Facilitating the Representation of Composite Structure, Active objects, Code Generation, and Software Component Descriptions in the Umple Model-Oriented Programming Language

Mahmoud Husseini Orabi

A Thesis submitted

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

University of Ottawa

Ottawa, Ontario, Canada

``

Abstract

For a long time, the development of component-based systems has been a crucial part of real-time software development required for embedded and automotive domains. However, most of the existing tools used in these fields are not only proprietary, but also expensive and not research-friendly. Open-source tools in this domain are so far quite limited in terms of the features supported, especially, code generation.

In this thesis, we demonstrate how we can improve the development of real-time and concurrent systems by the introduction of component-based modelling into Umple, an open-source modelling tool. Our work enables component-based modelling to be performed both textually and visually, as is the case with other Umple features.

We introduce a number of major features into Umple. First, we introduce support for real-time C++ code generation. This includes supporting all Umple features, such as class diagrams, associations, state machines, and attributes. In order to achieve this, we also introduce Umple Template Language (Umple-TL), which helps Umple developers to use Umple itself to emit text using easy-to-use constructs, such that the text emitted can be in different target languages such Java and C++. Umple-TL provides additional capabilities relying on Umple being a model-oriented and object-oriented language. Umple-TL has become the technology for all code generation in Umple, not just our real-time C++ generators. Umple-TL also plays a vital role easing writing component descriptions

Second, we support concurrency, which is crucial for the underlying architecture of composite structure. We have to avoid relying on any third-party libraries in order to make sure that the code generated will be deployable on embedded devices, which are limited and do not provide a lot of options. The concurrency pattern we follow extends the active object pattern aiming to enhance communication among active objects. Concurrency development in general, even if a programming language used is not real-time, is not easy. Hence, we simplify active object concepts, such as future, promise, and delay, using new Umple keywords.

We also add composite structure support to Umple, we believe that our syntax and language constructs are comprehensive, and do not require a wide knowledge of modelling and UML concepts. Additionally, we introduce a novel protocol-free approach that dynamically extracts communication protocols from ports, bindings, and active objects as a way to simplify development, and to lead to concise and optimized code generation.

We demonstrate the effectiveness of our work using cases studies, in which we implement Umple models using our new composite structure and concurrency constructs. We show that the amount of code required to specify complex concepts is reduced, and the generated systems are effective.

``

Acknowledgments

First and foremost, I wish to thank my supervisor Dr. Timothy C. Lethbridge. I have been honoured to be one of his students. He has always tried his best to teach and supervise me during my PhD. I appreciate all his contributions of time, suggestions, and funding during my study, which always kept me stay productive, focused, and motivated. Throughout my PhD years, he has always provided guidance and insights that helped me improve in the fields of my research. If there is anything I am thankful for in addition to getting my PhD, is knowing Tim as a friend in my life.

Very deep and special thanks go to my parents, who have always been supporting and available when I needed you. Being able to talk to you on a daily-basis has always been reassuring for me, enabling me to work better, and stay focused and motivated.

A heart-to-heart thank-you to my brother, Ahmed, with whom I have shared most of my academic and work life. Your help will be forever recorded in my work.

I would like to thank the University of Ottawa, The Ontario Graduate Scholarship (OGS), ORF, and NSERC for their collaboration and funding during my research.

Special thanks go to all of my close friends who have always been always encouraging, supporting, and impatient for me to finish my studies.

Finally, I give sincere thanks to all software researchers, around the globe, who participated or published in the related topics of my research. Your insights and findings have always provided the necessary substance for my work.

``

Table of Contents

``

``

``

``

List of Figures

``

``

List of Tables

``

List of Snippets

``

``

``

``

``

Chapter 1 Introduction

The complexity of real-time systems has been growing significantly over the past years [1]–[3]. Such systems are typically composed of a growing number of integrated components. Development of such systems requires numerous decisions that can be categorized as specification, design, and integration decisions. Integration decisions can be urgent or unavoidable in many cases, and include dealing with such issues as irreconcilable incompatibility between component libraries. Forced integration decisions are usually taken late in the development cycle, which makes them particularly difficult. Manually coded software is hard to adapt to accommodate such decisions. This is especially true in certain domains, such as embedded devices [4], in which systems have different levels such as software and hardware [5], [6].

A model-driven approach can be used to reduce development complexity [5], [6] and in particular to reduce the impact of integration issues. In such an approach, more time is spent at the design phase so errors can be detected earlier, and so code can be regenerated as needed. Furthermore, models can generate optimized code that can be compiled and run on different platforms and devices [4], [7].

A model is usually represented visually, as opposed to textually as in traditional programming languages. In this thesis, we investigate how model-driven engineering can be used to enhance real-time development by representing modelling abstractions in a textual form that can also be rendered visually using Umple (www.umple.org). The rationale for this is that programmers tend to be very comfortable with textual forms, and find them convenient for editing, commenting and change tracking; yet visual forms allow for easier manual understanding and verification.

Embedded devices and automotive system development requires introducing a new extension to Umple to support real-time development. UML concepts such as composite structure, active objects, and software component descriptions must be integrated into Umple; this integration includes adding new Umple syntax, metamodel elements, model analysis capabilities, and code generation.

We validate our real-time modelling using a variety of cases and examples. We develop different communication patterns to test complex communication protocols, which is necessary to support standard communication stacks. Our design objectives include reduction of the volume of code (including code describing modelling elements), portability, independence from third-party libraries, and ease of use. For each of these objectives we hope to exceed what has been achieved by non-generative approaches that require industrial libraries.

1.1 Motivation

In generative modelling, models can be written and then translated into the appropriate target language. Major difficulties faced in traditional programming (such as hardware support for diverse platforms) can be more easily handled in model-based development.

``

Starting in the 1990s, the model-oriented paradigm became widespread. For instance, in the automotive industry, 90% of the system functions in automotive applications are expected to be defined in software components as compared to other system components [8]. Any improvement to automotive software modelling ought therefore to have significant impact.

1.2 Research questions

Our research activities aim to improve real-time system development, guided by the following questions:

1.2.1 RQ1

How can we improve component-based modelling to enable better and easier development of real-time and distributed systems using Umple?

Component-based modelling depends on composite structure and active objects notions, which are now incorporated in UML specifications. Component-based concepts are used frequently in real-time development. Several UML tools support component-based development. This will be discussed in detail in Chapter 2.

The following are some issues and limitations that we experience when using the existing UML tools (even the high-end ones) to develop components:

1) **Lack of generative component-based modelling**: Most open-source UML tools do not support component-based modelling, and if they do, they do not support code generation from such models [9] (Section 2.6). In the next chapter, we will show a comparison of different UML tools in terms of their level of adoption of composite structure modelling. A key contribution of the thesis will be better generative modelling for component based systems.

2) **Model and textual gap**: Existing tools require specific diagrams, enhanced with code that is separated in some way from the model. We aim to avoid the round-trip process wherein generated code must be edited. Such an approach causes the relation between model and code to become convoluted or lost and thus increases complexity. The rule of thumb is not to edit the generated code, and let the user deal with modelling elements and code snippets as a single artefact but with different views (such as visual and code views).

3) **Code injection**: In UML, some elements (such as operations, transitions, states, and so on) can have a code body where the users can write their own code snippet. From the visual perspective, it is not possible for the user to know the code contents of such an element unless they select this element and switch to their code body view. Switching between code and visual views is not straightforward and adds more effort and consumes time while modelling, especially if the model is large and contains a lot of code-enabled

``

elements. Alternatively, users can add some visual note elements but they are still not enough for the users to visually spot errors (if there any) while debugging their models.

4) **Distributed environment:** Component-based applications are typically used in distributed environments. Sometimes, users may be limited to certain networking paradigms such as client/server. Developers may be required to deal with additional terms such as wiring and incarnation, as well as to regularly update the original design to deal with unforeseen networking obstacles. Transferring signals among components is not straightforward as it involves several stages such as serialization/deserialization and network configuration, as well as the frequent need to support multiple media formats such as JSON and XML.

1.2.2 RQ2

What are the major challenges in developing a real-time construct in a textual language?

The main points that we need to address when enabling a real-time software development in a textual notation can be listed as follows

1) **Usable and easy-to-use syntax**: We aim to provide a compact and friendly syntax. We want to maintain Umple's concept of treating model-code as a single artifact and avoid forcing developers to switch between textual and visual notations, while allowing them to see either at their preference.

