17, the definitions of the protocols allocated to the port types. Note that the name of the port type is given in square brackets. For example, the protocol *CTemperatureOPC* is defined to be allocated to the port type *CTemperatureOPT*.

The defined protocol indicates that, after creation of the component holding a port of this port type, the initial state implies that the port is ready to send a data using the *CelsiusTemperatureSD* action. The data flow between the send action *CelsiusTemperatureSD* and the *temp* pin of value type *CelsiusTemperature* specifies that the data sent will flow through this pin to the bound connector. The control flow that loops from/to the send action *CelsiusTemperatureSD* indicates that after sending a data, it may send the next one, and so on.

Similar protocols are defined for the other port types with out flows, as shown in Fig. 5.17: *FTemperatureOPC*, CommandOPC, and *PresenceOPC*. In Fig. 5.17,

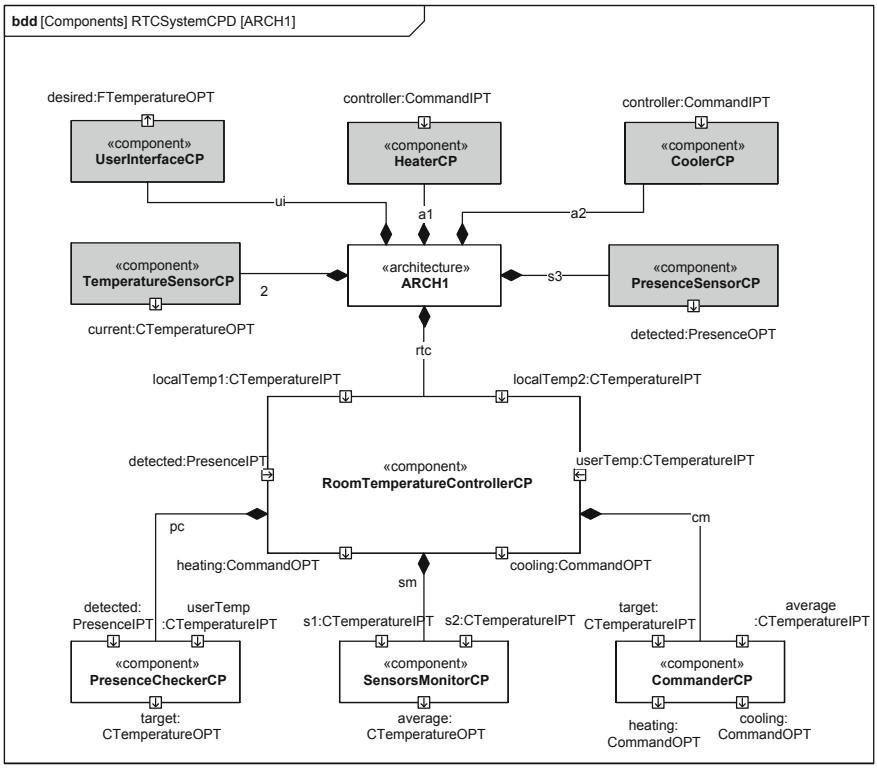


Fig. 5.16 Component types defined in the RTC architecture

we also define the protocols for the port types with in flows: *CTemperatureIPC*, CommandIPC, and *PresenceIPC*.

Now, that we defined the protocols allocated to the defined port types, we will define the activity to be allocated to the component types.

For defining an activity, we need two diagrams:

- a block definition diagram for defining the interface of the activity, and
- the activity diagram for specifying the body of the activity.

Note that, if the component is a boundary one, the body of the activity is not specified, only the interface is therefore defined, as shown in Fig. 5.18.

Once its interface specified in a block definition diagram, the activity diagrams show parameters and the action that compute the data received in the input parameters and send them to the output parameters. Considering that an activity diagram represents the behavior of a component, note that the parameters in the activity diagram represented by pins correspond to the ports of the component. This is also the case for connectors.

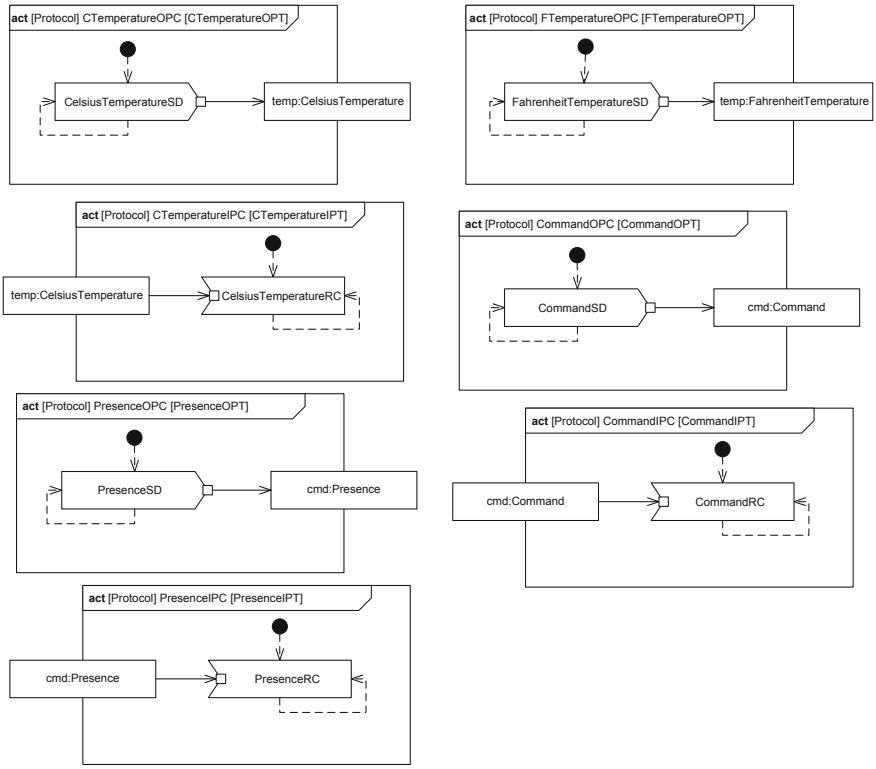


Fig. 5.17 Protocols defined in the RTC architecture

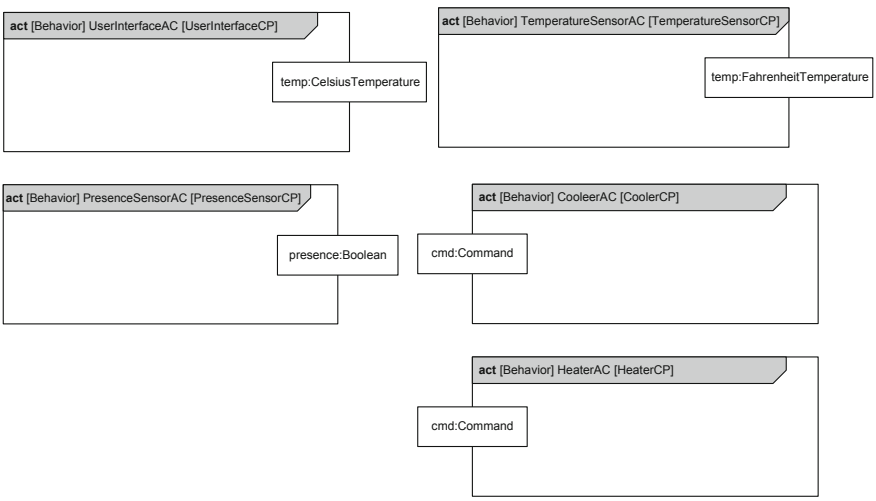


Fig. 5.18 Activities defined for the boundary components in the RTC architecture

Let us now define the activities to be allocated to internal component types. As shown in Fig. 5.16, at the architectural level, we have only one, the *Room TemperatureControllerCP*. This component type being composite, it will have other components as constituents (shown in Fig. 5.16), in this case the following three ones: *PresenceCheckerCP*, *SensorsMonitorCP*, and *CommanderCP*.

In Fig. 5.19 we show in a *bdd* the interface of the activities and the constituent actions. Three of them are activities defined for component types: *Calculate AverageTemperatureAC* for *SensorsMonitorCP, CheckPresenceToSetTemperature AC* for *PresenceCheckerCP,* and *DecideCommandAC* for *CommanderCP*. The fourth defined activity in Fig. 5.19 is the *FahrenheitToCelsiusAC* activity for the connectors linking sensors that output temperatures in °F and the *rtc* controller that input temperatures in °C.

Let us detail the definition of the activity *CalculateAverageTemperatureAC*. As shown in Fig. 5.19, the header of the activity has three parameters: (i) an input parameter *t1* of type *CelsiusTemperature*, (ii) an input parameter *t2* of type *CelsiusTemperature*, and (iii) an output parameter *averageTemp* of type *Celsius Temperature*. This activity calls, in its body, the action *CalculateAverage TemperatureAN* once. Such an action has similar parameters (the activity will pass its parameter values to the action): input parameters *t1* and *t2* of type *Celsius Temperature*, and an output parameter *averageTemp* of type *CelsiusTemperature*.

The post-condition of the *CalculateAverageTemperatureAN* action specifies the semantics of the action stating that the value of the output parameter *averageTemp* is equal to the sum of input parameters *t1* and *t2* divided by 2.

In a similar way, the post-conditions of the other actions define their semantics.

The *FahrenheitToCelsiusAC* activity, which expresses the behavior of the *FahrenheitToCelsiusCN* connector, links the *TemperatureSensorCP* component to the *RoomTemperatureControllerCP* component. Each activity, in this example, has only one action that provides the details of the behavior.

Now let us see the body of the defined activities. We show, in Fig. 5.20, the body of the following activities: *CalculateAverageTemperatureAC* for *Sensors MonitorCP, CheckPresenceToSetTemperatureAC* for *PresenceCheckerCP,* and *DecideCommandAC* for *CommanderCP*.

In Fig. 5.20, the *CalculateAverageTemperatureAC* activity specifies the behavior of the *SensorMonitorCP* component. The parameters show the type of data that flows in the corresponding ports. In this case, the specified behavior is the following one: each time a couple of temperature values are received by the activity in input pins *t1* and *t2*, the action *CalculateAverageTemperatureAN* is called and its resulting value is sent through the output pin *average*. It means that this activity will be executed continuously, where each execution will wait for the input pins to have values, for calculating and outputting the result in the out pin.

In Fig. 5.21, the *CheckPresenceToSetTemperatureAC* activity specifies the behavior of the *PresenceCheckerCP* component. Its behavior is similar to the previous one, with also two inputs and an output.

In Fig. 5.22, the *DecideCommandAC* activity specifies the behavior of the *CommanderCP* component. The parameters, expressed as pins, show the type of

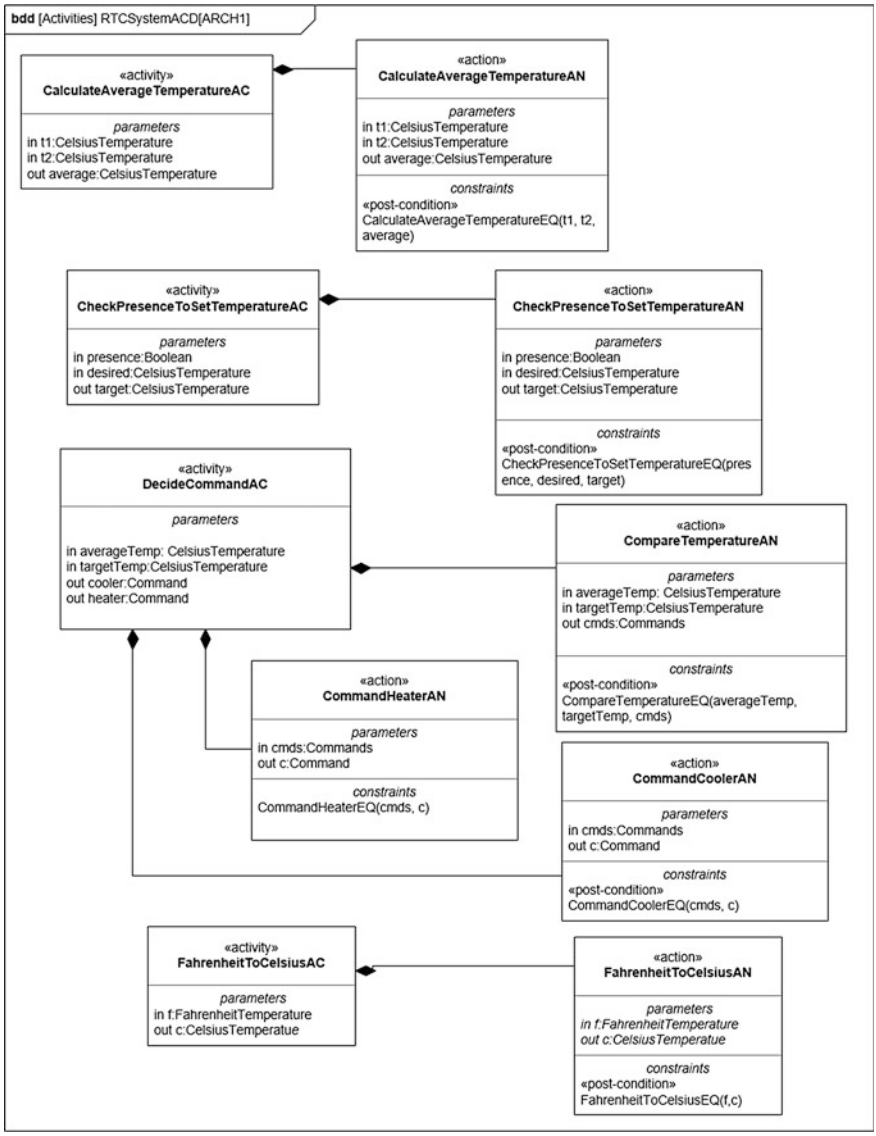


Fig. 5.19 Defining activities and actions in the RTC architecture

data that flows in the corresponding ports. In this behavior, from the input of the couple of the sensed temperature, *average*, and target temperature, *target*, the *SetCommandAN* action transmits commands for the heater and the cooler.

Finally, in Fig. 5.23, the *FahrenheitToCelsiusAC* activity specifies the behavior of the *FahrenheitToCelsius* connector. The behavior specifies that a connector of this type takes an input from a source port a value of temperature in °F, calls the

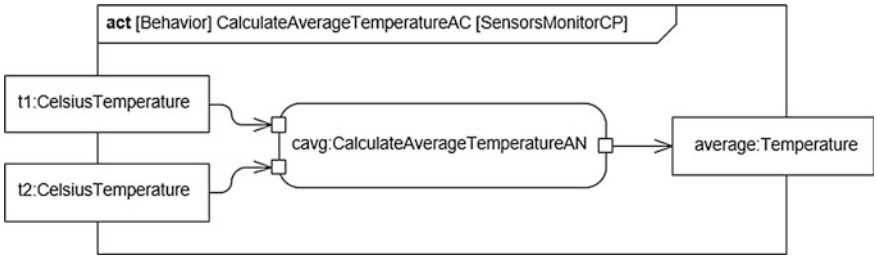


Fig. 5.20 Activity body of *CalculateAverageTemperatureAC*

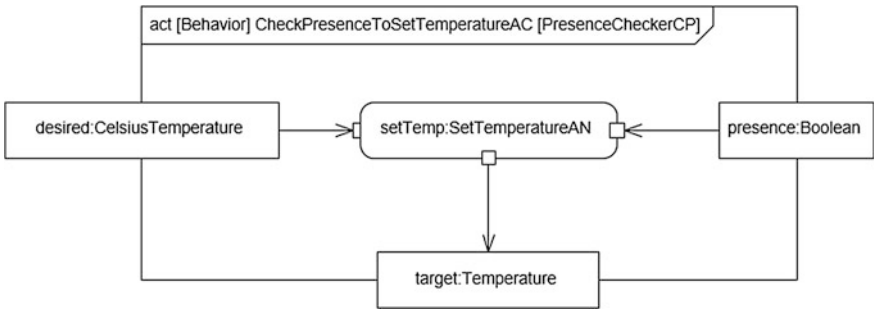


Fig. 5.21 Activity body of *CheckPresenceToSetTemperatureAC*

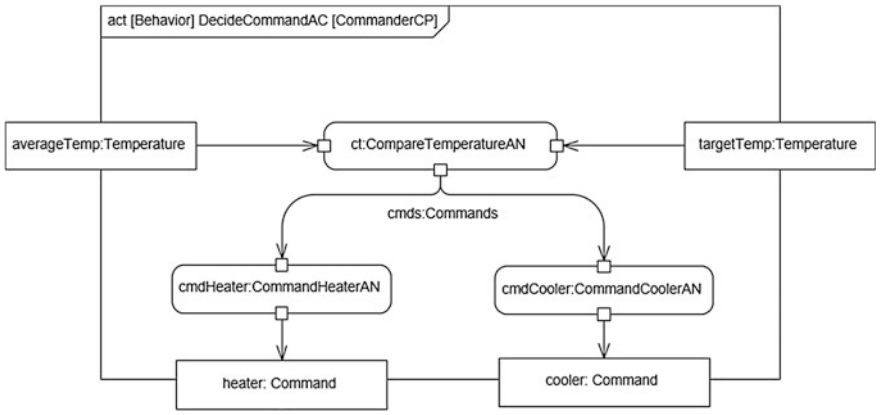


Fig. 5.22 Activity body of *DecideCommandAC*

action *FahrenheitToCelsiusAN* to transform this value in °C and then send it though its output pin to its destination port.

The set of definitions of protocols and activities are presented in different behavioral views that together form the description of the architecture from the

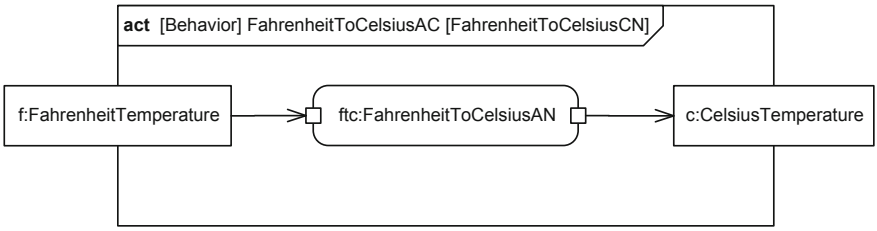


Fig. 5.23 Activity body of *FahrenheitToCelsiusAC*

behavioral viewpoint. Note that only simple activities and protocols are defined. All composite activities and protocols are derived from the definition of the simple activities and protocols based on the definition of the composite configurations.

5.7 Summary

In this chapter, you learned how to:

- define actions in terms of equations, i.e., pre- and post-conditions, to express atomic sequential behaviors;
- define activities to express the sequential behavior of components and connectors in terms of actions and interactions;
- define protocols that govern the interactions of ports of components and connectors;
- define configurations to express the concurrent behavior of a software architecture;
- organize these architectural definitions in the form of activity and parametric diagrams specifying different behavioral views of the software architecture.

Further Reading

1. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley Professional, Boston (1999)
2. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice, 1st edn. Wiley, New York (2010)

Chapter 6
Specifying Executable Software Architectures

In this chapter we present the executable viewpoint provided by SysADL. We explain the SysADL constructs that enable the description of the executable view. We describe, in details, the concepts underlying each of these constructs and how each one is applied. We illustrate each definition and how to use it with our running example.

You will learn the following:

- the architectural constructs to express the executable body of the actions;
- the data and control flow concepts used in the executable body of actions;
- the SysADL notation to represent these constructs.

6.1 Introduction

We presented, in the two previous chapters, the SysADL constructs provided by the structural and the behavioral viewpoints, respectively. In this chapter, we will present the third one: the *executable viewpoint*.

While the structural viewpoint declaratively describes the structure of the architecture and the behavioral viewpoint declaratively describes the behavior of the architecture, the executable viewpoint adds the executable elements of the architecture description.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_6

The executable viewpoint is expressed by describing the body of the actions expressing the computation. The SysADL notation to represent the body of the actions is based on ALF [1], part of UML and SysML.

Note that, even if executable, the level of abstraction of this viewpoint is independent of implementation concerns specific to programming languages.

From the executable viewpoint, the questions we will address are

- which are the SysADL constructs provided by the executable viewpoint?
- how to apply these constructs for describing the executable view of the architecture?

We will address each one of these questions in the sequel.

6.2 Executable Viewpoint and Views

6.2.1 Executable Viewpoint

As you know, SysADL comprises three viewpoints: (i) the structural viewpoint, (ii) the behavioral viewpoint, and (iii) the executable viewpoint.

The executable viewpoint enables the description of the executable elements of a software architecture to achieve the required system functionality.

In the behavior viewpoint, we saw how to express the activities and interactions to achieve the required system functionality. However, that behavior is not executable. To make an architecture executable, the executable viewpoint provides the SysADL constructs to describe the execution semantics of the body of actions. It comprises the data and control flow concepts.

Recall that the structural viewpoint provides the SysADL constructs to declare the structure of components, connectors, and configurations; and the behavioral viewpoint the SysADL constructs to declare the behavior of components, connectors, and configurations. To be able to execute them, an *action semantics* is needed.

Activities define the behavior of a component and connectors, and activities are defined in terms of actions that are specified in terms of pre- and post-conditions. An *action* is expressed by a sequential control flow of statements.

For instance, in the architecture description of the *RTC* System (our running example), in the activity *FahrenheitToCelsiusAC* of the *FahrenheitToCelsiusCN* connector, the action, *FahrenheitToCelsiusAN*, transforms the temperature from Fahrenheit to Celsius. The executable viewpoint completes this specification by providing the statements that actually executes the transformation.

An executable view is defined using the block definition diagram (bdd).

In the sequel, we will present the notation associated with the executable viewpoint and use the *RTC* system for illustrating the constructs.

6.2.2 Executable Views

Using the SysADL constructs of the executable viewpoint, we can define different executable views. An *executable view* shows a subset of the executable elements of a software architecture description.

For instance, in the *RTC* System, a view could show all the executable actions bodies invoked in the sensor activities, another view could show the executable actions bodies invoked in the controller activities and in the activities of its internal components and connectors, and yet another view could show the executable actions bodies invoked in the actuator activities. Views will be created according to the concerns of stakeholders.

6.3 Executable Constructs

6.3.1 Executable

In SysADL, we apply the *executable* construct to specify the action body. An executable depicts the action body by expressing

- its *parameters*: the pins that consume and produce data;
- its *body*: the statements that execute how the output pin is computed from the input pins.

The definition of the *CalculateAverageTemperatureEX* executable, illustrated in Fig. 6.1, includes thereby its *parameters* (*t1*, *t2*, *result*) and the *body* that calculates the average of temperatures coming from the inputs pins and returns the result in the output pin (*result*).

The definition of the *CompareTemperatureEX* executable, illustrated in Fig. 6.2, includes thereby its *parameters* (*averageTemp*, *targetTemp*, *result*) and the *body* that

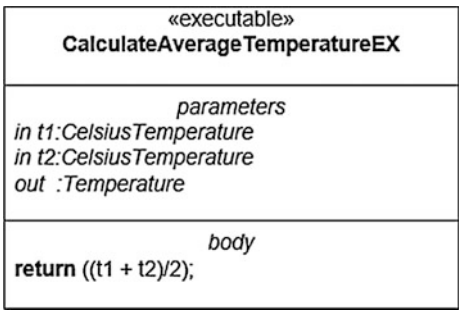


Fig. 6.1 Definition of the *CalculateAverageTemperatureEX* executable

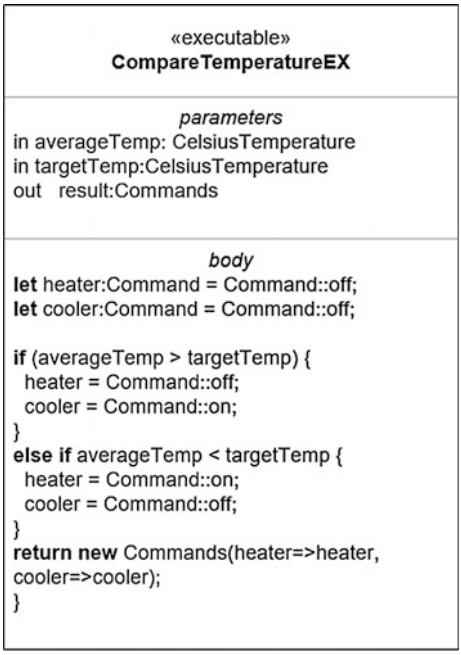


Fig. 6.2 Definition of the *CompareTemperatureEX* executable

- Uses the **let** statement to declare the *heater* and *cooler* variables of *Command* type and to initialize them with the *off* value.
- Uses the **if** statement to compare *averageTemp* with *targetTemp* to decide which commands must be set to *on* or *off* and set the values of each of them.
- Uses the **new** statement to create a value that is an instance of *Commands* datatype.
- Uses the **return** statement to return the *Commands* instance in the output pin.

6.3.2 Action Language

Let us now present the action language of the executable viewpoint that allows the definition of the executable action body. We will illustrate each of the statement constructs with our running example, i.e., the *RTC system*.

Statements. There are several kind of statements such as assignment, for, while, if, etc. In terms of grammar, statement is defined as in Fig. 6.3. In the following paragraphs, we view the grammar and examples for these statements.

Variables declaration requires the specification of a ***TypeUse*** and an optional ***Expression*** that returns a value and is expressed using constants, variables, and/or operators (logical, arithmetic, etc). In Fig. 6.4, we show the production rule of a variable declaration.

```
Statement ::=
     ( LocalNameDeclaration |
     Assignment |
     If |
     For |
     While |
     Switch |
     Expression |
     Return |
     ) ';'
```

Fig. 6.3 Statements in the SysADL grammar executable viewpoint

```
LocalNameDeclaration ::=
     TypeUse ('=' Expression)?

TypeUse::=
     ID ':' ID
```

Fig. 6.4 Variable declaration in SysADL grammar executable viewpoint

For instance, in the RTC System, we need to define a default temperature of type real. Its declaration and initial value assignment as 22 °C is expressed in SysADL as shown in Fig. 6.5.

Another example in the RTC System is to define a set of suggested temperature values, as shown Fig. 6.6.

In SysADL, we can assign values to variables using the assignment statement. In terms of grammar, the assignment production rule is defined in Fig. 6.7.

An example of assigning the value of an expression using the *defautTemperature* variable to another variable is shown in Fig. 6.8.

The internal control flow of a body is described using control flow statements: **while** and **for** are used to express iterative execution.

While is controlled by the evaluation of an expression. The statements are executed only if the *Expression* is true and remains executing while it is true. Figure 6.9 shows the SysADL grammar of *While*.

```
defaultTemperature:CelsiusTemperature=22;
```

Fig. 6.5 Variable declaration in the RTC System example

```
temperatures:CelsiusTemperature[] =
  new CelsiusTemperature[]{20,22};
```

Fig. 6.6 Declaration of a set in the RTC system example

```
Assignment ::=
     ID '=' Expression
```

Fig. 6.7 Assignment in SysADL grammar executable viewpoint

```
targetTemperature = defaultTemperature - 2;
```

Fig. 6.8 Assignment in the RTC system example

For instance, suppose that we need to search an element (*searchedTemp*) in a sequence of temperature values in the RTC System, stored in a variable named *temps*. As illustrate in Fig. 6.10, while is used to allow the searching loop until the searched temperature is found.

For is used to define a loop controlled by a local variable where the statements are repeatedly executed. It is typically used when the number of iterations is known or with sequences of values. Figure 6.11 shows the SysADL grammar of *For*.

For instance, suppose that we like to compute the sum of a sequence of temperature values in the RTC System, stored in a variable named *temps*. As illustrated in Fig. 6.12, *t* is a local variable that iterates over all the elements of the *temps* sequence. In each iteration, t refers to an element of the sequence (from the first to the last), and its value is added to the current sum.

If is a conditional construct that defines an expression to be evaluated during the execution. When the expression is true, the statements are executed. When the expression is false, the statements in the else block are executed. Figure 6.13 shows the SysADL grammar of *If*. Figure 6.10, previously presented, illustrates the use of *If*.

Switch is a control construct to express a number of possible execution paths. When the expression matches a case clause, the statements associated with that case are executed. Figure 6.14 shows the SysADL grammar of *Switch*.

Fig. 6.9 While in SysADL grammar executable viewpoint

```
While ::=
          'while' '(' Expression ')'
          '{'
                Statement*
          '}'
```

Fig. 6.10 While in the RTC system example

```
found = false;
i = 1;
while(not found) {
         if (searchedTemp == temps[i]) ;
            found = true;
          else
            i++;
 }
```

```
For ::=
  'for' ForControl '{'
        Statement*
  '}'

ForControl ::=
  LoopVariableDefinition*

LoopVariableDefinition ::=
  LoopTypeUse 'in' Expression ('..' Expression)?
  | LoopTypeUse ':' Expression

LoopTypeUse ::=
  ID ID /* the first ID refers to TypeUse, while the
second is the variable name*/
```

Fig. 6.11 For control elements in the SysADL grammar executable viewpoint

Fig. 6.12 For in the RTC system example

```
sum = 0;
for (t in temps) {
        sum = sum + t;
}
```

Fig. 6.13 If control element in SysADL grammar executable viewpoint

- **if** to express conditional execution

```
ConditionalBlock ::=
       'if' (Expression)
       ('{'
              (Statement)*
        '}'
       | Statement)
       (ElseBlock)?

ElseBlock ::=
       'else' '{'
              Statement*
       '}'
```

For instance, suppose that the user can define the fan level (low, medium, high). According to the level, the action sets a command, as illustrated in Fig. 6.15.

Return returns the results of the action given by an expression. Figure 6.16 shows the SysADL grammar of *Return*.

- **switch** to express the choice among different guarded branches

```
SwitchStatement ::=
       'switch' '(' Expression ')'
       '{' SwitchClause*
       SwitchDefaultClause? '}'

SwitchClause ::=
       SwitchCase+
       Statement*

SwitchCase ::=
       'case' Expression ':'

SwitchDefaultClause ::=
       'default' ':' Statement*
```

Fig. 6.14 Switch control element in SysADL grammar executable viewpoint

```
    switch(fanLevel) {
        case Level::low :
             fancommand(1);
        case Level::medium :
             fancommand(2);
        case Level::high :
             fancommand(3);

}
```

Fig. 6.15 Switch in the RTC system

```
Return ::=
       'return' Expression
```

Fig. 6.16 Return in SysADL grammar executable viewpoint

6.4 Summary

In this chapter, you learnt how to:

- define the executable body of actions;
- use the action language for expressing the data and control flows of actions;
- apply the SysADL notation to express the executable body of the actions

Further Reading

1. Concrete Syntax For A UML Action Language: Action Language For Foundational UML™ (ALF™). http://www.omg.org/spec/ALF/
2. Semantics Of A Foundational Subset For Executable UML Models (FUML™). http://www.omg.org/spec/FUML/
3. Precise Semantics Of UML Composite Structures™ (PSCS™). http://www.omg.org/spec/PSCS/1.0/

Reference

1. http://www.omg.org/spec/ALF/

Chapter 7
Executing Software Architectures

In this chapter, we present the execution semantics of architecture descriptions expressed with SysADL. We explain the SysADL constructs that enable execution as described from the different viewpoints and how the user observes this execution for validating the described architectures. In particular, we explain the operational semantics of each behavioral construct, specified in terms of transition rules where the premise declares the state before the execution and the conclusion, the state after the execution.

You will learn the following:

- the execution semantics of an architecture description expressed with SysADL;
- the execution semantics of components, ports, connectors, and configurations;
- the execution semantics of activities, actions, and related elements.

7.1 Introduction

We presented, in the previous chapters, the SysADL constructs provided by the structural, behavioral, and executable viewpoints. In this chapter, we will present the operational semantics of SysADL showing the execution of its elements.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_7

7.2 Executing an Architecture

We explain the execution of an architecture description using our running example, the *RTC* System, also used in the previous chapters. Figure 7.1 shows the architectural configuration of the system using a SysADL internal block diagram (ibd).

7.2.1 Executing Components

Considering the execution perspective, a component is a running process, seen as a black box communicating with its environment through ports.

Execution semantics of boundary components. The execution of a boundary component is given by the communication via its ports in conformance to the allocated protocol.

To understand our example, it is important to remind the port definition for the *s1* component. The *current:FTemperatureOPT* port provides Fahrenheit temperatures using a *temp* flow to send it out through the port. The *FTemperatureOPT* port definition is shown in Fig. 7.2.

Now we can illustrate how the execution of the *s1* component is given by the protocol of the *FTemperatureOPT* port. That protocol specifies that the port sends

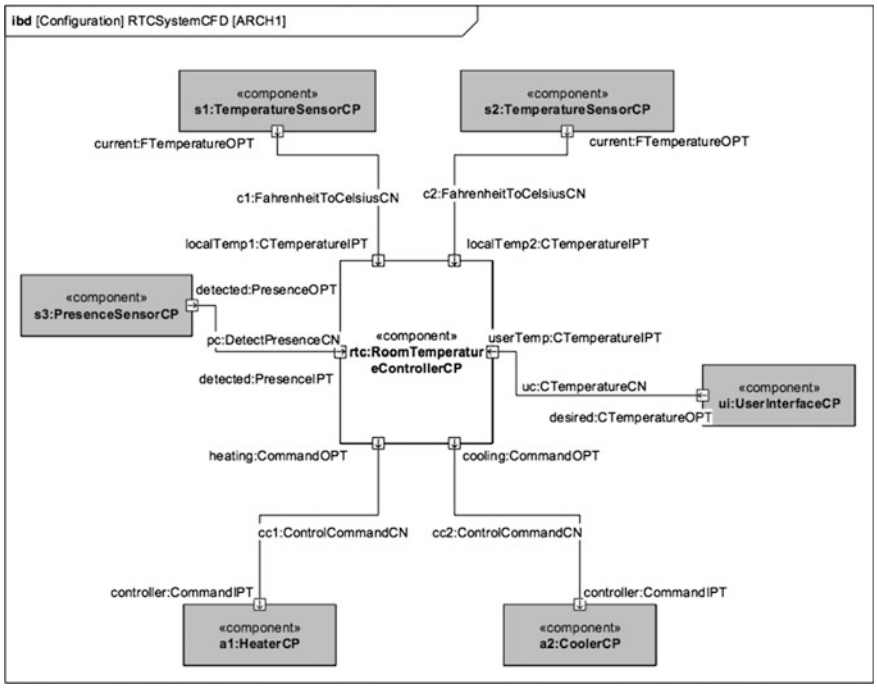


Fig. 7.1 The architecture of the RTC system

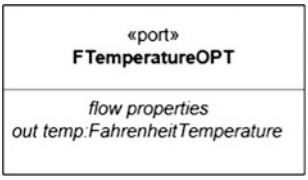


Fig. 7.2 *FTemperatureOPT* port type definition

Fahrenheit temperatures in a flow. From an observational point of view, the execution of *s1* produces a flow of temperatures, as shown in Fig. 7.3. Figure 7.3 illustrates a sequence of temperature values flowing out from the port. Initially, *temp* is empty. We use the *current* port in white color to represent the empty value. We also show the value of *temp* in the current port of *s1* component in a board at right side in Fig. 7.3. In the sequence, the first value out is a *temp=21*, which means a 21 °F temperature. The current port in dark gray represents that there is a data to be consumed. At the third instant of time, the value in the *temp* flow is empty again. After that, we have a sequence of *temp=22* and *temp=23*.

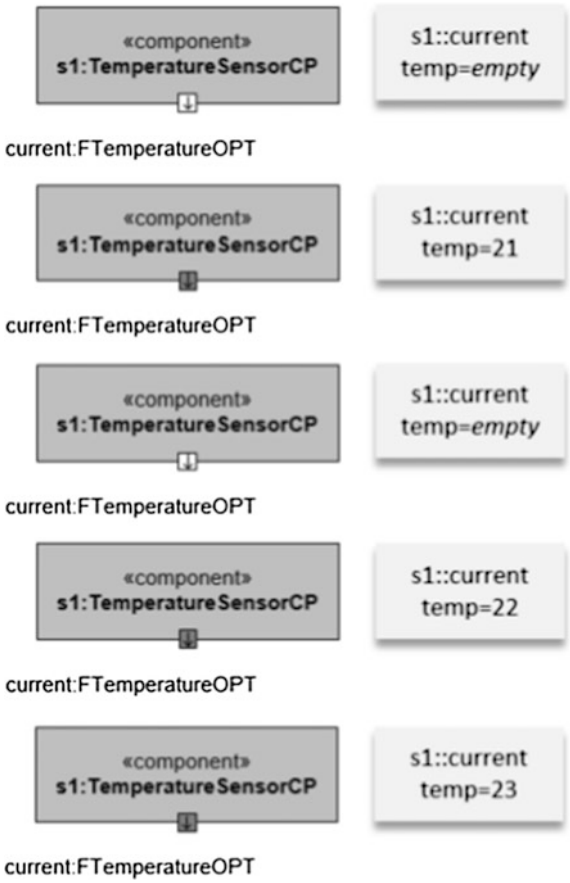


Fig. 7.3 The execution of a component

Execution semantics of simple components. The execution semantics of a composite component is given by the execution of its behavior as specified in an associated activity. That means the output data provided by the ports of a simple component must be in conformance to the specified the behavior considering the data in all its input ports.

For instance, considering the *sm* component in Fig. 7.4a. It has two input ports and one output port, all of them of *CTemperaturePT* type. As we have mentioned when describing the behavioral viewpoint (Chap. 5), a component starts its execution whenever all input ports receive values. Observing the component as a black box, once the input values in *s1* and *s2* ports are consumed, we see a value in the *average* port. This sequence is illustrated in Fig. 7.4. When a value is available in one of the ports—illustrated as dark gray in *s1* in Fig. 7.4—the component cannot begin its execution yet. The execution starts only when both *s1* and *s2*—Fig. 7.4c. In the next step Fig. 7.4d we can see the resulting value in the *average* port. We explain the execution of the internal behavior later in this chapter. The states of the

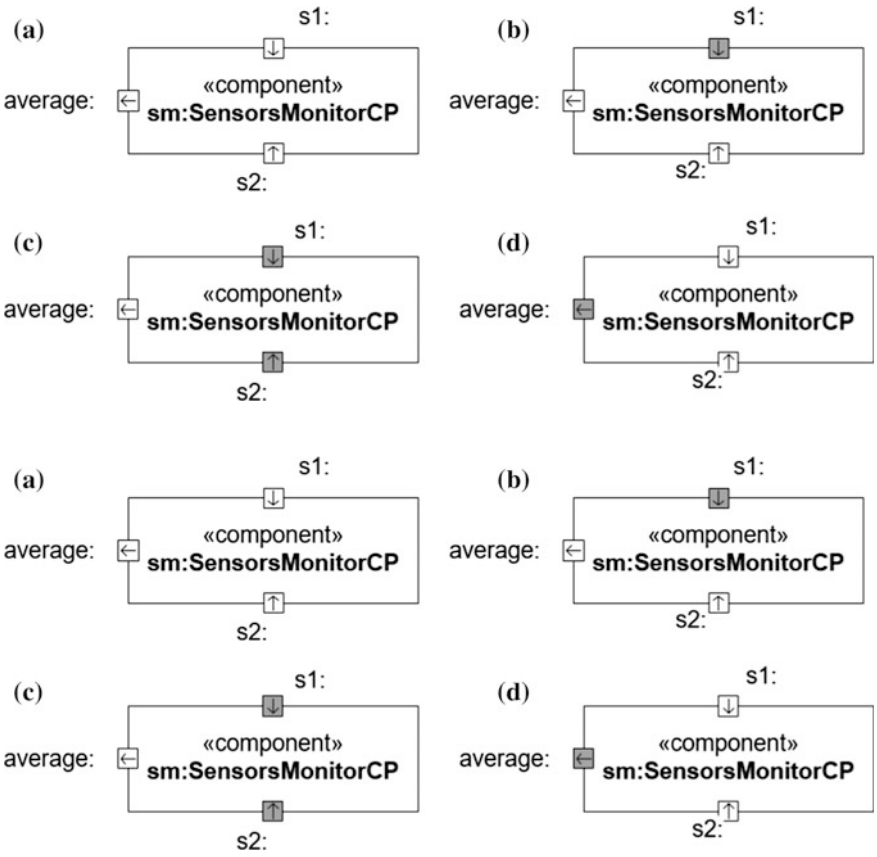


Fig. 7.4 The *sm:SensorsMonitorCP* execution sequence

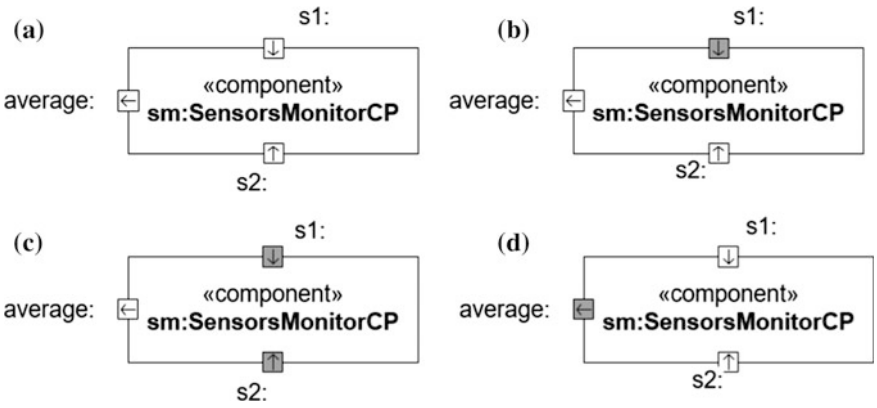


Fig. 7.4 (continued)

sm component in (a), (b), and (c) are all before the execution. The state (d) is reached after the execution. When the value in average is consumed, the *sm* component goes back to the state (a), (b), or (c), depending on the availability of data in its ports. The value in the *out* port must conform with the specified equations (constraint) associated with the component.

7.2.2 Executing Connectors and Delegations

Execution semantics of connectors. The execution semantics of a connector is given by the flow direction specification between its participant ports. Whenever there is a value in an *out* port that participates in a connector, that value flows through the connector to the participant ports. Figure 7.5 shows this sequence (from

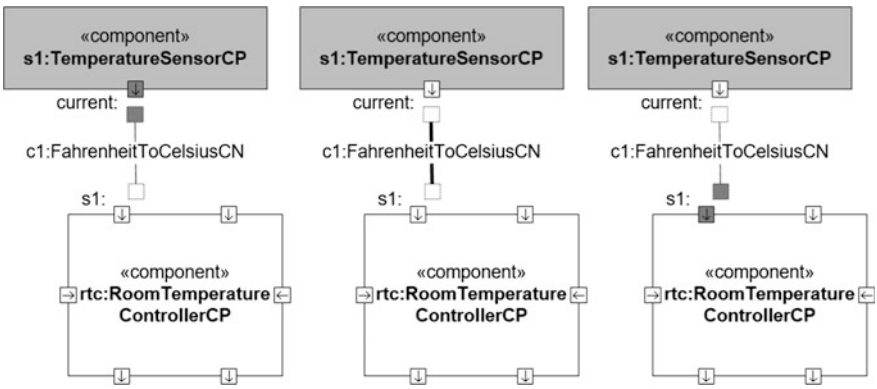


Fig. 7.5 The execution sequence of a connector

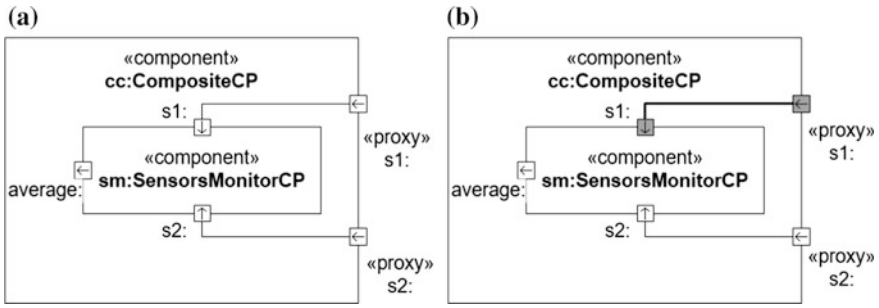


Fig. 7.6 Delegation execution sequence in composite components

left to right). We represent the port and its participation in the connector as two different figures, but they represent the same port: the port that belongs to a component is the same that participate in the connection (we use a dot line in the participant port).

In the leftmost figure, we show a value in the *current* out port of the *s1* component (shown in dark gray color). The same value is also represented in the connector as a participant port value. The next step shows the value flowing through the connector represented in strong dark line in the middle figure. The final step (the figure in the right) shows the value in the *s1 in* port of the *rtc* component.

Execution semantics of delegations. Each port of a composite component has a delegation to some of the ports of its internal component. The external port acts as a proxy. It is important to note that each port must be of the same type, e.g., same data type and direction. Whenever a value arrives in the proxy port of the composite component, it is immediately sent to the corresponding port. Figure 7.6 illustrates the execution of a delegation before (a) and after (b) a value arrives in the *s1* proxy port.

7.2.3 Executing Configurations

Composite component and connectors have a configuration of components and connectors. The semantics of its configurations are based on the semantics of each individual element.

Execution semantics of composite components. The execution of a composite component is given by the concurrent composition of the behavior of the interconnected component. Each individual component begins when all input ports have their values available. This condition depend on the semantics of the way those ports are connected to other components or by delegation to proxy ports. When a component has input ports connected to an external proxy ports its execution begins immediately when the proxy ports receive theirs values. When a component has

input ports connected to others, component execution of its ports depends on the execution semantics of the others output ports of the connected components.

In the RTC System, the execution of the *RoomTemperatureControllerCP* composite component (*rtc*) is given by the concurrent composition of the sequential behaviors of *pc*, *sm*, and *cm*, taking into account that they communicate only through the connectors *target* and *average* and interact only using its ports *s1*, *s2*, *detected*, *userTemp*, *heating*, and *cooling* according to the delegations.

7.3 Executing Activities

In this section, we explain the semantics of activities and its elements. Activities are the core elements to specify the behavior of component, connectors, and configuration, their operational semantics defines the core rules to the execution of an architecture description.

We first define the semantics of each activity element and then we illustrate the execution of an activity.

7.3.1 The Semantics of Activity Elements

Activities. We present the execution semantics of activities considering the execution of each element used in an Activity Diagram. The elements in that diagram are *flows*, *delegations*, *decisions*, *buffers*, *data stores*, and *actions*. We explain the semantics of each element individually, as illustrated in Table 7.1. A *premise* is the state before the execution. The *conclusion* is the state after the execution.

Flow. A flow is a specification that a value can flow from an element (except parameters, delegations or actions) to another element (except parameters, delegations or actions). The semantics of a flow is that when there is a value in an element it flows to the element in the other element. Table 7.1 shows the semantics in terms of premise and conclusion.

Delegation. A delegation specifies that a value in a parameter is directly also in the pin linked to it. Table 7.2 shows the premise (no value) and the conclusion (value in both the parameter and pin). The same semantics applies when a pin has a delegation to a parameter.

Decision. The semantics of a decision states that when there is a value in a pin that is part of a decision element, if the condition is true, the value flows to the true side

Table 7.1 The semantics of flow element

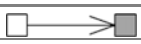
Name	Premise	Conclusion
Flow		

Table 7.2 The semantics of delegation element

Name	Premise	Conclusion
Delegation		

of the decision. If the condition is false it flows to the false side, as illustrated in Table 7.3.

Buffer. A buffer is an element that stores a value until it is consumed by another element. The semantics of a buffer is illustrated in Table 7.4. When there is a value in a pin, it flows to the buffer. The buffer stores it until it is consumed by another element that has a pin connected to it.

Datastore. A datastore is an element that stores a value even when it is consumed by another element. The semantics of a datastore is illustrated in Table 7.5. When there is a value in a pin, it flows to the datastore. It stores the value even if it is consumed by another element that has a pin connected to it.

Table 7.3 The semantics of decision element

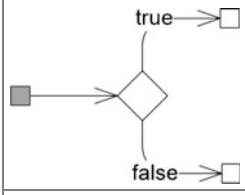
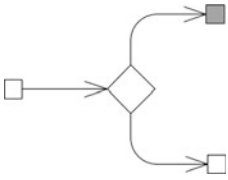
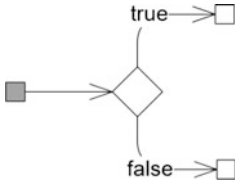
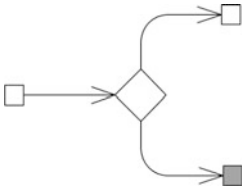
Name	Premise	Conclusion
Decision		
		

Table 7.4 The semantics of the buffer element

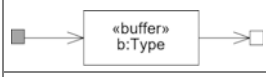
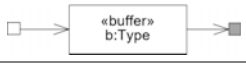
Name	Premise	Conclusion
Buffer		
		

Table 7.5 The semantics of the datastore element

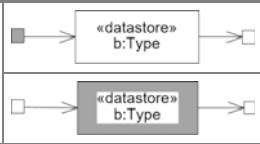
Name	Premise	Conclusion
Datastore		

Table 7.6 The semantics of action element

Name	Premise	Conclusion
Action		

Action. An action is a behavioral element that executes a body of statements using the input pins (parameters of the action) and returns a value to an output pin. Table 7.6 shows an illustration of the semantics of an action. When there is a value in the input pin (premise), the action begins to execute (conclusion). In the execution of an action (premise), the result is provided in the *out* pin (conclusion).

7.3.2 Executing an Activity

We now show the execution of a simple activity in the *RTC* System example. The *CommanderCP* component receives values from two components. The *SensorsMonitorCP* component sends the current average temperature in the room. The *PresenceCheckerCP* component sends the target temperature that is to be used to control the temperature in the room. The *CommanderCP* component uses these values to compare *target* and *current* temperature and to decide if it should turn the heater or the cooler *on* (or *off*).

Figure 7.7 shows the initial state of the activity execution with no value in the parameters. It is important to remember that each parameter in the activity can be values of the same type that can flow from or to the corresponding ports.

The *averageTemp* parameter in dark gray, in Fig. 7.8, represents a value available in the component *average* port. Considering the semantics of the delegation element, this value is immediately available in the corresponding pin in the *ct:CompareTemperatureAN* action. However, since there is no value in the other pin, the action does not start yet.

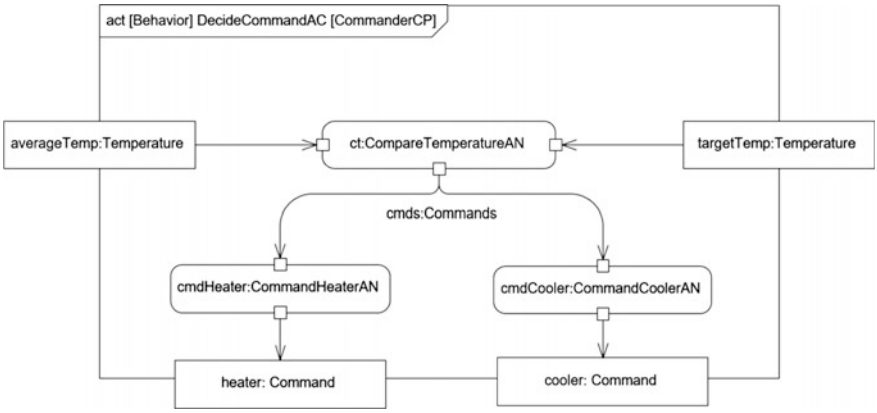


Fig. 7.7 Execution of the *DecideCommandAC* activity of the *CommanderCP* component—Step 1

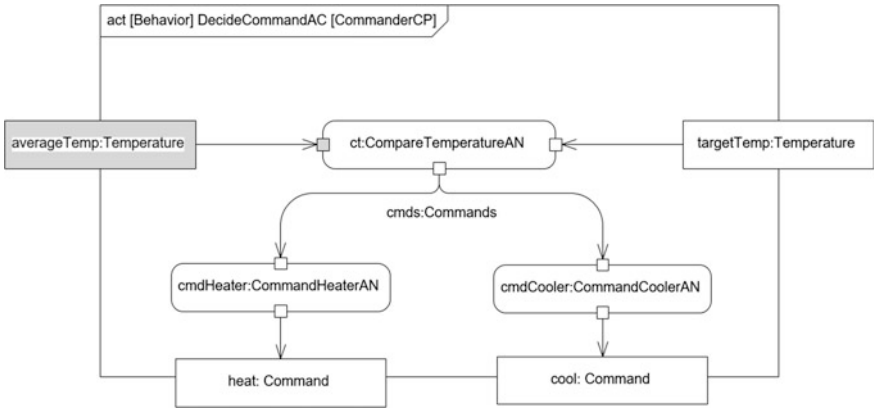


Fig. 7.8 Execution of the *DecideCommandAC* activity of the *CommanderCP* component—Step 2

Figure 7.9 shows when the value is in the *targetTemp* temperature it flows directly to the other pin. So, the *ct* action is ready to execute. Figure 7.10 represents the action executing in dark gray. When it finishes, the result is provided in the out pin (Fig. 7.11) and it simultaneously flows to the pins of the *cmdCooler* and *cmdHeater* actions (Fig. 7.12).

The two last steps show the *cmdCooler* and *cmdHeater* actions concurrently running (Fig. 7.13) and providing the *out* value through their respective pins and directly to *cool* and *heat* parameters (Fig. 7.14).

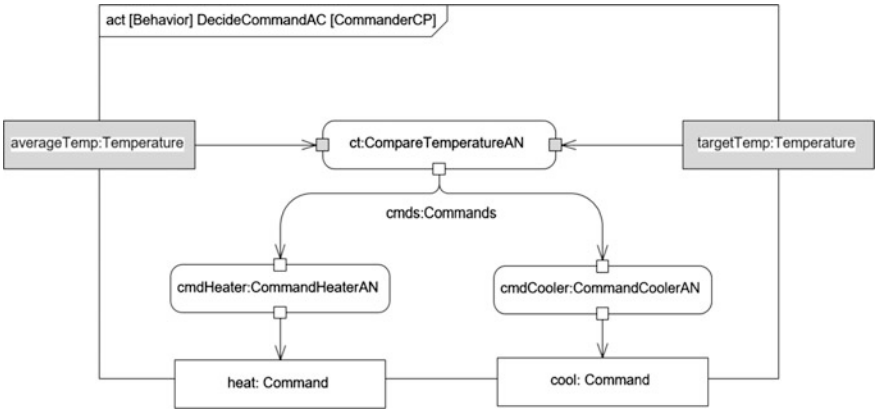


Fig. 7.9 Execution of the *DecideCommandAC* activity of the *CommanderCP* component—Step 3

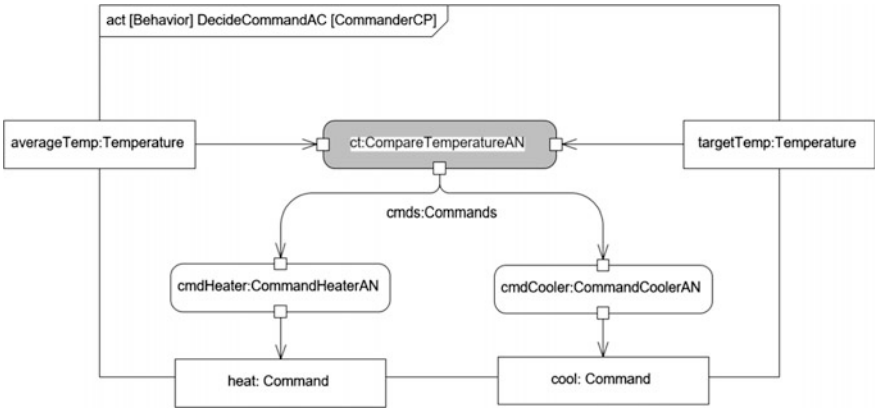


Fig. 7.10 Execution of the *DecideCommandAC* activity of the *CommanderCP* component—Step 4

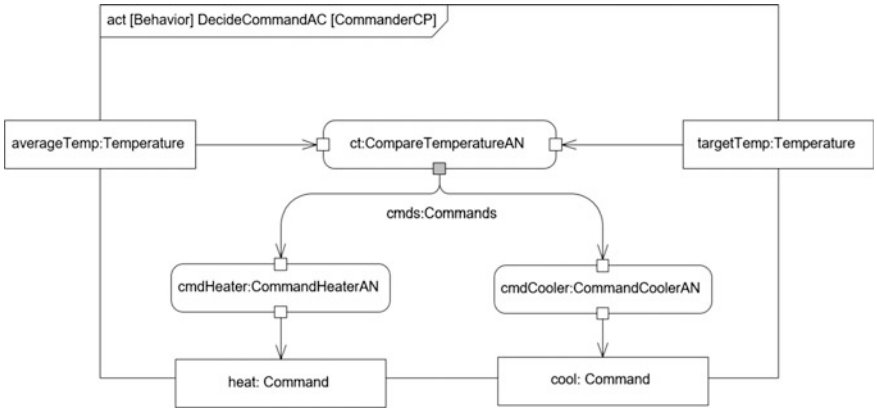


Fig. 7.11 Execution of the *DecideCommandAC* activity of the *CommanderCP* component—Step 5

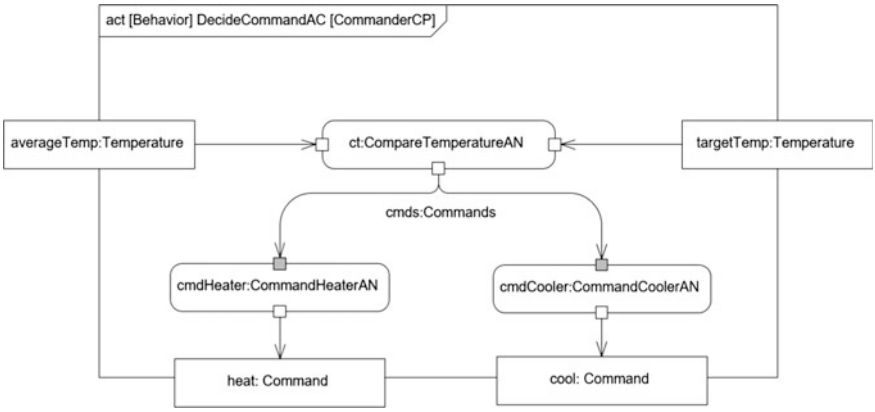


Fig. 7.12 Execution of the *DecideCommandAC* activity of the *CommanderCP* component—Step 6

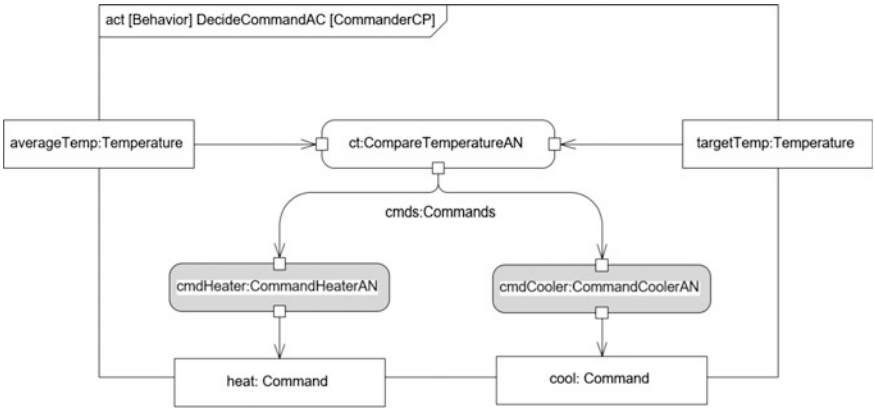


Fig. 7.13 Execution of the *DecideCommandAC* activity of the *CommanderCP* component—Step 7

7.4 Executing the RTC System Example

We now present a specific scenario of architecture execution to illustrate a complete example of a running architecture. In this scenario, the system is installed in a single room with two temperature sensors in different locations.

We consider the initial state of the running system a situation in which both the cooler and the heater are turned off, the room temperature is 80 °F in average, and a person is in the room and sets the temperature to 25 °C using the remote control. The temperature sensors continuously provide the values of the current temperature.

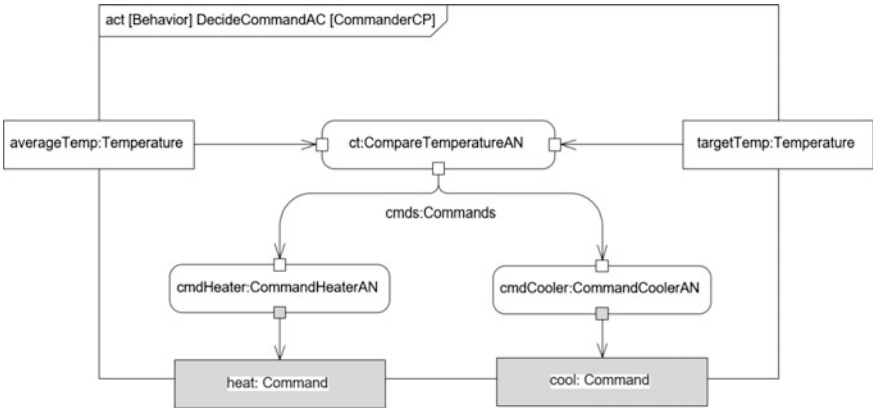


Fig. 7.14 Execution of the DecideCommandAC activity of the *CommanderCP* component—Step 8

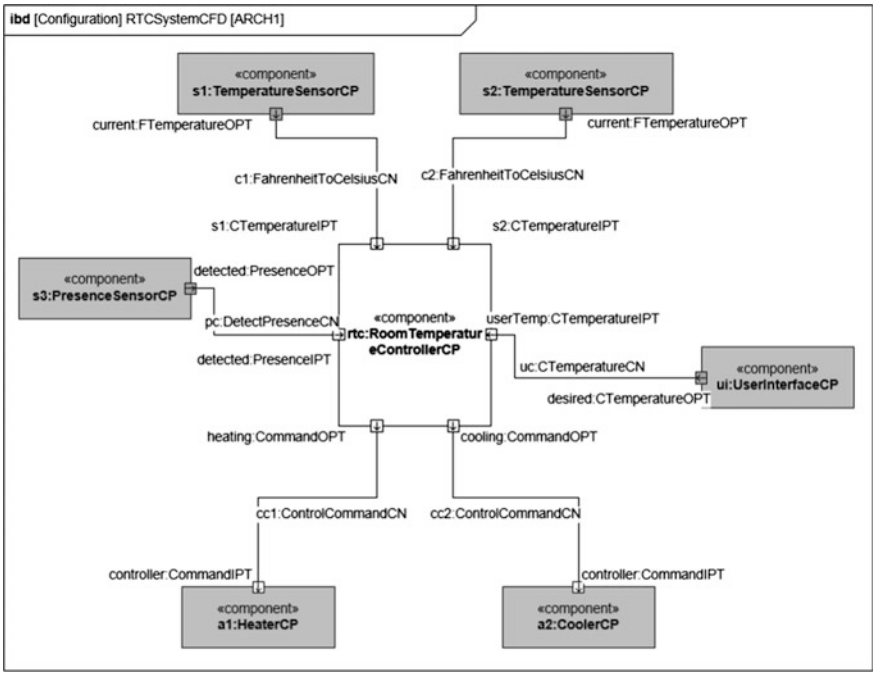


Fig. 7.15 Executing a scenario in the RTC system architecture—Step 1

Figure 7.15 shows that initial state. The *s1* and *s2* sensors provide the value 78 and 83, respectively, in their out port. The *ui* component provides the value 25 in the *desired* port and the *s3* presence sensor provides a true value in the *detected* port.

These four ports are connected to the *rtc* component. The *uc* and *pc* connectors are simple connectors and they allow the values to flow directly to their, respectively, connected ports (see Fig. 7.16). So, the *detected* and *userTemp* ports of the *rtc* component contain, respectively, the value *true* and *25*.

The current port of both *s1* and *s2* components are connected to *rtc* by the *c1* an *c2* connectors. These connectors are of type *FahrenheitToCelsiusCN* and their behavior is to convert the temperature in Fahrenheit to the temperature in Celsius. Thus, the value provided in the *localTemp1* port of the *rtc* component is 25.6 and in *localTemp2* of the same component is 28.3. As all values are in the four input ports of the *rtc* component, its execution can now start (see Fig. 7.17).

The execution of the *rtc* component begins with the concurrent execution of the *pc:PresenceCheckerCP* and *sm:SensorsMonitorCP* components since all their input ports are directly linked to the ports in the *rtc* component by delegations (see Fig. 7.18). The *sm* component provides a value 27 in its *average:CTemperature OPT* out port that is the average of the values provided in its input port (see Fig. 7.19). This result flows to the *average:CTemperatureIPT* port of the *cm* component through the *average:CTemperatureCN* connector (see Fig. 7.20).

The *pc* component receives the desired temperature (25) in its *userTemp* port and the *true* value in the *detect* port. The result of this activity is the 25 value in the

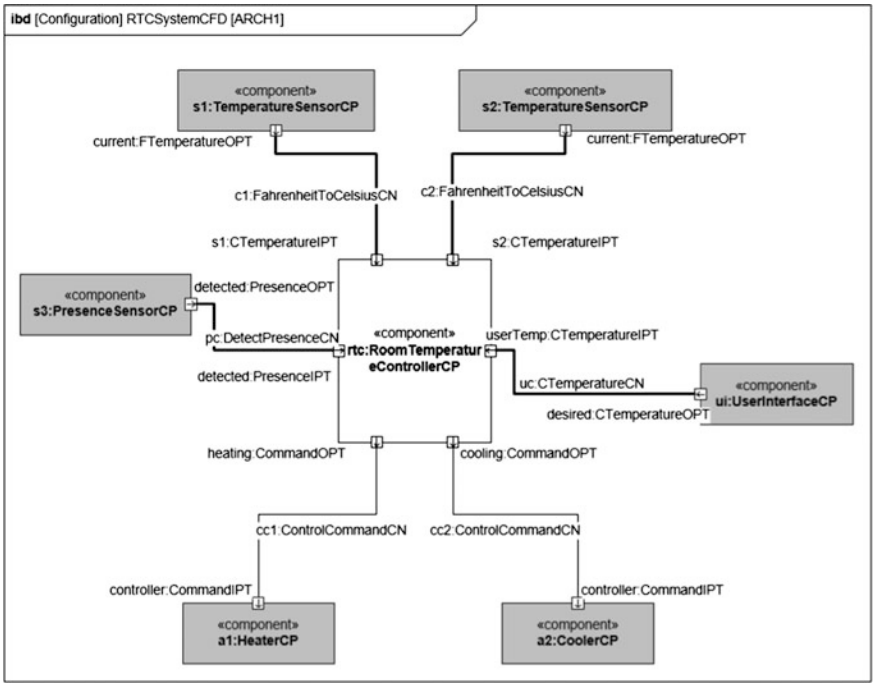


Fig. 7.16 Executing a scenario in the RTC system architecture—Step 2

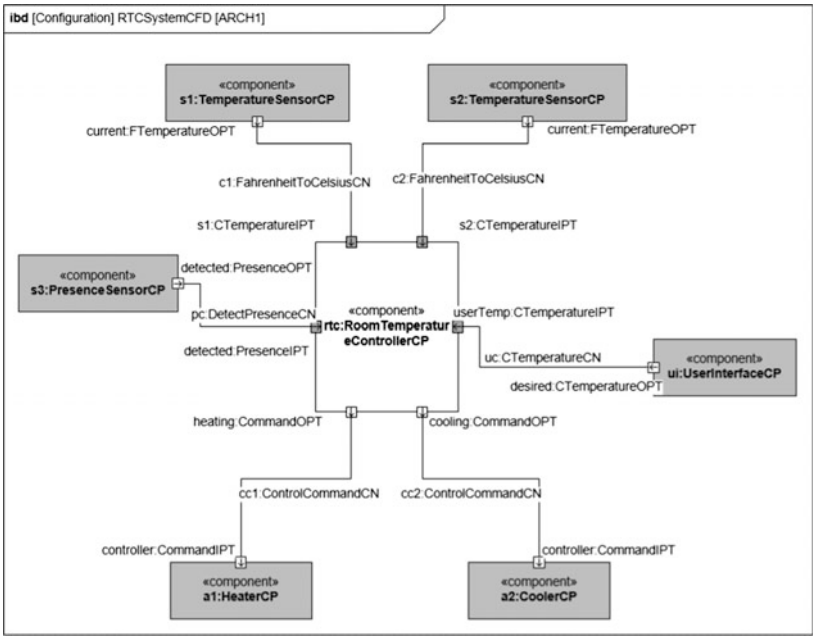


Fig. 7.17 Executing a scenario in the RTC system architecture—Step 3

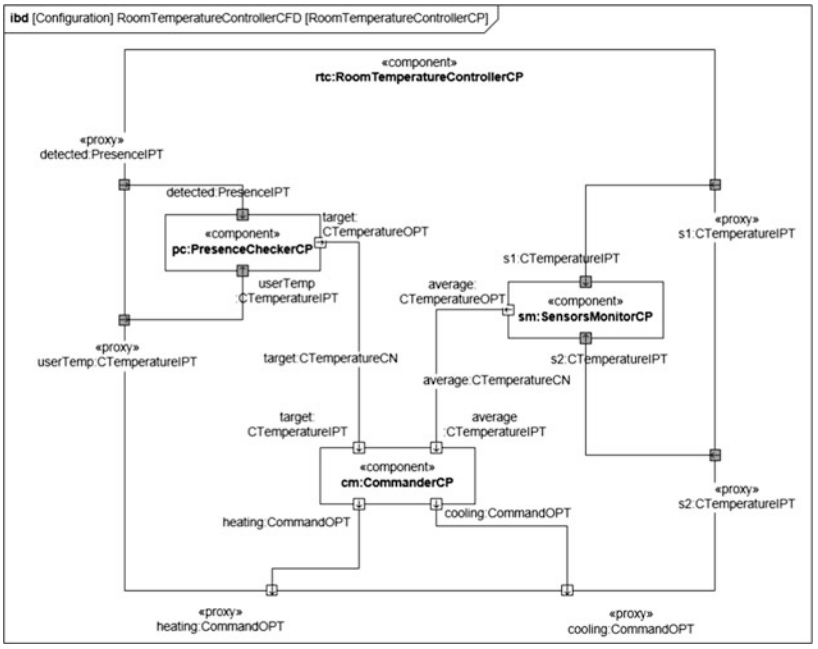


Fig. 7.18 Executing the RTC component internal elements—Step 1

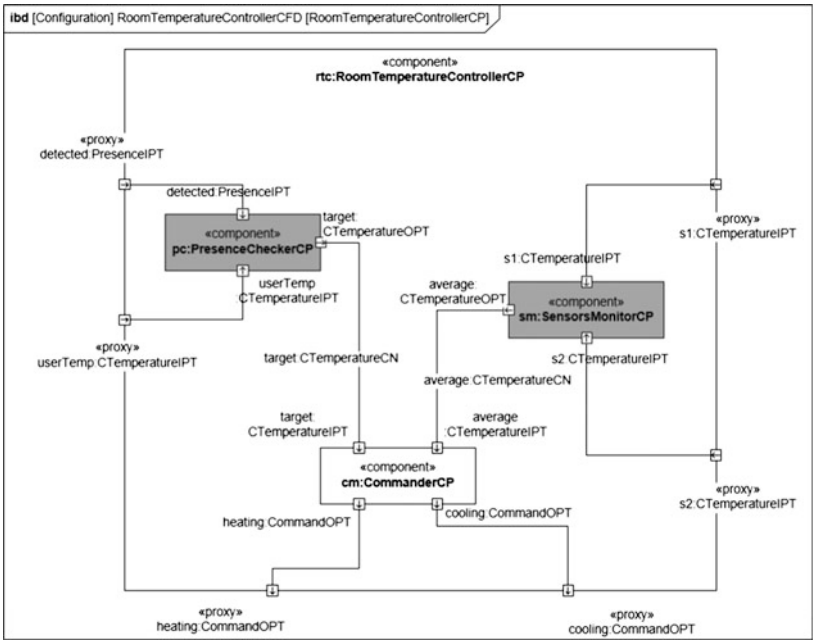


Fig. 7.19 Executing the RTC component internal elements—Step 2

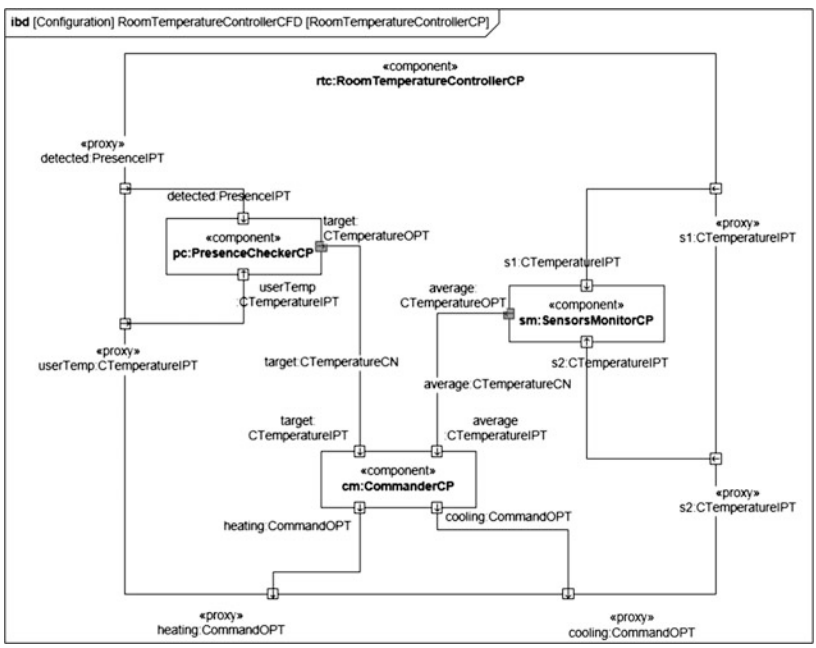


Fig. 7.20 Executing the RTC component internal elements—Step 3

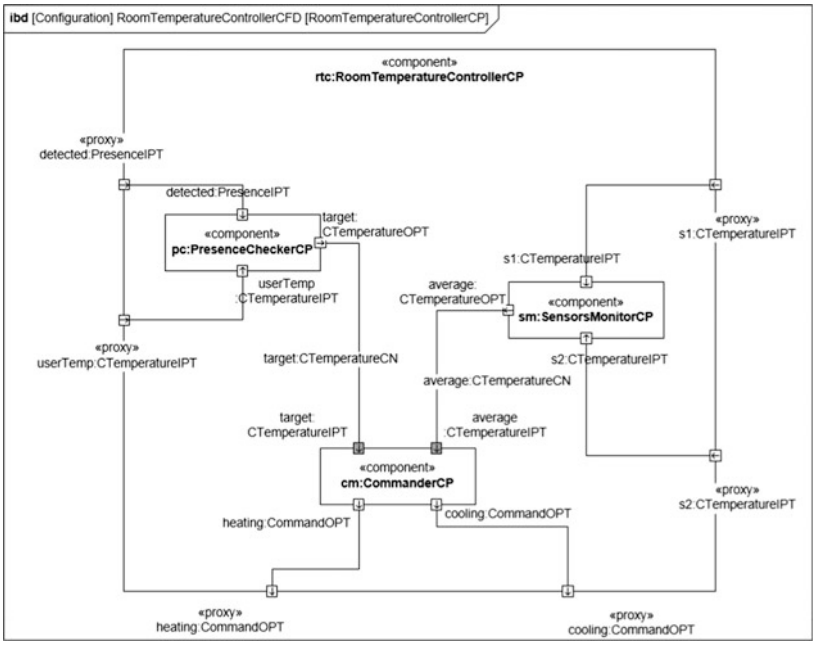


Fig. 7.21 Executing the RTC component internal elements—Step 4

target port. This value flows to the *target:CTemperatureIPT* port of the *cm* component through the *target:CTemperatureCN* (also in Fig. 7.21).