2) **Visualization of textual code representation**: We aim to enable developers to visualize their code, which is necessary for documentation and inspecting work to find errors.

3) **Scalability**: Textual representation is useful for tasks such as editing and version control. However, as models get bigger, it becomes more challenging to scale such tasks. Various features that rely on variability modelling and separation of concerns can combat this by allowing developers to split their code/model in whatever way is most suitable. These features include aspect-orientation, mixins, and traits [10]. In our work we enable all of these features.

4) **Validation**: Existing tools tend to lack native validation and tracing capabilities. We aim to provide comprehensive validation of our constructs, based on our textual language. Our work in particular enables users to specify validation rules and best practices rules which cannot easily defined diagrammatically.

1.2.3 RQ3

How can we evaluate our proposed technology?

``

Our evaluation approach is based on the model executability in terms of feature validation, semantics, and code quality evaluation. We investigate and evaluate a set of real-time system examples that demonstrate our features and show that our language is compact, expressive and integrates nicely with other features of Umple.

1.3 Contributions

The top-level contributions of our work are.

1. **Introducing real-time software development into Umple:** We fully implemented real-time code generation in Umple, as our work is fundamentally dependent on real-time modelling. We covered all Umple features, such as attributes, associations, state machines, mixins, patterns, and aspects.

2. **Umple as a Template Language (Umple-TL)**: This is a model-based text-generations technology; developing this was crucial for our research in order to improve and ease the development process of its other aspects. Umple-TL now is one of the core features in the Umple project, and it is used to generate all Umple target languages.

3. **Enable component modelling in Umple**: We added composite structure and active object features to Umple, including a textual syntax, diagram generation as well as generation of executable code and configuration files. We made this available through the command line interface, in UmpleOnline (try.umple.org) and in Eclipse. The core implementation modules are a real-time extension, composite structure, active objects, and their associated component descriptions. Our work is designed to enable future integration with Autosar so it can be used in the automotive industry [11].

4. **Distributable environment:** Our component-based models are enabled for distributed environments. A crucial aspect of this is that Umple users do not need to change or apply any special design on their models to handle distributed concepts such wired or unwired communication.

5. **Assessment of component modelling tools, and comparison of them to our Umple-based approach**: We explore different modelling tools to investigate their existing features for component-based development. We will show how we incorporated the major features of these tools. Additionally, we show how our newly introduced Umple constructs can meet aspects of common specifications such as UML [12], AUTOSAR [13], MARTE [14], and EAST-ADL [15], which all focus on time management. We as well show how we can overcome limitations such as enabling the three call types, synchronous, one-way asynchronous, and two-way asynchronous. This is as opposed to specifications such as UML, which only supports one-way asynchronous calls, due to the complexity of supporting other call types.

``

1.4 Thesis structure

In Chapters 2, we provide the required literature reviews for our research.

In Chapter 3, we explain the implementation of Umple as a Template Language (Umple-TL), which is used to ease the process of generating templates by consolidating template definitions into Umple.

In Chapter 4 we explain the implementation of concurrency and active objects in Umple, which are fundamental for the introduction of component-based modelling.

Chapter 5 is the core of our research: We present the syntax, semantics, and implementation of our composite structure and distributed extensions to Umple.

Finally, in Chapter 6, we discuss some case studies showing how they can be written in compact Umple models using our new constructs. Also, we evaluate the quality of our generated code

``

Chapter 2 Component Modelling

In this chapter, we highlight the key contribution of this research, the introduction of component-based modelling into Umple. We show a motivating example to illustrate how we can use Umple to develop component-based models. The development is available both textually and visually, as is the case with other Umple features. Second, we show a comparison between Umple and other modelling tools, in which we pinpoint the features that are crucial for the support of component modelling, and their level of support in our implementation.

2.1 Key concepts of component modelling

To design a component-based software system, developers must split the system into abstract independent components. A component is also referred to as a part or building block [16], and can typically be replaced by some other component with a different implementation that adheres to the same specification [17]. A self-contained component contains an object or a set of objects, and encapsulates certain information and/or functions. Self-containment means that a component can be compiled, stored, linked, and run separately. Dependencies on other components should be minimized and carefully managed.

A software component is considered a black box, since its implementation details are not externally visible by default. Each component can be accessed using an access point or interface which can be either *provided* or *required* [12]. A provided interface specifies the services that a component provides to other components. Alternatively, the elements of a provided interface can be called *provisions*. A required interface specifies what services a component requires from other components. A well-designed component must be reusable and configurable by other applications, similarly to how it was deployed in its original application.

A component can either be designed to execute sequentially or concurrently [16]. If concurrent, that means that it is *active*, executing in its own thread. Different types of communication can be applied in concurrent design, such as the active object pattern [18]. A sequential design means that a component acts as a passive class, and does not have its own thread of control. A call/return pattern is the only possible communication pattern in sequential design.

2.2 UML component modelling

In this section, we will discuss component modelling in UML and show how it can be handled in a real-time environment. We will give an idea about the major UML elements that are required to develop component-based models.

``

2.2.1 Composite structure

Composite structure was identified in UML 2.0 to extend modelling capabilities beyond classes, and to better handle decomposition [12].

A composite structure element is owned by a class instance, and it encapsulates a set of internal components [16]. Internal components are also called parts [12]. A component is a type of class, which means that it has class features such as generalization, association, and operations. Components can be accessed using interfaces in order to enforce encapsulation and loose coupling. Component behaviour relies on its interface's specification and implementation.

A component can have a composite structure diagram, a container component, that [12], [16] 1) shows whole/part relationships for its internal parts, while other class entities, such as attributes, can be omitted, and 2) describes the system configuration requirements.

Relationships between a component and its internal components are context-sensitive, since they are only represented based on their owning class and decomposition level.

Internal parts within a composite structure can define behaviour in the context of their container. Class diagram refactoring becomes more reliable, as changes are insulated from internal component behaviour.

Each component acts as a physical and logical container at the same time, since it can have more than one instance of the same part class type but with different view information (Figure 2-1).

Figure 2-1. Multiple instances of the same component

It should be noted that figures in this chapter, such as Figure 2-1, are generated in Umple as an element of the work of this thesis.

2.2.2 Structured classes and interfaces

A structured class is a simple root package that allows elements, such as classes, to have internal structure and port capabilities.

Interfaces are used to enforce abstraction between software components. An interface defines a construct to be implemented by its components properly. Components are not necessarily required to use or implement all interface constructs (specifically, a component only needs to implement those constructs that other components actually need;

``

if a component C implements interface I, but not completely, then C would be an abstract component). The required and provided interfaces (discussed below) of interacting components must be compatible [19].

An interface construct contains operations, services (for client-server architecture) and data items (to define component behaviour) [16]. Interface construct implementation must describe how information can be transmitted between components by properly configuring data items [19].

An operation has a unique tuple that consists of its name as well as a parameter list [16]. An interface does not provide implementations, which exist in its implementers such as components and ports.

Interfaces describe the interaction points of their implementers. Upon operation invocation, interfaces are bound to their implementers [12].

A provided interface is shown diagrammatically using the "lollipop" notation, a small a circle on a stick (Figure 2-2), while a required interface uses the "socket" symbol, a semi-circle on a stick (Figure 2-3). Both symbols appear together for provided/required interfaces (Figure 2-4).

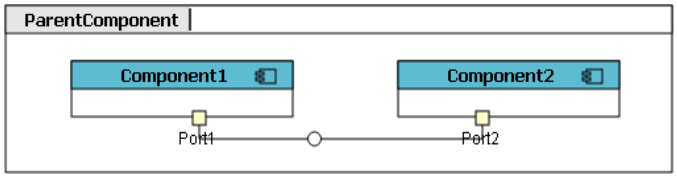


Figure 2-2. The circle notation for a "provided" interface

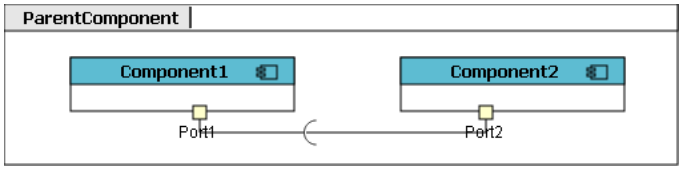


Figure 2-3. A semicircle notation for a "required" interface

``

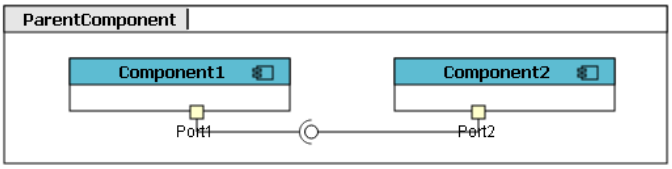


Figure 2-4. The "lollipop" notation for a provided/required interface

2.2.3 Part or subcomponent

Subcomponents define hierarchical composition and internal structure of their owing components [16]. A part can have more than one instance that can exist in an instance of its container at runtime [16].