The *cm* component begins its execution when these two values are available in its *target* and *average* ports (Fig. 7.22). We explained the execution of this component in Sect. 7.3.2 showing the behavior of the *DecideCommandAC* activity. The results of the execution are the commands in the *heating* and *cooling* ports of the *cm* component and also in the *rtc,* respectively, proxy ports (see Figs. 7.23 and 7.24). The values in these ports are, respectively, *Command::off* and *Command::on*.

In Fig. 7.24, we see again the rtc component connected to the boundary components and the values represented in the out ports. These ports—*heating* and *cooling*—are connected to the *a1:HeaterCP* and *a2:CoolerCP* components. The values flow to each of them (see Fig. 7.25) and, in this case, the command values keep the heater off and cooler on. We finish this scenario showing the values in the *commander* ports of both *a1* and *a2* in Fig. 7.26.

Note that we presented a view of the architecture execution where we accompanied the execution sequentially. It is important to highlight that, the architecture being composed of concurrent behaviors, in parallel, with this scenario the sensors

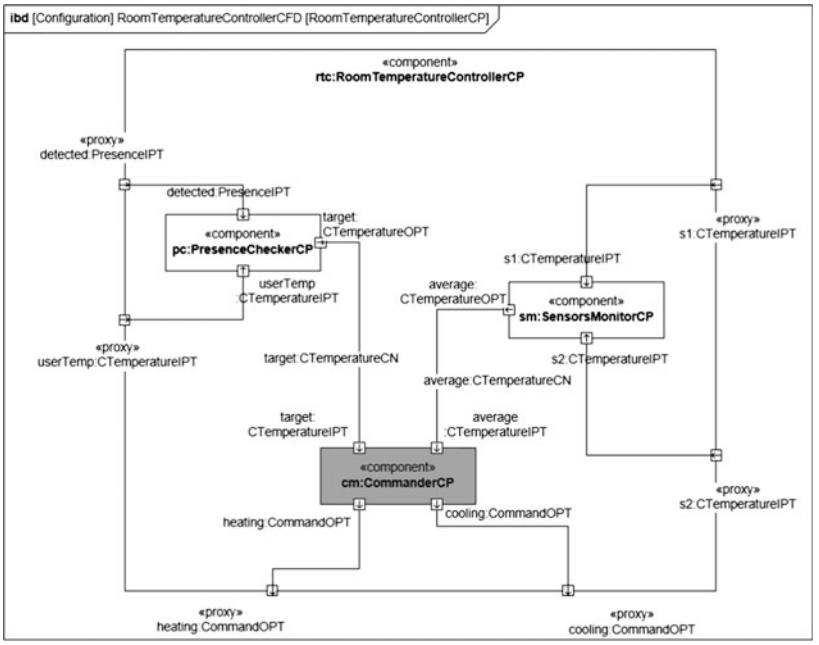


Fig. 7.22 Executing the RTC component internal elements—Step 5

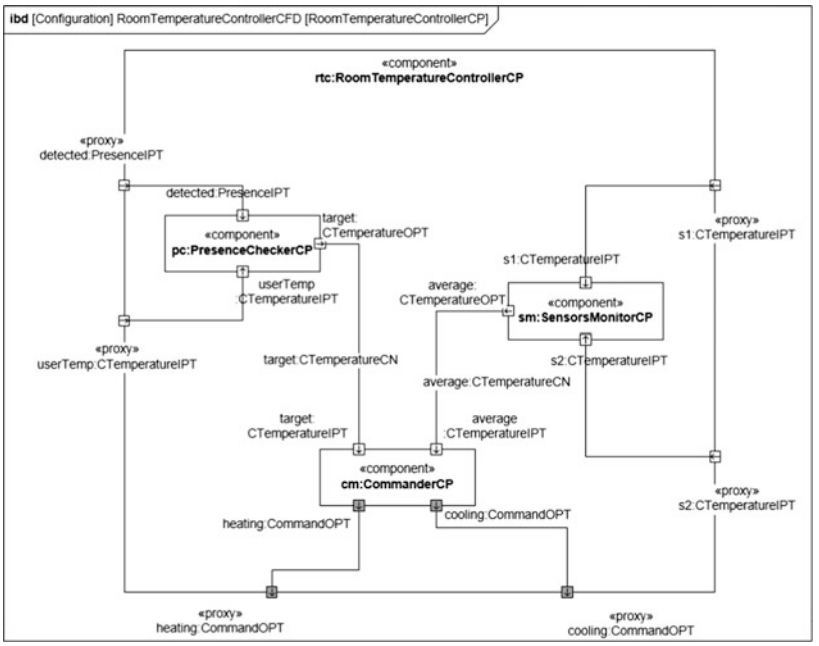


Fig. 7.23 Executing the RTC component internal elements—Step 6

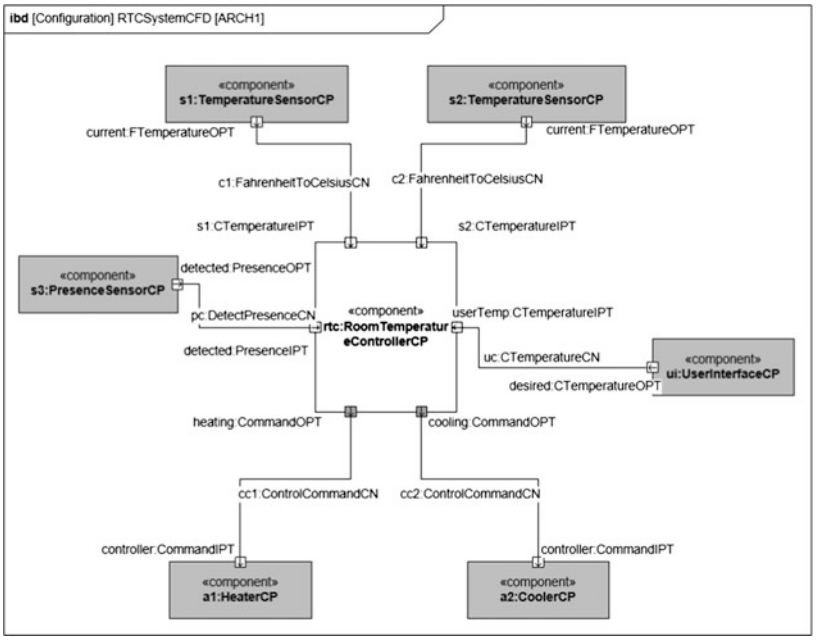


Fig. 7.24 Executing a scenario in the RTC system architecture—Step 4

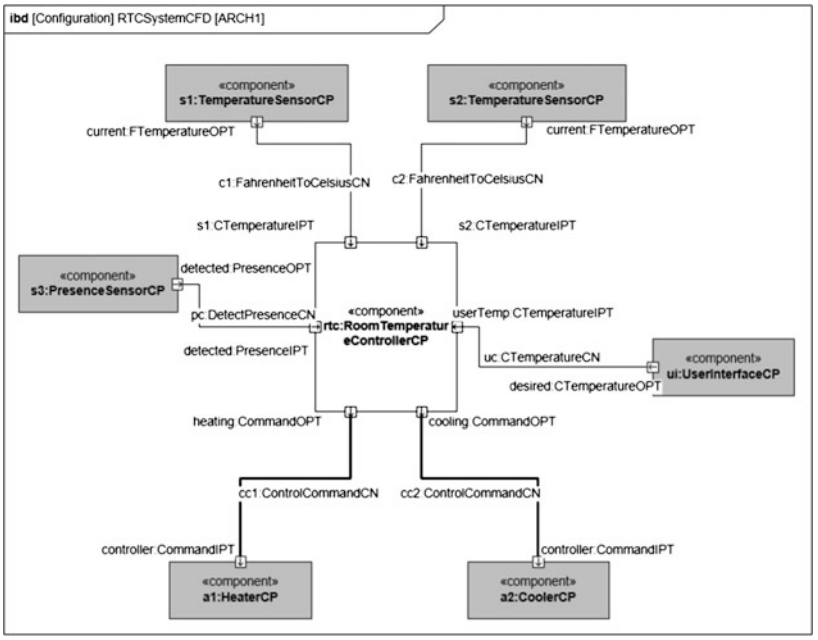


Fig. 7.25 Executing a scenario in the RTC system architecture—Step 5

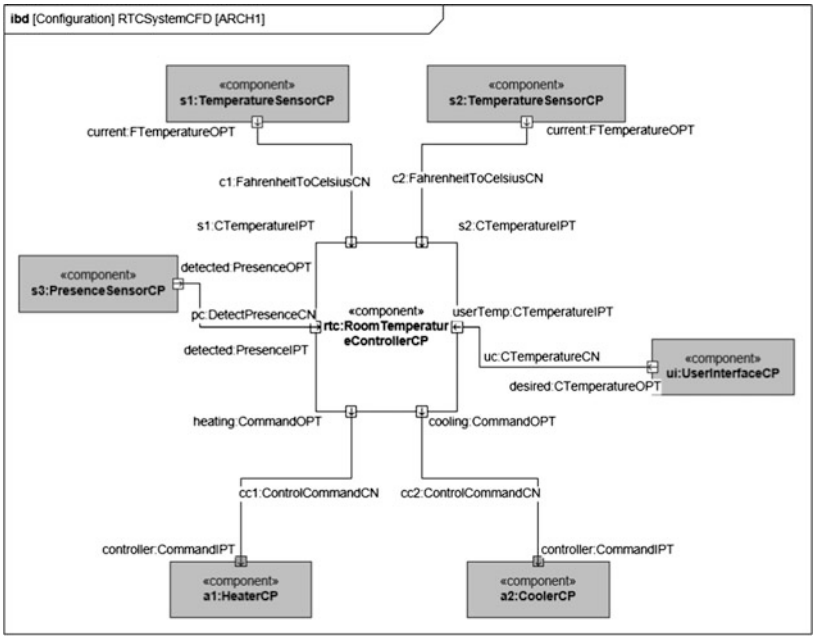


Fig. 7.26 Executing a scenario in the RTC system architecture—Step 6

could have provided new values of temperatures some instants later which will also result in new commands for the heater and the cooler. This RTC architecture thereby continuously and concurrently senses values of temperature, of presence/absence of a person in the room and possibly new values of desired temperatures resulting in commands for the heater and the cooler.

7.5 Summary

In this chapter, you have learnt the following:

- how an architecture execution is expressed with SysADL;
- the operational semantics of the SysADL constructs;
- a scenario of an architecture execution of our running example.

Further Reading

1. Concrete Syntax For A UML Action Language: Action Language For Foundational UML™ (ALF™). http://www.omg.org/spec/ALF/
2. Semantics Of A Foundational Subset For Executable UML Models (FUML™). http://www.omg.org/spec/FUML/
3. Precise Semantics Of UML Composite Structures™ (PSCS™). http://www.omg.org/spec/PSCS/1.0/

Part II
Quality-Based Architectures

Chapter 8
Introduction to Quality-Based Architectures

Part I introduced the modelling of requirements and presented how to describe a software architecture to meet functional requirements. In Part II, we will present how to design a software architecture with SysADL to satisfy nonfunctional requirements known as quality attributes.

You will learn the following:

- the concept of quality;
- the concept of quality-based architectures;
- the activities involved in analysing quality-based architectures;
- the introduction to three quality attributes that will be addressed in the next chapters.

8.1 What Is a Quality

As we saw in Chap. 3, a functional requirement is a service that a system must provide. An example of a functional requirement in the *RTC System* is that the controller must maintain the temperature in a room.

As we also saw in Chap. 3, a quality is a nonfunctional requirement that a system must satisfy. More precisely, the quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value [ISO 25010].

Several architectures can be designed to satisfy functional requirements. However, these architectures may vary in the degree in which the non-functional requirements are satisfied. For instance, one of these architectures can be easily modified and another could have a high cost to allow modifications.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_8

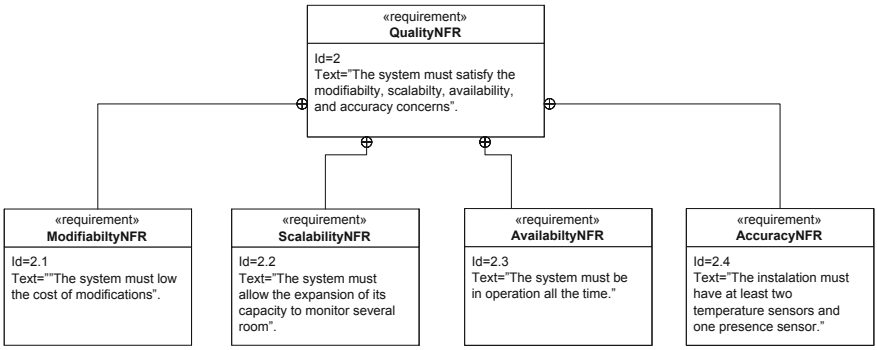


Fig. 8.1 The nonfunctional requirements of the RTC System

In our running case study, the *RTC system* has the following nonfunctional requirements, which quality attributes: (i) the architecture must minimize the cost of modifications; (ii) the architecture must allow the expansion of its capacity to monitor several rooms; and (iii) the system must be in operation all the time.

Figure 8.1 shows the nonfunctional requirements elicitated previously in Chap. 3.

8.2 What Is Quality-Based Architectures

A quality-based architecture is one designed to satisfy a quality attribute or a set of quality attributes. In most cases, it is impossible to maximize all of them and the architect should consider a trade-off. For instance, consider an architecture that must satisfy both security and performance requirements. An architecture to increase security must add architectural elements to implement the security policies. For instance, adding an architectural element to encrypt information that is transmitted between components. That architecture decision implies in performance degradation. For addressing the trade-off, the architect should design an architecture that both security and performance has acceptable values.

8.3 Analysing Quality-Based Architectures

The analysis of quality-based architectures involves the following activities:

- To derive an **architectural requirement** for each nonfunctional requirement related to quality attribute;
- To define the **causes** for each architectural requirement;
- To analyse the **ripple effects** in different architectures for the same cause.

Indeed, quality attributes can be defined as a cause–effect relationship, so we need to express both the causes and the effects.

For instance, considering the quality attribute of modifiability applied to our *RTC System*, one possible architectural requirement is that the system must allow the modification of the temperature sensors to improve temperature measures.

Two possible modification causes related to this requirement are: (C1) to add a new temperature sensor of a defined type to improve accuracy, or (C2) to replace a temperature sensor for another one of a new type, e.g., to support other temperature units. Many other causes can also be included.

In terms of ripple effect associated to the C1 cause includes: to replace the room temperature controller by another that is a specialization of the previous one with a new temperature port.

8.4 Quality Attributes: Modifiability, Scalability, and Fault Tolerance

This Part II focuses on discussing three quality attributes: Modifiability, Scalability, and Fault Tolerance.

Modifiability refers to the degree to which a system can be effectively or efficiently modified without introducing defects or degrading existing system quality. In Chap. 9, we discuss architectural requirements involved in the modifiability for the *RTC System* with the causes and effects on other architectural elements (ripple effect). We present an example of modifiability where a Fahrenheit temperature sensor is replaced by a Celsius temperature sensor. We also analyse the cost of modifying two architectures of the *RTC system*, i.e., the number of additional architectural elements that need to be modified when a given modification is needed (ripple effects).

Scalability refers to the degree to which a system can effectively and efficiently be scaled for increasing operational usage. In Chap. 10, we discuss architectural requirements involved in scalability, with the causes and effects on other architectural elements. We explain how an architecture of the *RTC system* can be scaled to cope with an increasing number of component usage that may be of several orders of magnitude.

Fault tolerance refers to the degree to which a system can maintain operational service even in the presence of faults. In Chap. 11, we discuss architectural requirements involved in fault tolerance, with the causes and effects on other architectural elements. We will analyse the fault tolerance on two different architectures of the *RTC system*.

8.5 Summary

In this chapter, you learned the following:

- the quality concept;
- the quality-based architecture concept;
- the activities to analyse quality-based architectures.

Further Reading

1. Bass, L., Clements, P., Kazman, R.: Software Architectures in Practice, 2nd ed. Addison Wesley, Reading (2003)
2. Gorton, I.: Essential Software Architecture. Springer (2011)
3. Mistrik, I., Bahsoon, R., Eeles, P., Roshandel, R., Stal, M.: Relating System Quality and Software Architecture. Elsevier (2015)

Chapter 9
Designing Modifiability in Software Architectures

In this chapter, we present modifiability as an architectural quality and how to express modifiability in software architecture. We explain how to analyse architecture to evaluate its modifiability and how to apply tactics to improve modifiability.

You will learn the following:

- what is modifiability as an architectural quality;
- what are the architectural causes and effects of modifiability;
- tactics to improve modifiability;
- a taxonomy of modifiability primitives; and
- a comparison technique to evaluate the modifiability in alternative architectures.

9.1 Introduction

Some nonfunctional requirements are expressed as quality attributes in the architecture. Modifiability is a quality to refer to the degree to which a system can be effectively or efficiently modified without introducing defects or degrading existing quality [1].

Examples of modifiability in the case of a room temperature controller system, as defined in our running case of the *RTC System*, are

- to maintain its functionality when adding a temperature sensor to improve accuracy;

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_9

- to remove a temperature sensor to remove a damaged sensor; or
- to replace a temperature sensor to change a Fahrenheit temperature sensor to a Celsius sensor.

The modification in an element can impact other elements (ripple effect) requiring new modifications in them. The *degree of modifiability* is related to the number of elements that need to be modified. The best case is to have no additional modifications in the architecture.

9.2 Expressing Modifiability Using Software Architecture Concepts

9.2.1 Modifiability Causes and Effects

Modifiability can be defined as a cause–effect relationship, so we need to express the causes and the effects.

Architectural causes refer to identify the need of architectural modifications addressing nonfunctional requirements. *Architectural effects* refer to the impact of each modification in other elements of the architecture. All causes of modification need to be identified and classified. For each cause, all effects need to be identified and quantified.

Modifiability Causes. We need to consider causes that may require modifications both in the use and in the definition of elements. *Modifications in the use* mean to add, remove, and replace an element in a defined architecture without modifying their types. *Modifications in the definition* mean to add, remove, and replace, an element based on a new type definition.

In the RTC system, an example of modification in the use is to add a new *Fahrenheit temperature sensor* to increase measure precision. An example of modification in definition is to add a new port to the sensor monitor component type to receive data from a new temperature sensor of a new temperature unit, e.g., Kelvin.

Modifiability Effects. Effects are the consequences of the causes in architecture. We consider the effects of modifiability in terms of the impact on the effectiveness and efficiency of the system. Effectiveness refers to meeting the purpose of the system. Efficiency refers to meeting this purpose optimizing the use of resources.

According to the requirements of the *RTC System*, effectiveness is to assure that the control-loop correctly adjust the room temperature according to the user desired temperature, the current average temperature and the presence of a person. Efficiency refers to the resources used to adjust the temperature.

9.2.2 Modifiability Quality Attributes

Modifiability quality attributes refers to a quality used to quantify the effects.

The *ripple effect* refers to a spreading effect caused by a modification in an architectural element. As an example, if we add a new sensor, we need to add a new port to the sensor monitor, to create a new connector, and to connect the sensor to the monitor. To add that new port, we could need to modify the sensor monitor definition.

The cost of modifications refers to the global cost implied by the modifications. As an example, if we add a new sensor, the cost of this modification includes the cost of adding the sensor, the cost of modifying the architecture to insert the new sensor, the cost of refining the architecture towards the implementation, and the cost of deploying the system.

9.2.3 A Classification of Modifiability Effects

We can express modifiability effects using action primitives.

When considering definitions, the following actions should be considered:

- add, remove, or replace a component definition,
- add, remove, or replace a port type, a connector definition,
- add, remove, or replace a value definition,
- add, remove, or replace a configuration definition.

When considering configuration definitions, the following actions should be considered:

- add, remove, or replace a component,
- add, remove, or replace a connector,
- add, remove, or replace a port in a component.

When considering component definitions, the following action should be considered:

- add, remove, or replace a port.

When considering the use of elements, the following actions should be considered:

- add, remove, or replace a component in a configuration,
- add, remove, or replace a port in a component,
- add, remove, or replace a connector between the ports of components in a configuration.

9.2.4 Examples of the Add Primitive

Let us take the architecture described in part I. We call it ARCH1.

We can add existing instances using the *add* primitive. To add an existing port (of type *CTemperatureIPT*) in a component definition (*RoomTemperature ControllerCP*) we use

- add *localTemp3*:*CTemperatureIPT* in *RoomTemperatureControllerCP* type.

We can also add instances in other instances. To add a component (of type *TemperatureSensorCP*) in a configuration (*RTCSystemARCH*):

- add *s3*:*TemperatureSensorCP* in *RTCSystem* of *ARCH2* configuration.

To add a connector (of type *FahrenheitToCelsiusCN*) in a configuration:

- add *c3*:*FahrenheitToCelsiusCN* between *localTemp3*:*CTemperatureIPT* of *rtc*: *RoomTemperatureControllerCP* and *current*:*FTemperatureOPT* of *s3*:*TemperatureCP*.

Figure 9.1 shows the new configuration with those additional elements. Later in this chapter, we present all these modification in details.

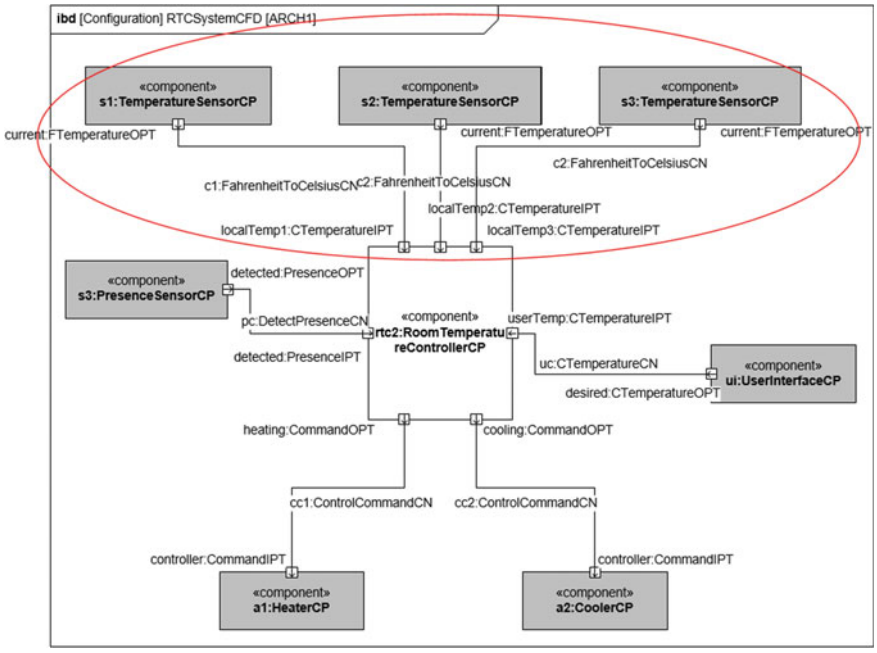


Fig. 9.1 The new configuration of the RTC system ARCH1

9.3 Modifiability Tactics

Tactics are design decisions that influence quality attributes. Tactics are used to minimize effects.

9.3.1 Using Modifiability Tactics

In the sequel, we present two modifiability tactics (T1 and T2):

T1—localize modifications

To assign responsibilities to elements during design such that anticipated changes will be limited in scope.

As an example, in the *RTC System*, we can design a component to each responsibility: a component to monitor sensor, another to monitor presence, and a third component to control the heater and cooler. Thus, the modifications will be localized in the component that is responsible for monitoring or controlling.

T2—minimize ripple effects

To avoid that a modification in an architectural element causes a series of other modifications.

As an example, in the *RTC System*, we can design a component using a multiplex port: a monitor sensor temperature having a multiplex port does not need to be modified when adding a new sensor.

9.4 Analysing Modifiability in the RTC System

In this example, we present two modifiability requirements. For each requirement, we enumerate the causes and then we depict the effects. Our analysis is not complete. We explain the analysis for one cause in two different architectures of the RTC system.

First, we analyse the *ARCH1* architecture. This architecture has two simple ports in the controller to receive data from two sensors. Then, we design and analyse the *ARCH2* architecture. This architecture has one port that allows multiple connectors.

9.4.1 RTC System Requirements

We use an AND-OR tree to depict the requirements (R), causes (C) and effects (E).

In the *RTC System*, the following modifications are required:

- R1—The system must allow the modifications of the temperature sensors to improve temperature measures,
- R2—The system must allow the modification of the user interface to display the current average temperature.

9.4.2 RTC System—Causes

We express the following alternative causes (C) for the first requirement (R1):

- C1—Add a new temperature sensor of a defined type to improve accuracy,
- C2—Add a new temperature sensor of a new type to include sensors supporting other temperature metric systems,
- C3—Replace a temperature sensor of the same type,
- C4—Replace a temperature sensor for a new type to improve accuracy or to support other temperature units,
- C5—Remove a temperature sensor when it is not needed anymore.

Considering the second requirement (R2), the cause to be considered is:

- C6—Replace the user interface.

9.4.3 Analysing the Ripple Effect in ARCH1

We first analyse the ripple effect on two different architectures of the *RTC* System considering the same cause of modification. The modification is to improve temperature measures (Requirement *R1*) by adding a new sensor to the *RTC System* (Cause *C1*).

The effects to cause *C1* in *ARCH1* are described below:

C1—add a new temperature sensor component of a defined type to improve accuracy implies to (AND),

- E1—add a new *RoomTemperatureControllerCP* component type, which is to (AND),
  - E1.1—add a new *RoomTemperatureControllerCP* as a subtype of *Room TemperatureController*,
  - E1.2—add the *localTemp3*:*CTemperatureIPT* port in the new *Room TemperatureControllerCP* (see Figs. 9.3 and 9.4).

- E2—add a new *SensorsMonitorCP* component type, which is to (AND)
  - E2.1—add a new *SensorsMonitorCP* as a subtype of *SensorsMonitorCP*,
  - E2.2—add the *s3*:*CTemperatureIPT* port to the *SensorsMonitorCP* component type (Fig. 9.5).
- E3—modify the *RoomTemperatureControllerCP* configuration (see Fig. 9.6), which is to (AND)
  - E3.1—replace the *sm*:*SensorsMonitorCP* component by the *sm*:*NewSensors MonitorCP* component,
  - E3.2—add a delegation between *s3* and *localTemp3*.
- E4—modify the *ARCH1* configuration (see Fig. 9.1), which is to (AND)
  - E4.1—replace *rtc*:*RoomTemperatureControllerCP* by *rtc2*:*RoomTemperature ControllerCP*,
  - E4.2—add the *s3*:*TemperatureSensorCP* component in *ARCH1* configuration,
  - E4.3—add *c3*:*FahrenheitToCelsiusCN* between *localTemp3*:*CTemperature IPT* of *rtc2*:*RoomTemperatureControllerCP* and *current*:*FTemperatureOPT* of *s3*:*TemperatureCP*.

Figure 9.2 shows the cause C1 and its corresponding effects in an AND-OR tree.

Quantifying the modifications

Considering the analysis above, we have nine modifications: seven add actions and two replace actions.

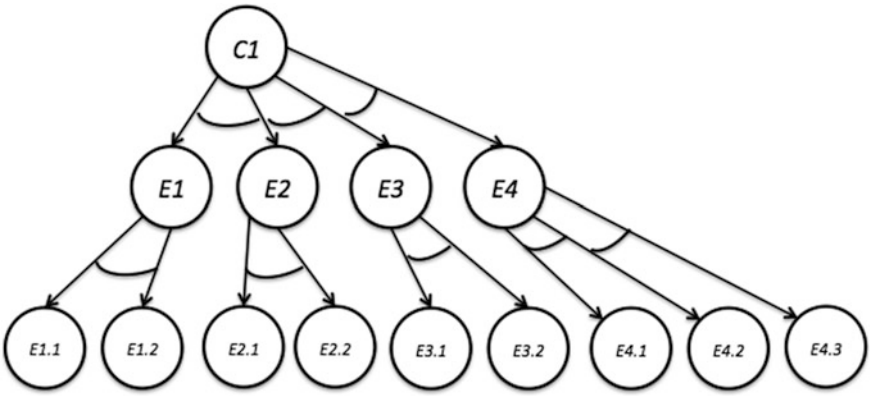


Fig. 9.2 A AND-OR tree of the C1 cause and the corresponding effects

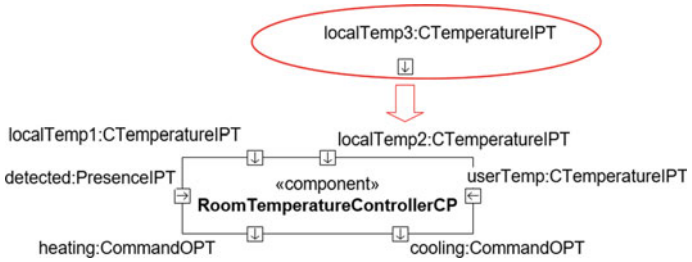


Fig. 9.3 Adding a new port to the *RoomTemperatureControllerCP* definition

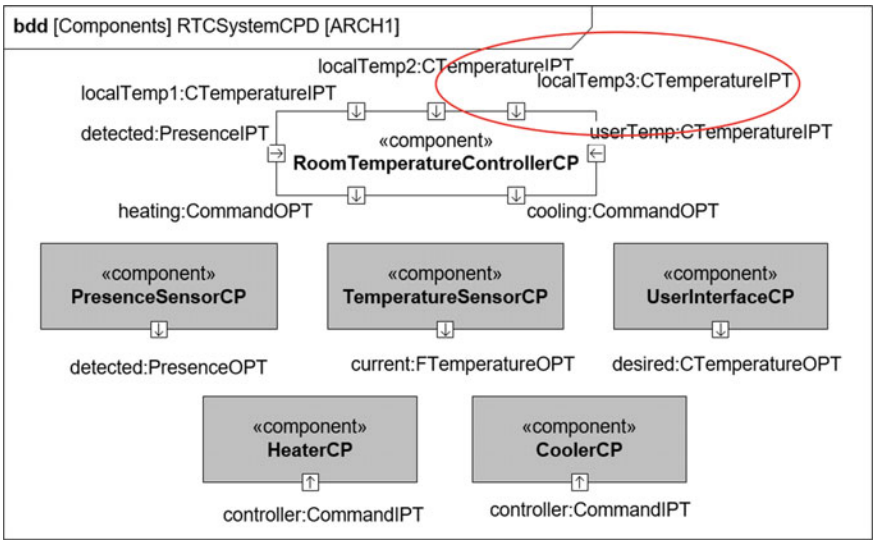


Fig. 9.4 The new BDD showing the new port

Summarizing, the cause C1 states that to add a new temperature sensor we need to perform the effect actions E1, E2, E3, and E4. Let us now explain each of the effects.

E1 states that we need to define a new *RoomTemperatureControllerCP*. In other to do it, we must add a new *RoomTemperatureControllerCP* as a subtype of *RoomTemperatureController* and add the *localTemp3*:*CTemperatureIPT* port in the new *RoomTemperatureControllerCP*. Figure 9.3 illustrates these modifications.

Figure 9.4 shows the new BDD showing the new created port.

E2 states that we need to add a new *SensorsMonitorCP* component *definition*, which involves adding a new *SensorsMonitorCP* as a subtype of *Sensors MonitorCP* and add the *s3*:*CTemperatureIPT* port to the *SensorsMonitorCP* component definition. Figure 9.5 illustrates an instance of this new component.

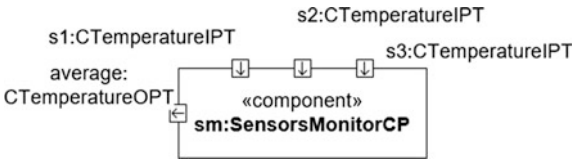


Fig. 9.5 An instance of a new *SensorsMonitorCP*

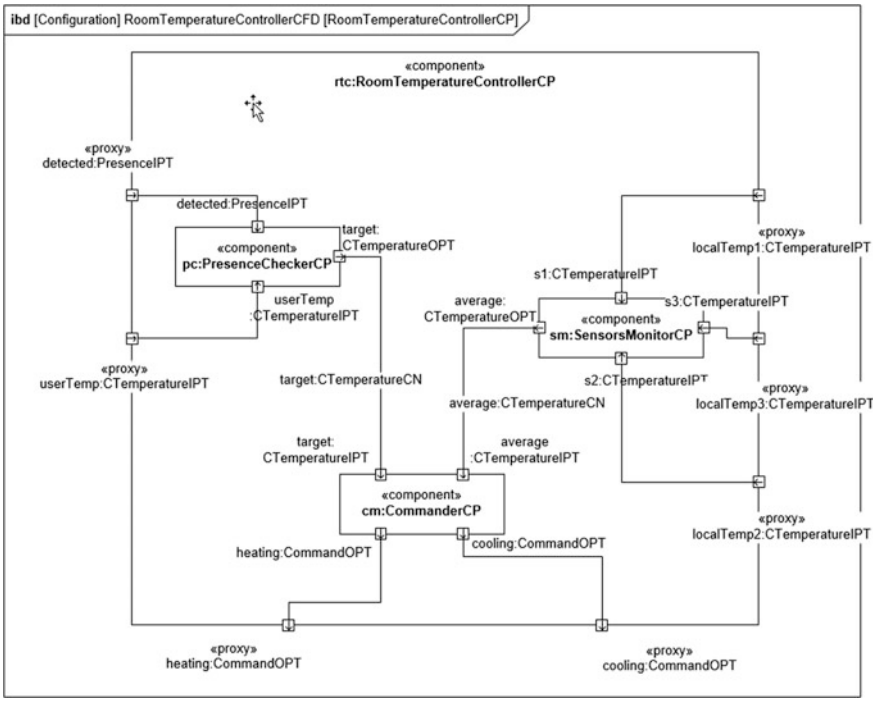


Fig. 9.6 The configuration of the *RoomTemperatureControllerCP* component

E3 states that we need to modify the *RoomTemperatureControllerCP* configuration, which is to replace the *sm*:*SensorsMonitorCP* component by the *sm*:*NewSensorsMonitorCP* component and to add a delegation between *s3* and *localTemp3*, as illustrates the Fig. 9.6. The figure shows also the whole configuration of the *RoomTemperatureControllerCP*.

E4 states that we need to modify the *ARCH1* configuration, which is to replace *rtc*:*RoomTemperatureControllerCP* by *rtc2*:*NewRoomTemperatureControllerCP*; to add the *s3*:*TemperatureSensorCP* component in *ARCH1* configuration; and to add *c3*:*FahrenheitToCelsiusCN* between *localTemp3*:*CTemperatureIPT* of *rtc*:*RoomTemperatureControllerCP* and *current*:*FTemperatureOPT* of *s3*:*Temperature*

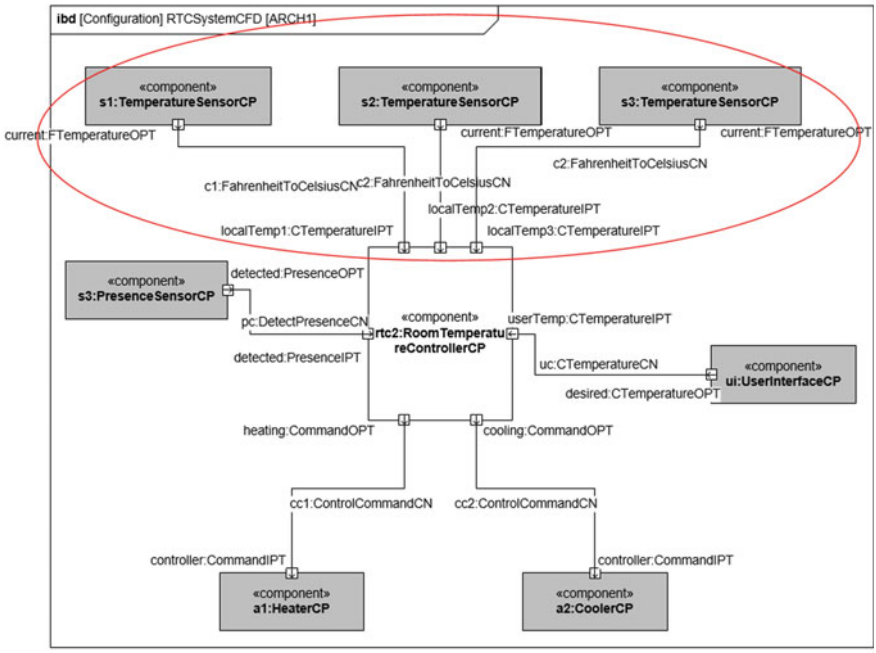


Fig. 9.7 The new configuration of the RTC system ARCH1

CP. The resulting configuration is in the Fig. 9.7 that we show in the beginning of this chapter.

9.4.4 Design a New Architecture: ARCH2

In this section we illustrate the application of the T2 modifiability tactic (to minimize the ripple effect) in the *ARCH1* architecture of the *RTC System*. The new *ARCH2* is designed to be more modifiable (cheaper to modified) than *ARCH1*. We first describe the new architecture and then we analyse the modifiability of both ones.

To minimize the ripple effect (T2), we redesign the architecture of the *RTC System*—called *ARCH2*—using a multiplex port in the *SensorsMonitorCP* component. Recall that using multiplex port; we can add new temperature sensors without modifying the *RoomTemperatureControllerCP* component.

The BDDs that defines ARCH1 and ARCH2 components are illustrated in Figs. 9.8 and 9.9, respectively. Each figure highlights the ports of the *RoomTemperatureControllerCP* component that receives the Celsius temperature.

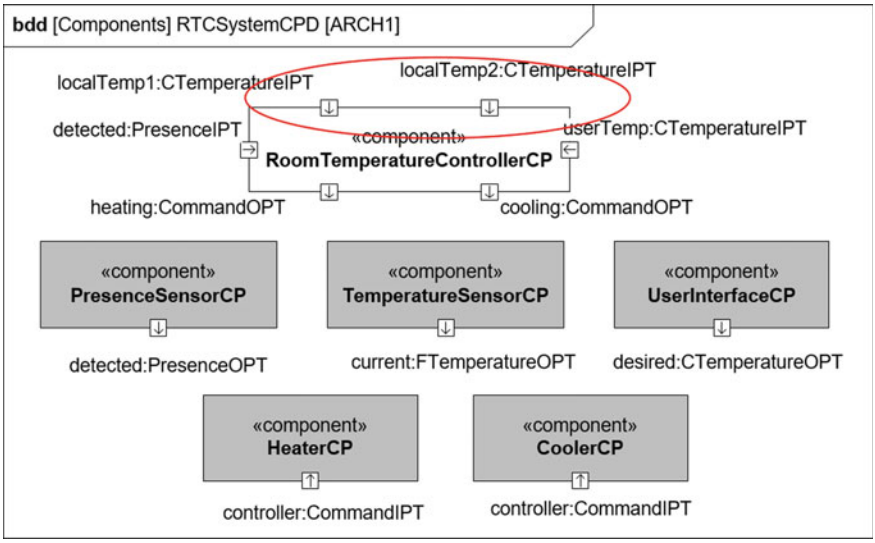


Fig. 9.8 ARCH1 Components BDD

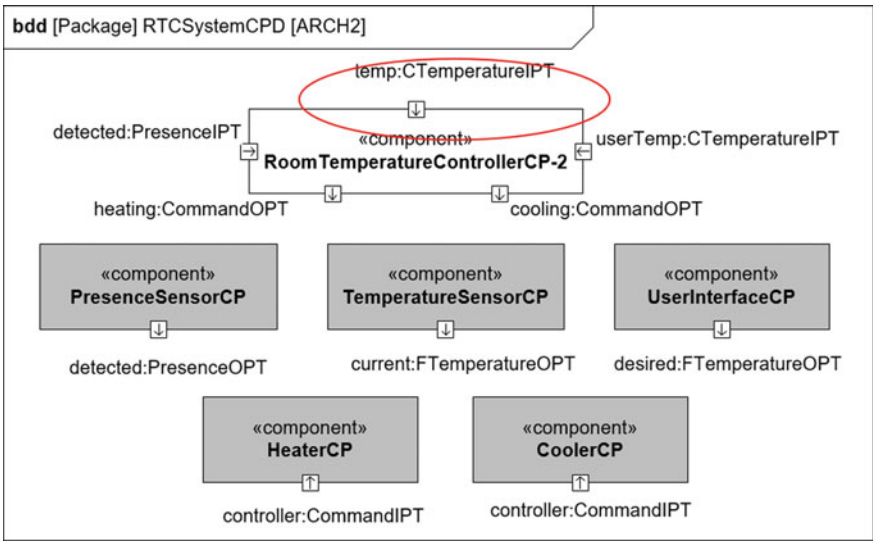


Fig. 9.9 ARCH2 Components BDD

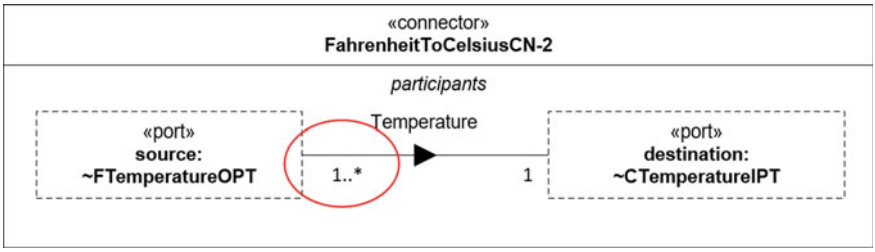


Fig. 9.10 The *FahrenheitToCelsiusCN-2* Connector

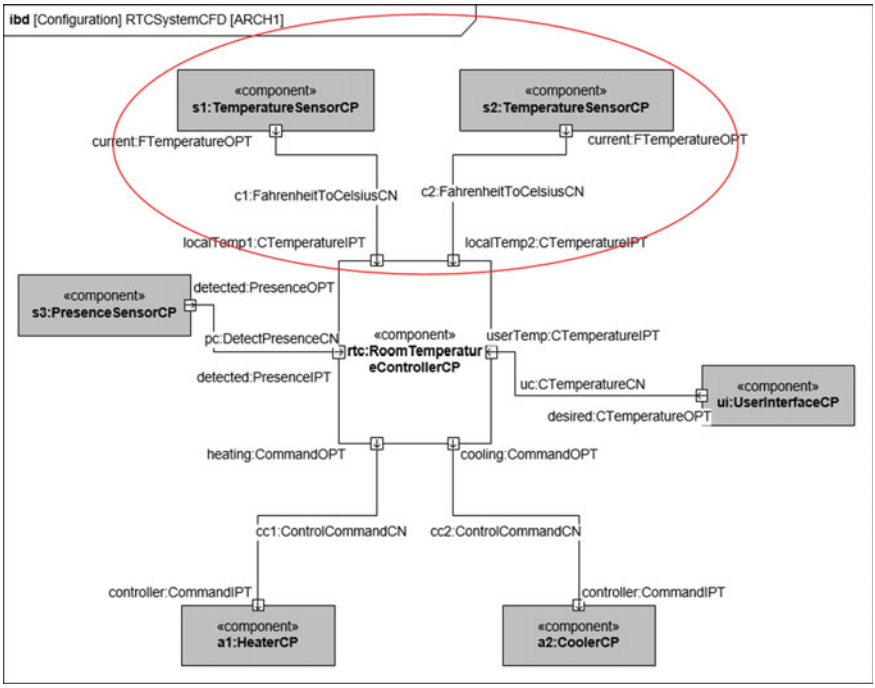


Fig. 9.11 The ARCH1 configuration

Figure 9.10 shows the definition of the connector that converts Fahrenheit temperature to Celsius temperature. The first connector implements a one-to-one conversion. This second connector shows that it is possible to have 1 or more source ports providing Fahrenheit values to be converted.

Figures 9.11 and 9.12 present, respectively, the *ARCH1* and *ARCH2* configurations.

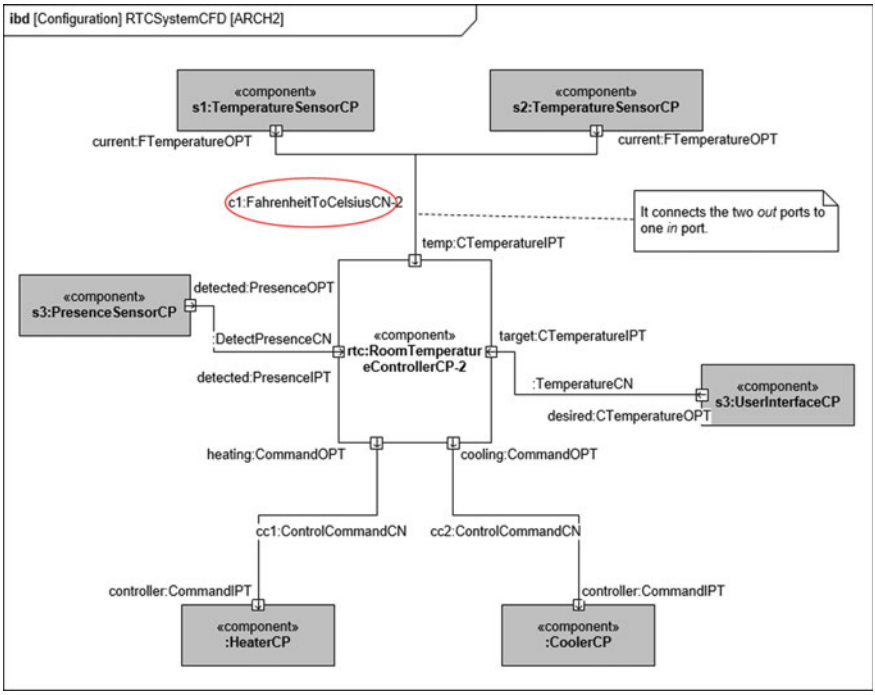


Fig. 9.12 The ARCH2 configuration

9.4.5 Analysing the Ripple Effects in ARCH2

In ARCH2, the *FahrenheitToCelsiusCN* connector supports connection from 1 or more *FTemperatureOPT* ports to 1 *CTemperatureIPT*. Figure 9.10 shows the new connector.

In the ARCH2 definition, we have a *RoomTemperatureController* CP with just one *CTemperatureIPT* Port. Figure 9.9 shows the new *bbd* to the *ARCH2* definition.

We need now to depict the effects to cause C1, considering the *ARCH2*:

C1—add a new temperature sensor component of a defined type to improve accuracy requires to

E1—modify the architecture description (*ibd configuration*)

- add *s3*:*TemperatureSensorCP* in *ARCH2*,
- add *c3*: *FahrenheitToCelsiusCN* between the *localTemp3*:*CTemperatureIPT* of *rtc*:RoomTemperatureControllerCP and *current*:*CTemperatureOPT* of *s3*: *TemperatureSensorCP*.

The resulting architecture after the modification of *ARCH2* to add a new temperature sensor is show in Figs. 9.12 and 9.13.

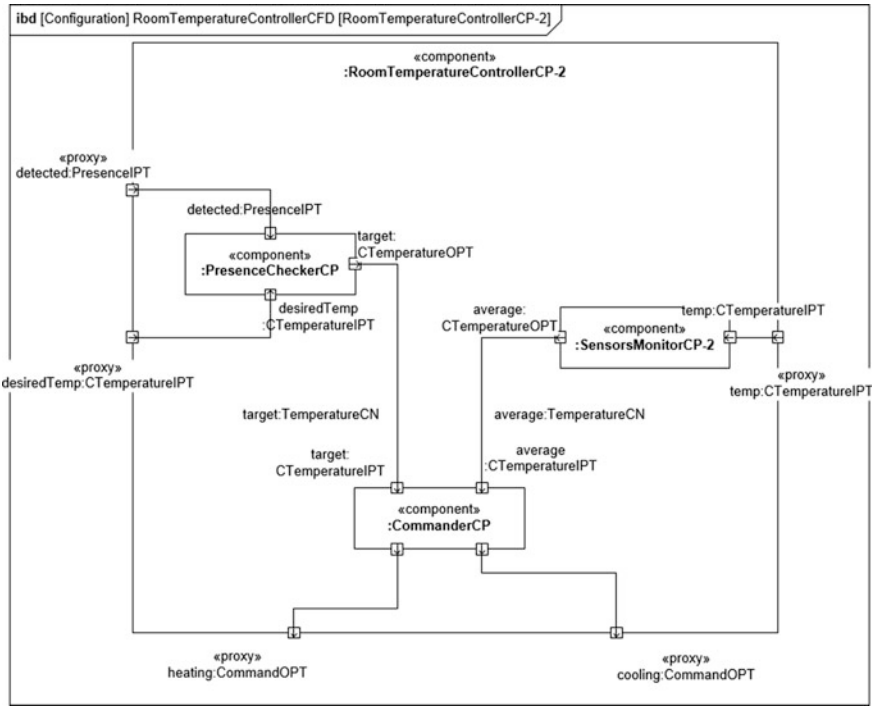


Fig. 9.13 The configuration of the *RoomTemperatureControlerCP-2* Component

9.4.6 Comparing Modifiability in ARCH1 and ARCH2

To add a new sensor in ARCH1, the ripple effect determines that we need to perform nine modifications: seven additions and two replaces. The same initial modification, i.e., add a new sensor, in *ARCH2* requires only two modifications: the addition of two elements.

Therefore, *ARCH2* is more modifiable than *ARCH1*.

9.5 Summary

In this chapter, you learnt the following:

- the modifiability concept,
- the architectural causes and effects of modifiability,
- tactics to improve modifiability,

- a taxonomy of modifiability primitives,
- a comparison technique to evaluate the modifiability in alternative architectures.

You learnt how to do the following:

- express modifiability in software architecture using a cause–effect relationship,
- compare two alternative architectures to evaluate their modifiability by quantifying the ripple effect quality attribute.

Further Reading

1. Bass, L., Clements, P., Kazman, R.: Software Architectures in Practice, 2nd edn. Addison Wesley, Reading (2003)
2. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)

Reference

1. ISO/IEC 25010 http://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed March 2016

Chapter 10
Designing Scalability in Software Architectures

In this chapter we present scalability as an architectural quality and how to express scalability in software architecture. We explain how to analyze an architecture to evaluate its scalability and how to apply tactics to improve scalability.

You will learn

- what is scalability as an architectural quality;
- what are the architectural causes and effects of scalability;
- tactics to improve scalability;
- a comparison technique to evaluate the scalability in alternative architectures.

10.1 Introduction

Scalability is the quality that refers to the degree to which a system can effectively and efficiently be scaled for increasing operational usage. In Software Architecture, scalability refers to how the architecture can be scaled to cope with an increasing number of component usage that may be of several orders of magnitude.

An example of scalability in the case of a temperature monitor system is to maintain its performance when the number of sensors increases in orders of magnitude. Increasing from 1 to 10 sensors, and then from 10 to 100, and then from 100 to 1000, and so on.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_10

10.2 Scalability Causes and Effects

We need to express the causes and the effects of scalability. Causes refer to identify the component types to which their uses (number of instances) can increase in orders of magnitude. Effects refer to the impact on the effectiveness and efficiency of the system implied by its architecture when the number of component uses increases.

Component types can have their use (number of instances) significantly increased in the software architecture. *Connector types* may have their use (number of instances) significantly increased as a consequence of the rise of component use. *Port types* may have their use (number of instances) significantly increased as a consequence of the rise of component use. It is important to note that, if the architecture defines that a component type can have only one instance, we cannot increase the number of instances.

In the RTC system, we can increase the number of temperature sensors and presence sensors. We cannot increase the user interface and the controller component types as the architecture defines only one instance of each type.

Effects are the consequences of the causes in the architecture. In other words, they are the impact on the effectiveness and efficiency of the system.

According to the requirements of the *RTC System*, *effectiveness* is to assure that the control-loop correctly adjusts the room temperature according to the user desired temperature, the current average temperature and the presence of a person. *Efficiency* refers to the time of the control-loop to adjust the temperature.

In the architecture *ARCH2* of the *RTC System*, the cause is the increase in the number of temperature sensors.

The effects in the same architecture should consider the cardinality of the temperature sensor. According to the cardinality, the minimum number of temperature sensor is 1. When the number of temperature sensors is increased by 10, the number of connectors is also multiplied by 10, and they are connected to the same port. Then, the time to read all sensors is multiplied by 10. As a consequence, the efficiency is divided by 10 (effect) and the system is not effective if the efficiency it too low.

10.3 Scalability Quality Attribute

The scalability quality attribute refers to the quality used to quantify the effects. Usually, we use the following ripple effects in system performance

- response time,
- throughput,
- resource consumption.

As an example, if we increase the number of temperature sensors by 10, we expect not to impact the response time of the feedback control-loop.

10.4 Scalability Tactics

Tactics are design decisions that influence quality attributes. We use tactics to minimize effects. An example of a scalability tactic is to introduce concurrency to improve scalability. We can introduce concurrence in order to keep performance.

As an example, in the *RTC System*, we can introduce concurrency in the sensor monitor component to concurrently read the temperature from the sensors.

10.5 Applying the Scalability Tactics

To allow a significant rise of components (by order of magnitude) we design a new architecture of the *RTC System*. In the new architecture, *ARCH3*, we apply concurrency tactic.

We design a *CompositeMonitorCP* component that can read all sensors data concurrently. *CompositeMonitorCP* has one *SensorReadersCP* component for each *TemperatureSensorCP*. An n-ary connector (*AllTemperaturesCN*) joins all output from the *SensorReadersCP* components to the *AverageCalculatorCP* component.

We present the differences between ARCH3 and ARCH2 (presented in last chapter) in the following section.

10.5.1 The Component Definitions of ARCH2 and ARCH3

ARCH2 has a simple component to read data from all sensors. ARCH3 has a *CompositeMonitorCP* composed of a *SensorReadersCP* component and an *Average CalculatorCP* for each *TemperatureSensorCP*. Figure 10.1 shows the definition of architecture *ARCH2* (previously shown in the last chapter) and Fig. 10.2 shows the definition of architecture *ARCH3*, with the new *CTemperaturePT* port having a cardinality [1..*].

10.5.2 The Configuration of ARCH2 and ARCH3

The new architecture seems similar when we view a system configuration using an *ibd*. However, the main difference is the way the temperature sensors are connected

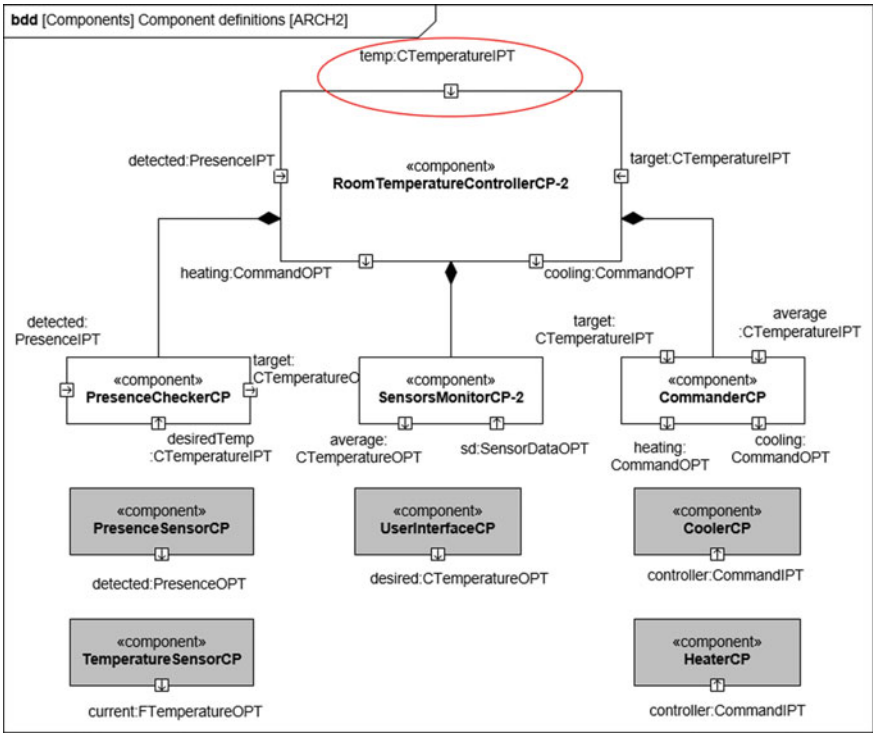


Fig. 10.1 *ARCH2* component definition

to the *CTemperatureIPT* port. In *ARCH2* (Fig. 10.3), the port supports several connectors and it is responsible for receiving data and providing them to the internal components.

In *ARCH3*, we have multiples instances of *CTemperatureIPT* port. The difference is that the *RoomTemperatureControllerCP* needs a modification to receive data from several port instances, as illustrated in Fig. 10.4. In that figure, we have three instances of the *CTemperatureIPT—temp*, *temp2,* and *temp3*.

10.5.3 The Configuration of RoomTemperatureControllerCP

Internally, the *RoomTempertureControllerCP*, in *ARCH3* (see Fig. 10.6), has a new component to read the values from the sensors. Whereas, in *ARCH2* (see Fig. 10.5), the *SensorMonitorCP* component reads the data from all sensor sequentially, in

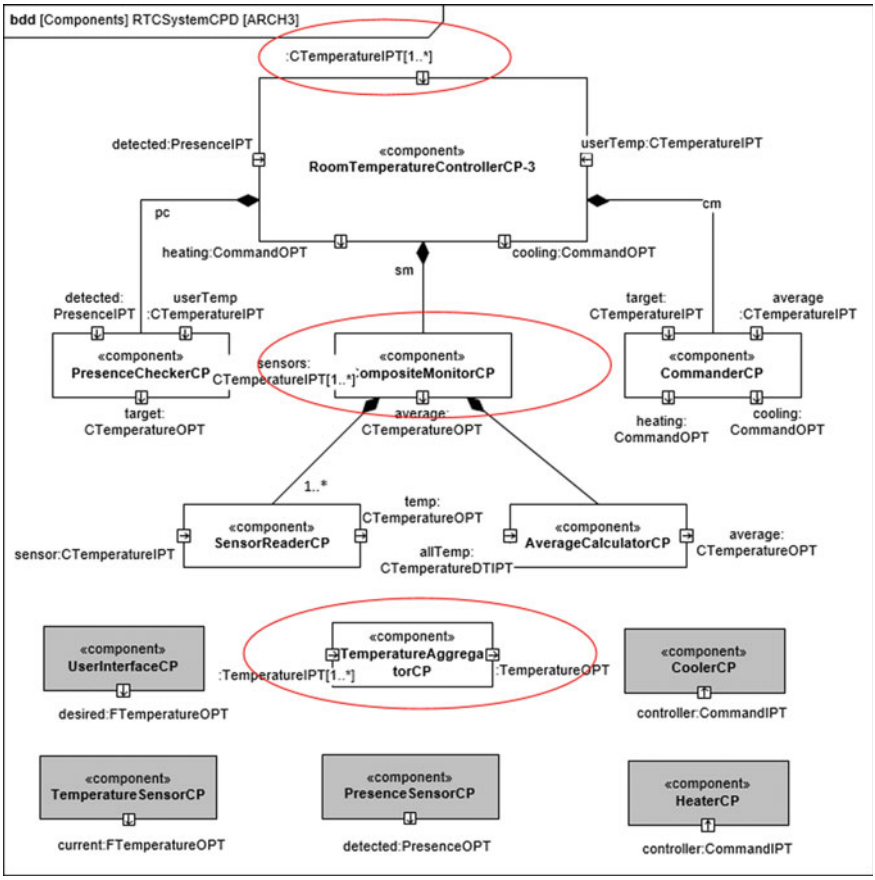


Fig. 10.2 *ARCH3* component definitions

ARCH3 the new *CompositeMonitorCP* component can concurrently read all sensors data. *SensorMonitorCP* is a simple component and it does not have a configuration.

10.5.4 The Definition of CompositeMonitorCFD

In *ARCH3*, as depicted in Fig. 10.7, *CompositeMonitorCFD* is a composite component that has one *SensorReadersCP* component for each *TemperatureSensorCP*.

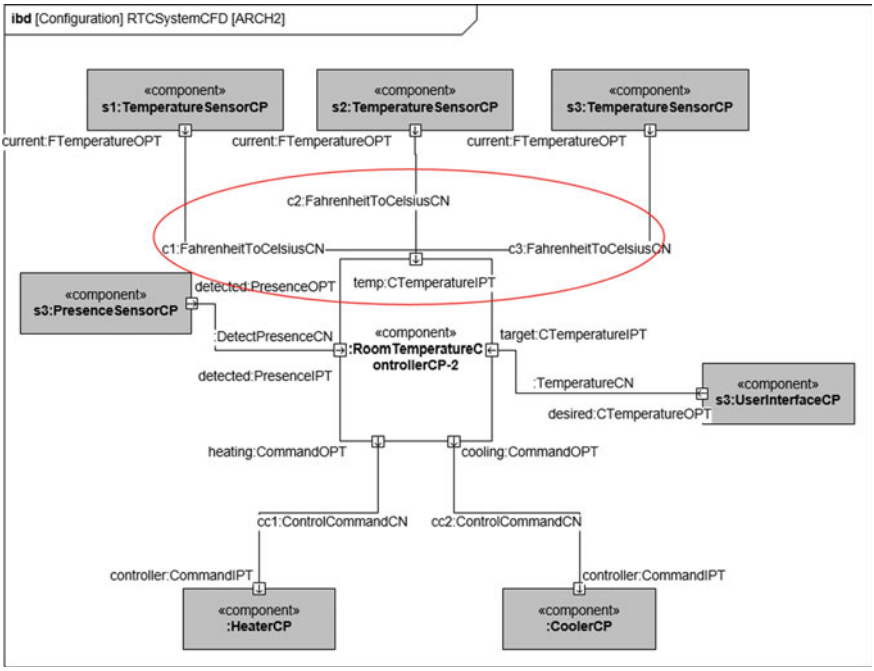


Fig. 10.3 *ARCH2* configuration

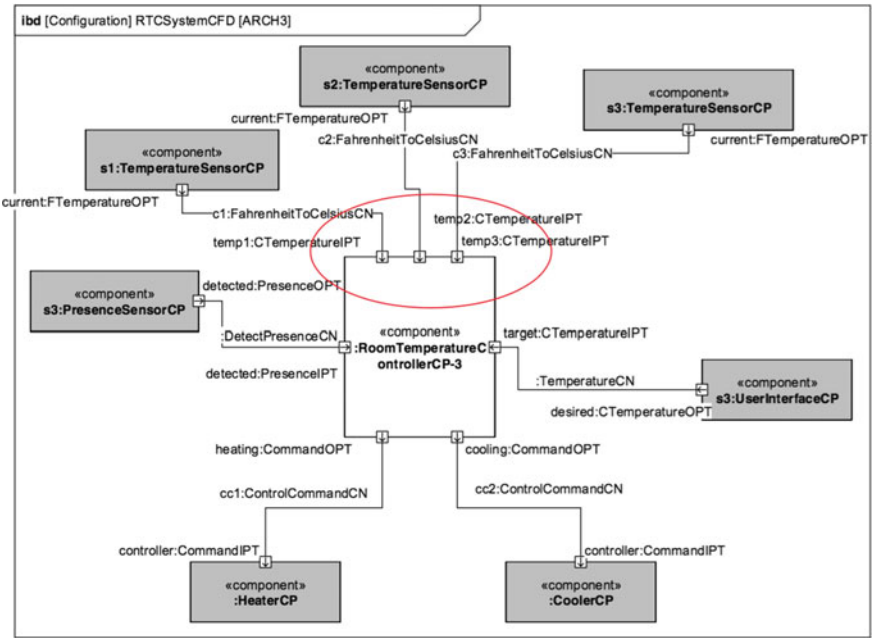


Fig. 10.4 *ARCH3* configuration

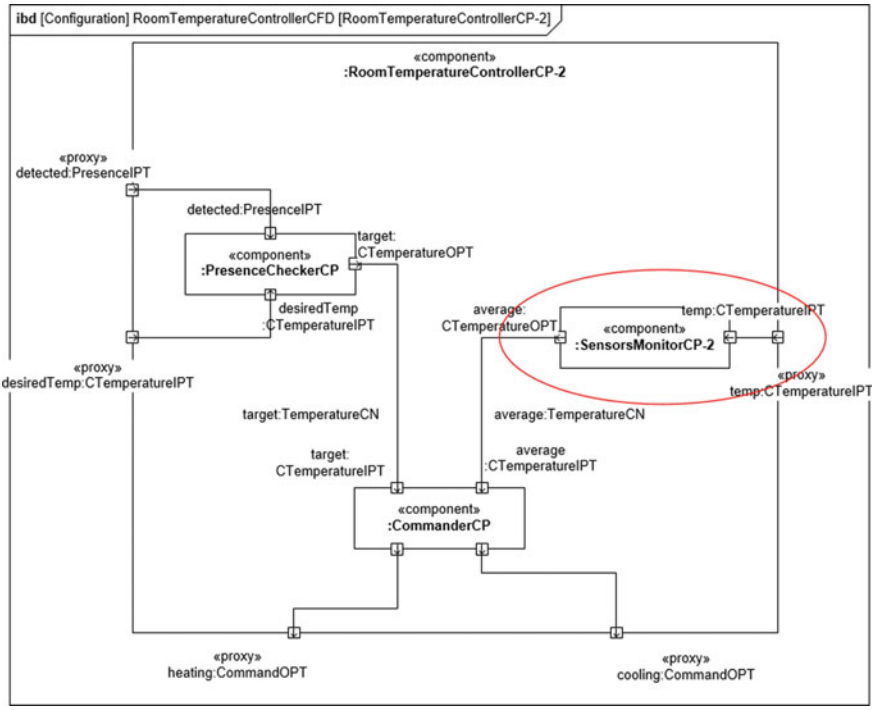


Fig. 10.5 *RoomTemperatureControlerCP* in ARCH2

The *AllTemperaturesCN* (n-ary) connector joins all output from the *Sensor ReadersCP* components and forwards them to the *AverageCalculatorCP* component.

10.5.5 The AllTemperaturesCN Connector in ARCH3

The configuration of the *AllTemperaturesCN* connector, shown in Fig. 10.8, includes a component to aggregate the data from the ports that are linked to the sensors. The *TemperatureAggregatorCP* connector provides a data structure with all data through the *TemperatureDTOPT* port.

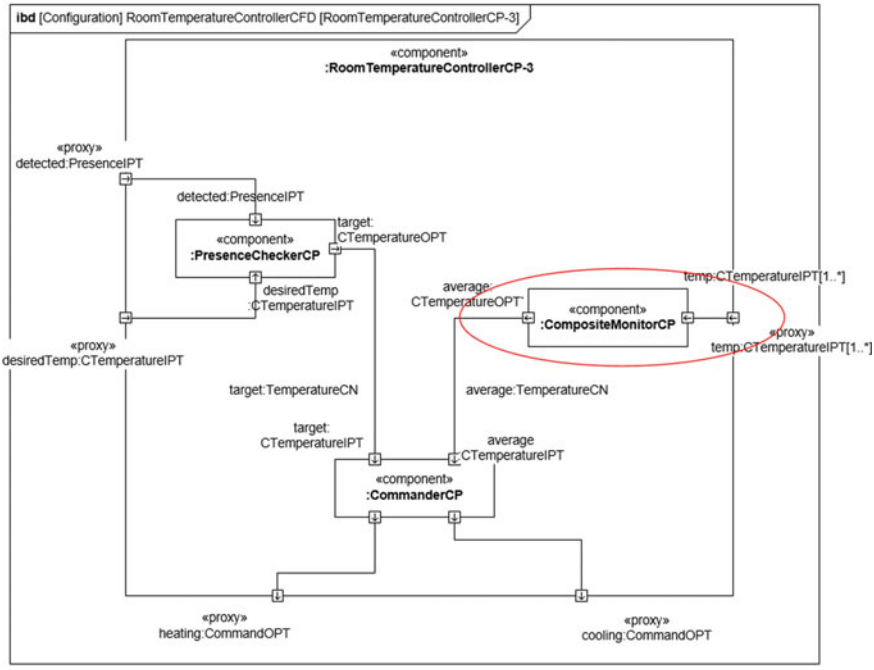


Fig. 10.6 *RoomTemperatureControlerCP* in ARCH3

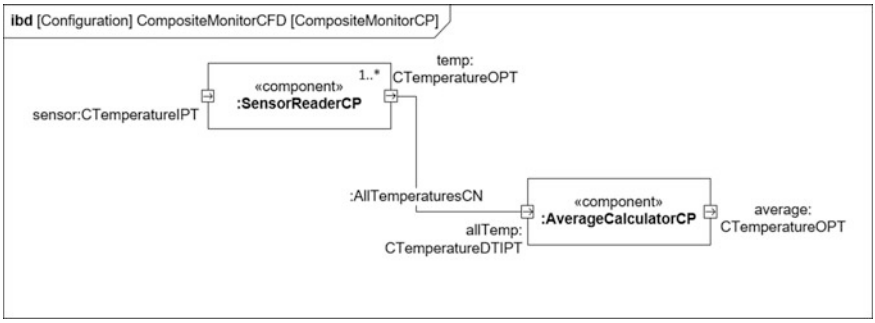


Fig. 10.7 *CompositeMonitorCFD* configuration in *ARCH3*

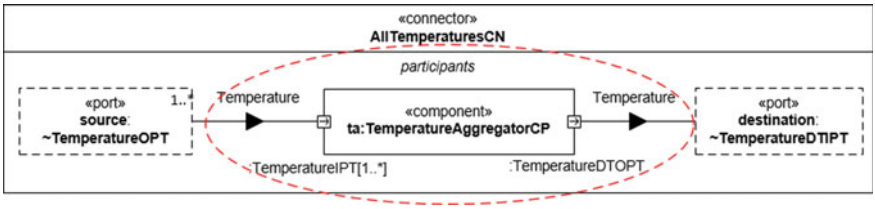


Fig. 10.8 The *AllTemperaturesCN* connector in *ARCH3*

10.6 Scalability Analysis

In this section, we present an example of scalability analysis using the previous defined architectures, *ARCH2* and *ARCH3*. To motivate the analysis of scalability we introduce a new requirement to the RTC system.

10.6.1 RTC System Requirements

The RTC system has the following scalability requirement:

- R1—The system must allow the rise of the temperature sensors of a defined type in an order of magnitude (multiply by 10) to improve accuracy.

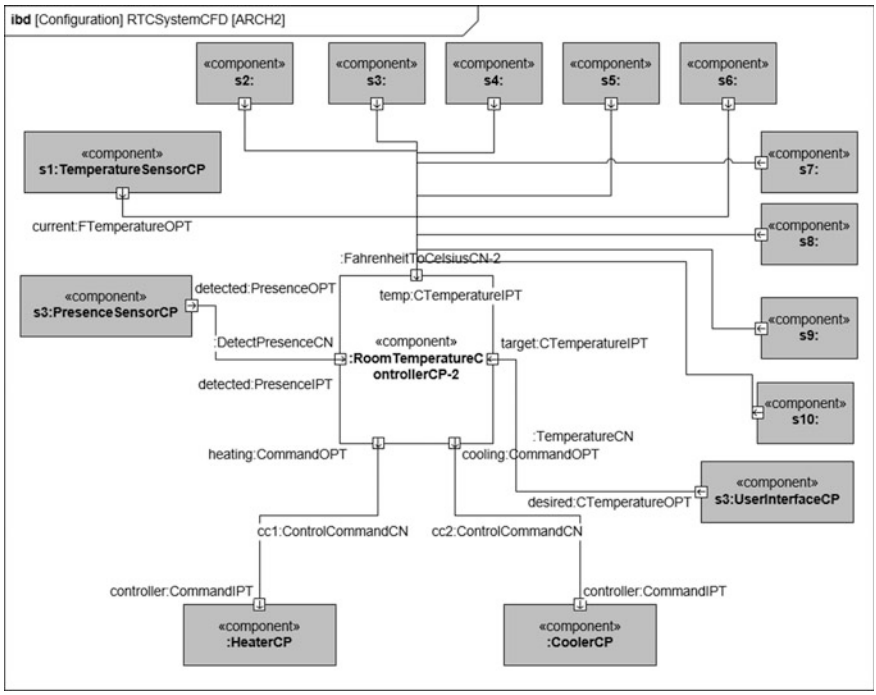


Fig. 10.9 *ARCH2* configuration

10.6.2 RTC System—Causes

Our analysis also considers the cause–effect approach we have used in this book. The cause to the R1 requirement is:

- C1—To increase the temperature sensors of a defined type in an order of magnitude (multiply by 10) to improve accuracy.

10.6.3 Analyzing the Ripple Effect

We now consider the effects. Considering the ripple effect discussed in the beginning of this chapter, we analyze the response time on two different architectures of the *RTC System* considering the same cause.

The required modification is to improve temperature measures (R1 Requirement) by increasing the temperature sensors of the same type by an order of magnitude to the *RTC System* (Cause C1).

First, we analyze the *ARCH2* architecture presented in Fig. 10.9. This architecture has one multiplex port in the controller to receive data from many sensors. To read one sensor, one unit of time is necessary. If we have 10 sensors, the monitor reads one temperature at a time, so it needs 10 units of time. If we multiply the number of sensors by 100, the response time is proportionally increased by 100.

Then, we analyze the *ARCH3* architecture depicted in Fig. 10.10. This architecture has a composite monitor (inside *the RoomTemperatureControllerCP* component) that concurrently reads all temperature. To read one sensor, one unit of time is necessary. If we have 10 sensors, the monitor reads all temperature at the same time, so it reads all them in 1 unit of time. Therefore, if we multiply the number of sensors by 10, the response time does not increase.

According to the aforementioned analysis, we can conclude that *ARCH2* is not scalable and *ARCH3* is scalable.

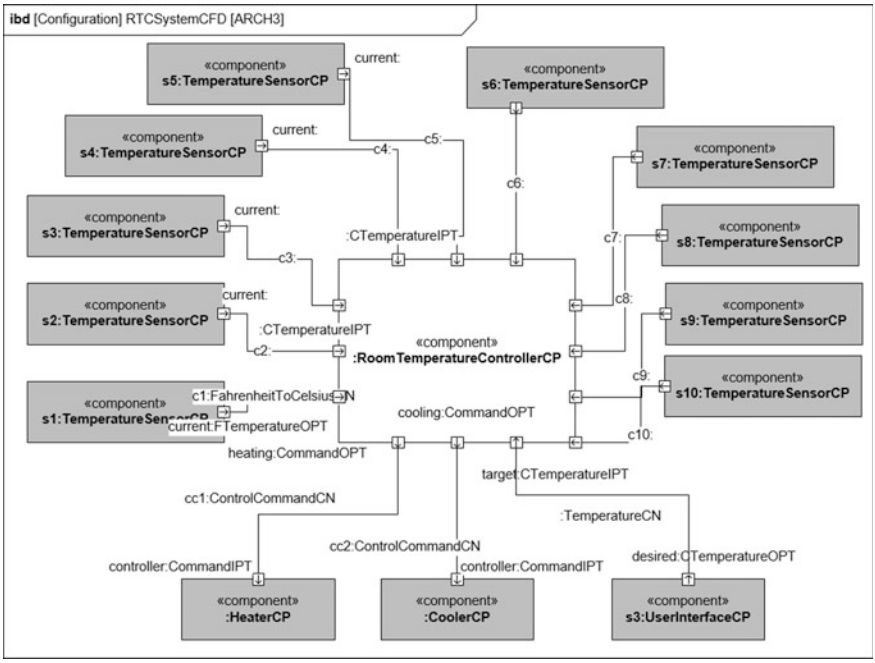


Fig. 10.10 *ARCH3* configuration

10.7 Summary

In this chapter, you will learn the following:

- the scalability concept;
- the architectural causes and effects of scalability;
- tactics to improve scalability;
- x architectures.