A part is used to represent a role of its type within its containing class. For example, in Figure 2-5, part "a" shows an instance of type "A", and part "b" shows an instance of its type "B" while both instances exist in the context of their owner instance "c" that is of type "C".

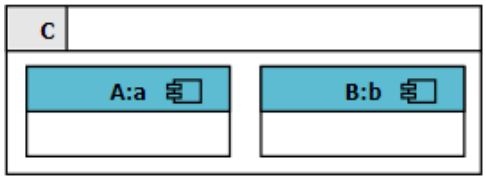


Figure 2-5. Multiple instances of different components

When one or more part instances are added to the internal structure of a class, this means that this class will have a composition relationship to the owning type of these instances. A composition relationship is required, since the part instances cannot exist independently of their parent instance.

Considering an example, in which a composite structure "A" contains two instances of the same part "B" (Figure 2-6). A composition relationship is denoted as the "black diamond" symbol. Figure 2-7 shows the structure diagram of "A", which has two instances of B. The "B" instances are what caused the composition relationship to be drawn in Figure 2-6.

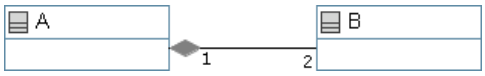


Figure 2-6. The composition relationship of Figure 2-5

``

Figure 2-7. The structure diagram of "A" shown in Figure 2-6

Class diagram constraints and properties must be applied in the context of their owned composite structure.

Composite relationships between a container and its parts are implicitly present; in other words, they cannot be optionally omitted. Other complex cases may require relationships to be specified explicitly. Examples include refinements, constraints, and associations as will be shown below.

Refinements (inheritance) still can be applied at the composite structure level such as adding component constraints. For example, a part can have multiplicity, which can be represented in the composite structure but must be enforced in the corresponding class diagram.

Considering the following example, a part "b" of type "B" exists in a composite structure of type "A". When setting the multiplicity of part "b" to be "4", this means that the implicit composition relationship between classes "A" and "B" must have the same multiplicity "4" (or less constrained) in the class diagram.

It is important to mention that composite structure diagrams and class diagrams are at the same level, but each represents different views.

A class can have references to other parts but not necessarily own them. A reference to another part can simply be defined as a normal association. Composite structure diagrams can represent part references as dashed rectangles (Figure 2-8) [12].

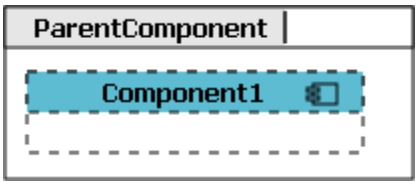


Figure 2-8. A reference to a class

Alternatively, an aggregation relationship can be drawn to imply references; however, it is not recommended because of its semantic limitations as compared to associations.

Reference relationship directions are not defined directly, but additional constraints can be applied to prohibit the referenced parts to call the referencing class. For example, in Figure 2-6, the cardinality of the composite relationship is set implicitly to one-to-two. When additional instances are added, the upper bound of that relationship will be increased accordingly. However, when a developer decides to define this relationship directly, as an

``

association, the number of instances will be enforced based on that association's multiplicity. For example, if an association between two classes sets the upper bound to be three, then a developer will not be able to add more than three instances of the targeted class.

2.2.4 Ports

A port is owned by a component, and it is used to specify a dedicated communication point between its owner and other elements [12]. Any communication between a component instance and its environment will pass through this port.

A port specifies communication using "provided" and "required" interfaces [16], [19]. Interface implementation depends on the port communication type, which can be:

1. **Provide Port (P-Port):** a port provides a service for the implemented interface. A P-Port is commonly used to transmit data to other ports, but in some cases, the P-port can receive data.
2. **Require Port (R-Port):** a port requests a service from the implemented interface.

A port can comprise (group) one or many interfaces [16]. A complex port is a port that comprises more than one interface [16]. For each port, there must be at least one provided and/or one required interfaces.

By default, a port maintains a high level of model abstraction since it is typed by its interface types [16]. Interface types must be semantically related but not necessarily of the same type in order for the port to model interactions. An interaction is a sequence of invocations between port interfaces and their environment [20]. A port interface can have a state, which represents a current phase of the invocation sequence [20]. The port object is used to bind states to its interfaces [20].

In UML 2.0, a port can have multiplicity indicating the number of port instances that can be created within this port's owning element.

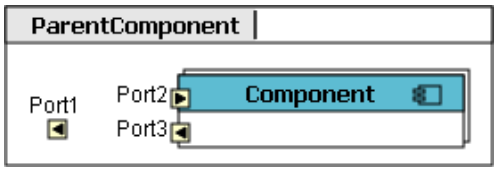


Figure 2-9. Ports of different types

A port shape is denoted as a rectangle, commonly placed on its owner's border, and in some cases on its owing component structure diagram. In Figure 2-9, Port1 is located on the diagram because it is private, while both Port2 and Port3 are placed on the border because they are public. Port2 is a required port, while Port3 is provided port; each one is distinguished by a unique direction symbol (Figure 2-9).

``

2.2.5 Connectors

A connector is used to connect components by acting as a junction between those component port interfaces [16]. Ports cannot be connected if there is any interface type mismatch. Connectors provide a high level of abstraction since the details of the connected components are not of interest. Connectors between ports are used to send and receive signals between ports. A connector can also be called a wire.

A component that requires an external service from another component must have at least one required port interface, while the other component must have at least one corresponding provided port interface [16]. This diagram notation is shown as a lollipop in a slot (Figure 2-2).

Connectors are used to control the flow of data between components that becomes more frequent in distributed systems. A wire is used to send and receive signals via ports of the connected components [20].

A connector can be either unidirectional or bidirectional based on the port interface types [16]. If two connected ports are both required and provided, then the connection is bidirectional; otherwise, it becomes unidirectional and its source is the port with the provided interface, and its target is the port with the required interface. Note that this does not correspond to direction of data flow, which is a separate concept.

There are various ways to express the semantics of connectors. The three basic notations that are commonly used in UML are [12]:

1. **Port**: When the port types of two components are unique, those ports can be simply wired directly with the possibility to ignore unused port interface artifacts. The port notation is usually linked to the callback mechanism. The connector shape is represented as a simple wire that connects the corresponding ports.
2. **Usage relation**: Required and provided interfaces at two components can be connected with a usage relation instead of using a wire. The usage notion confirms that a component does not only connect to the other component but also has a dependency relationship for full implementation of that component. The usage relation is denoted at dotted line in UML (Figure 2-8).
3. **Ball-And-Socket**: This is a simplified notation of the usage relation, and it is designed for local communication. This notation is widely used to depict component dependencies due to its simplicity. Port interfaces are joined directly into one symbol (Figure 2-4).

There are two kinds of connectors that UML 2.0 specifies [12]:

1. **Assembly (binding):** Connects required interfaces of a component to provided interfaces of another component. A binding connector is usually used (but not restricted) to connect subcomponents that report to the same parent.

``

2. **Delegate:** Is usually used to connect provided interfaces of a component to provided interfaces of one of its subcomponents, or required interfaces of subcomponents to required interfaces of their owing component. For the provided-to-provided scenario, a component uses a delegate connector to delegate requests from one provided port to another, implemented as a subcomponent. For the required-required scenario, a subcomponent that requires functionality delegates to a required port of its parent component, also using a delegate connector. The concept of delegate connector is not described directly by UML 2.0, so a "delegate" stereotype can be added to the connector.

2.2.6 Composite component

There is a common confusion between composite component and composite structure, since a composite component inherits component structure capabilities [12]. Composite structure focuses on depicting the internal structure of components, while composite component is more about the communication behaviour among different components, and showing the system as a whole. A composite component tends to use connector capabilities more frequently, such as delegation and assembly connector types.

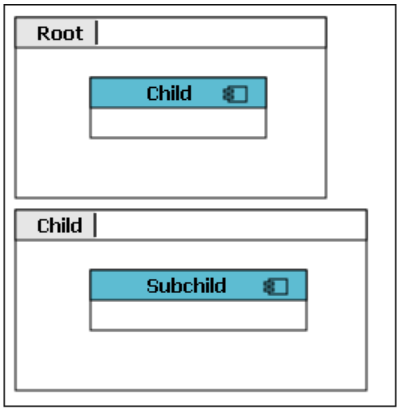


Figure 2-10. An example of a passive component by its parent

A composite component can be used to show the deployment view of the system's components [16]. Components can be either active or passive [16]. Active components can be deployed independently from other components, while passive components' deployment depends on other components to be deployed first [16]. The concept of active component is closely related to concurrency, as an active component has its own thread of control.

There are two main cases when a component is considered passive. These are:

1. A subcomponent can be a passive component to its parent. In Figure 2-10, Subchild is a passive component to its parent, "Child".
2. If a composite component defines a required interface, then it will be a passive component, as it will rely on the other component's provided interface (Figure 2-3).

``

2.3 Component-based modelling and standards

Many of the existing component-based modelling tools (Section 2.62.6) adapt specifications such as UML [12], SysML [69], Specification and Description Language (SDL), and Real-time Object-Oriented Modelling (ROOM) [12].

In our work, we are more inspired by UML, since, 1) OMG uses UML as the de facto modeling notation for object-oriented systems [69], and 2) UML is well-known for design specification for real-time systems, especially embedded devices. As well, in terms of real-time modelling, we follow many of the ROOM specifications. However, we have an extended active object pattern to handle concurrency (Chapter 4)

SysML provides a lightweight profile of UML in terms of aspects such as stereotypes, constraints, and tagged values hence, it can ease and reduce UML restrictions, and support a wide range of systems, either software or hardware.

SysML only uses seven of the UML diagrams (Figure 2-12) in addition to two diagram types, requirement and parametric (Figure 2-11). A class diagram is called a "block definition" diagram and a component (composite) structure diagram is called an "internal block" diagram.

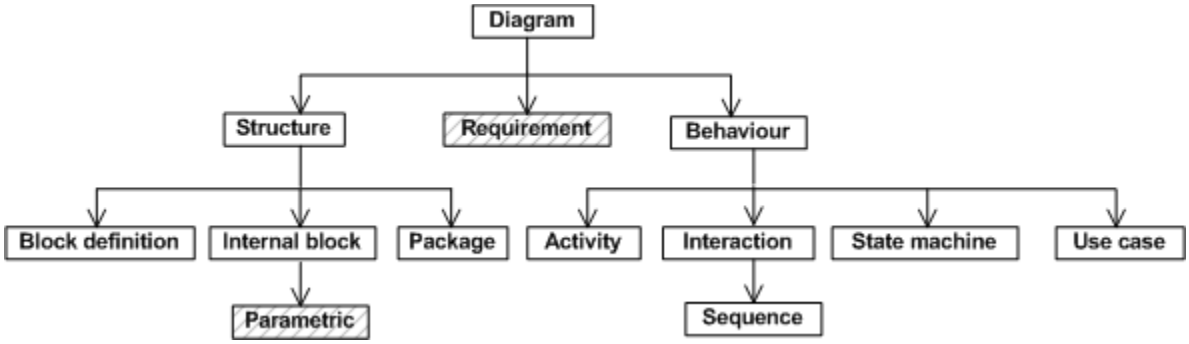


Figure 2-11. A hierarchical representation of SysML diagram types
Additional SysML diagrams, as compared to UML (Figure 2-12) are highlighted

``

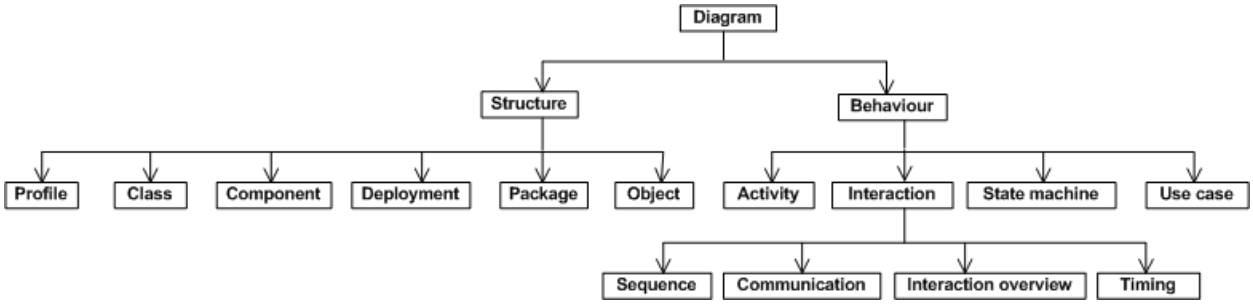


Figure 2-12. A hierarchical representation of UML diagram types

Table 2-1. UML to SDL term mapping

UML	**SDL**
Class	Type
Interface	Interface
Associations	Channels
Operations	Signal List
Variable	Attribute
Sub	Block
Abstract	Abstract
Implementation	Process
Type	Gate
Inheritance	Inheritance

SDL's main purpose is to provide unambiguous software system specifications [25]. SDL has mainly been used in the modeling of real-time communication systems. SDL provides similar functionality as UML but with different terminology and notation (Table 2-1).

ROOM was developed by ObjectTime and introduced in the ObjectTime Developer (ODT) tool [12]. ROOM incorporates a variant of Harel's statecharts. In 1996, RoomLanguage represented the actor as the primary element that communicates with other elements using port interfaces. A port is an instance of a protocol class that defines the message communication between actors. The concept of Actor was later referred to as Capsule in UML-RT. ROOM supports hierarchical modeling and incremental refinement of complex behaviour.

2.4 Component modelling in practice

There are some disadvantages that developers commonly face when using a component-based approach. An entire system must be rebuilt when one of its components imports external libraries. Shared libraries can partially solve this limitation, as they can be loaded as separate entities.

``

The same shared library can have different versions. Some components can use the same shared libraries; however, this does not mean that they are necessarily using the same version or even compatible versions. Upon version conflicts, a developer will need to go through complicated configuration processes in order to resolve such issues. In some cases, those conflicts may be irreconcilable forcing a developer to find a workaround or to disrupt some major features of functions of certain components.

Procedural functions can be inflexible since new libraries can introduce changes to parameter signatures. It will require extra effort to adapt to these changes.

Detecting the root cause of defects in component systems can be tricky in many cases when debugging existing issues [21], [22].

Although a component can interconnect directly with a nested component in another composite component, it is good idea to connect components that only belong to the same root component. Complicated interconnection at different levels can cause future changes to the system to be unmanageable [21], [22].

2.4.1 Component modelling specification

The specifications of component modelling can be summarized as follows:

- **Independence**: Component interfaces and implementation details should be separated, so any implementation changes will not cause major changes. Component models must adequately describe properties, constraints, and the proper building blocks based on the interfaces that the component will implement.
- **Infrastructure requirements**: Separation of concerns must be maintained across different components. Any implementation details that seem to be service-oriented (such as life-cycle management communication) must be migrated to the appropriate middleware components.
- **Composition rules**: Component models must precisely define the composition rules such as how, where, and, when components can communicate, and the required interfaces. Composition rules are necessary for component integration. Composition rules must be language-independent.
- **Interaction rules**: These are used to define the flow of control, as well as data transmission among components at run-time.

2.4.2 Component modelling objectives

We can summarize the basic steps that we need to develop a component modelling system as follows:

- **Environment requirement**s: Component-based systems must be designed to operate in a distributed environment where a component is considered a physical node. Each node must be able to execute as a standalone entity.

``

- **Component development and integration**: A component must be developed in a way that matches component-modelling specification so component integration becomes an easy task.
- **System deployment**: The system can be deployed as a single entity after its component integration process. Deployment requires hardware configuration to physical nodes in the system.

2.5 A motivating Umple component-modelling example

In this section, we show a simple ping-pong example (Snippet 2-1 and Figure 2-13). There are two peers denoted as Component1 (Lines 1-18) and Component2 (Lines 20-35). Both components are contained in another component (Lines 37-48), which will send an initial message via a port (pIn1) to get the communication started (Line 43).

The value received will be incremented by 1 (Line 10), and sent back to the other component (Line 10). The message propagation will continue between the two peers until the value becomes 10 (Line 25), at which point back-and-forth propagation will end.