You will learn how to

- express scalability in software architecture using a cause–effect relationship;
- compare two alternative architectures to evaluate their scalability by analyzing the ripple effect of the response time quality attribute.

Further Reading

1. Bass, L., Clements, P., Kazman, R.: Software Architectures in Practice, 2nd edn. Addison Wesley, Reading (2003)
2. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)

Chapter 11
Designing Fault Tolerance in Software Architectures

In this chapter, we present fault tolerance as an architectural quality and how to express fault tolerance in software architecture. We explain how to analyze an architecture to evaluate its fault tolerance and how to apply tactics to improve fault tolerance.

You will learn the following:

- what is fault tolerance as an architectural quality;
- what are the architectural causes and effects of fault tolerance;
- tactics to improve fault tolerance;
- a comparison technique to evaluate fault tolerance in alternative architectures.

11.1 Introduction

Fault tolerance is a quality to refer to the degree to which a system can maintain operational service even in the presence of faults. In software architecture, fault tolerance refers to the degree to which an architecture can maintain its operation even if components fail.

An example of fault tolerance in the case of a temperature monitor system is to maintain its operational service when the controller fails.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_11

11.2 Fault Tolerance Causes and Effects

We need to express fault tolerance causes and effects. Causes refer to identify the components that can fail. Effects refer to the impact in the system implied by its architecture when one or more components fail.

Components can fail due to several causes. One cause may be the halt of an operation (loss of service) due to a hardware crash or a software fail. Another cause may be an incorrect operation due to a hardware or software fail.

In the *RTC System*, we can identify components that can fail. For instance, a sensor temperature component or a room temperature controller component may fail. In the first case, it causes an incorrect operation. The latter causes the system to stop its operation.

Effects are the consequences of the causes in the architecture. In other words, they are the impact on system operations that lead to a failure.

We need to analyze faults at both component level and architecture level.

At the component level, an internal fault can cause an error in the following conditions:

- If the fault is recovered, there is no error.
- If the fault is not recovered and it is propagated out of the component causing a behavior that is not in conformance to the specification, it is an error. Outage of component means that a component stops to operate or it continues to operate in an incorrect way.

At the architecture level, a component error can cause a failure.

- If the error is recovered, there is no failure.
- If the error is not recovered and it is propagated out of the system causing a behavior that is not in conformance to the specification, it is a failure. Outage of system means that it stops to operate or it continues to operate in an incorrect way.

In the architecture *ARCH3*, the causes of errors may be associated to the following components:

- a temperature sensor that fails;
- a presence sensor that fails; or
- a controller that fails.

In terms of effects

- If a temperature sensor fails, the effect is that the system continues to operate, but incorrectly, causing the monitor to stop waiting for data from the failed sensor, and the controller does not change the room temperature anymore;
- If a presence sensor fails, the system continues to operate, but incorrectly, causing the system to maintain the room temperature in 22 °C, even in the presence of a user;
- If a controller fails, the system stops to operate.

11.3 Fault Tolerance Quality Attributes

Fault tolerance quality attribute refers to a quality used to quantify how much the system tolerates faults. The ripple effects in system operation are

- no impact in the system services—the services continue to be provided in nominal mode (according to the specification);
- some impact in system services—the services continue to be provided but in a degraded mode;
- total impact in system services—the services stop to be provided.

As an example, in the *ARCH3* architecture of the *RTC System* we can analyze three cases

- If a temperature sensor fails, there is a total impact because the system continues to operate but it is blocked and the services are not provided anymore;

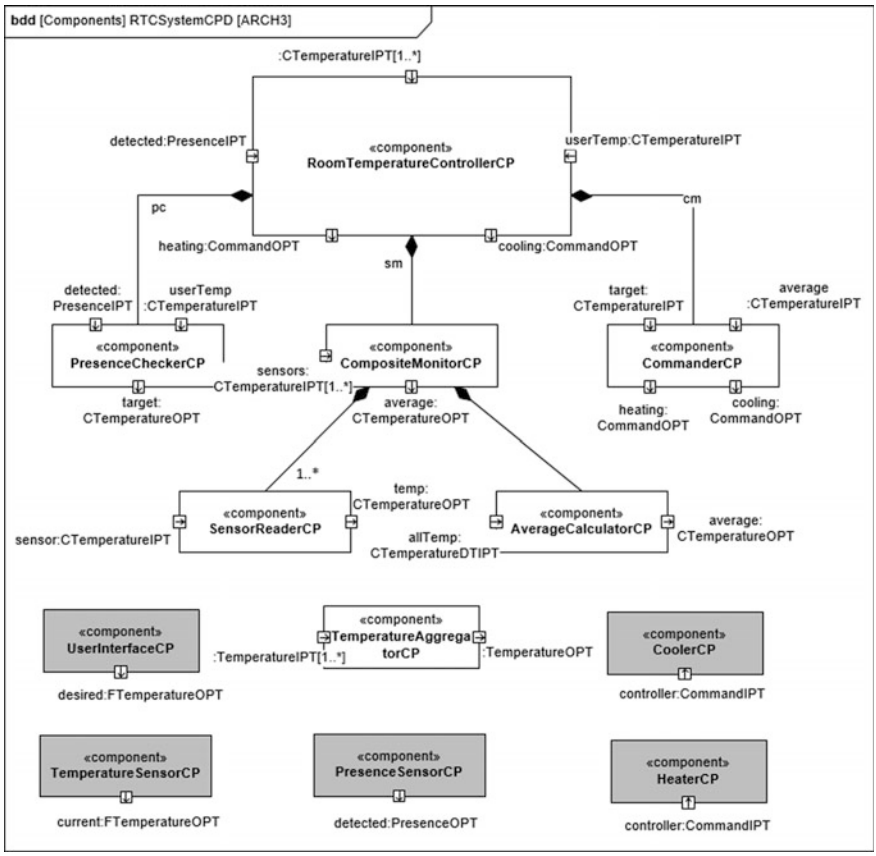


Fig. 11.1 *ARCH3* component definitions

- If a presence sensor fails, there is a total impact because the system continues to operate but it is blocked and the services are not provided anymore.
- If the controller fails, there is a total impact because the system stops to operate.

Thus, the *ARCH3* architecture, depicted in Fig. 11.1, is not fault tolerant with respect to those three components.

11.4 Fault Tolerance Tactics

Tactics are design decisions that influence quality attributes. We use two fault tolerance tactics (T1 and T2):

T1—introduce redundancy

An example of fault tolerance tactic is to introduce *redundancy*. In software architecture, to improve fault tolerance we can introduce component redundancy in order to keep system services.

As an example, in the RTC System, we can introduce redundancy in the *RoomTemperatureControllerCP* component to keep the system services in operation.

T2—heartbeat

Another fault tolerance tactic is the *heartbeat*. The heartbeat consists of a component periodically sending a message to notify it is operating.

As an example, in the *RTC System* the controller periodically sends a status message to the connector informing that it is up.

11.5 Applying Fault Tolerance Tactics

To detect and recover when a controller fails, we design a new architecture of the *RTC System*. In the new architecture *ARCH4*, we apply the redundancy and the heartbeat tactics. We create two instances of the *RoomTemperatureControllerCP*, one is considered primary and the other is the secondary. A composite connector receives the controller status and uses that information to decide to which one it sends the temperature values. The *RoomTemperatureControlerCP* component has a composite port, *StatusTempCPT*, to send a message to the connectors and to receive the temperature value, as illustrated in Fig. 11.2.

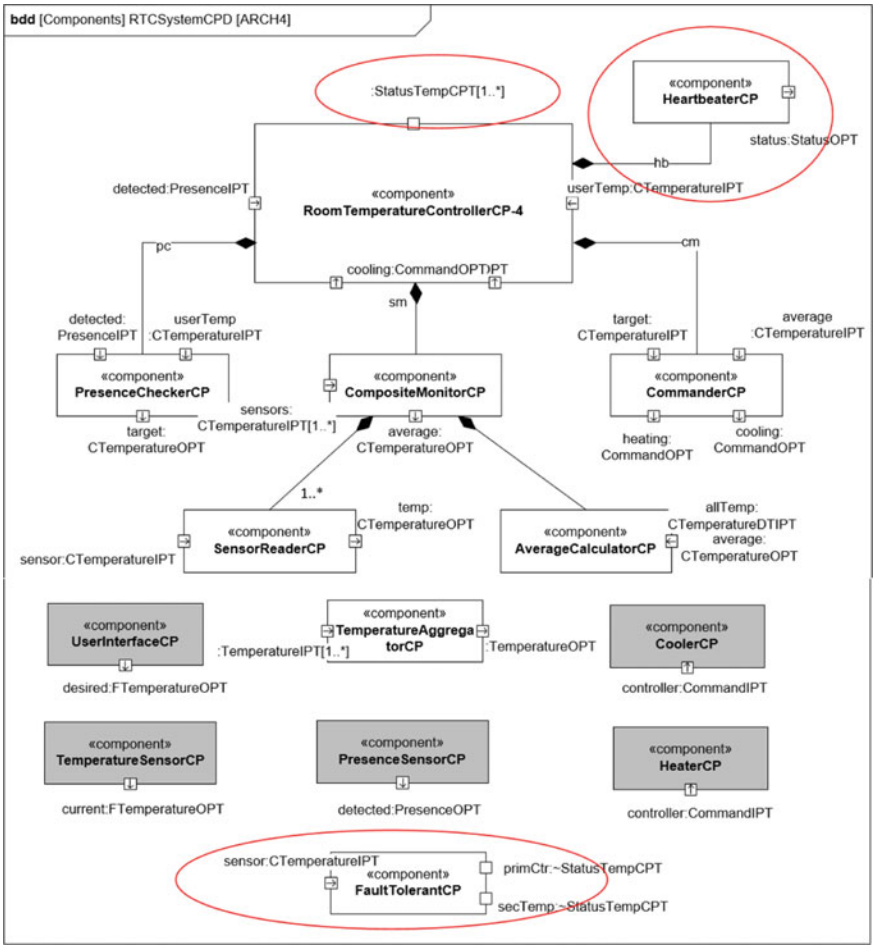


Fig. 11.2 ARCH4 introduces a composite port *StatusTempCPT* and *HeartbeaterCP*

11.5.1 The Configuration of ARCH3 and ARCH4

In *ARCH3*, depicted in Fig. 11.3, there is no component to inform the system status. In *ARCH4*, shown in Fig. 11.4, the *RoomTemperatureControllerCP* has a *HeartbeaterCP* component that sends a status message informing it is up or down. The *StatusTempCPT* composite port has an *out* port to send the status and an *in* port to receive the temperature.

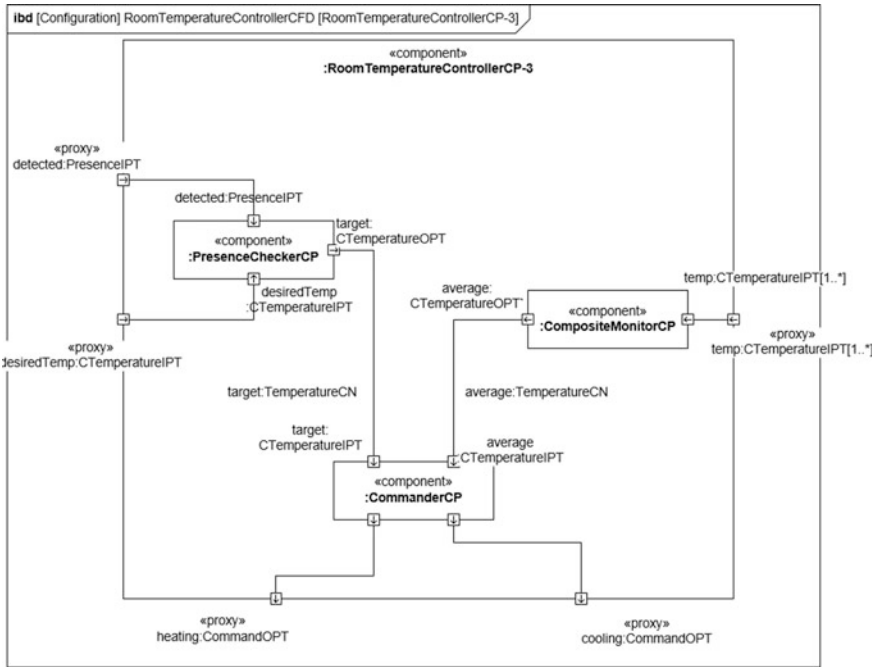


Fig. 11.3 The *RoomTemperatureControllerCP* configuration in *ARCH3*

11.5.2 The HeartbeaterCP Component

The *HeartbeaterCP* component (see Fig. 11.5) periodically emits a message to notify it is operating. It periodically sends a status message through its *StatusOPT* port to inform that it is up. Figure 11.6 shows the *HeartbeaterAC* activity specifying the *setStatus* action to send a status message through that port.

The *RoomTemperatureControllerCP* in *ARCH4* has a composite port—*StatusTempCPT*—composed of two ports as shown in Fig. 11.7. *CTemperature OPT* is an input port to receive the temperature from the sensors. *StatusOPT* is an output port to send the status to the external connection.

In *ARCH4*, we have two instances of the *RoomTemperatureControllerCP* component. Figure 11.8 shows the two instances *rtc1* and *rtc2*. Each instance is connected to the s1 and s2 sensors using the *StatusTempCCN* composite connector.

StatusTempCCN is a complex connector that receives the temperature value from *FTemperatureOPT* port of a *TemperatureSensorCP* and sends it to the *StatusTempCPT* port of the *RoomTemperatureControllerCP*. It additionally receives the status of the *HeartbeaterCP* in the *RoomTemperatureControllerCP*. It uses the status to decide to which component *rtc1* or *rtc2* it should send the temperature. In order to decide it has a *FaultTolerantCP* component, as shown in Fig. 11.9.

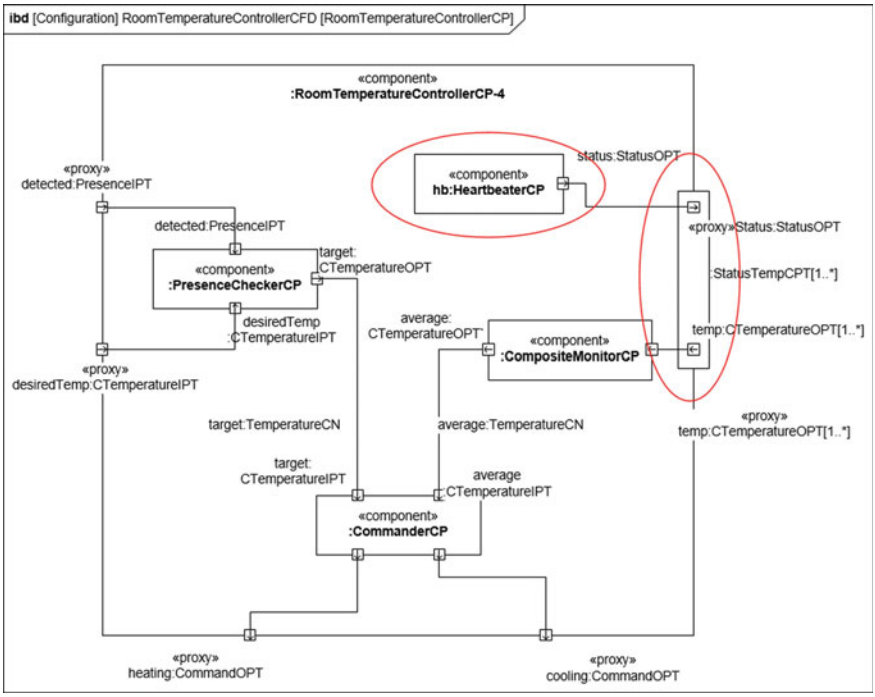


Fig. 11.4 The *RoomTemperatureControllerCP* configuration in *ARCH4*

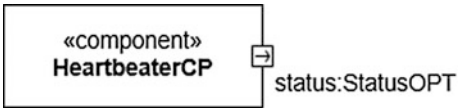


Fig. 11.5 The *HeartbeaterCP* component in *ARCH4*

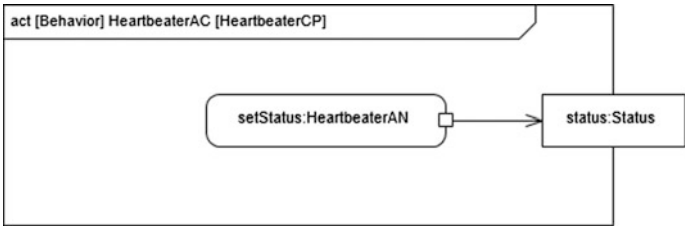


Fig. 11.6 The *HeartbeaterCP* behavior in *ARCH4*

Figure 11.10 shows the activity diagram that specifies its behavior. It receives a status message from the *RoomTemperatureControllerCP* component that informs its status (up or down). The status message is received in the *StatusOPT* port of the

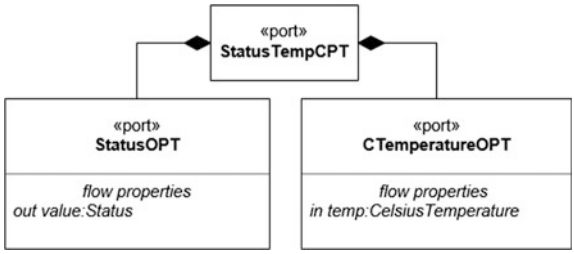


Fig. 11.7 The *StatusTempCPT* composite port definition in *ARCH4*

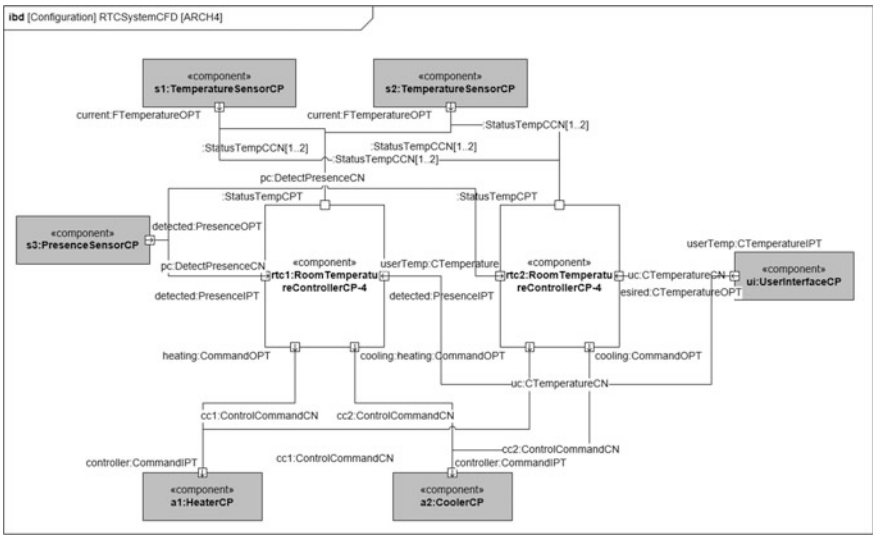


Fig. 11.8 A configuration of ARCH4 using two *RoomTemperatureControllerCP*

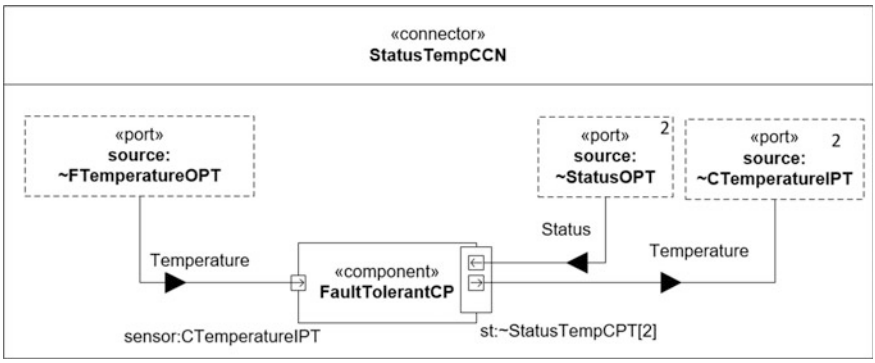


Fig. 11.9 The *StatusTempCCN* complex connector definition in ARCH4

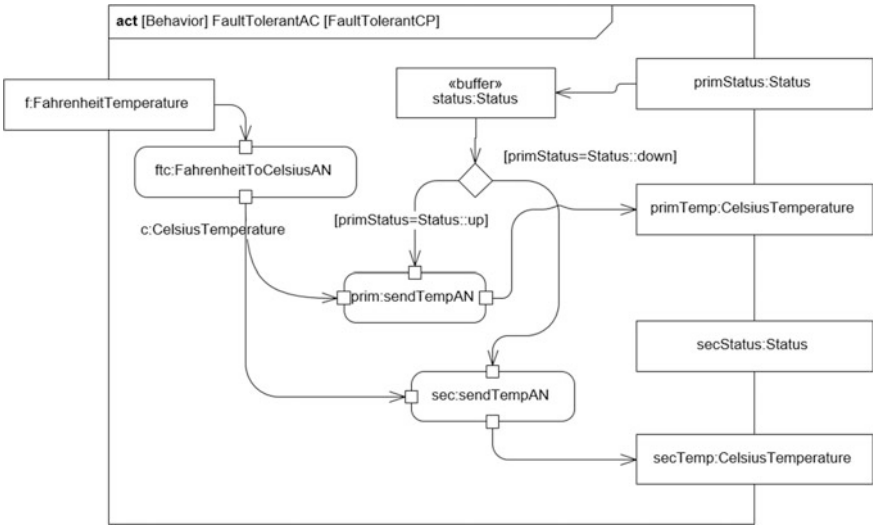


Fig. 11.10 The behavior of the *FaultToleranceCP* component of the *StatusTempCCN* connector

StatusTempCPT composite port. In the activity diagram, the ports are represented as pins. Thus, the status message is received in the *primStatus* and *secStatus* pins.

The *FaultTolerantCP* component uses the status message to decide if the temperature value should be sent to the *rtc1* or *rtc2* controllers, through the *primTemp* and *secTemp* pins.

11.6 Fault Tolerance Analysis

We use an AND/OR tree to depict the requirements (R), causes (C) and effects (E).

11.6.1 RTC System Requirements

The RTC system has the following fault tolerance requirement:

- R1—the system must tolerate the failure of the controller.

11.6.2 RTC System—Causes

Our analysis also considers the cause-effect approach we have used in this book. The cause to the R1 requirement is

- C1—The controller failure.

11.6.3 Analyzing the Ripple Effect

We analyze the fault tolerance on two different architectures of the *RTC System* considering the same cause: the tolerance to the failure of the controller (Requirement R1) studying the ripple effect of this failure (Cause C1).

First, we analyze the *ARCH3* architecture. In this architecture the failure of the controller causes the failure of the whole system.

Then, we analyze the *ARCH4* architecture. In this architecture the failure of the controller does not cause the failure of the whole system. This architecture has two instances of a controller. When the first fail, the system continues to operate using the second instance.

According to the aforementioned analysis, considering the failure, we can conclude that *ARCH3* is not fault tolerant and *ARCH4* is fault tolerant regarding the controller.

11.7 Summary

In this chapter you learnt the following:

- the fault tolerance concept,
- the architectural causes and effects of fault tolerance,
- tactics to improve fault tolerance,
- a comparison technique to evaluate the fault tolerance in alternative architectures.

You learnt how to

- express fault tolerance in software architecture using a cause-effect relationship,
- compare two alternative architectures to evaluate their fault tolerance by analyzing the ripple effect of the response time quality attribute.

Further Reading

1. Bass, L., Clements, P., Kazman, R.: Software Architectures in Practice, 2nd edn. Addison Wesley, Reading (2003)
2. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)

Part III
Style-Based Architectures

Chapter 12
Introduction to Style-Based Architectures

Part I presented how to describe a software architecture with SysADL to meet functional requirements. Part II presented how to design a software architecture with SysADL to satisfy nonfunctional requirements known as quality attributes. In this, we introduce the concept of architectural style and how a software architecture can be designed in conformance to a style. In particular, we present how to design a software architecture style with SysADL that can be used in an architecture design process for supporting architecture description.

You will learn the following:

- the concept of architectural style;
- the introduction to four architectural styles that will be addressed in the next chapters: Pipe-Filter, client–server, feedback control loop, and blackboard.

12.1 What Is an Architectural Style

An architectural style can be seen as a collection of principles that shapes the design of a software architecture to achieve a set of related quality attributes. For instance, Unix-like shells use the pipeline architectural style for composing commands, where the output of a command is piped to the input of the next command and so using standard output and input supporting the quality attribute of composability of

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_12

commands. Hence, in a pipeline architectural style a sequence of components is chained together by their standard streams, so that the output of each process feeds directly as input to the next one. In the case of Unix-like shells these components are called commands and the connectors are the pipes, represented in the shells by the '|' character. For instance, to list files having "architecture" in their names in a current directory, we can use the command *ls* to output the list of all filenames, then use the command *grep* to select only the lines of the *ls* output containing the string "architecture", and use the command *less* to display the result in a scrolling page, which gives in the command line of a terminal: ls –l | grep architecture | less.

Operationally, an architectural style can be seen as a collection of architectural design decisions and constraints on architectural decisions to achieve beneficial qualities in the resulting software architecture.

An architectural style is expressed by a vocabulary of design elements in terms of component types and connector types and a configuration type determining the allowed compositions of components and connectors.

Note that an architectural style has an associated semantic interpretation supporting defined analyses of architectures designed in the style.

12.2 What Is a Style-Based Architecture

The application of architecture styles in the design of software architectures allows to achieve expected qualities more easily. Indeed, an architectural style guides how to organize the components of an architected system so that one can design the complete architecture and achieve the qualities intrinsic to the style.

A style-based architecture is thereby defined as an architecture description that has been designed to cope one or a combination of different architectural styles.

For instance, let us take again our running example of the RTC System. The architecture that was designed in the Part I of the book was designed without following a specific style. Then in Part II we introduced quality-based architectures, i.e., an architecture that is designed to satisfy quality attributes using specific tactics (see for instance Chap. 9 for tactics to achieve modifiability). In Part III, we enhance the approach of designing quality-based architectures by the one of style-based architectures which allows to achieve different qualities reusing architectural decisions that are embodied in a defined architectural style. For instance, by designing architectures according to the pipeline style, the quality attribute of modifiability is granted as pipelines imply modifiability.

12.3 Architectural Styles

Part III focuses on discussing four architectural styles: Pipe-Filter, Client-Server, Feedback Control Loop, and Blackboard.

The *Pipe-Filter style* allows a sequential processing of data. Components are "filters" that read streams of data on its inputs and produces streams of data on its outputs. Connectors are "pipes" that transmits the output streams of one filter to inputs of another. In Chap. 13, we discuss the structural and behavioral specification of the Pipe-Filter style. We present an example of the Pipe-Filter style in an architecture of the RTC System with distributed temperature sensors with no direct communication with the controller. Instead, they are connected with their adjacent sensors in a sequence. We also present the Pipeline substyle where all filters are sequentially connected.

The *Client–Server style* allows components, called "clients," to send requests to a component, called "server," and wait for a reply. In Chap. 14, we discuss the structural and behavioral specification of the client–server style. We present an example of the Client–Server in the *RTC* System with a client that requests to the server (the controller) the average temperature.

The *Feedback Control Loop style* allows a central component to control several actuators by analyzing information from sensors. In Chap. 15, we discuss the structural and behavioral specification of the Feedback Control Loop style. We present an example of the Feedback Control Loop style in an architecture of the RTC System with three types of components (sensors, controller, actuators) and two types of connectors, characterizing a feedback loop where the controller uses sensors data from the environment to command actuators to act in the environment. This is a continuous loop where the actuators change the environment that will be then sensed by the sensors.

The *Blackboard style* defines a shared data structure (the blackboard) and multiple knowledge sources that interact with each other via the blackboard. In Chap. 16, we discuss the structural and behavioral specification of the Blackboard style. We present an example of the Blackboard style in an architecture of the RTC System where the blackboard is a connector that receives data from sensors and provides the received data to the central controller.

12.4 Summary

In this chapter you learnt the following:

- the quality concept
- the quality-based architecture concept
- the activities to analyze quality-based architectures.

Further Reading

1. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)
2. Buschmann, F., Meunier, R., Rohnert, H. Sommerlad, P., Michael Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns, vol. 1. Wiley (1996)
3. Vogel, O., Arnold, I., Chughtai, A., Kehrer, T.: Software Architecture: A Comprehensive Framework and Guide for Practitioners. Springer (2011)

Chapter 13
Pipe-Filter Architectural Style

In this chapter, we present and explain the Pipe-Filter architectural style and how to specify it in SysADL. We specify the style using the structural and behavioral viewpoints. We also present the pipeline architectural style as a substyle of Pipe-Filter. Finally, we illustrate the Pipe-Filter style and how to use it with our running example.

You will learn the following:

- what is a Pipe-Filter architectural style;
- what are its architectural elements, structure, and behavior;
- the pipeline substyle.

13.1 Conceptual Overview

In the Pipe-Filter style, components and connectors have a particular behavior and should be configured to allow a sequential processing of data.

Components, called "filters," read streams of data on its inputs and produces streams of data on its outputs, typically applying a local transformation to each element of the input streams and incrementally computing the corresponding elements of the output streams.

Connectors, called "pipes," serve as conduits for the streams, transmitting outputs of one filter to inputs of another.

We can use the Pipe-Filter style to model a part of a system that has a sequential dataflow that is transformed by filter components. The filter component acts as a filter to transform the flowing data. Each filter component has at least an *in* port (*Source*) or an *out* port (*Sink*). The connector acts as a pipe to conduct data from a

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_13

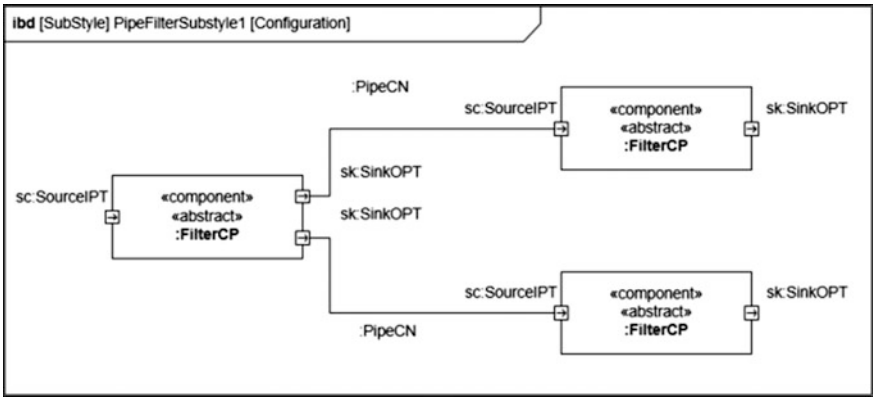


Fig. 13.1 The Pipe-Filter architectural style

sink in one component to a source in other component. Components are sequentially connected. Figure 13.1 shows an overview of these elements specified as abstract components. It shows three filters connected by two pipes.

An example of the use of the Pipe-Filter style in the *RTC* System is an architecture with distributed temperature sensors with no direct communication with the controller. Instead, they are connected with their adjacent sensors in a sequence.

In this example, depicted in Fig. 13.2, we have composite temperature sensors composed of a sensing component and a transmitting component.

The *CompositeTempSensorCP* component has an internal temperature sensor and a component that sends temperature values that it receives in its in port. *s1*, *s2*, and *s3* components are the filters, and *c13* and *c23* connectors are the pipes in this architecture. The *s1* and *s2* components are connected to the s3 component. The *s3* component receives the temperature and sends it to the *outTemp* port.

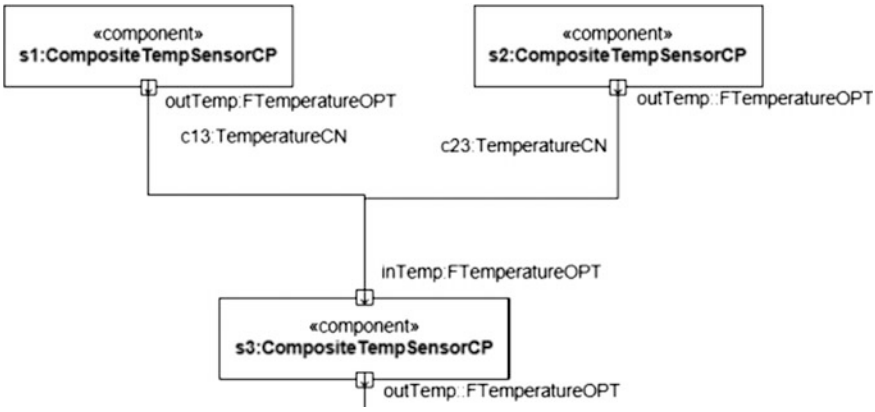


Fig. 13.2 A Pipe-Filter example

13.2 Pipe-Filter Structural Viewpoint

We define the Pipe-Filter architectural style using a *bdd*, as illustrated in Fig. 13.3. The Pipe-Filter Architecture Style (*PipeAndFilterARCH*) is composed of at least two filter components (*FilterCP*). Each filter component must have at least one *in* port (*SourceIPT*) and one *out* port (*SinkOPT*). As a convention we named the *bdd* as *PipeAndFilterSCPD—SCPD* means *Style Component Definition*. All components must be connected by at least one filter.

In the Pipe-Filter architectural style definition, we must also define the ports of a filter. Figure 13.4 shows the ports definitions. The *SourceIPT* port receives values of any type. The *SinkOPT* port sends values of any type.

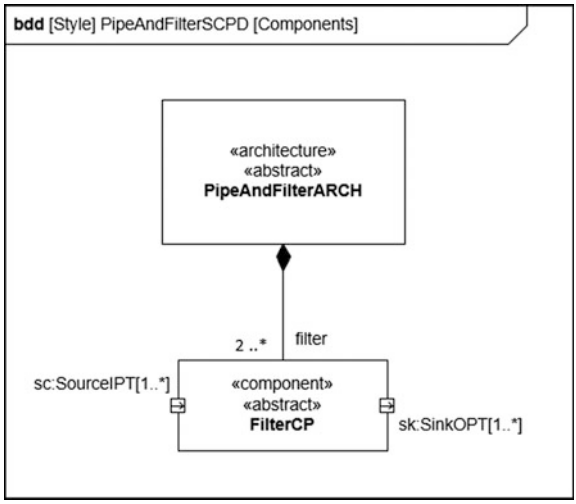


Fig. 13.3 A Pipe-Filter definition

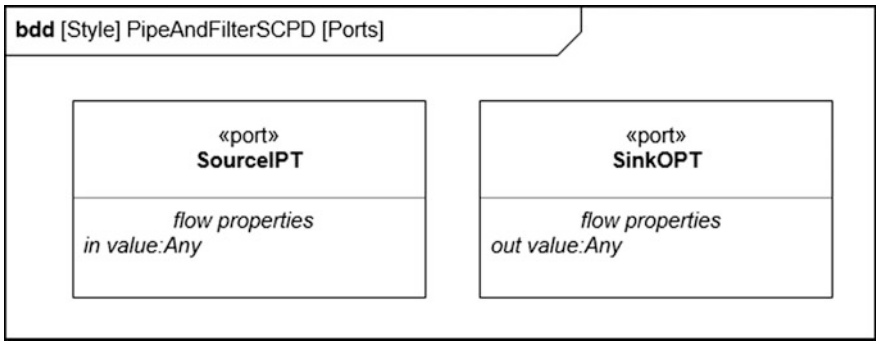


Fig. 13.4 Pipe-Filter port definitions

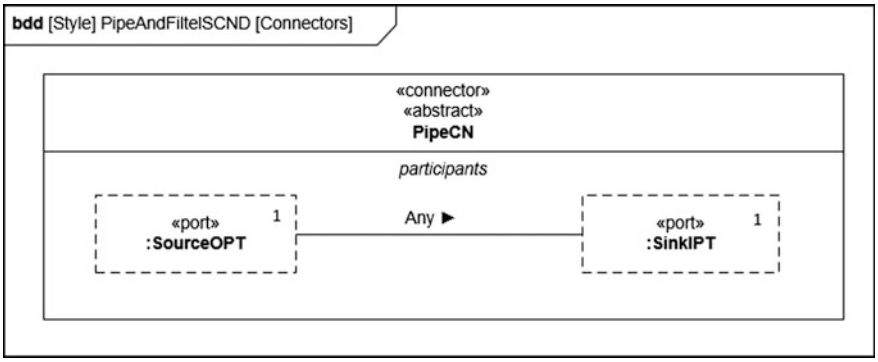


Fig. 13.5 Pipe-Filter connectors definitions

In the Pipe-Filter architectural style definition, we must also define the pipe connector. Figure 13.5 shows the connectors definitions of the style. The *PipeCN* connector has two participants: one *SourceOPT* port and one *SinkIPT* port. The *PipeCN* connector allows any type of data to flow from the *SourceOPT* port to the *SinkIPT* port.

13.3 Pipe-Filter Behavioral Viewpoint

We specify the behavior of the pipe connector using activity diagrams, as shown in Fig. 13.6. The connector sends a value directly from the input pin to the output pin.

We define the protocol of the ports using activity diagrams, shown in Fig. 13.7. The *SourceIPC* protocol states that the port repeatedly receives values of any type. The *SinkOPC* protocol states that the port repeatedly sends values of any type.