The events keep going through four ports, pIn1 and pOut1 (Lines 2 and 3) in the first component, and pIn2 and pOut2 (Lines 21 and 22) in the second component (Figure 2-14). The basic event methods in this example are pingIncrement (line 9), pongIncrementOrStop (Line 26), logOutPort1 (Line 15) and logOutPort2 (Line 33).

The increment (Lines 9 and 28), and log (Lines 16 and 33) methods are defined as active methods, which means that each one has its own thread.

Umple

```
1  class Component1 { // A component
2    public in Integer pIn1;  // An in port
3    public out Integer pOut1;  // An out port
4    pIn1 -> pOut1;  // A connector
5
6    // A watch constraint defining the in port on which a signal triggers the following.
7    //An active method is invoked when a signal is received
8    [pIn1]
9    active pingIncrement {
10     pOut1(pIn1 + 1);
11   }
12 // A watch constraint; the following is triggered when a signal is sent
13 // through this out port
14   [pOut1]
15   active logOutPort1 {
16     cout << "CMP 1 : Ping Out data = " <<  pOut1 << endl;
17   }
18  }
19
20 class Component2 {
21 public in Integer pIn2;
22 public out Integer pOut2;
23   pIn2 -> pOut2;
```

``

```
24    // A watch constraint, and a value constraint
25    [pIn2, pIn2 < 10]
26    active pongIncrementOrStop {
27        // Get a read-only copy of the current cached value at port Pin2
28        pOut2(pIn2 + 1);
29    }
30
31    [pOut2]
32    active logOutPort2 {
33      cout <<"CMP 2 : Pong Out data = "<< pOut2 << endl;
34    }
35  }
36
37  class Atomic {
38    Component1 cmp1;
39    Component2 cmp2;
40    Integer startValue;
41
42    after constructor {
43      cmp1->pIn1(startValue); // Initiates communication in the constructor
44
45    }
46    cmp1.pOut1 -> cmp2.pIn2;
47    cmp2.pOut2 -> cmp1.pIn1;
48  }
```

Snippet 2-1. An Umple component-modelling example

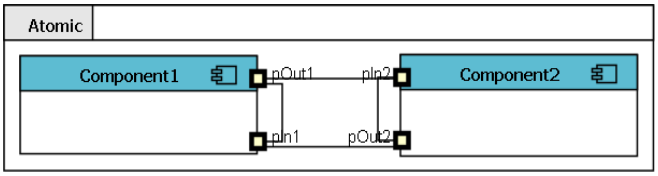


Figure 2-13. A simple ping-pong example using Umple

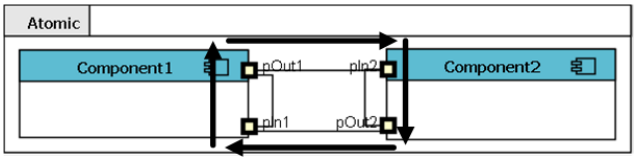


Figure 2-14. The flow of information in the ping-pong example (Snippet 2-1)

2.6 A comparison with other tools

During our research, it was important to compare Umple to other modelling tools in order to make sure that we fulfilled the core requirements of component-based modelling (Table 2-2).

``

The selected items in our comparison include industrial and research tools. The tools that we conducted our comparisons against are starUML [23], eTrice [24], ArgoUML [25], Rational Software Architect Real-Time (RSARTE) [26], IBM Rhapsody [27], Papyrus-RT [28], and PragmaDev.

Our comparison focuses on the composite structural features. The main such features that we found most widely supported during our research include decomposition, data sharing, communication, and type management. Decomposition is the top-level core feature that determines whether a tool supports breaking a system into a number of components. Data sharing, as the name suggests, refers to the ability to share data among physical and logical components. Communication refers to the concepts used to define information flow among components, which are typically protocols or gates/interfaces; ports/active methods in our case. Type management refers to tool support for creating custom types or using complex types for data transmission. During our research, we found that other modelling features could affect the development process of component-based applications even if they are not structural; examples include behaviour and simulation features. For instance, behavioural features (primarily state machines) are required for behaviour ports. Simulation features are required for many customers that rely on component-based development. For example, in automotive development, simulation is crucial for a complete end-to-end testing process among software and hardware components.

Table 2-2 summarizes the results of our comparison. In Table 2-2, the symbol "X" indicates a moderate level of support; "~" indicates a weak level of support, and “-” means no support.

In our comparison, we do not go into deep detail. For example, we mention if a tool supports code generation rather than mentioning the quality of the generated code.

``

Table 2-2. A comparison of some component modelling tools

	Functional Features									Simulation	Code Generation		
	Behaviour				Structural								
Tool	**Decomposition**	**Composite states**	**Concurrent states**	**State class**	**Decomposition**	**Concept**	**Data Sharing**	**Communication**	**Type Definition**		**Java**	**C++**	**C**
starUML	X	X	X	-	~	~	~	~	~	-	-	-	-
eTrice	X	-	-	-	X	X (Actors)	-	X (Port/ protocol)	X	X	X	-	X
ArgoUML	X	-	-	-	-	-	-	-	-	-	~	~	-
RSA-RT	X	X	X	-	X	X(Aggregation/ Composition)	X	X (Port/ protocol)	X	X	X	X	X
Papyrus-RT	X	X	X	-	X	X(Aggregation/ Composition)	X	X (Port/ protocol)	X	X	X	X	X
IBM Rhapsody	X	X	X	-	X	X(Aggregation/ Composition)	X	X (Port/ protocol)	X	X	X	X	X
PragmaDev	X	X	X	X	X	X(Agents)	X	X (Gate/Interface)	X	X	-	X	X
Umple	X	X	X	-	X	X(Aggregation/ Composition)	X	X (Port/ Active method)	X	X	X	X	-

In terms of communication, tools can differ based on the standards that they support. For example, tools that follow UML standards use ports and protocols for their communication; those tools include eTrice, RSARTE, Papyrus-RT, and IBM Rhapsody. On the other hand, tools that support Specification and Description Language (SDL) [29] use gates and interfaces for communication; PragmaDev is an example.

In Umple, we follow UML, which means that communication should rely on ports and protocols. However, we found that requiring a user to define their protocols explicitly, adds unnecessary overhead, which is our main motivation to introduce a *protocol-free* approach in Umple. By this, we mean that Umple uses inference to extract the required protocol. Thus, in Table 2-2, we mentioned that our communication depends on ports and the pattern of active object.

For state machines, Umple support the major features listed in Table 2-2 except for the "State Class". However, this is a deliberate design decision to maximize the efficiency of Umple state machines [30].

``

In terms of code generation, the most important target languages supported include Java, C++, and C. Umple supports real time code generation in C++ for all modelling features. Code generation for C and possibly Java will be supported in the future (Umple supports Java already for non-real-time features).

Most commercial tools support behavioural and structural decompositions, while the only open-source tool other than Umple that supports them is eTrice and Papyrus-RT. eTrice support is partial, while Papyrus-RT has all the major features supported. Two tools (starUML and ArgoUML) have little or no code generation support for component-based modelling.

``

Chapter 3 Umple as a Template Language (Umple-TL)

In this chapter, we show how we extended Umple to act as a template language. This means that Umple itself can be used to generate textual output based on a template that follows a specific set of custom rules and constraints. When we use the word ‘template’, we are referring to the generation of output, and *not* to ‘template parameters’ or 'generic types', the other common use in computer science of the term, referring to specifying generic types, typically in angle brackets.

The purpose of developing this capability in Umple is to support the work in subsequent parts of this research. In particular, we needed to create a more sophisticated code-generation capability to support C++ code generation, active objects, and composite structure. Another motivation was that Jet, which had been used previously by Umple, was deprecated and we wanted to avoid dependencies on other tools, particularly Eclipse-based ones that make it hard to build software outside of Eclipse using scripts.

We focus on showing the key features of Umple-TL, and illustrating how it can overcome common limitations of competing technologies that restrict developers to a specific development language. The users of Umple can use any development environment or even directly use the UmpleOnline editor. When using Umple-TL, the generated code can be in any of the available options provided by Umple; e.g. Java, Ruby, C++, and PHP. During our discussion in this chapter, we focus on Java and C++.

We use the term *escape characters* to refer to special sequences of characters used to indicate the beginning and end of special sections of the Umple input text marking places such as strings to substitute. We show how Umple-TL can provide additional capabilities stemming from Umple being a model-oriented and object-oriented language.

Finally, we show a comparison between Umple-TL and common tools used for text emission.

Umple-TL is documented further in the Umple User manual [31]. It has been used by other researchers as a replacement for Jet to generate other Umple target languages such as Java, PHP and Ruby, as well as to develop various features of Umple such as metrics generators and state tables. These examples can be found in the codebase of Umple on Github [32].

3.1 Overview

Fundamentally, templates are textual macros, a very old concept, that has been around for years in languages such as C and C++ [33]. As well, some languages such as PHP, in fact, were specifically designed with generation of textual output as their motivating use case. Other languages such as Java do not come with built-in template mechanisms

``

and rely on somewhat verbose method calls to generate text. However, there are a wide variety of contexts where generating formatted textual content is an essential requirement, including generation of data formats such as XML and html, generation of modelling and programming languages (in metaprogramming and code generation), generation of messages for inter-process communication, and generation of user interfaces. A key objective of Umple is to make software development simpler by adding modelling and other constructs to base languages. Templates are just one of the many abstractions that have been added to Umple as a necessary step to achieve Umple's overall goals.

For simplicity, we will refer to any text emission tool, extension, or project as a text emission tool; someone using a text emission tool is called either a developer or user. What a developer writes will be called a source file; a text emission tool is used to generate target files based on the content of the developer's source files. These target files are typically used at run time to generate specific string output, based on data in the running program. This process is summarized in Figure 3-2.

The basic two elements in Umple-TL are *templates* and *emitter methods*. A template, as its name suggests, is used to define some text content to be appended to or substituted within the final output text. An emitter method is a method that uses one or more templates to construct a text output.

Grammar 1 shows the grammar definition of Umple-TL in RailRoad syntax [34].

```
Grammar  ::= templateAttributeDefinition*
templateAttributeDefinition ::= templateIdentifier templateBody
templateIdentifier   ::= ( classname'.' )? name (templateParameters)?
templateParameters::= '(' templateParameter (',' templateParameter)* ')'
templateParameter::= parameter | "" parameterValue ""
emitMethod::= (modifier)? ('static')? type 'emit' methodDeclarator templateList ';'
modifier::= 'public' | 'protected' | 'private'
templateList::= '(' (templateName ( ',' templateName )* )? ')'
templateBody::= '<<!' (templateBodyContent)* '!>>'
templateBodyContent::= templateExpression | templateComment | templateCodeBlock | templateText | templateExactSpaces
templateTagContent::= '<<' templateTagChoice
templateTagChoice::= '=' | '#' | '/*' | '$'
templateText::= ( text [^templateTagContent])*
templateComment::= '<<//*' commentText '*//>>'
templateExpression    ::='<<='((expressionText)[^templateTagContent])+'>>'
templateCodeBlock    ::= '<<#' ((codeText)  [^templateTagContent])*  '#>>'
templateExactSpaces  ::= '<<$' S*  (templateText [^templateTagContent])* '>>'
```

Grammar 1 Umple-TL. Details are in Appendix A

``

3.2 Fundamental concepts of Umple-TL

When describing Umple-TL we will refer to three types of text:

- **The original Umple source text**: This is written in Umple, with embedded code written in the target language.
- **Generated target language code**: This is generated by the Umple compiler from the original Umple source text. It is designed to be readable, so it can be inspected and debugged. It is also designed to execute efficiently
- **Final output text**: This is generated at run time when the generated target language code is executed. Producing this is the whole objective. Umple-TL has some special features to help format this nicely; for example, to enable whitespace to be appropriately output.

As mentioned in the last section, to use Umple-TL, a developer will write two types of entities:

- **Templates**: These describe the output to be generated. A template is a specialized Umple attribute with a unique label and a body defined within the top-level block type, as indicated in Table 3-1.
- **Emitter methods**: These are invoked to create output based on the templates. The keyword 'emit' is used to indicate these.

Both of the above must be defined within a class.

Since a template is an Umple attribute, it must be named following the rules for naming attributes. In particular, an attribute's name cannot start with a number, and it is not allowed to have the same name as any other attribute or template.

Similarly, an emitter method is a specialized method type, which means that it will follow the same restrictions applied on normal Umple methods.

At least one emitter method and one template are required for a class to be recognized as a template class.

Umple-TL templates can contain arbitrary text, plus a number of escape character sequences specifying *expression*, *exact space*, *comment*, and *code blocks* (Table 3-1).

``

Table 3-1. The types of blocks found in templates

Block type	Description	Escape Sequence
Top Level	Define the start and end of a template body. The following are all nested in the body.	<<! >>
Expression	Used to append strings to the final output text directly; typically, the strings to be appended are variables or the return values of method invocations in the target language.	<<= >>
Exact space	Used to specify whitespace that will appear at the beginning of every line in the final output text.	<<$ >>
Comment	Add a comment to the generated target language code; the comments will be in the syntax of the target language used.	<</* */>>
Code	Inject code segments in generated target language code. Such code blocks enable developers to define logical conditions and loop statements. The content of code blocks are not appended to the final output text, hence are quite different from expression blocks.	<<# #>>

Snippet 3-1 shows a simple template labelled as "templateLabel", which is used by the emitter method named generate1.

Umple

```
1  class TemplateTest{
2      templateLabel<<! My Template !>>
3      emit generate1()(templateLabel);
4  }
```

Snippet 3-1. A simple Umple-TL example

Snippet 3-2 shows the generated code of Snippet 3-1 with the assumption that the selected target language is Java. In Snippet 3-2, we only show the associated portions of code related to the generated emitter method, and omit other code not necessary to understand the example.

In Snippet 3-2, there is a method named generate1 (Line 4), which is the same as the emitter method defined in Snippet 3-1. There is an additional method named _generate1 (Snippet 3-2- Line 9) used by generate1 (Snippet 3-1- Line 6).

The method _generate1 has the main implementation, and it contains an additional parameter for the number of spaces to be appended before each line of the output text. The call for _generate1 made by generate sets the number of spaces as zero. If a user wants to add spaces to each line of the output text, they can call the method _generate1 directly and set a value for the number of spaces required.

``

The output of generate1 and _generate1 will be the text content of templateLabel defined in Snippet 3-1- Line 2. Specifically, the text content is assigned to the static constant declared in Snippet 3-2- Line 3. The example shown in Snippet 3-1 and Snippet 3-2 is simple. In later examples, such as Snippet 3-29, a higher number of string constants will be generated; eventually those constants are all used by the generated emitter method to construct the output text.

```
1  public static final String NL =                                    Java
2      System.getProperty("line.separator");
3      public static final String TEXT_0 = " My Template ";
4      public String generate1() {
5          StringBuilder sb = new StringBuilder();
6          return this._generate1(0, sb).toString();
7      }
8
9  public StringBuilder _generate1(Integer numSpaces,
10     StringBuilder sb) {
11     String spaces = "";
12     StringBuilder newCode = new StringBuilder();
13     StringBuilder realSb = sb;
14     if (numSpaces > 0) {
15         realSb = newCode;
16         spaces = _createSpacesString(numSpaces);
17         newCode.append(spaces);
18     }
19     realSb.append(TEXT_0);
20     if (numSpaces > 0) {
21         newCode.replace(0, newCode.length(),
22         Pattern.compile(NL).matcher(newCode).replaceAll(NL +
23          spaces));
24         sb.append(newCode);
25     }
26     return sb;
27 }
```

Snippet 3-2. A simple Java generation example of an emitter method using Umple-TL

An emitter method is designed to be called as a helper method (Snippet 3-3).

```
1  System.out.println(new TemplateTest().generate1());                Java
```

Snippet 3-3. An example of invoking a generated Java emitter method

Snippet 3-4 and Snippet 3-5 show the equivalent code generation for Snippet 3-2 and Snippet 3-3 in C++.