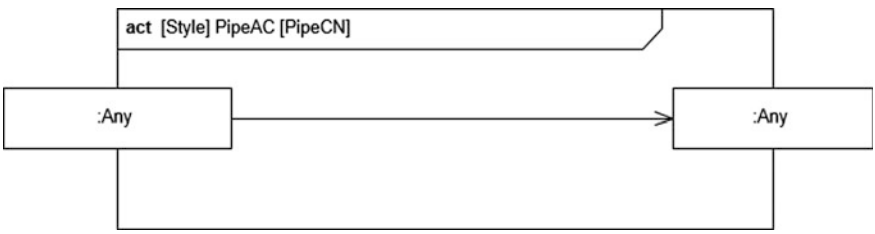


Fig. 13.6 Filter connector behavior

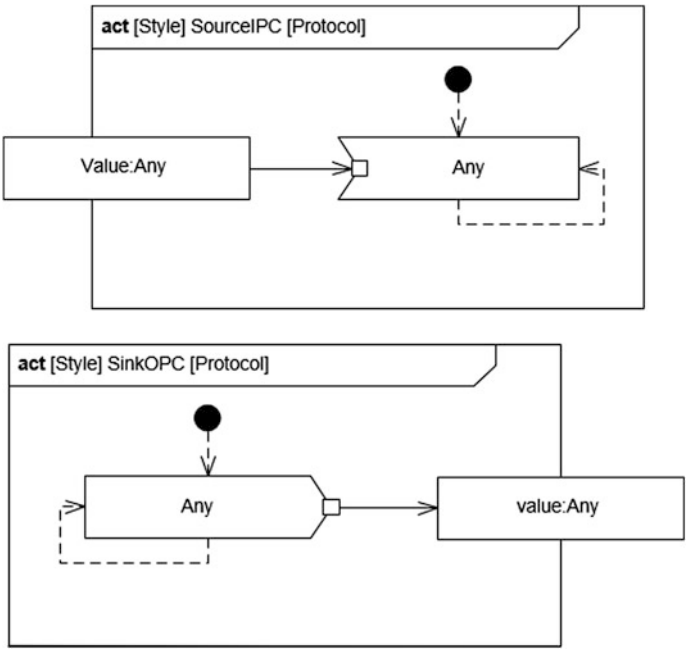


Fig. 13.7 Port protocol behavior

13.4 The Pipeline Substyle

A Pipeline is a substyle of the Pipe-Filter style where all filters are sequentially connected. Each filter has at most one predecessor and one successor.

An example of the use of the pipeline style in the RTC System is an architecture with distributed temperature sensors with no direct communication with the controller. Instead, they are connected to their adjacent sensors in a sequence.

We can use the pipeline substyle to model a part of the system that has a sequential dataflow that is transformed by the filter component. The components are, thereby sequentially connected, as depicted in Fig. 13.8.

In the example illustrated in Fig. 13.9, we can see composite temperature sensors in a pipeline architecture. The *CompositeTempSensorCP* component has an internal temperature sensor and a component that sends temperature values that it receives in its *in* port. The *s1*, *s2*, and *s3* components are filters, and the *c12* and *c23* connectors are pipes in this architecture; *s1* is connected to *s2*, and *s2* is connected to *s3*; *s2* and *s3* receive the temperature and send it to the *outTemp* port.

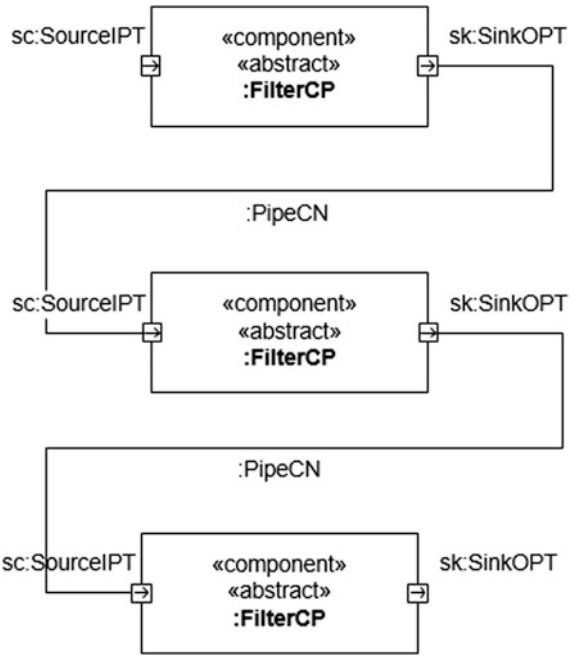


Fig. 13.8 Pipeline style overview

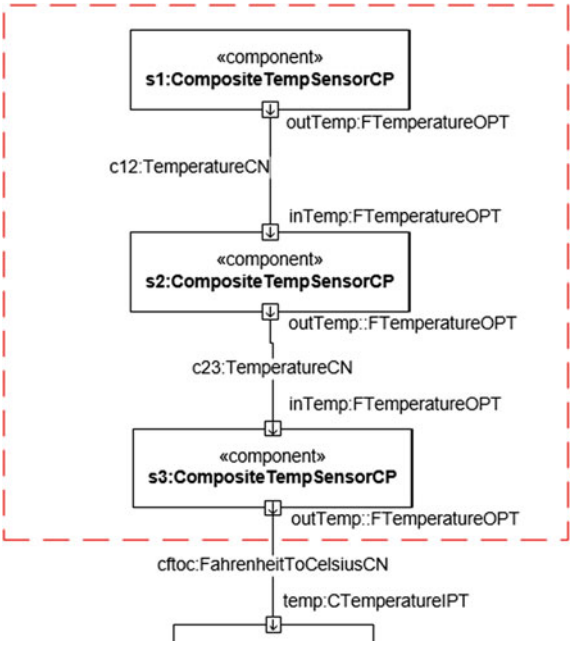


Fig. 13.9 Pipeline style example

13.5 Summary

In this chapter, you learnt the following:

- the Pipe-Filter architectural style;
- the elements, structure, and behavior of the Pipe-Filter style;
- the Pipeline substyle.

You learnt how to:

- apply the Pipe-Filter style in an architecture design.

Further Reading

1. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Michael Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns, vol. 1. Wiley (1996)
3. Vogel, O., Arnold, I., Chughtai, A., Kehrer, T.: Software Architecture: A Comprehensive Framework and Guide for Practitioners. Springer (2011)

Chapter 14
Client Server Architectural Style

In this chapter, we present and explain the client–server architectural style and how to specify it in SysADL. We specify the style using the structural and behavioral viewpoints. Finally, we illustrate the client–server style and how to use it with our running example.

You will learn the following:

- what is a client–server architectural style;
- what are its architectural elements, structure, and behavior.

14.1 Conceptual Overview

In a client–server style, components and connectors have a particular behavior. Components, called "clients," send requests to a component, called "server," and wait for a reply. A server component receives a request from a client and sends it the reply. We can use the client–server style to model a part of a system that has many components sending requests (clients) to another component (server) that offer services.

An example of client–server style is the use of a component—the client—that requests to the controller the average temperature. The controller is a server that provides the temperature value as a reply to the client component.

Figure 14.1 depicts the essential elements in a client–server configuration style. The *ClientCP* component sends requests to a server using the *ClientServerCN* connector. The *ServerCP* component offers services to reply the clients' requests. There is only one server and zero or more clients.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_14

Fig. 14.1 *Client–server* overview

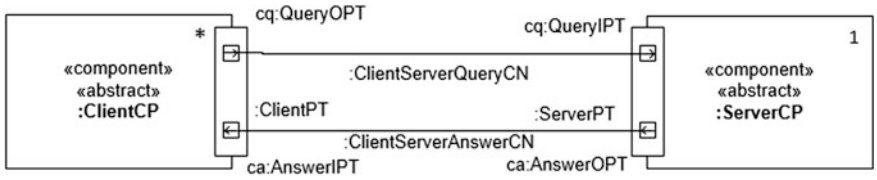


Fig. 14.2 The composite ports and connector in the *client–server* configuration

Both components have a composite port. The *ClientPT* port is composed by one *out* port, *cq:QueryOPT*, to send a request to the server, and one *in* port *ca: AnswerIPT*, to receive the answer from the server. The *ServetPT* port is composed by one *in* port, *cq:QueryIPT*, to receive the request from the clients, and one *out* port *ca:AnswerOPT* to prove the answer.

ClientServerCN is a composite connector. It is composed of two components to transmit the request and response. The *ClientServerQueryCN* connector links the *cq:QueryOPT* port to the *cq:QueryIPT* port. The *ClientServerAnswerCN* connector links the *cq:AnswerOPT* port to the *cq:AnswerIPT* port. Figure 14.2 shows the details of the composite ports and the two connectors of *ClientServerCN*. This configuration is an alternative visualization of the configuration illustrated in Fig. 14.1.

14.2 An Example of Client–Server in RTC System

To illustrate the use of a client–server in the *RTC* System, we add a new component that is a client to display the current room average temperature, which is shown in Fig. 14.3. *TemperatureDisplayCP* sends a query via the *ClientServerCN* composite connector and waits for the answer. The *ServerPT* port is a composite proxy port to an internal component that stores the last current average temperature.

The *TemperatureStorageCP* internal component is the server in this example. It receives the average temperature from the *SensorsMonitorCP* component (not shown here) in its average port (see Fig. 14.4). When it receives a query in its *sq: QueryIPT* port, it sends the average temperature in its *sa:QueryOPT* port.

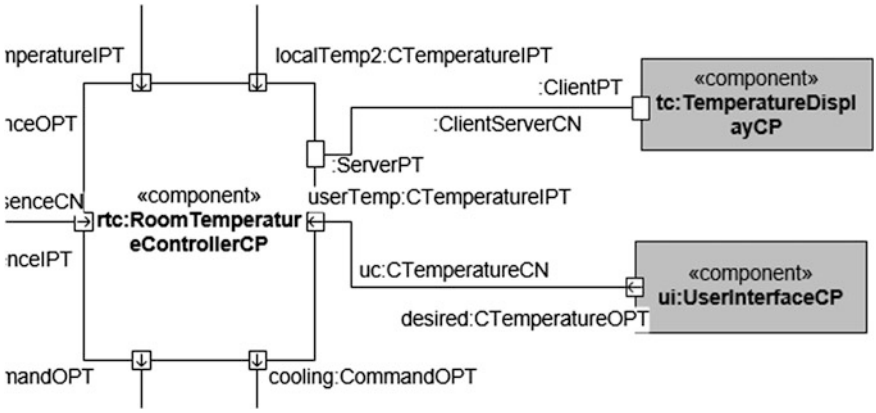


Fig. 14.3 A fragment of the *RTC System* architecture configuration showing the new TemperatureDisplayCP component

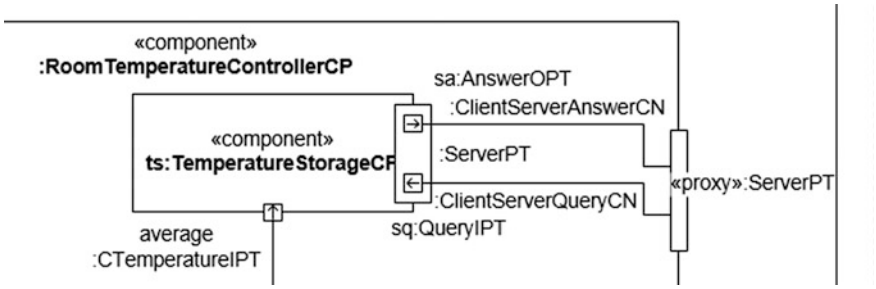


Fig. 14.4 The *TemperatureSotrageCP* component that acts as a server providing the current temperature

14.3 Client–Server Structural Viewpoint

We define the client–server architectural style using a *bdd*, as illustrated in Fig. 14.5. The client–server architecture style (*ClientServerARCH*) is composed of one *ServerCP* component and zero or more *ClientCP*.

The *ClientPT* port, depicted in Fig. 14.6, is composed by a *QueryOPT out* port and an *AnswerIPT in* port. Both ports allow values of any type. The *ServerPT* port, shown in Fig. 14.7, is composed of an *AnswerOPT out* port and a *QueryIPT in* port. Both ports allow values of any type.

In the client–server architectural style definition, we must also define the *ClientServerConnectorCN* connector, illustrated in Fig. 14.8. This connector is composed of two connectors: *ClientServerQueryCN* and *ClientServerAnswerCN*, each one is linked to the composite ports of the *ClientCP* and *ServerCP* components.

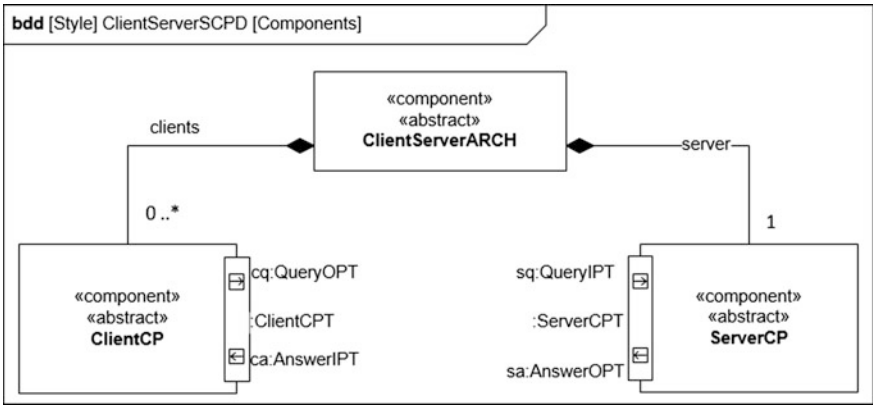


Fig. 14.5 The *client–server* component definitions

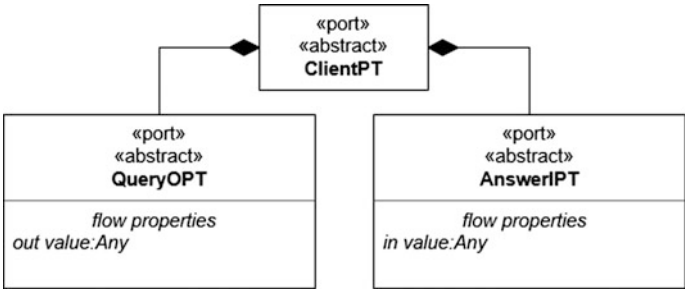


Fig. 14.6 The *ClientPT* composite port definitions

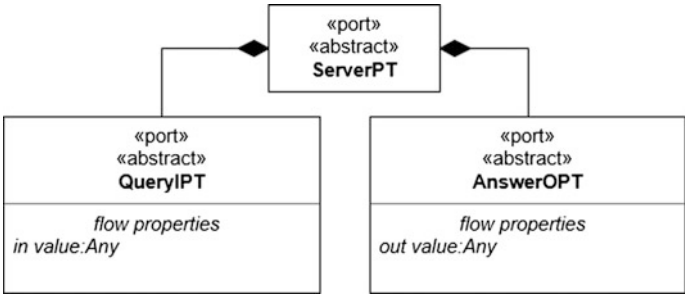


Fig. 14.7 The *ServerPT* composite port definition

The *ClientServerQueryCN* connector, depicted in Fig. 14.9, has two participants ports: *client:QueryOPT* and *server:QueryPortIPT*. The connector conveys data of any type from the client port to the server port.

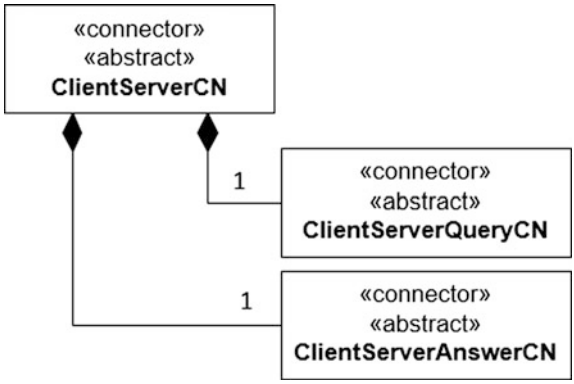


Fig. 14.8 The *ClientServerCN* composite connector definition

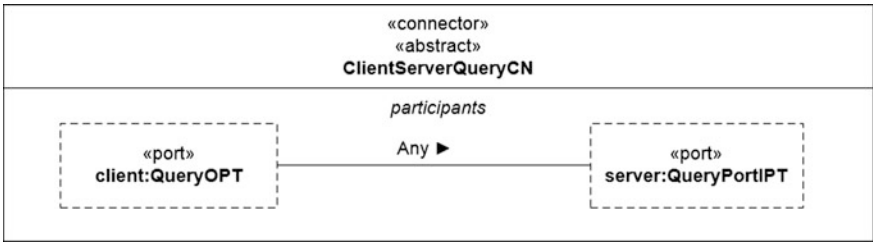


Fig. 14.9 The *ClientServerQueryCN* connector definition

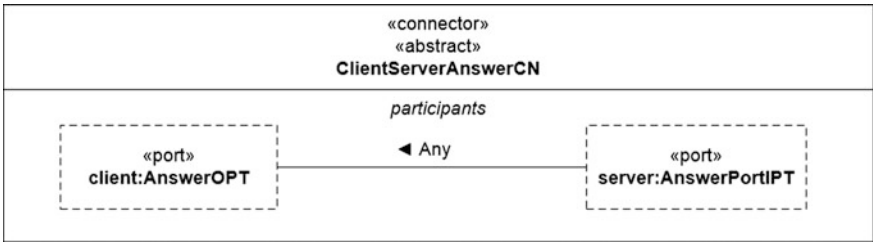


Fig. 14.10 The *ClientServerAnswerCN* connector definition

The *ClientServerAnswerCN* connector, illustrated in Fig. 14.10, has two participants ports: *client:AnswerOPT* and *server:QueryAnswerIPT*. The connector conveys data of any type from the server port to the client port.

14.4 Client–Server Behavioral Viewpoint

We can specify the behavior of a client–server style using activity diagrams specifying the behavior of the connectors and the protocols of the ports.

We define the behavior of the *ClientServerCN* connector by defining the behavior of the two compound connectors.

We define the behavior of the *ClientServerQueryCN* connector using the activity diagram (*ClientServerQueryAC*), shown in Fig. 14.11. The connector sends to its *out* pin any data received in its *in* pin.

We define the behavior of the *ClientServerAnswerCN* connector using the activity diagram (*ClientServerAnswerAC*), illustrated in Fig. 14.12. The connector sends to its *out* pin any data received in its *in* pin.

We define the protocols of the ports using activity diagrams. Since we have two composite ports, we define the protocol of the composite port and the protocols of the individual ports. Figure 14.13 shows the specification of the protocol of the *ClientPT* port.

The *QueryOPT* internal port initiates by sending a value of any type to its out pin and it waits until the *AnswerIPT* receives data. The *AnswerIPT* internal port receives data from its in pin and the gives the control to the QueryOPT port.

Figure 14.14 shows the specification of the protocol of the *ServerPT* port. The *QueryIPT* internal port initiates by receiving a value of any type from its *in* pin. After that, the control is given to the *AnswerOPT* internal port to send data (the answer) to its *out* pin. Finally, it the gives back the control to the *QueryIPT* port.

We now present the specification the protocol of the individual internal ports of the previous composite ports.

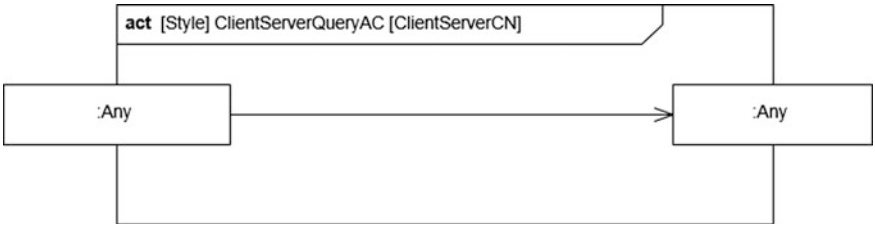


Fig. 14.11 The *ClientServerQueryCN* connector behavior specification

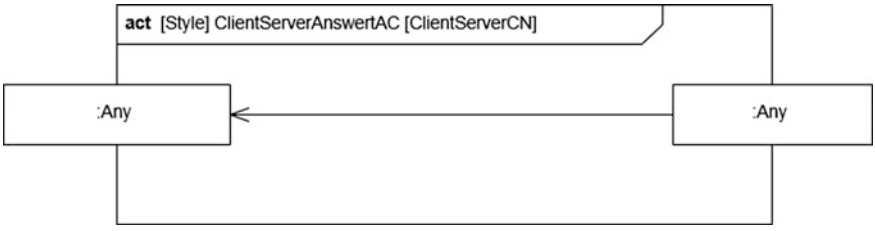


Fig. 14.12 The *ClientServerAnswerCN* connector behavior specification

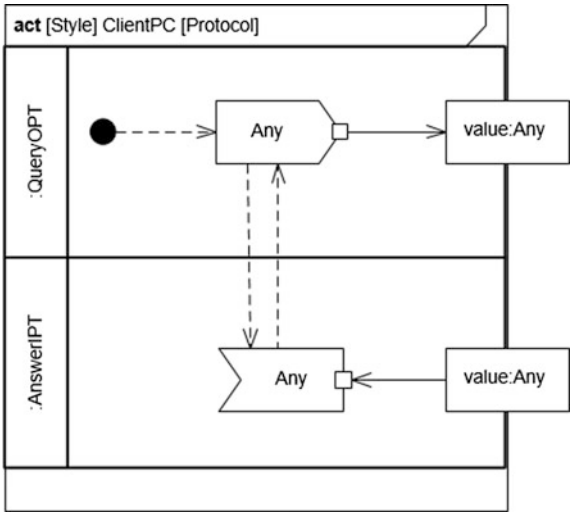


Fig. 14.13 The specification of the *ClientPC* protocol of the *ClientPT* port

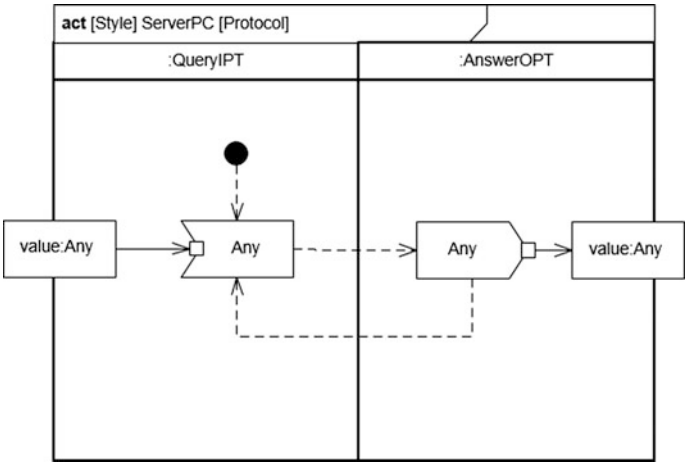


Fig. 14.14 The specification of the *ServerPC* protocol of the *ServerPT* port

Figure 14.15 shows the internal ports of *ClientPT*. The *QueryOPT* internal port initiates by sending a value of any type to its *out* pin, and then it waits to send another value. The *AnswerIPT* internal port initiates by receiving data from its *in* pin, and then it waits to receive another value.

Figure 14.16 shows the internal ports of *ServerPT*. The *QueryIPT* internal port initiates by receiving a value of any type from its *in* pin, and then it waits to receive another value. The *AnswerOPT* internal port initiates by sending data to its *out* pin, and then it waits for another value to send.

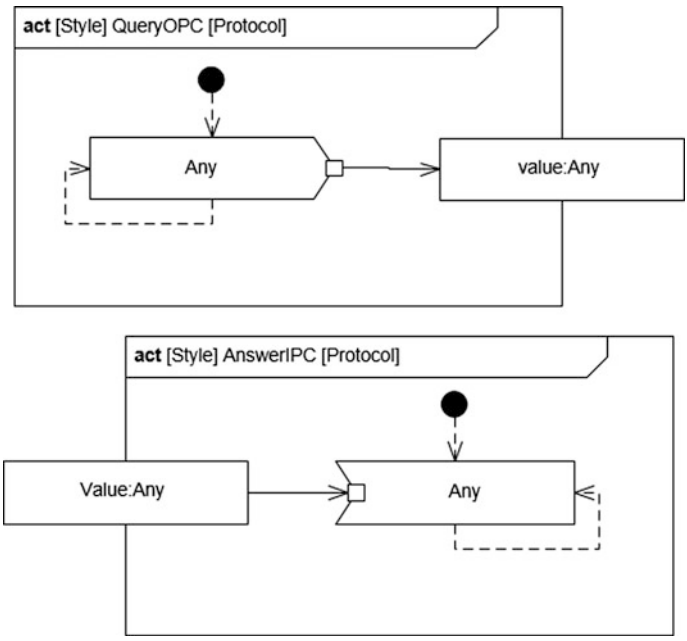


Fig. 14.15 The specification of the protocols of the *QueryIPT* and *AnswerOPT* ports

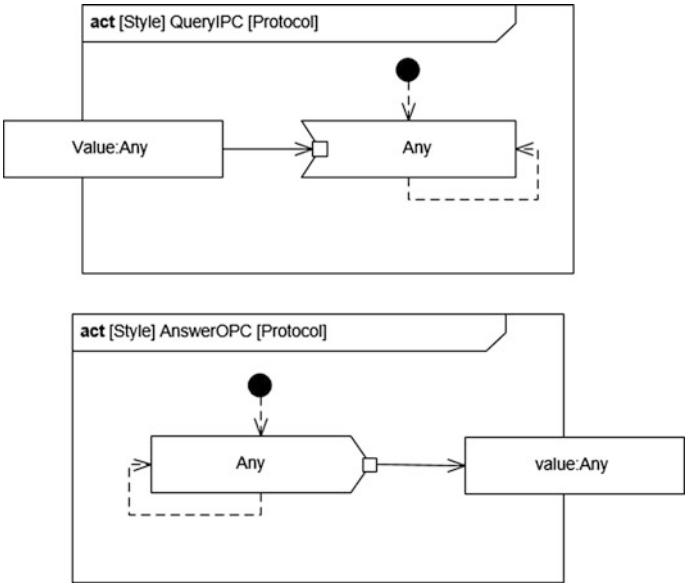


Fig. 14.16 The specification of the protocols of the *QueryOPT* and *AnswerPT* ports

14.5 Summary

In this chapter, you have learnt the following:

- the client–server architectural style;
- the elements, structure, and behavior of the client–server style.

You learnt how to:

- apply the client–server style in an architecture design.

Further Reading

1. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Michael Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns, vol. 1. Wiley (1996)
3. Vogel, O., Arnold, I., Chughtai, A., Kehrer, T.: Software Architecture: A Comprehensive Framework and Guide for Practitioners. Springer (2011)

Chapter 15
Feedback Control Loop Architectural Style

In this chapter, we present and explain the Feedback Control Loop architectural style and how to specify it in SysADL. We specify the style using the structural and behavioral viewpoints. Finally, we illustrate the Feedback Control Loop style and how to use it with our running example.

You will learn the following:

- what is a Feedback Control Loop architectural style;
- what are its architectural elements, structure, and behavior.

15.1 Conceptual Overview

In a Feedback Control Loop style, components and connectors are configured to allow a central component to control several actuators by analyzing information from sensors.

The Feedback Control Loop has three types of components and two types of connectors. Sensors components read information from environment. The controller uses this information to control the actuators by sending commands. Actuators components change the environment executing these commands. This process is a feedback loop where the controller uses sensors data from the environment to command actuators to act in the environment. Note that this is a continuous loop where the actuators change the environment that will then be sensed by the sensors.

We can use the Feedback Control Loop style to model a part of a system that has a central controller to control one or more actuators by using data from one or more sensors.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_15

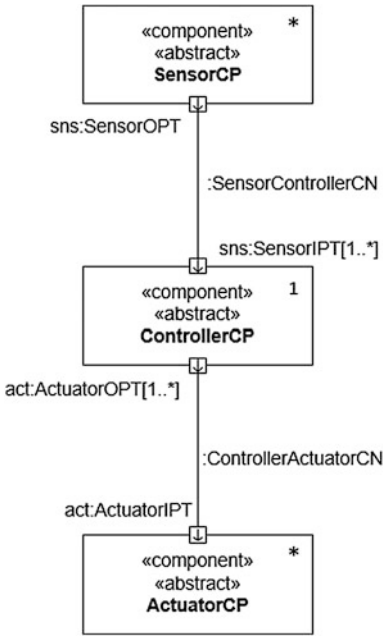


Fig. 15.1 The Feedback Control Loop architectural style

Figure 15.1 shows an overview of the Feedback Control Loop. The *Controller CP* component receives sensor data from one or more *SensorCP* components. The *ControllerCP* component sends commands to one or more *ActuatorCP* components.

An example of the Feedback Control Loop style is the *RTC* System that we have been using in this book. The sensors monitor the temperature and the presence of persons in the room. The heater and cooler are the actuators that change the temperature of heating or cooling the room. The controller controls the temperature by turning the heater or the cooler on or off, according to the current temperature, the desired temperature, and the presence of a person in the room.

To illustrate the use of a Feedback Control Loop in the *RTC* System, Fig. 15.2 presents the *RTC* System Configuration previously presented in Chap. 4.

15.2 Feedback Control Loop Structural Viewpoint

We define the Feedback Control Loop architectural style using a *bdd*, as shown in Fig. 15.3. The Feedback Control Loop Architecture Style (*FeedbackControlARCH*) is composed by one central *ControllerCP* component, one or more *SensorCP* component, and one or more *ActuatorCP* component.

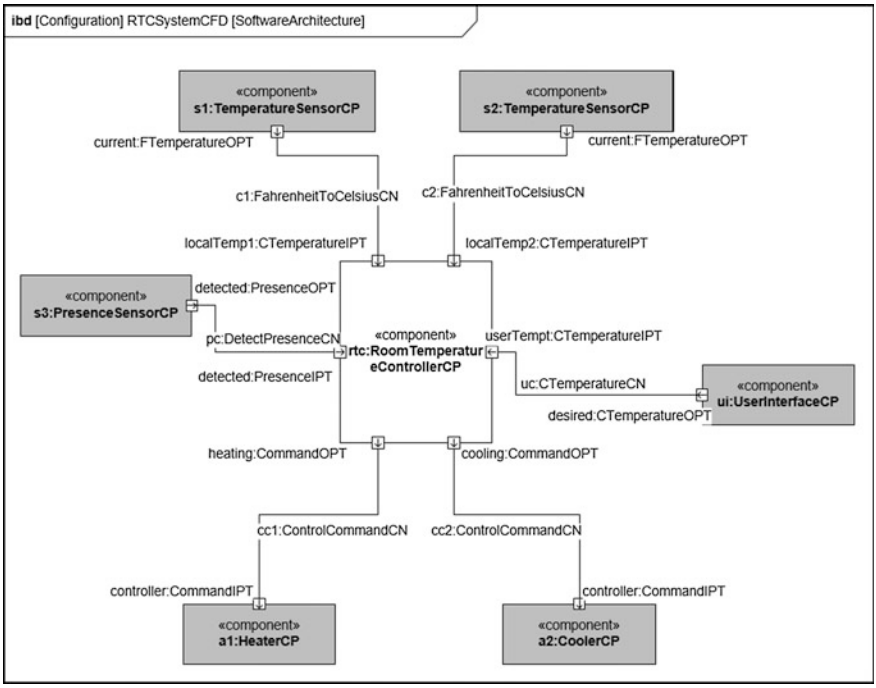


Fig. 15.2 The *RTC* system as an example of Feedback Control Loop architectural style

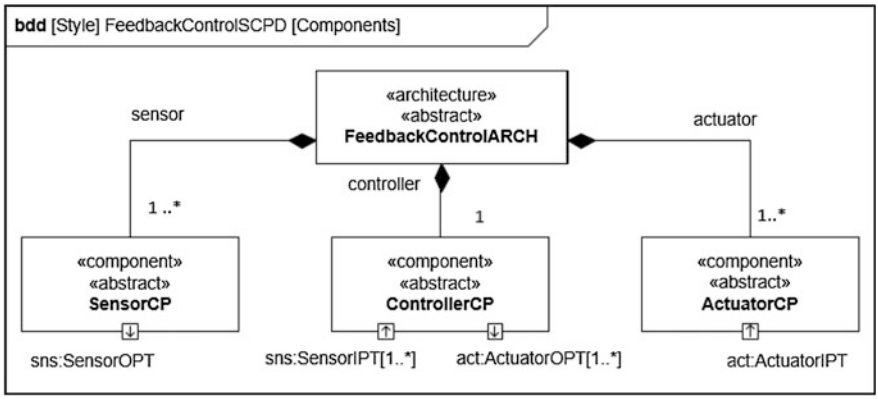


Fig. 15.3 The Feedback Control Loop architectural style structural definition using BDD

In the Feedback Control Loop architectural style definition, we must also define the ports, as shown in Fig. 15.4. The *SensorCP* component has an *out* port (*SensorOPT*) to provide data of any type. The *ActuatorCP* component has an *in* port (*ActuatorIPT*) to receive data of any type. The *ControllerCP* component has

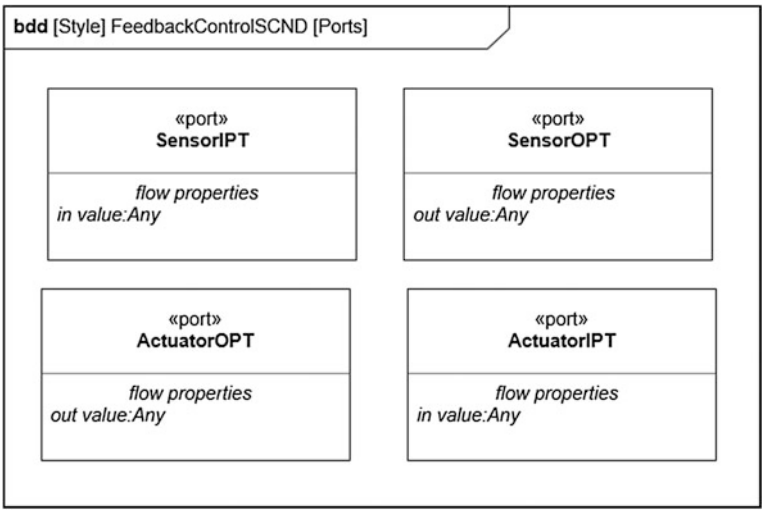


Fig. 15.4 The ports definition in the Feedback Control Loop architectural style

one or more *SensorIPT in* port to receive data from sensor and one or more *ActuatorOPT out* port to send data to one or more actuators.

In the Feedback Control Loop architectural style definition, we must also define the connectors that link the *ControllerCP* to the other two components. Figure 15.5

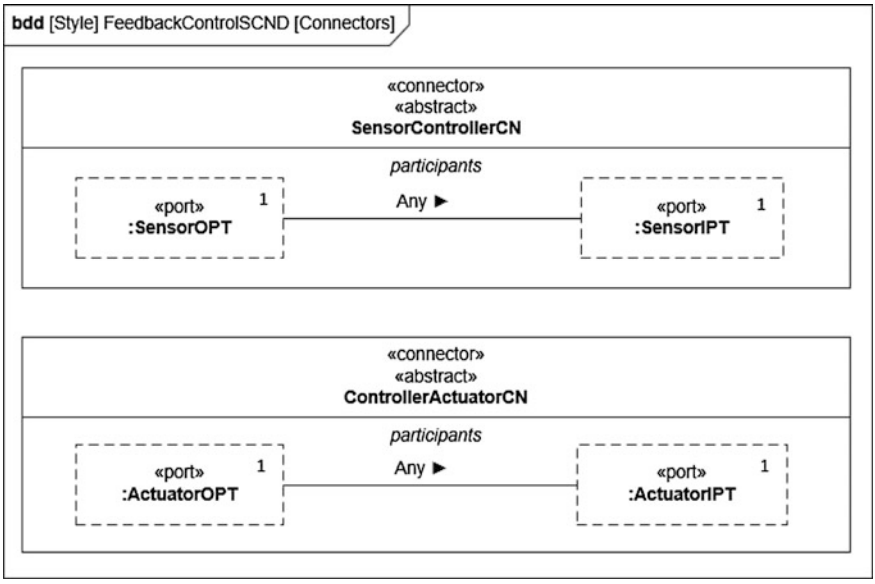


Fig. 15.5 Connectors definition in the Feedback Control Loop architectural style

illustrates the connectors definition. The *SensorControllerCN* connector conveys data of any type from the *SensorOPT out* port of the *SensorCP* component to the *SensorIPT in* port of the *ControllerCP* component. The *ControllerActuatorCN* connector conveys data of any type from the *ActuatorOPT out* port of the *ControlerCP* component to the *ActuatorIPT in* port of the *ActuatorCP* component.

15.3 Feedback Control Loop Behavioral Viewpoint

We can specify the behavior of a Feedback Control Loop style using activity diagrams specifying the behavior of the controller, the connectors, and the protocols of the ports.

The behavior of the controller can be depicted in terms of one action, called *ControllerAN* as shown in Fig. 15.6. This action waits to read the data from all sensors, then decides the commands to send to the actuators. It sends the commands to the actuators to enable each of them to act on the environment. Figure 15.6 depicts this behavior in an activity diagram.

The behavior of the connectors is simple. It sends to its *out* pin any data received in its *in* pins, as illustrated in Fig. 15.7.

We specify the protocols of the ports using the activity diagrams in Fig. 15.8. Figure 15.8a specifies the *SensorOPC* protocol of the *SensorOPT* port. The protocol states that the port sends data to its *out pin* in sequence. Figure 15.8b specifies that the *SensorIPT* port receives data from its *in pin* in sequence. In Fig. 15.8c, we see the specification of the *ActuatorOPT* port sending data to its *out pin* in sequence. Figure 15.8d, specifies the *ActuatorIPC* protocol of the *ActuatorIPT* port that receives data from its *in pin* in sequence.