``

```
1  const string TemplateTest::NL = string("\n");                                    C++
2  const string TemplateTest::TEXT_0 = " My Template ";
3  string TemplateTest::generate1(){
4          string sb;
5         return this->_generate1(0,sb);
6  }
7  string TemplateTest::_generate1(int numSpaces, string sb){
8      string spaces = "";
9      string newCode = "";
10     string *realSb = &sb;
11     if(numSpaces > 0) {
12         realSb = &newCode;
13         spaces = _createSpacesString(numSpaces);
14         newCode+=spaces;
15     }
16
17     (*realSb)+= TEXT_0;
18
19     if(numSpaces > 0) {
20         string replacement = NL + spaces;
21     for(string::size_type _szIdx_ = 0; (_szIdx_
22         = newCode.find(NL, _szIdx_))
23             != string::npos;newCode.replace(_szIdx_, NL.length(),
24         replacement),
25             _szIdx_ += replacement.length() - NL.length() + 1);
26             sb+=newCode;
27     }
28     return sb;
29 }
```

Snippet 3-4. A simple C++ generation example of an emitter method

```
1  TemplateTest test;                                                               C++
2  cout << test.generate1();
```

Snippet 3-5. An example of invoking a generated C++ emitter method

A developer is able to add comments before templates and emitter methods (Snippet 3-6- Lines 2, 3, 6, and 7). The same comment content will document in the methods in code generated.

``

```
1  class TemplateTest{                                                    Umple
2      //Template comment1
3      /*Template comment2 */
4      templateLabel1<<! a simple template !>>
5
6      //Emit comment1
7      /*Emit comment2 */
8      emit generate1()(templateLabel1);
9  }
```

Snippet 3-6. Commenting for templates and emitters

3.3 Details of the use of emitter methods

If an emitter method modifier is not specified, this emitter method will be assumed public. During our discussions, for almost all examples, we will not specify the modifier of emitter methods. There are a few places where we will define private emitter methods such as Snippet 3-19, Snippet 3-35, and Snippet 3-42.

An emitter method can be defined as static, such that the generated method will also be static (Snippet 3-7 - Line 3), and accessed statically (Snippet 3-8).

```
1  class TemplateTest{                                                    Umple
2      templateLabel<<! My Template !>>
3      static emit generate1()(templateLabel);
4  }
```

Snippet 3-7. A static emitter method

```
1  System.out.println(TemplateTest.generate1());                          Java
```

Snippet 3-8. An example of invoking a static generated emitter method

3.4 Details of the use of the different type of blocks

In the next sections, we will show how expression, code, and comment blocks are used in the top-level escape characters.

3.4.1 Code blocks

Content enclosed within a code block is written in the syntax of the target language (Snippet 3-9- Line 2). Target language code is emitted as is in the code generated (Snippet 3-10- Line 21). The assumption is that the selected target language is Java. The code block written in Snippet 3-9 can work on both Java and C++. It is the developer's responsibility to write valid code according to the target language of their selection.

``

```
1  class TemplateTest{                                                    Umple
2     templateLabel<<!<<#if(true)#>>This will always be reached !>>
3     emit generate1()(templateLabel);
4  }
```

Snippet 3-9. A simple example of expressions using Umple-TL

We will shorten the next snippets by only showing the content used for text manipulation. We will as well avoid showing redundant line of code such as Lines 1-2 in Snippet 3-2.

```
1  public static final String NL =                                        Java
2     System.getProperty("line.separator");
3     public static final String TEXT_0 = "This will always be reached ";
4
5  public String generate1() {
6     StringBuilder sb = new StringBuilder();
7     return this._generate1(0, sb).toString();
8  }
9
10 public StringBuilder _generate1(Integer numSpaces,
11    StringBuilder sb) {
12    String spaces = "";
13    StringBuilder newCode = new StringBuilder();
14    StringBuilder realSb = sb;
15    if (numSpaces > 0) {
16       realSb = newCode;
17       spaces = _createSpacesString(numSpaces);
18       newCode.append(spaces);
19    }
20
21    if (true)
22       realSb.append(TEXT_0);
23
24    if (numSpaces > 0) {
25       newCode.replace(0, newCode.length(),
26       Pattern.compile(NL).matcher(newCode).replaceAll(NL +
27          spaces));
28       sb.append(newCode);
29    }
30    return sb;
31 }
```

Snippet 3-10. Java generation of simple expressions in Umple-TL

3.5 Expression blocks

The methods invoked or variables found in an expression block either define strings, or are used to translate into strings. They can only be valid accessible methods or attributes.

In Snippet 3-11, there are two variables string1 and string2. Umple creates a getter method for any public attribute in a class; this means that both variables will have their getter methods. These getter methods are designed to

``

encapsulate generated attributes. A generated class can still access its variable directly without using a getter method. In the expression block in Snippet 3-11, we use expression blocks in two occasions. The first one uses the getter method of string1, and the second one directly appends the value of string2 without using its getter method.

```
1  class TemplateTest{                                                    Umple
2      String string1;
3      String string2;
4
5      templateLabel<<! String1=<<=getString1()>>; String2=      <<=string2>>!>>
6      emit generate1()(templateLabel);
7  }
```

Snippet 3-11. Using assign statements in Umple-TL

```
1  public static final String TEXT_0 = "String1=";                       Java
2  public static final String TEXT_1 = "; String2= ";
3  public StringBuilder _generate1(Integer numSpaces,
4      StringBuilder sb) {
5       ....
6      realSb.append(TEXT_0);
7      realSb.append(getString1());
8      realSb.append(TEXT_1);
9      realSb.append(string2);
10     ....
11 }
```

Snippet 3-12. A generation example of used assign statements using Umple-TL

Snippet 3-13 shows an example of using an instance of "TemplateTest" generated from the Snippet 3-11 Umple model.

The constructor of a class in Umple is generated automatically based on the public attributes defined in this class, and the order of definition of those attributes. Thus, the constructor of an instance of TemplateTest1 expects two parameters; the first one assigns the value of string1, and the second one assigns the value of string2.

```
1  System.out.println(new TemplateTest1("Test1",                          Java
2      "Test2").generate1());
```

Snippet 3-13. An example of using a generated template class with some attributes

The output of Snippet 3-13 is shown in Snippet 3-14.

```
1  String1=Test1; String2= Test2                                          Console
```

Snippet 3-14. The output of Snippet 3-13

A developer can use a combination of expression and code blocks, as they require. Snippet 3-15 shows an example of an expression bock that uses a for loop statement to print out the content of a template block 4 times.

``

```
1  class TemplateTest{
2      templateLabel <<!<<#
3          for(int index=0; index<4; index++){
4              #>>Iteration; <<=index>>;<<#
5          }#>>!>>
6
7      emit generate1()(templateLabel);
8  }
```
Umple

Snippet 3-15. An Umple-TL example using expression and assign statements

The generated C++ of Snippet 3-15 is shown in Snippet 3-16. The output expected when using the generated emitter method is shown in Snippet 3-17.

```
1  const string TemplateTest::TEXT_0 = "Iteration ";
2  const string TemplateTest::TEXT_1 = ";";
3
4  string TemplateTest::_generate1(int numSpaces, string sb){
5        ....
6       for(int index=0; index<4; index++){
7          (*realSb)+= TEXT_0;
8          (*realSb)+= index;
9          (*realSb)+= TEXT_1;
10      }
11      ....
12 }
```
C++

Snippet 3-16. A generated template class contains expression and assignment statements

```
1  Iteration 0;Iteration 1;Iteration 2;Iteration 3;
```
Console

Snippet 3-17. The output of Snippet 3-15

3.6 Comment blocks

Snippet 3-18 shows an example of a comment block. An Umple user does not need to worry about how to write comment in the syntax of the target language; this will be generated and handled automatically.

```
1  templateLabel1<<! <</*Comment */>> !>>
```
Umple

Snippet 3-18. A comment example for a template expression

The content of comment blocks is not a part of the output string, but they are used for documentation purposes. Technically, a developer can use code blocks to add comments but in that case, it will be the developer's responsibility to follow the commenting convention of the target language selected.

3.6.1 Exact space blocks

In order to increase usability in Umple-TL, we allow developers to set the number of spaces preceding each line of the final output string.

``

By default, whitespace in templates is output as it appears. Sometimes, however, a developer wants to ensure that specific indentation appears on every line of output, no matter how that output is generated (e.g. from an expression, variable, or plain text). To achieve this, the developer can write an exact space block.