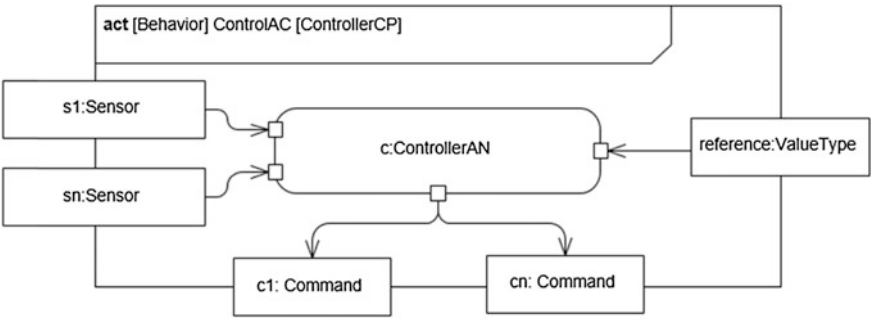


Fig. 15.6 The Feedback Control Loop architectural style behavior specification

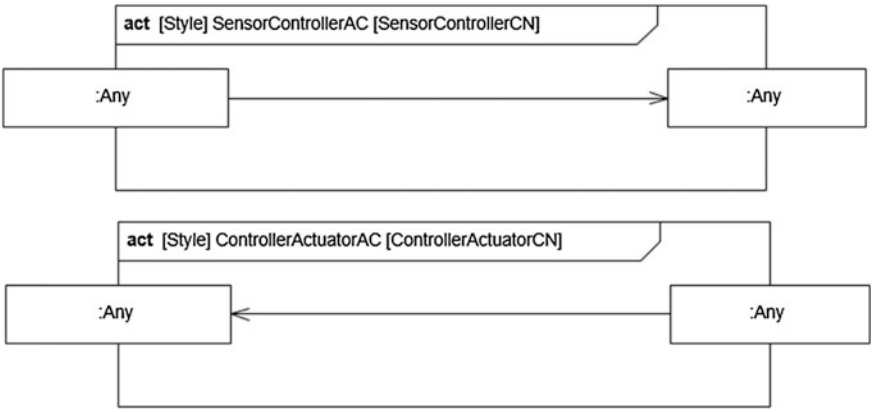


Fig. 15.7 The connectors behavior specification

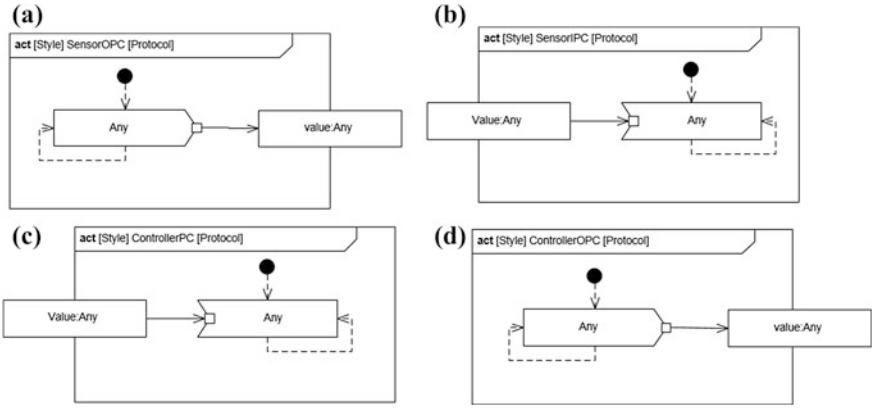


Fig. 15.8 The ports behavior specification

15.4 Summary

In this chapter, you learnt the following:

- the Feedback Control Loop architectural style
- the elements, structure, and behavior of the Feedback Control Loop style.

You learnt how to

- apply the Feedback Control Loop style in an architecture design.

Further Reading

1. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)
2. Buschmann, F., Meunier, R., Rohnert, H. Sommerlad, P., Michael Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns, vol. 1. Wiley (1996)
3. Vogel, O., Arnold, I., Chughtai, A., Kehrer, T.: Software Architecture: A Comprehensive Framework and Guide for Practitioners. Springer (2011)

Chapter 16
Blackboard Architectural Style

In this chapter, we present and explain the Blackboard architectural style and how to specify it in SysADL. We specify the style using the structural and behavioral views. Finally, we illustrate the Blackboard style and how to use it with our running example.

You will learn the following:

- what is a Blackboard architectural style;
- what are its architectural elements, structure, and behavior.

16.1 Conceptual Overview

In the Blackboard architectural style, there is a shared data structure—the blackboard—and multiple knowledge sources that interact with each other using the blackboard. The knowledge sources write information in the blackboard and read information from the blackboard (inspect the blackboard). The blackboard is the central element that coordinates the interaction between the knowledge sources.

For instance, one could use a Blackboard style to design the architecture of an image processing system where the image acting as the Blackboard is annotated with recognition elements by different knowledge sources. Each knowledge source could use a different algorithm of image processing cooperatively achieving the image recognition.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_16

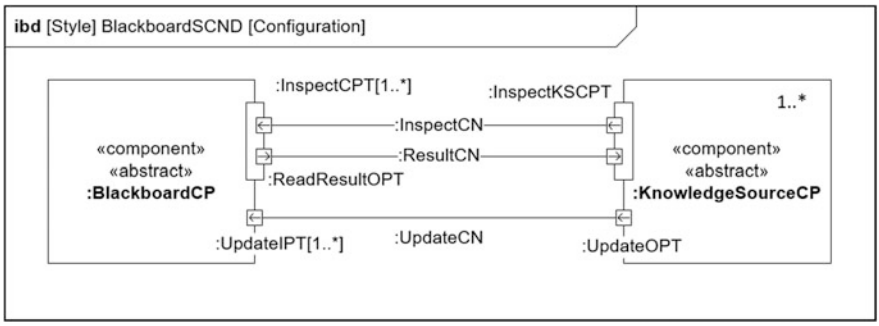


Fig. 16.1 Blackboard overview

Figure 16.1 shows an overview of the Blackboard. The *BlackboardCP* component sends data to one or more *KnowledgeSourceCP* components. The *Knowledge SourceCP* components send data to update the *BlackboardCP* component.

16.2 Blackboard Structural Viewpoint

We define the Blackboard architectural style using a bdd, as shown in Fig. 16.2. The Blackboard architectural style (ARCH7) is composed by one *BlackboardCP* component and one or more *KnowledgeSourceCP* component.

In the Blackboard architectural style definition, we must also define the ports, as shown in Fig. 16.3. The *BlackboardCP* component has an *in* port (*UpdateIPT*) to receive data of any type from one or more *KnowledgeSourceCP*, an out port

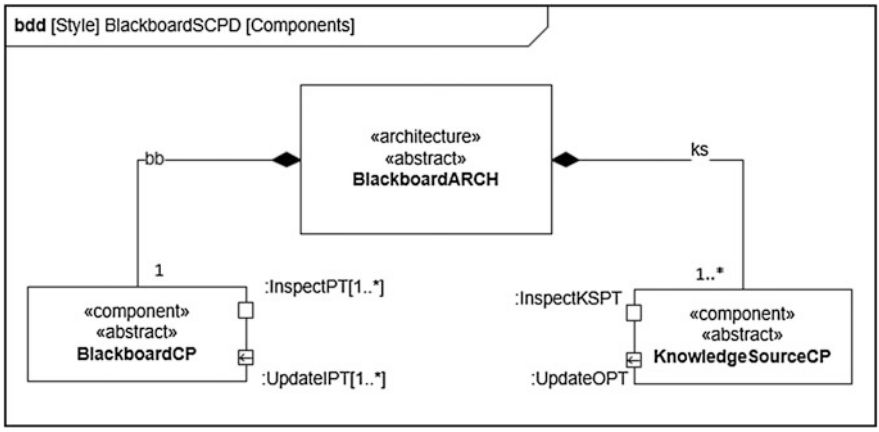


Fig. 16.2 The Blackboard architectural style structural component definitions using BDD

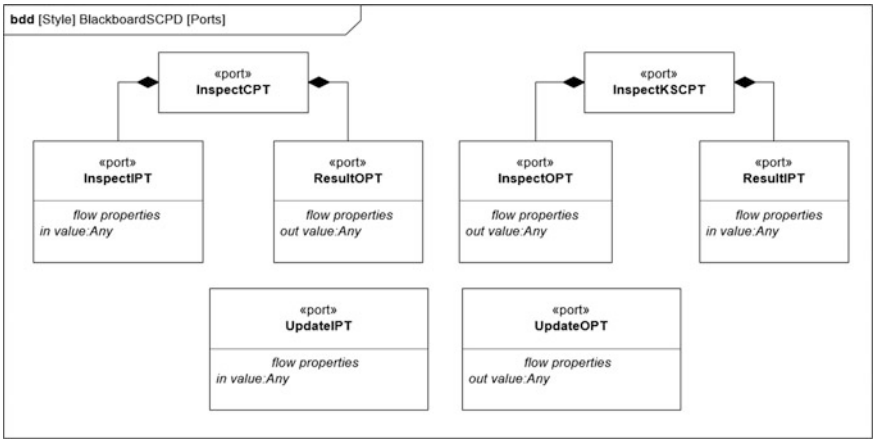


Fig. 16.3 The ports definition in the Blackboard architectural style

(*UpdateOPT*) to send data of any type to one or more *KnowledgeSourceCP*, a *ReadPT* composite port, composed of a *ReadCmdIPT in* port and a *ResultOPT* out port, a *ReadKSPT* composite port, composed of a *ReadCmdOPT out* port, and a *ResultIPT in* port.

In the Blackboard architectural style definition, we must also define the connectors that link the *BlackboardCP* to the *KnowledgeSourceCP* components, as illustrated in Fig. 16.4. The *InspectReadCN* connector is composed of two connectors: *InspectCN* and *ResultCN*, each one is linked to the composite ports of the *BlackboardCP* and *KnowledgeSourceCP* components. The *UpdateCN* connector conveys data of any type from the *UpdateOPT out* port of the *KnowledgeSourceCP* component to the *UpdateIPT in* port of the *BlackBoardCP* component.

16.3 Blackboard Behavioral Viewpoint

We can specify the behavior of a Blackboard style using activity diagrams specifying the behavior of the connectors and the protocols of the ports.

We define the behavior of the *InspectResultCN* connector by defining the behavior of the two compound connectors: *InspectCN* and *ResultCN*. We define the behavior of the *InspectCN* connector using the *InspectAC* activity diagram, shown in Fig. 16.5. The connector sends to its *out* pin any data received in its *in* pin. The *ResultAC* activity diagram represents the behavior of the *ResultCN* connector, by sending to its *out* pin any data received in its *in* pin.

We define the behavior of the *UpdateCN* connector using the *UpdateAC* activity diagram, shown in Fig. 16.5. The connector sends to its *out* pin any data received in its *in* pin.

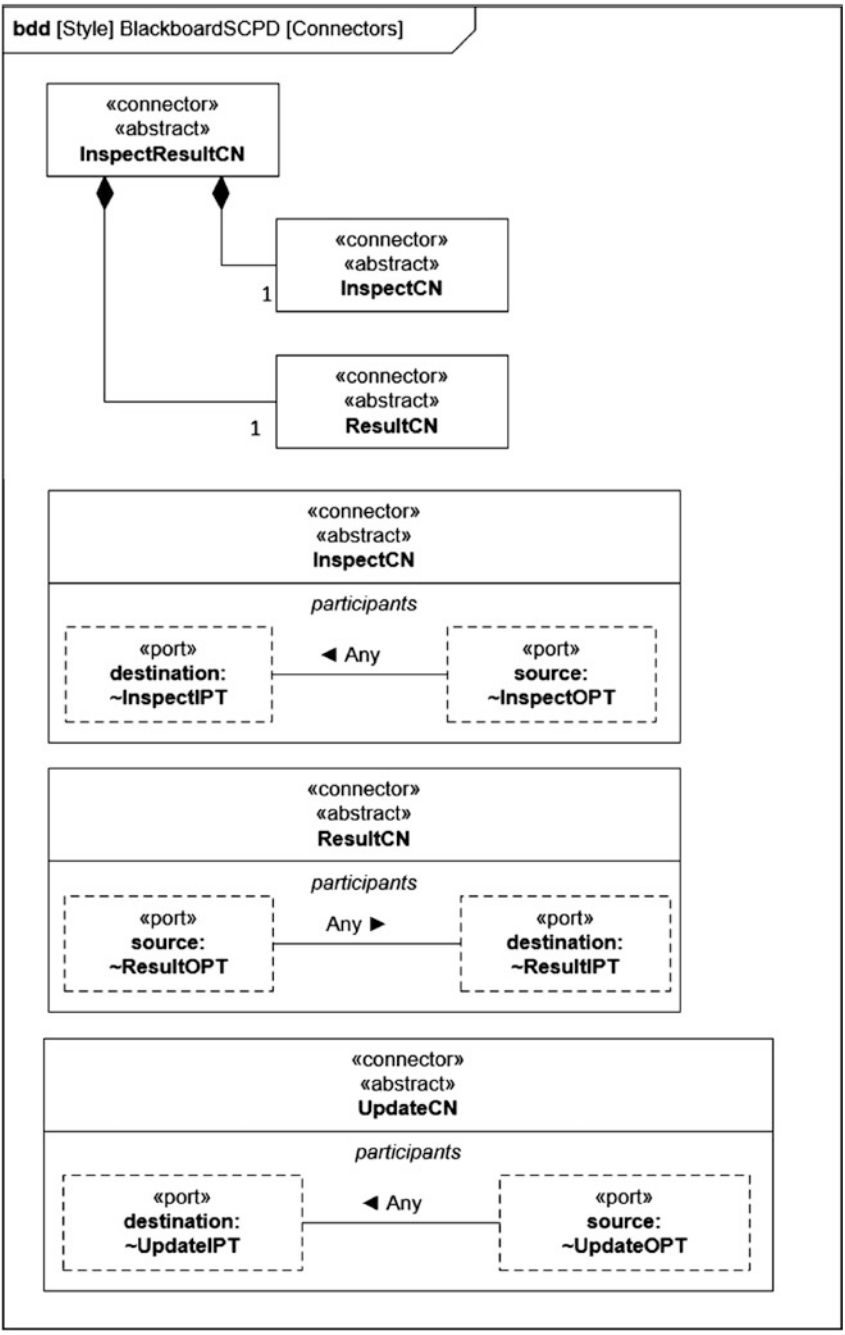


Fig. 16.4 The connectors definition in the Blackboard architectural style

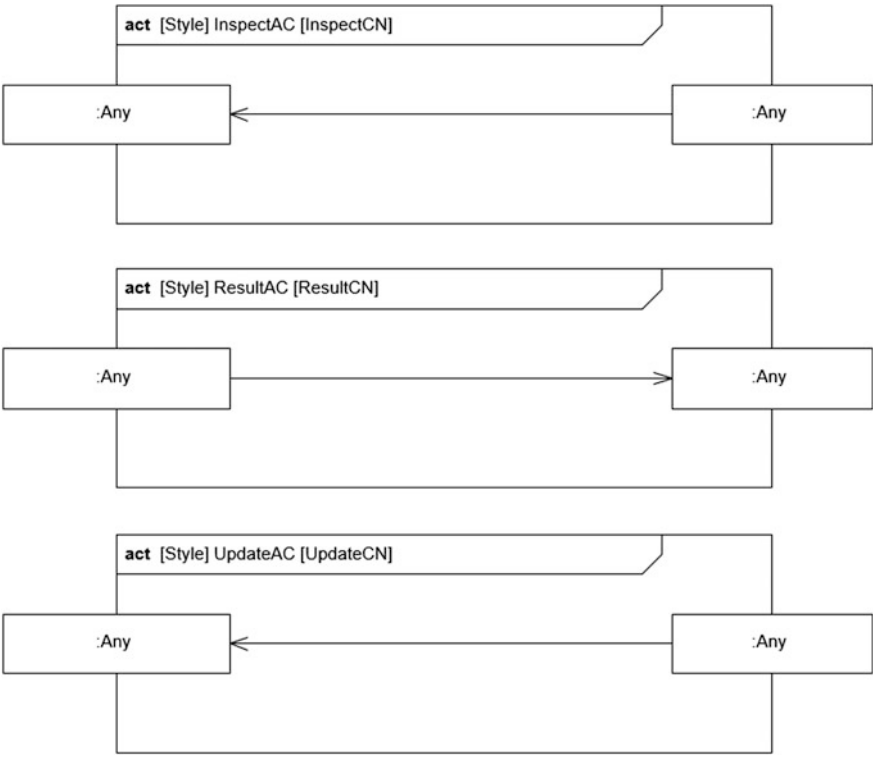


Fig. 16.5 The connectors behavior specification in the Blackboard architectural style

We define the protocols of the ports using activity diagrams. Since we have two composite ports, we define the protocol of the composite port and the protocols of the individual ports. Figure 16.6 shows the specification of the protocol of the *InspectCPT* port. It states that first a value is received via the *InspectIPT* port, then a value is sent via the *ResultOPT* port and so on.

Figure 16.7 shows the specification of the protocol of the *InspectKSPT* composite port. It states that first a value is sent via the *InspectOPT* port, then a value is received via the *ResultIPT* port, and so on.

Figure 16.8 shows the specification of the protocol of the individual *Inspect* ports. The *InspectIPT* internal port initiates by receiving data from its *in* pin, and then it waits to receive another value. The *InspectOPT* internal port initiates by sending a value of any type to its *out* pin, and then it waits to send another value.

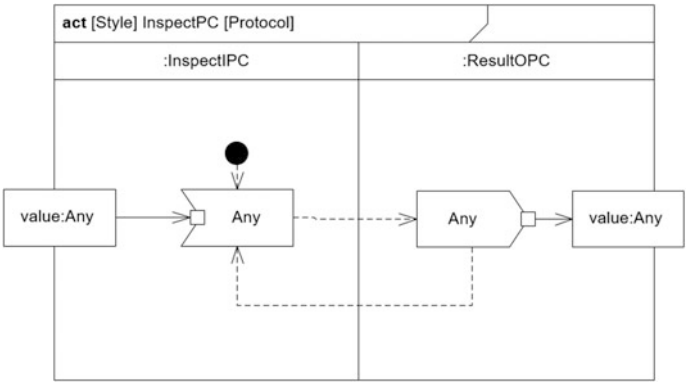


Fig. 16.6 The protocol specification of the *InspectPC* composite port

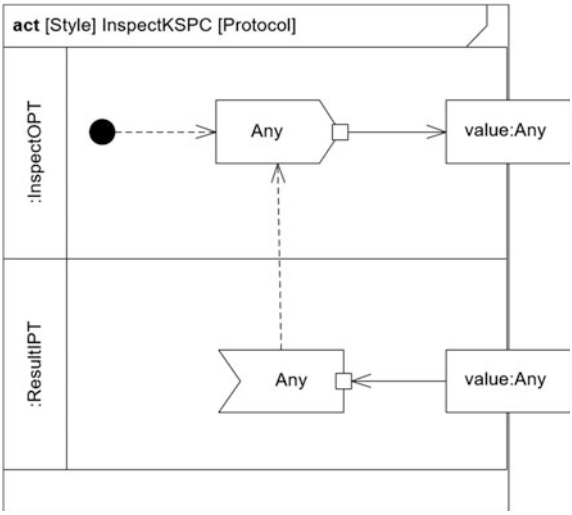


Fig. 16.7 The protocol specification of the *InspectKSPC* composite port

Figure 16.9 shows the specification of the protocol of the individual *Result* ports. The *ResultIPT* internal port initiates by receiving data from its *in* pin, and then it waits to receive another value. The *ResultOPT* internal port initiates by sending a value of any type to its *out* pin, and then it waits to send another value.

Figure 16.10 shows the specification of the protocol of the individual *Update* ports. The *UpdateOPT* port initiates by sending a value of any type to its *out* pin, and then it waits to send another value. The *UpdateIPT* port initiates by receiving data from its *in* pin, and then it waits to receive another value.

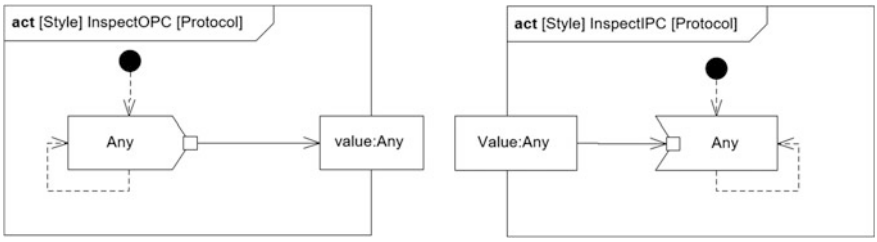


Fig. 16.8 The protocol specifications of the *InspectOPC* and *InspectIPC* ports

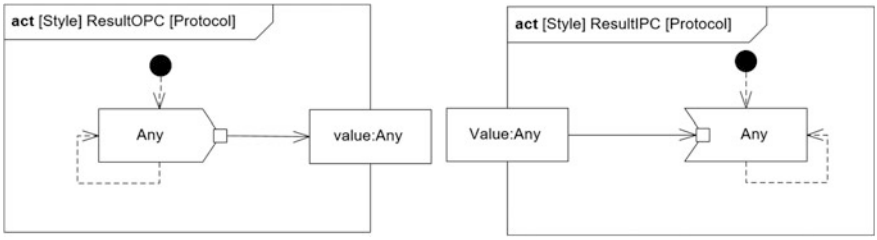


Fig. 16.9 The protocol specifications of the *ResultIPC* and *ResultOPC*

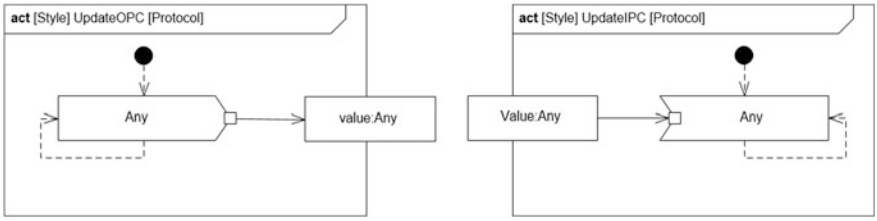


Fig. 16.10 The protocol specification of the (**a**) *UpdateOPT* and (**b**) *UpdateIPT* ports

16.4 Tuple Space

The Tuple space style is a substyle of the Blackboard style that defines a tuple space providing a shared memory (a blackboard) where tuples (data) are inserted and retrieved by other components. Figure 16.11 shows the specialization definition of the Blackboard style where the *TupleSpaceSP* component is a specialization of the *BlackboardCP* and the WorkerCP component is a specialization of the *Knowledge SourceCP* component.

An example of the use of the Tuple space style in the RTC System is an architecture with the temperature sensors inserting temperatures (tuples) in a connector (the tuple space) that are retrieved by the controller.

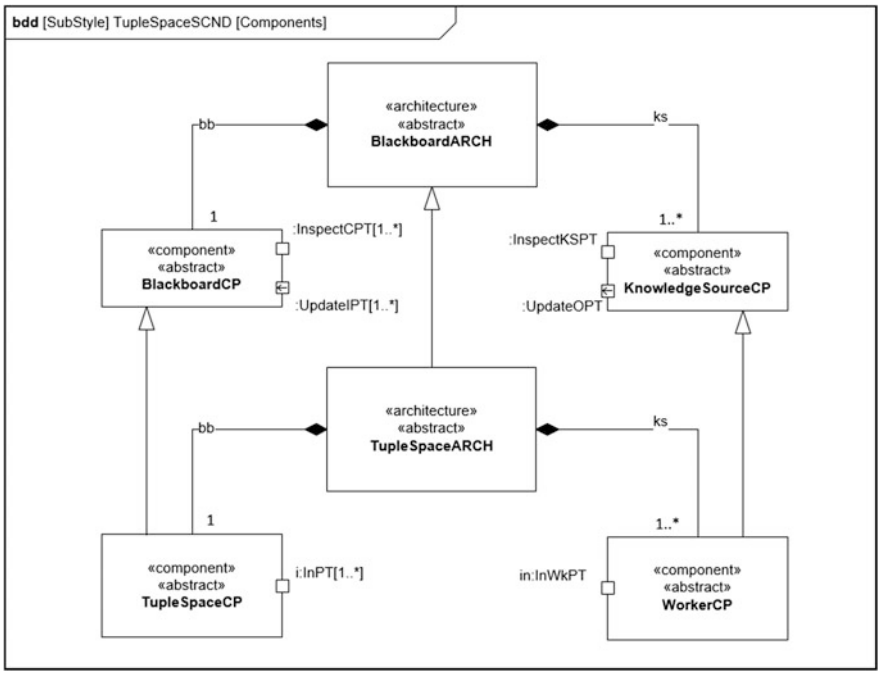


Fig. 16.11 The Tuple Space substyle as a specialization of blackboard

In Fig. 16.12 we define a new connector used in the Tuple Space style, the *InCN* composite connector. It is composed of two connectors: *InCmdCN* and *ReadResultCN*, each one is linked to the composite ports of the *TupleSpaceCP* and *WorkerCP* components. The *InCmdCN* connector conveys data of any type from the *InCmdOPT out* port of the *WorkerCP* component to the *InCmdIPT in* port of the *TupleSpaceCP* component. The *ReadResultCN* connector conveys data of any type from the *InResultIPT in* port of the *TupleSpaceCP* component and the *InResultOPT out* port of the *WorkerCP* component.

Figure 16.13 shows a configuration where the *TupleSpaceCP* component is connected to *WorkerCP* components (one or more) via three connectors: (i) the *InspectCN* composite connector that allows the *WorkerCP* components to inspect (i.e., nondestructively read) the *TupleSpaceCP* component; (ii) the *InCN* composite connector, that allows the *WorkerCP* components to consume (i.e., read and remove) a tuple from the *TupleSpaceCP* component; (iii) the *UpdateCN* connector that allows the WorkerCP components to insert a new tuple into the *TupleSpaceCP* component.

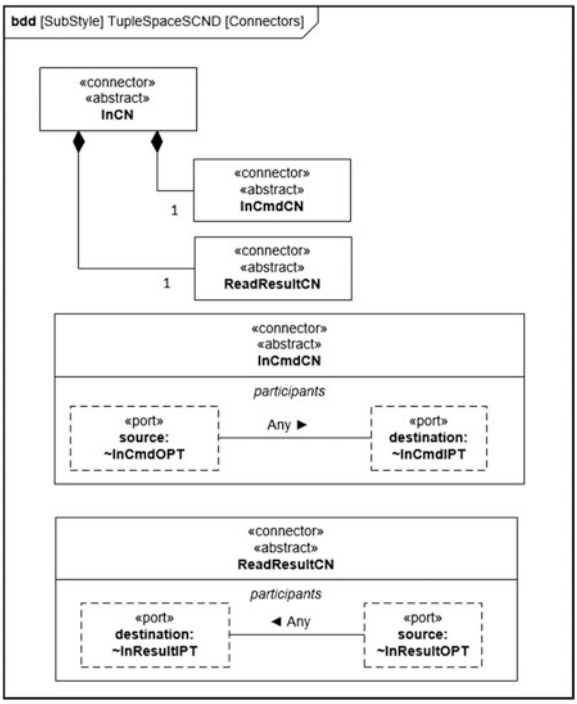


Fig. 16.12 The new connectors of the Tuple Space substyle

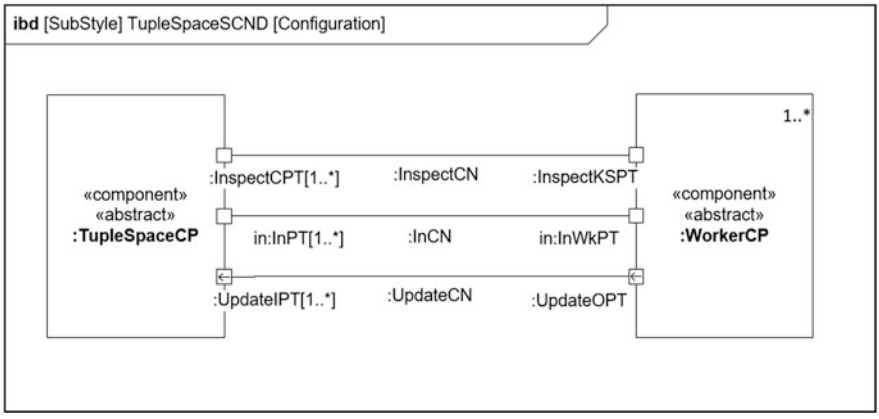


Fig. 16.13 An abstract configuration to the Tuple Space substyle

16.5 An Example of Blackboard in RTC System

To illustrate the use of the Tuple space style in the RTC System, Fig. 16.14 depicts the RTC System configuration previously presented in this book. In this new architecture (ARCH7), we use a connector as a *TupleSpace*. In this example, the connector acts as a repository for the temperatures provided by the sensor. In the configuration in ARCH7 (Fig. 16.14) there are two temperature sensors providing values to the tuple space connector. The values are read by the *RoomTemperature ControllerCP-7* component in the *temp:CTemperatureCPT* port.

Figure 16.15 depicts the definition of the RTC components used in the *TupleSpace* style. We highlight the *SensorsMonitorCP-7* component and the new port *CtemperaturesCPT* in the *RoomTemperatureControlerCP-7* component. This new port allows the connection with the *TempTupleSpaceCN* connector.

Figure 16.16 shows the definition of the *CtemperaturesCPT* composite port that is composed of two *CTemperatureIPT in* ports. These ports are used to receive temperatures from the *TempTupleSpaceCN* connector (described below).

Figure 16.17 shows the definition of the new RTC connectors: the *Temp TupleSpaceCN* connector and the *ControlCommandCN* connector. The *Temp TupleSpaceCN* connector conveys temperature data from the ∼*FTemperatureOPT*

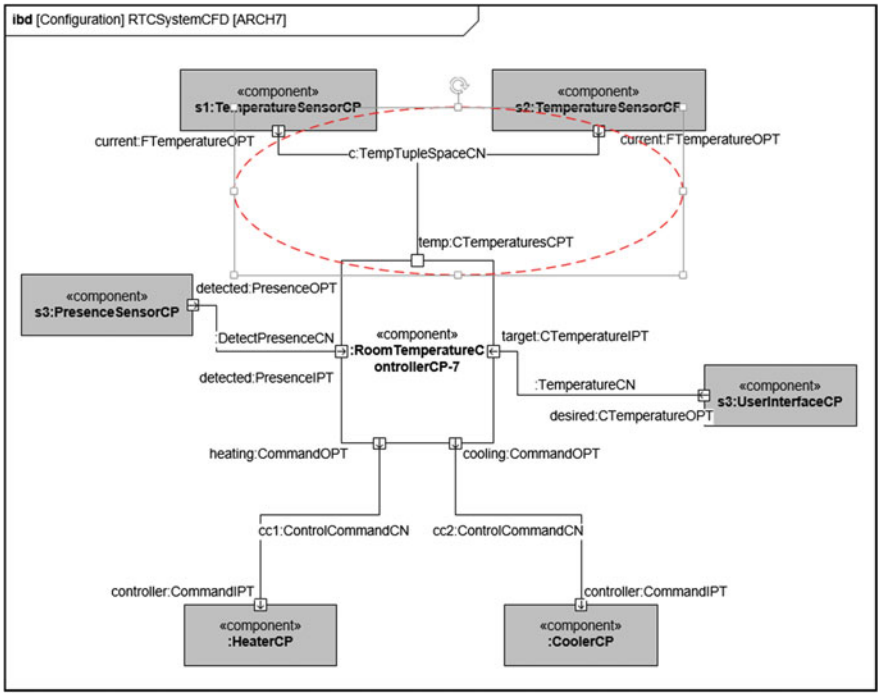


Fig. 16.14 The RTC System as an example of the Blackboard architectural style

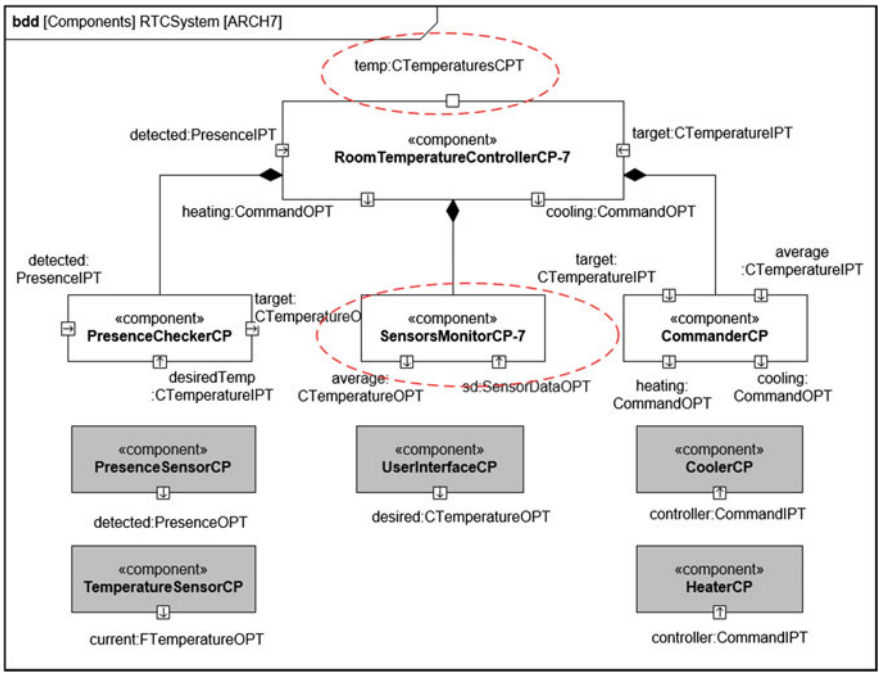
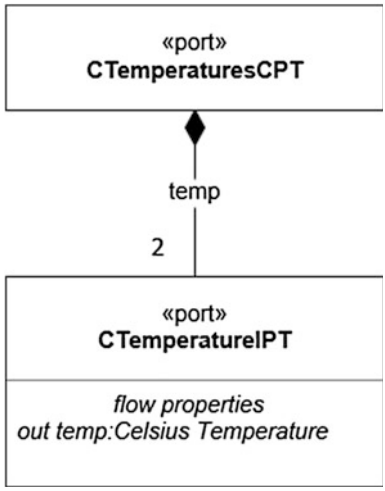


Fig. 16.15 The components definitions of the RTC System applying a Tuple space

Fig. 16.16 The new ports definitions of the RTC System applying a Tuple space



out port of the *TemperatureSensorCP* component to the ∼*CTemperaturesCPT composite* port of the *RoomTemperatureControllerCP-7* component. The *ControlCommandCN* connector conveys Command data from the ∼*CommandOPT*

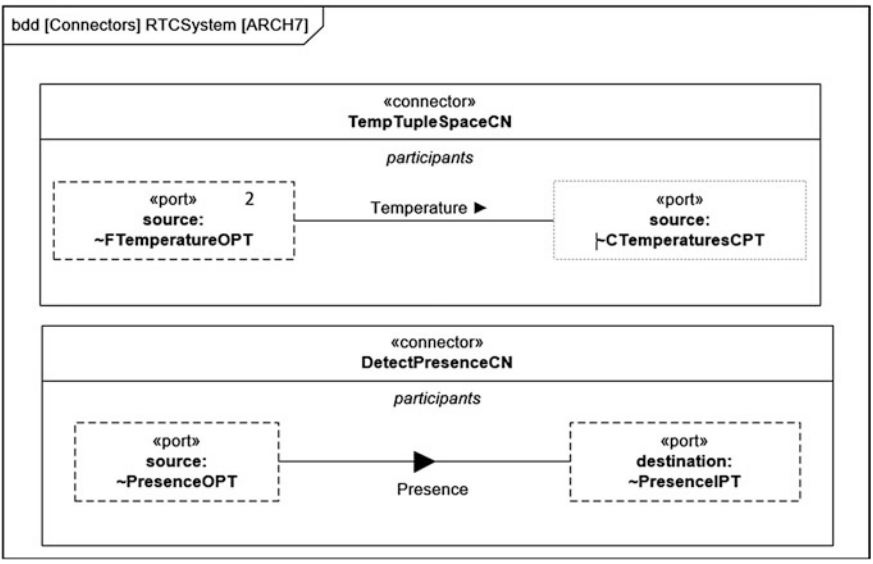


Fig. 16.17 The new connectors definitions of the RTC System applying a Tuple space

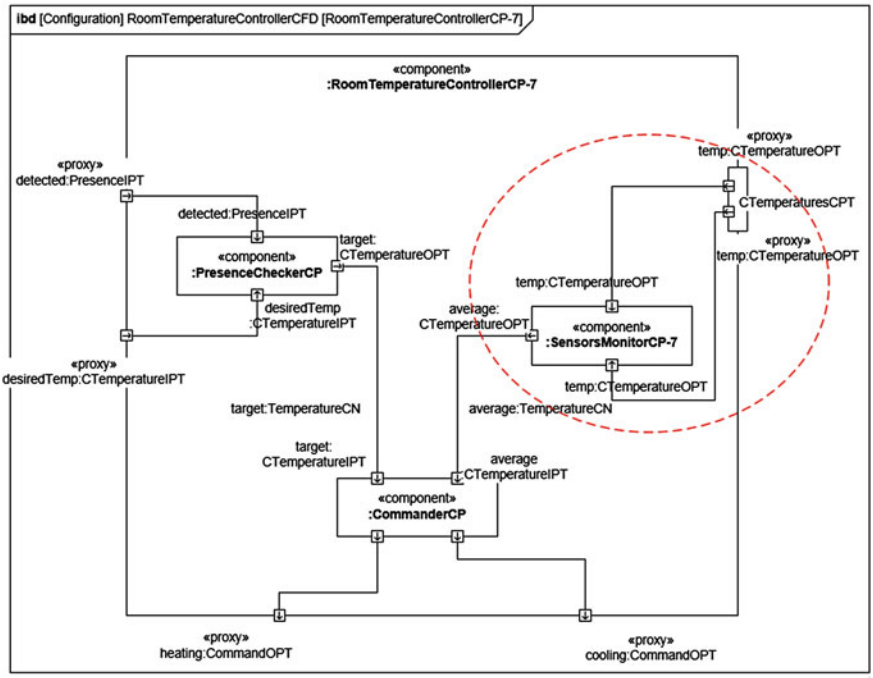


Fig. 16.18 The configuration of RoomTemperatureControllerCP-7

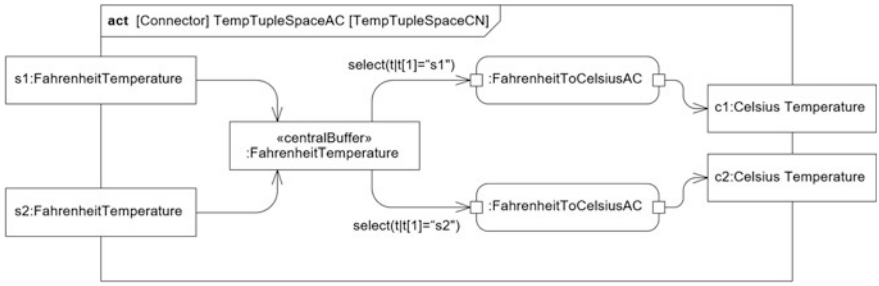


Fig. 16.19 The behavior specification of the *TempTupleSpaceAC* activity to the *TempTupleSpaceCN* connector

out port of the *RoomTemperatureControllerCP-7* component and ∼*CommandIPT* *in* port of the *Heater or Cooler* components.

Figure 16.18 highlights part of the configuration that connects the *SensorsMonitorCP-7* component to the *CTemperaturesCPT* composite port.

In Fig. 16.19 we specify the behavior specification of the *TempTupleSpaceCN* connector using the activity diagram. It receives two *FahrenheitTemperature* in its *in* pins (*s1*, *s2*) and stores them in a buffer. It selects *s1* to send to the *FahrenheitToCelsiusAC* action that will convert it to Celsius and send the result to the *c1 out* pin. Similarly, it also selects *s2* to send to the *FahrenheitToCelsiusAC* action that will convert it to Celsius and send the result to the *c2 out* pin.

16.6 Summary

In this chapter, you learnt the following:

- the Blackboard architectural style;
- the elements, structure, and behavior of the Blackboard style;
- the Tuple space architectural substyle of the Blackboard style;
- the elements, structure, and behavior of the Tuple space style.

You learnt how to

- apply the Blackboard style in an architecture design;
- apply the Tuple space substyle in an architecture design.

Further Reading

1. Clements, P., Bachmann, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R.: Documenting Software Architecture: Views and Beyond. SEI Series in Software Engineering (2003)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P.: Michael Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns, vol. 1. Wiley (1996)
3. Vogel, O., Arnold, I., Chughtai, A., Kehrer, T.: Software Architecture: A Comprehensive Framework and Guide for Practitioners. Springer (2011)

Part IV
Textual Description of Architectures

Chapter 17
Textually Representing Software Architectures

In this chapter we present the textual notation of SysADL. We describe the SysADL constructs (structural, behavioral, and executable) in the textual notation. We describe, in details, the concepts underlying each of these constructs and how each one is applied to textually describe the architecture. We illustrate each definition and how to use it with our running example.

You will learn the following:

- the SysADL textual notation to express the structural, behavioral, and executable viewpoints;
- the SysADL grammar for each architectural construct;
- how to use the textual notation to define an architecture;

17.1 Introduction

We presented, in Part I, the SysADL constructs in terms of visual diagrams. In this chapter, we will present the textual notation that corresponds to the SysADL constructs represented in the diagrams. The expressiveness of both diagrammatic and textual notation is equivalent as all the architectural abstractions expressed in the diagrammatic notation can be also expressed in the textual notation.

SysADL provides both notations to allow the architect choose the most suitable to describe the different parts of the architecture.

Typically, the textual notation is used for detailing the diagrammatic representation of the overall picture of an architecture.

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3_17

The questions that we will address are the following:

- what is the SysADL grammar for specifying the textual description of a software architecture?
- how to apply the SysADL grammar for describing different views of an architecture?

Each of these questions will be tackled in this chapter.

17.2 Textual Notation

Let us now present the SysADL textual notation. We will illustrate each of the constructs with our running example, i.e., the *RTC* system.

17.2.1 Properties and Data

Data. SysADL provides the following primitive data types: **Boolean**, **Integer**, **Real**, and **String**. These scalar data types are completed with user-defined enumerated types, defined with the **enum** construct. Data types, defined with the **datatype** construct, are compound types. A value type represents a quantity in a **dimension** according to a **unit** of measure. A value type may be defined by extending another defined value type.

In terms of grammar, as shown in Fig. 17.1, *DataDef* rule defines the different kinds of data supported by SysADL.

Dimension and unit definitions are expressed with the production rules shown in Fig. 17.2.

Figure 17.3 shows the definition of a dimension, *TemperatureDimension*, and the associated units: *Fahrenheit* and *Celsius*.

Fig. 17.1 *DataDef* in the SysADL grammar executable viewpoint

```
DataDef ::=
        DimensionDef |
        UnitDef |
        ValueTypeDef |
        DataTypeDef |
        Enumeration
```

Fig. 17.2 *Dimension* and *Unit* definitions in the SysADL grammar executable viewpoint

```
DimensionDef ::=
        'dimension' ID
        (‘{‘ (Property)*'}')?

UnitDef ::=
        'unit' ID ‘for’ ID
        (‘{‘ (Property)*'}')?
```

```
dimension TemperatureDimension
unit Fahrenheit for TemperatureDimension
unit Celsius for TemperatureDimension
```

Fig. 17.3 *Dimension* and *Units* definitions in the RTC system

```
ValueTypeDef ::=
        'valuetype' ID (('extends' ID) | (‘=’ ID))?
                (‘<’ 'unit' '=' ID ‘>’)?
                (‘<’ 'dimension' '=' ID ‘>’)?

        )?
```

Fig. 17.4 Value type definition in the SysADL grammar executable viewpoint

For value type, in terms of grammar, as defined hereafter, the definitions are expressed with the *ValueTypeDef* production rule. The **valuetype** keyword introduces the value type definition specifying the type name (**ID**) and the name of the type it inherits of, through the **extends** keyword. It enables to specify the unit, the dimension for the value type. Figure 17.4 shows the syntax to define a value type in the SysADL executable viewpoint.

Figure 17.5 illustrates value type definitions in the *RTC* system. *Temperature* is defined as a value type of **Real** type and *TemperatureDimension* dimension. *Ftemperature* is a value type that extends the *Temperature* value type defining the *Fahrenheit* unit. Similarly, *Ctemperature* is a value type that extends the *Temperature* value type defining the *Celsius* unit.

Data types are expressed with the *DataTypeDef* production rule. The **datatype** keyword introduces the name (**ID**). Figure 17.6 shows the syntax to define a data type in SysADL executable viewpoint.

The example in Fig. 17.7 shows a data type definition for *Commands*, which is a structure of two properties of the *Command* enumeration (see later).

Enumerations are expressed through the *Enumeration* production rule. The **enum** keyword introduces a name for the enumeration, the values are defined using the ***EnumLiteralValue*** rule, which assigns a name for an integer. Additionally, enumerations can be annotated with properties. Figure 17.8 shows the syntax to define an enumeration in the SysADL executable viewpoint.

```
valuetype Temperature = Real <dimension = TemperatureDimension>;
valuetype FTemperature extends Temperature <unit = Fahrenheit>;
valuetype CTemperature extends Temperature <unit = Celsius>;
```

Fig. 17.5 Value type definitions in the RTC system

```
DataTypeDef ::=
        'datatype' ID '{'
                (TypeUse)*
        '}'
```

Fig. 17.6 Data type definition in the SysADL grammar executable

```
datatype Commands {
  heater: Command;
  cooler: Command;
}
```

Fig. 17.7 Data type definitions in the RTC system

```
Enumeration ::=
        'enum' ID '{'
                (Property)*
                (EnumLiteralValue ( "," EnumLiteralValue)*)?
        '}'
EnumLiteralValue ::=
        ID
```

Fig. 17.8 Enumeration definition in the SysADL grammar executable viewpoint

```
enum Command {on, off}
enum TemperatureUnit {Celsius, Fahrenheit}
```

Fig. 17.9 Enumeration definition in the RTC system

```
dimension TemperatureDimension
unit Fahrenheit for TemperatureDimension
unit Celsius for TemperatureDimension
valuetype Temperature = Real <dimension = TemperatureDimension>
valuetype FTemperature extends Temperature <unit = Fahrenheit>
valuetype CTemperature extends Temperature <unit = Celsius>
datatype Commands {
  heater: Command
  cooler: Command
}
enum Command {on, off}
enum TemperatureUnit {C, F}
```

Fig. 17.10 Properties and data definitions in the RTC system

The example in Fig. 17.9 shows an **enum** definition for a *Command*, which value can be *on* or *off*. It also shows an **enum** definition for a *TemperatureUnit*, which value can be *Celsius* or *Fahrenheit*.

We show now, in Fig. 17.10, the complete declaration of the definitions, previously explained, used in the RTC System.

17.2.2 Components and Ports

Components. Components are defined with the **component def** construct. It defines the ports used to interact with components of the type, and for each port the direction of the flow through the port (**in** or **out**).

In terms of grammar, the **component def** keyword introduces the component definition (that can be **boundary** or not) specifying the component name (**ID**), and its **ports** (**PortUse)**. The grammar of component definitions is expressed with the *ComponentDef* production rule as shown in Fig. 17.11.

We use this same production rule to define simple and composite components, (composite components are discussed in the next section). Ports, components, connectors (for composite components), inside a component definition using the ***StructuralDef*** rule. We can also define data types using the *DataDef* rule. The internal configuration of composite components is defined using the ***Configuration*** rule.

Hereafter, in Fig. 17.12 we give an excerpt of the component types of our running example.

Fig. 17.11 Component definition in the SysADL grammar executable viewpoint

```
ComponentDef ::=
        'boundary'? 'component' 'def' ID '{'
                (StructuralDef
                | DataDef)*
                ('ports'
                 PortUse*)
                (Configuration)?
        '}';
```

Fig. 17.12 Component definitions example in the RTC system

```
component def SensorsMonitorCP {
  ports
    in s1: CTemperatureIPT
    in s2: CTemperatureIPT
    out average: CTemperatureOPT
}
component def CommanderCP {
  ports
    target: CTemperatureIPT
    average: CTemperatureIPT
    heating: CommandOPT
    cooling: CommandOPT
  }
component def PresenceCheckerCP {
  ports
    detected: PresenceIPT
    userTemp: CTemperatureIPT
    target: CTemperatureOPT
  }
```

Fig. 17.13 Boundary component definitions example in the RTC system

```
boundary component def TemperatureSensorCP {
  ports
    current: FTemperatureOPT
}
boundary component def HeaterCP {
  ports
    controller: CommandIPT
}
```

Fig. 17.14 Simple port definition in the SysADL grammar executable viewpoint

```
SimplePortDef ::=
        'port' 'def' ID '{'
               'flow' FlowProperty TypeUse
               (DataDef)*
        '}'
enum FlowProperty ::=
        in = 'in' | out = 'out'
```

Boundary components are those that do not have observable behavior, representing an interaction with the environment. The **boundary** keyword is used to express those components. Figure 17.13 shows the declaration of a boundary component type *TemperatureSensorCP*: it has a port of type *FTemperatureOPT* that provides an outgoing flow of temperature values in °F. The other boundary component types *PresenceSensorCP*, and *UserInterfaceCP* have a similar semantics.

Ports. Simple ports are defined with the **port def** construct. It defines the port name, the direction and the datatype of the flow, and the protocol that characterizes the behavioral specification of the port.

In terms of grammar, simple port definitions are expressed with the *SimplePortDef* production rule as shown in Fig. 17.14. The **port def** keywords introduce the port definition specifying the port name (**ID**) and the **flow** clause. **flow** establishes the direction of the flow (***FlowProperty***) and the value type of the data flow (***TypeUse***). The ***TypeUse*** can be defined as a shared type of the architecture or it can be defined inside the port, using the *DataDef* rule. The **FlowProperty** can be **in** or **out**. Note that when the data type used to type the flow is defined inside the port, it is locally defined.

Figure 17.15 shows the definition of the ports for the RTC system.

17.2.3 Connectors

Let us now handle the case of connectors.

Connectors. Simple connectors are defined with the **connector def** construct. It specifies the ports of the components that will participate in the connector using the **participants** clause. It also specifies the direction of the flow that traverses the

```
port def CTemperatureOPT {
 flow out temp:CelsiusTemperature
}
port def CTemperatureIPT {
 flow in temp:CelsiusTemperature
}
port def CommandIPT {
 flow in cmd:Command
}
port def CommandOPT {
 flow out cmd:Command
}
port def PresenceIPT {
 flow in presence:Boolean
}
port def PresenceOPT {
 flow out presence:Boolean
}
```

Fig. 17.15 Example of port definitions in the RTC system

connector (from **in** to **out**). Note that the flow itself is implicitly typed by the flow definition of participants.

In terms of grammar, as defined hereafter, illustrated in Fig. 17.16, the simple connector definitions are expressed with the *ConnectorDef* production rule. The keywords **connector def** introduces the connector definition specifying the connector name (**ID**), the **participants** ports (**PortUse_Reverse**), and a flow (**Flow**). The **PortUse_Reverse** represents the conjugated port. If the connector is composite, we need to define its configuration. We see composite connectors in the next section.

For instance, we show in Fig. 17.17 the declaration of a connector type *CTemperatureCN*: it has an incoming flow coming from participant *source* of port type conjugated of *CTemperatureOPT* that traverses the connector toward

```
ConnectorDef ::=
       'connector' 'def' ID '{'
              DataDef*
              'participants' PortUse_Reverse*
              (‘flow’ Flow*)?
              Configuration?
       '}'
Flow ::=
       ID ‘from’ ID ‘to’ ID
```

Fig. 17.16 Simple connector definition in the SysADL grammar executable viewpoint

Fig. 17.17 Connectors definitions in the RTC system

```
connector def CTemperatureCN {
  participants
    source: ~CTemperatureOPT
    destination: ~CTemperatureIPT
  flow Temperature from source to destination
}
connector def DetectPresenceCN {
  participants
    source: ~PresenceOPT
    destination: ~PresenceIPT
  flow Presence from source to destination
}
connector def ControlCommandCN {
  participants
    source: ~CommandOPT
    destination: ~CommandIPT
  flow Command from source to destination
}
```

participant *destination* of port type conjugated of *CTemperatureIPT*. The inferred value type of the flow is *CelsiusTemperature*.

In these three examples of connector types' definitions, a connector has the default semantics of transferring values from **in** to **out**.

17.2.4 Compositions and Architecture

Composite components. Composite components are components too and thereby have ports. As the ports of composite components are always *proxy ports*, they need to be delegated to the ports of internal components using the **delegation** clause (see the definition of delegation in the configuration grammar).

The grammar for defining composite components is the same for component definition (shown in Fig. 17.11). The difference between the simple component and a composite component is the existence of a configuration in the case of composite components.

For composite components, Fig. 17.18 shows the definition of the *RoomTemperatureControllerCP* composite component. The component type is defined as a composition of components from different component types. The configuration is described by instantiating components *pc* of *PresenceCheckerCP* type, *sm* of *SensorMonitorCP* type, and *cm* of *CommanderCP* type, and then by linking these components using connectors *target* and *average* of *CTemperatureCN* type: *pc* is connected to *cm* using their ports named *target*, and *sm* to *cm* also using their ports named *target*.

Composite connectors. As components, connectors can also be composite. Composite connectors can be a composition of others defined connectors.

```
component def RoomTemperatureControllerCP {
   ports
        localTemp1: CTemperatureIPT
        localTemp2: CTemperatureIPT
        detected: PresenceIPT
        userTemp: CTemperatureIPT
        heating: CommandOPT
        cooling: CommandOPT

   components
        pc:PresenceCheckerCP
        sm:SensorMonitorCP
        cm:CommanderCP

   connectors
        target:CTemperatureCN bind source to pc.target, destination
to cm.target
        average:CTemperatureCN bind source to sm.target, destination
to cm.target

   delegations
        localTemp1 to sm.s1
        localTemp2 to sm.s2
        detected to pc.detected
        target to pc.userTemp
        heating to cm.heating
        cooling to cm.cooling
}
```

Fig. 17.18 Composite components definition example in the RTC system

Figure 17.19 illustrates a composite connector named *ClientServerCN* that is composed of two connectors: *query*, of type *ClientServerQueryCN*, and *answer*, of type *ClientServerAnswerCN*.

Composite connectors can also have a configuration of others defined components and connectors.

Figure 17.20 illustrates a composite connector named *AllTemperaturesCN* that is composed of the *ta* component of type *TemperatureAggregatorCP*. Note that the flows on ports go to and come from this component.

Figure 17.21 illustrates a composite connector named *StatusTempCCN* that is composed of the *ft* component of type *FaultTolerantCP*. Note that the flows on the *st* ports are managed by subports (*ft.st.t* and *ft.st.s*).

Architecture. Architecture types are defined with the **architecture def** construct. An architecture is defined as a special kind of composite component: it is the composite component at the root of the hierarchy of composite components representing a software architecture. As composite components, an architecture is defined in terms of instantiated components and connectors forming a configuration. Note that the architecture represents its interaction with the environment of the system through boundary components.

In terms of grammar, as defined hereafter, the architecture definitions are expressed with the *ArchitectureDef* production rule, as shown in Fig. 17.22.

```
port def ClientCPT {
  ports
    query:QueryOPT
    answer:AnswerIPT
}

port def ServerCPT {
  ports
    query:QueryIPT
    answer:AnswerOPT
}


connector def ClientServerCN {
  participants
    client: ~QueryCPT
    server: ~ServerCPT
  connectors
    query: ClientServerQueryCN bind client.query to server.query
    answer: ClientServerAnswerCN bind server.answer to client.answer
}

connector def ClientServerQueryCN {
  participants
    client: ~QueryOPT
    server: ~QueryPortIPT
  flow Any from source to destination
}

connector def ClientServerAnswerCN {
  participants
    client: ~AnswerOPT
    server: ~AnswerPortIPT
  flow Any from source to destination
}
```

Fig. 17.19 Composite connector definition in the RTC system—Example A

The **architecture def** keyword introduces the architecture definition specifying the architecture name (**ID**), its ports (**PortUse**), and the internal **configuration**.

We can define structural elements, such as ports, components, connectors, inside an architecture definition using the *StructuralDef* rule. We can also define data types using the *DataDef* rule.

An architecture has a configuration defined using the ***Configuration*** rule. We need to specify the **components**, **connectors**, and **delegations**. Connectors instances are used to bind (**ConnectorBinding**) two or several ports of connectors.

For instance, we show in Fig. 17.23 the declaration of the software architecture of the *RTC* system. It has seven components of which two of *TemperatureSensor CP* type and one of each of the following types: *PresenceSensorCP*, *HeaterCP*, *CoolerCP*, *UserInterfaceCP*, and *RoomTemperatureControllerCP*. It has six connectors: *c1*, *c2*, *uc*, *pc*, *cc1*, and *cc2*.

Fig. 17.20 Composite connector definition in the RTC system—Example B

```
datatype TemperatureDT {
  temp:Temperature[1..*]
}

port def TemperatureIPT {
  flow out temp:Temperature
}

port def TemperatureDTOPT {
  flow out tempdt:TemperatureDT
}

component def TemperatureAggregatorCP {
  ports
    p1:TemperatureIPT
    p2:TemperatureDTOPT
}

connector def AllTemperaturesCN {
  participants
    source: ~TemperatureOPT[1..*]
    destination: ~TemperatureDTIPT
  components
    ta:TemperatureAggregatorCP
  flow
    Temperature from source to ta.p1
    Temperature from ta.p1 to destination
}
```

17.2.5 Activities

Activities. Activities specify the behavior of components. They possibly execute repeatedly waiting for incoming values, executes calling internal actions when these values are received and then sends the results in outgoing flows.

In terms of grammar, as defined in Fig. 17.24, the activity definitions are expressed with the *ActivityDef* production rule. The **activity def** keywords introduce the activity definition specifying the activity name (**ID**), the input and output pins (expressed by **ActivityParam**).

We can define behavioral elements, such as actions, inside an activity definition using the *BehavioralDef* rule. We can also define data types using the *DataDef* rule.

The invariants of an activity are expressed using the ***ConstraintInvariant*** rule.

The behavior of an activity is expressed in terms of ***Statements*** (such as conditional statements, assignment, sequence, loop statements) using the ***Body*** rule.

Let us now address the case of user-defined behavior of component types through the example in Fig. 17.25. In this example, the activity is defined in terms of *t1* and *t2 in* flows and *average out* flow. This activity itself calls an action, named *CalculateAverageTemperatureAN*. Note that all the input flows in the activity are delegated to the action parameters and its result is delegated to the output flow.

```
port def FTemperatureOPT {
  flow out temp:FahrenheitTemperature
}
port def CTemperatureIPT {
  flow in temp:CelsiusTemperature
}
port def StatusOPT {
  flow out s:Status
}
port def StatusTempCPT {
  ports
    s:StatusOPT
    t:CTemperaturaIPT
}
component def FaultTolerantCP {
  ports
    sensor:TemperatureIPT
    st:~StatusTempCPT
}
connector def StatusTempCCN {
  participants
    source: ~FTemperatureOPT
    destination: ~StatusTempCPT[2]
  components
    ft:FaultTolerantCP
  flows
    Temperature from source to ft.sensor
    Temperature from ft.st.t to destination.t
    Status from destination.s to ft.st.s
}
```

Fig. 17.21 Composite connector definition in the RTC system—Example C

```
ArchitectureDef ::=
       'architecture' 'def'  ID '{'
              (StructuralDef
              | DataDef
              | PortUse)*
              Configuration
       '}';

Configuration ::=
       'components' ComponentUse*
       & 'connectors' ConnectorUse*
       & 'delegation' Delegation*

ConnectorUse::=
       ID ‘:’ ID 'bind' ConnectorBinding ("," ConnectorBinding)*

ConnectorBinding ::= ID ‘to’ ID
```

Fig. 17.22 Architecture definition in the SysADL grammar executable viewpoint

```
architecture def ARCH1 {

components
   s1:TemperatureSensorCP
        {sensorTemperatureUnit = TemperatureUnit::Fahrenheit}
   s2:TemperatureSensorCP
        {sensorTemperatureUnit = TemperatureUnit::Fahrenheit}
   s3:PresenceSensorCP
   ui:UserInterfaceCP
   a1:HeaterCP
   a2:CollerCP
   rtc:RoomTemperatureController

connectors
   c1:FahrenheitToCelsiusCN bind source to s1.current.heating, des-
tination to rtc.localTemp1
   c2:FahrenheitToCelsiusCN bind source to s2.current.heating, des-
tination to rtc.localTemp2
   uc:CTemperatureCN bind source to ui.desired, destination to
rtc.userTemp
   pc:DetectPresenceCN bind source to s3.detected, destination to
rtc.detected
   cc1:ControlCommandCN bind source to rtc.heating, destination to
a1.controller
   cc2:ControlCommandCN bind source to rtc.cooling, destination to
a2.controller
```

Fig. 17.23 Architecture definition in the RTC system

Let us now examine the case of user-defined activities specifying the behavior of connector types through the example in Fig. 17.26. In this example, the activity has two flows, *f* and *c*, and calls an action, named *FahrenheitToCelsiusAN*. Note that the input flow is delegated to the action input parameter, and the output flow receives the result of the action.

17.2.6 Executable Advanced Examples

Figure 17.27 illustrates another activity definition. In this example, the activity is defined in terms of *averageTemp* and *targetTemp* in flows and heater and cooler out flows. This activity itself calls three actions: *CompareTemperatureAN*, *CommandHeaterAN*, and *CommandCoolerAN*. Note that all the input flows in the activity are delegated to actions parameters and their results are delegated to the output flows. In this example, two flows connect the result of the *ct* action invocation with the input of the *cmdHeater* and *cmdCooler* actions, respectively.

```
ActivityDef ::=
       'activity' 'def' ID '(' ((ActivityParam) (','
(ActivityParam))*)? ')' '{'
       ( BehavioralDef
       | DataDef
       | ConstraintUse)*
       (ActivityBody)?
       '}'

ActivityParam ::=
       ActivityParamIn | ActivityParamOut

ActivityParamIn ::=
       ‘flow’? 'in' ID ':' ID

ActivityParamIn ::=
       ‘flow’? 'out' ID ':' ID

ConstraintUse :: =
       ConstraintKind ID ‘(‘ID (‘,’ ID )* ‘)’

ConstraintKind ::=
       ‘pre’ | ‘post’ | ‘invariant’

ActivityBody ::=
    'actions' ActionUse*
    ('data' DataObject*)?
    ('delegations' ActivityDelegation*)?
    ('flows' ActivityFlow*)?

ActionUse ::=
    ID ':' ID '(' ID (',' ID))? ')'
    ('{' Property*
    '}')?;

DataObject ::=
    DataStore | DataBuffer;

DataStore ::=
    'datastore' ID ':' ID ('=' Expression)?;

DataBuffer ::=
    'buffer' ID ':' ID ('=' Expression)?;

ActivityDelegation ::=
    ID 'to' ID;

ActivityFlow ::=
    ID 'from' ID 'to' ID;
```

Fig. 17.24 Activity definition in SysADL grammar executable viewpoint

```
activity def CalculateAverageTemperatureAC(
        flow in t1:CelsiusTemperature,
        flow in t2:CelsiusTemperature,
        flow out average:CelsiusTemperature)  {

  actions
        av:CalculateAverageTemperatureAN(x, y)

  delegations
        t1 to  av.x
        t2 to av.y
        average to av.out
}
```

Fig. 17.25 Activity definition in the RTC system example

```
activity def FahrenheitToCelsiusAC(
             flow in f : FahrenheitTemperature,
             flow out c : CelsiusTemperature) {

    actions
       ftc:FahrenheitToCelsiusAN(f);

    delegations
       f to ftc.f
       ftc.c to c
}
```

Fig. 17.26 Activity definition for the connector in the RTC system example

```
activity def DecideCommandAC(
        flow in averageTemp:Temperature,
        flow in targetTemp:Temperature,
        flow out heater:Command,
        flow out cooler:Command)  {

  actions
        ct:CompareTemperatureAN(t1, t2)
        cmdHeater:CommandHeaterAN(cmds)
        cmdCooler:CommandCoolerAN(cmds)

  delegations
        averageTemp to  ct.t1
        targetTemp to  ct.t2
        cmdHeater.heater to  heater
        cmdHeater.cooler to  cooler

  flows
        Commands from ct.cmds to cmdHeater.cmds
        Commands from ct.cmds to cmdCooler.cmds
}
```

Fig. 17.27 Another activity definition in the RTC system example

```
Protocol ::=
       'protocol' ID '{'
              ProtocolBody
       '}'
ProtocolBody ::=
      ('several' | 'once' | 'perhaps' | 'always')
      (ProtocolBody | PredefinedActions | '('ProtocolBody')') (
(';' | '|') ProtocolBody)?

PredefinedActions ::=
       ActionSend | ActionReceive

ActionSend ::=
       ID 'send' Expression

ActionReceive ::=
       ID 'receive' TypeUse
```

Fig. 17.28 Protocol definition in SysADL grammar executable viewpoint

17.2.7 Protocols

Protocols. The behavioral specification of a port is expressed by the ***Protocol*** production rule defined in Fig. 17.28. The ***Protocol*** rule defines the name of the protocol (ID) and the ***ProtocolBody.*** The ***ProtocolBody*** rule defines the protocol in terms of regular expressions on send/receive actions with operators (***several***, ***once***, ***perhaps***). **Once** means that the action will be executed once (it means that in case of the send action, a data item will be sent via the port. In case of the receive action, a data item will be received via the port). **Several** means that the action will be executed several times (none, one, or many times). **Perhaps** means that the action will be executed once or not. **Always** means that the action will be executed forever. For instance, in the case of energy management of the Temperature Sensor in the *RTC* System, the *CTemperatureOPT* port could notify when the energy level is low (represented by a threshold value). In this case, the protocol body expression could be **several** (**once via p send temp; perhaps via p send threshold**).

Figure 17.29 shows examples of protocols definitions.

17.2.8 Actions

Actions. In SysADL, an action is the basic unit of executable functionality. The execution semantics of actions is the one of start-stop execution: it executes from beginning to end, atomically, only once when called. An action forms thereby an abstraction of a computation which is an atomic execution and therefore completes without interruption.

```
protocol FTemperatureOPC (out temp:FTemperatureOPT {
 several send temp
}
protocol CTemperatureOPC(out temp:CelsiusTemperature) {
  several send temp;
}
protocol CTemperatureIPC (in temp:CTemperatureIPT) {
  several receive temp;
}
protocol CommandIPC (in cmd:CommandIPT) {
  several receive cmd;
}
protocol CommandOPC (out cmd: CommandOPT) {
  several send cmd;
}
protocol PresenceIPC (in presence:PresenceIPT) {
  several receive presence;
}
protocol PresenceOPC (out presence:PresenceOPT {
  several send presence;
}
```

Fig. 17.29 Example of protocol definitions in the RTC system

The internal control flow of an action is also described using control flow constructs without any interaction during its execution.

An action definition is the specification of basic executable statements. After it is defined it then can be called in activities or other actions. Once called, an action execution represents the run-time behavior of executing an action definition with actual parameters in an encompassing context. An action describes a run-time behavior that can be considered as almost instantaneous and atomic.

In terms of grammar for actions, as defined in Fig. 17.30, the actions definitions are expressed with the ***UserDefinedAction*** production rule. The keywords **action def** introduces the action definition specifying the action name (**ID**), the input and output pins (expressed by ***TypeUse***).

We can also define data types using the *DataDef* rule.

The invariants, pre-, and post-conditions of an action are expressed using the ***ConstraintUse*** rule.

We exemplify now the definition of the actions we use in previous example (see Figs. 17.25 and 17.26). Figure 17.31 shows the action definition *CalculateAverage TemperatureAN* with three parameters *t1*, *t2*, and *result*. An action is defined in terms of pre-, and post-condition. In this example we do not have pre-conditions. The post-condition is specified using the *CalculateAverageTemperatureEQ* equation, which we explain later in this chapter.

```
ActionDef ::=
'action' 'def'   ID   '(' (ActionPin (',' ActionPin)*)? ')' (':'
ID)? '{'
       ( DataDef
       | ConstraintUse)*
       '}'

ConstraintUse ::=
       ConstraintKind ID '(' ID (',' ID)* ')' /* the first ID re-
fers to a constraint ID */

ActionPin ::=
        'in'|'inout' ID ':' ID


ConstraintKind ::=
       'pre' | 'post' | 'invariant'
```

Fig. 17.30 Action definition in SysADL grammar executable viewpoint

```
action def CalculateAverageTemperatureAN (t1: CelsiusTemperature,
   t2: CelsiusTemperature, result: CelsiusTemperature) {
   pre //optional
   post CalculateAverageTemperatureEQ(t1, t2, result)
}
```

Fig. 17.31 Action definition to the *CalculateAverageTemperatureAC* activity in the RTC system example

```
action def FahrenheitToCelsiusAN (f: FahrenheitTemperature, c:
CelsiusTemperature) {
    post FahrenheitToCelsiusEQ(f, c)
}
```

Fig. 17.32 Action definition to the *FahrenheitToCelsiusAC* activity in the *RTC* system example

The action defined in Fig. 17.32 is similar to the previous one. *Fahrenheit ToCelsiusAN* action has a post-condition expressed by the *FahrenheitToCelsiusEQ* equation. The action has two parameters and is used in the activity FahrenheitToCelsiusAC explained in Fig. 17.26.

17.2.9 Constraints

Constraints. Constraints definitions are expressed with the *ConstraintDef* production rule, as illustrated in Fig. 17.33. The keywords **constraint def** introduces

```
ConstraintDef ::=
       'constraint' 'def' ID ('(' TypeUse (',' TypeUse)*')' '{'
              (BooleanExpression)

       '}'
```

Fig. 17.33 Constraints definition in SysADL grammar executable viewpoint

```
constraint def CalculateAverageTemperatureEQ(t1:CelsiusTemperature,
       t2: CelsiusTemperature, average: CelsiusTemperature) {
       average==(t1+t2)/2;
}

constraint def FahrenheitToCelsiusEQ (f: FahrenheitTemperature,
     c: CelsiusTemperature) {
     c == (5 * (f - 32) / 9);
}
```

Fig. 17.34 Constraint definitions in the *RTC* system example

the constraint definition specifying the constraint name (**ID**) and the constraint parameters (**TypeUse**). **BooleanExpression** is used to define the constraint in terms of a logical expression. In addition, properties could be defined to characterize a constraint.

We now show the definitions of the constraints previously used in the action definitions in Figs. 17.31 and 17.32. In the constraint definition, we need to provide an equation to express the constraint uses in an action (Fig. 17.34).

17.2.10 Executables

Executable definitions. We define the executable body of an action using the ***ExecutableDef*** construct (Fig. 17.35).

We now show, in Fig. 17.36, the definitions of the executables to the previously action definitions presented in Figs. 17.31 and 17.32. In the executable definition, we need to provide the statements defining the execution of an action.

```
ExecutableDef ::=
   'executable' 'def' ID '(' (ActionPin (',' ActionPin)*)? ')' (':'
ID)? '{'
       Statements*
   '}'
```

Fig. 17.35 Executable definition in SysADL grammar executable viewpoint

```
executable def CalculateAverageTemperatureEX(in t1:
CelsiusTemperature, in t2: CelsiusTemperature):CelsiusTemperature {

        return ((t1 + t2)/2);

   }
executable def FahrenheitToCelsiusEX(f:FahrenheitTemperature):
CelsiusTemperature {

        return 5*(f - 32)/9;
}
```

Fig. 17.36 Executable definitions in the RTC system example

17.2.11 Executable Advanced Examples

In this section we present some executable advanced examples. Figure 17.37 illustrates the executable specification of the *AggregateTemperaturesAN* action. It returns the temperature of all sensors.

Figure 17.38 illustrates the executable specification of the *CalculateAverage Temperature*AN action. It receives the temperatures of all sensors and returns the average temperature.

```
executable def AggregateTemperaturesEX( in temps:Temperature[1..*]):
TemperatureDT {

        let allTemp:TemperatureDT = ;
        let index:Integer=1;

        for (t in temps) {
                allTemp[index] = t;
                index = index + 1;
        }
        return allTemp;
}
```

Fig. 17.37 Executable specification of the *AggregateTemperaturesAN* action

```
executable def CalculateAverageTemperatureEX(in alltemp:
TemperatureDT): Temperature {

        let sum:CelsiusTemperature=0;
        let numOfElem:Integer=0;

        for (t in alltemp) {
                sum = sum + t;
                numOfElem = numOfElem + 1;
        }
        return sum/numOfElem;
```

Fig. 17.38 Executable specification of the *CalculateAverageTemperatureAN* action

```
executable def CalculateAverageTemperatureEX(
        in alltemp: TemperatureDT): Temperature  {

        return alltemp->sum()/alltemp->size();

    }
```

Fig. 17.39 Executable specification of the *CalculateAverageTemperatureAN* action—alternative version

```
executable def CheckPresenceToSetTemperatureEX(
          presence: Boolean,
          desiredTemp: CelsiusTemperature)
          result:CelsiusTemperature)  {

               if (presence) {
                         return desiredTemp;
               }
               else {
                         return 22;
               }
    }
```

Fig. 17.40 Executable specification of the *CheckPresenceToSetTemperatureAN* action

Figure 17.39 illustrates another possibility for the executable specification of the *CalculateAverageTemperature*AN action. It receives the temperatures of all sensors and returns the average temperature.

Figure 17.40 illustrates the executable specification of the *CheckPresence ToSetTemperature*AN action. It checks if there is someone in the room and, in the positive case, it returns the desired temperature, otherwise, it returns 22 °C.

Figure 17.41 illustrates the executable specification of the *DecideCommand*AN action. It compares the average temperature with the target temperature and decides how to set the heater and the cooler.

```
executable def DecideCommandEX (in averageTemp:CelsiusTemperature, in
targetTemp:CelsiusTemperature):Commands {

          let heater:Command = Command::off;
          let cooler:Command = Command::off;

          if (averageTemp > targetTemp) {
                  heater = Command::off
                  cooler = Command::on
          } else if averageTemp < targetTemp {
                  heater = Command::on
                  cooler = Command::off
          }
          return new Commands(heater=>heater, cooler=>cooler)
  }
```

Fig. 17.41 Executable specification of the *DecideCommandAN* action

17.3 Summing up

As we saw, SysADL provides a textual notation and an action language to textually describe execution views from the executable viewpoint. Glossary summarizes the keywords used by this language presented by this chapter.

17.4 Summary

In this chapter, you have learnt the following:

- organize architectural definitions in the form of a textual description;
- the SysADL notation to express the architecture provided by the executable viewpoint.

Further Reading

1. Concrete Syntax for a UML Action Language: Action Language for Foundational UML™ (ALF™). http://www.omg.org/spec/ALF/
2. Semantics of a Foundational Subset for Executable UML Models (FUML™). http://www.omg.org/spec/FUML/
3. Precise Semantics of UML Composite Structures™ (PSCS™). http://www.omg.org/spec/PSCS/1.0/

Glossary

Summary of Keywords Presented in this Chapter 17

action def Action definition

activity def Activity definition

architecture def Architecture definition

bind Define a binding between connectors' and components' ports

boundary Boundary component

component def Component definition

constraint def Constraint definition

connector def Connector definition

datatype Defines a data type

enum Enumeration

for Control loop based on enumerated values

if Conditional statement

in, out Data direction: input/output

invariant Define the conditions that must be true during the execution of the action or activity

once Specify that an action or a sequence of actions will be executed once in the protocol behavior

participants Participant ports of a connector

perhaps Specify that an action or a sequence of actions will be executed once or not in the protocol behavior

port def Port definition

F. Oquendo et al., *Software Architecture in Action*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-44339-3

post Define the conditions that must be true after the execution of the action

pre Define the conditions that must be true before the execution of the action

property Defines an annotation for any named element

protocol Protocol definition

select Multiple-choice for port synchronization

several Specify that an action or a sequence of actions will be executed several times (noneone, or many times) in the protocol behavior

switch Multiple-choice conditional statement

valuetype Defines a quantity with a dimension and a unit

while Control loop based on a boolean expression