In Snippet 3-19, there are two templates, internalTemplate and templateTest. In Line 2, templateTest internally invokes the generated emitter method internalGenerate () using the exact space escape characters. There are four whitespaces after the escape characters <<$.

```
1  class TemplateTest{                                                    Umple
2      templateTest<<!<<$    internalGenerate()>>,!>>
3      internalTemplate<<!Some content!>>
4
5      emit generate()(templateTest);
6      private emit internalGenerate()(internalTemplate);
7  }
```

Snippet 3-19. An example of exact space handling

The generated method generate(), which uses templateTest, will make sure to set the value of the number of spaces to four (Snippet 3-20- Line 3).

```
1  public String generate() {                                             Java
2      StringBuilder sb = new StringBuilder();
3      return this._generate(4, sb).toString();
4  }
```

Snippet 3-20. A generated emitter method from Snippet 3-19

Snippet 3-21 shows the output of the snippet above. The text content will be the same as specified but it will be preceded by additional four whitespaces.

```
1      Some content                                                       Console
```

Snippet 3-21. The output of Snippet 3-19

Alternatively, developers can directly use the method _generate1 and pass in the number of spaces required. Snippet 3-22 shows an updated version of Snippet 3-19. The output of Snippet 3-19 will be exactly as in Snippet 3-21.

``

```
1  class TemplateTest{                                                    Umple
2      templateTest<<!<<#_internalGenerate(4, sb);#>>!>>
3      internalTemplate<<!Some content!>>
4
5      emit generate()(templateTest);
6      emit internalGenerate()(internalTemplate);
7  }
```

Snippet 3-22. An alternative example of exact space handling

3.7 Dynamic definitions

An emitter method without parentheses is treated as a method with void parameters. For example, the defined emitter methods in Snippet 3-23 will be equal to Snippet 3-1.

```
1  class TemplateTest{                                                    Umple
2      templateLabel<<! My Template !>>
3      emit generate1(templateLabel);
4  }
```

Snippet 3-23. An emitter method with parameters

An emitter method can have multiple parameters and can refer to multiple templates. Snippet 3-24 shows an example of an emitter method referring to three templates.

```
1  class TemplateTest{                                                    Umple
2      templateLabel1<<!Content1!>>
3      templateLabel2<<!Content2!>>
4      templateLabel3<<!Content3!>>
5
6      emit generate1()(templateLabel1, templateLabel2, templateLabel3);
7  }
```

Snippet 3-24. Multiple references for templates in an emitter method

An emitter method generated will encompass the content of all of the referenced templates. The templates will be concatenated in the order that they are passed in to the emitter function. Snippet 3-25 shows the generated C++ of the model shown in Snippet 3-24.

``

```
1  const string TemplateTest::TEXT_0 = "Content1";
2  const string TemplateTest::TEXT_1 = "Content2";
3  const string TemplateTest::TEXT_2 = "Content3";
4  string TemplateTest::_generate1(int numSpaces, string sb, string param1, string param2){
5      ....
6      (*realSb)+= TEXT_0;
7      (*realSb)+= TEXT_1;
8      (*realSb)+= TEXT_2;
9      ....
10 }
```

C++

Snippet 3-25. The generated code of Snippet 3-24

A developer must make sure to pass the appropriate parameters when invoking an emitter method. For instance, the generated C++ code for Snippet 3-26 will cause compilation errors; this is because generate1 defined in Line 2; param1 and param2). The generated code of Snippet 3-26 is shown in Snippet 3-27.

```
1  class TemplateTest{
2      templateLabel<<!Parameter1:<<= param1>>;Parameter2<<=
3          param2>>!>>
4      emit generate1()(templateLabel);
5  }
```

Umple

Snippet 3-26. Using an arbitrary number of variables within a template

```
1  public static final String TEXT_0 = "Parameter1:";
2  public static final String TEXT_1 = ";Parameter2";
3  string TemplateTest::_generate1(int numSpaces, string sb){
4      ....
5      (*realSb)+= TEXT_0;
6      (*realSb)+= param1;
7      (*realSb)+= TEXT_1;
8      (*realSb)+= param2;
9      ....
10 }
```

C++

Snippet 3-27. The generation results of Snippet 3-26

Snippet 3-28 is a corrected version of Snippet 3-26. The generated code of Snippet 3-28 will not cause compilation errors, since the appropriate parameters are passed in, as shown in Snippet 3-29- Line 3.

``

```
1  class TemplateTest{
2      templateLabel <<!Parameter1:<<= param1>>;Parameter2<<=
3          param2>>!>>
4      emit generate1(String param1, String param2)(templateLabel);
5  }
```
Umple

Snippet 3-28. An updated version of Snippet 3-26

```
1  public static final String TEXT_0 = "Parameter1:";
2  public static final String TEXT_1 = ";Parameter2";
3  string TemplateTest::generate1(string param1, string param2){
4      string sb;
5      return this->_generate1(0,sb,param1,param2);
6  }
7
8  string TemplateTest::_generate1(int numSpaces, string sb, string param1, string param2){
9      ....
10     (*realSb)+= TEXT_0;
11     (*realSb)+= param1;
12     (*realSb)+= TEXT_1;
13     (*realSb)+= param2;
14     ....
15 }
```
C++

Snippet 3-29. The generation results of Snippet 3-28

A generated emitter method will contain the same parameters specified in the model. Snippet 3-30 shows an example of invocating the emitter method generated from Snippet 3-29.

```
1  TemplateTest test;
2  cout << test.generate1("parameter1 value", "parameter2 value");
```
Console

Snippet 3-30. An example of using the generated emitter method of Snippet 3-29

The output of Snippet 3-30 is shown in Snippet 3-31.

```
1  Parameter1:parameter1
2  value;Parameter2parameter2 value
```
Console

Snippet 3-31. The output of Snippet 3-30

A developer must make sure to pass the variables required for the templates used in an emitter method. Snippet 3-32 shows an example of an emitter method that uses three templates.

``

```
1  class TemplateTest{                                                    Umple
2      templateLabel1<<!Parameter1:<<= param1>>;Parameter2<<=
3      param2>>!>>
4      templateLabel2<<!
5      !>>
6      templateLabel3<<!;SameParameter1:<<=
7      param1>>SameParameter2<<= param2>>!>>
8
9      emit generate1(String param1, String param2)(templateLabel1,
10     templateLabel2, templateLabel3);
11 }
```

Snippet 3-32. An example of using multiple templates of different parameters

In Snippet 3-32, the first and third templates use the same parameters, while the second template does not use any parameters. Thus, the emitter method needs to pass two parameters named after the shared two parameters identified in the first and third template. Snippet 3-33 shows the expected output when using the emitter method generated from Snippet 3-32.

```
1  Parameter1:value1;Parameter2value2                                     Console
2  SameParameter1:value1;SameParameter2value2
```

Snippet 3-33. The output when using the emitter method generated from Snippet 3-32

Some templates can share some or all of their parameters; a developer will not need to repeat the references to the shared parameters. Table 3-2 shows an example of an assumed number of templates of different parameters; the shared parameters are set in bold. We assume that parameters are all of the string type.

Table 3-2 Templates of multiple parameters

Name	Parameters
template1	param1, param2, param3, and param4
template2	***param1, param2,*** template2Param1
template3	param1, param3
template4	template4Param1
template5	param3, and param4

The expected call for those parameters should be as in Snippet 3-34.

``

```
1  emit generate1(String param1, String param2, String                    Umple
2      param3, String param4, String
3      template2Param1, String template4Param1)
4      (template1,template2,template3,
5       template4, template5);
```

Snippet 3-34. An example of emitter method based on the templates defined in Table 3-2

Defining parameters of an emitter method is done manually as specified above. We decided to make this manual, since automatic assigning of parameters will restrict developers from specifying the order of their emitter method parameters.

3.8 Invocation and template reusability

A developer can add more variations such as using static modifiers or limiting the visibility of some emitters so they can be used only internally in a class. For instance, in Snippet 3-35, the private emitter method internalGenerate is used internally by another template templateLabel.

```
1  class TemplateTest{                                                    Umple
2    internalTemplate<<! Some content  !>>
3    templateLabel<<!<<=internalGenerate()>>!>>
4
5    emit generate()(templateLabel);
6    private emit internalGenerate()(internalTemplate);
7  }
```

Snippet 3-35. An internal invocation of an emitter method

In many situations, developers will need to use the same text content repeatedly. For a good design, a developer can implement a helper class that will contain common templates to be shared. For instance, in Snippet 3-36, HelperTemplate has two static emitter methods used to add carriage returns and tabs. These helper methods are invoked statically in another class, TemplateTest.

``

```
1  class HelperTemplate {                                                    Umple
2      newLineTemplate<<!
3  !>>
4
5      newTabTemplate<<!     !>>
6
7      static emit newLine()(newLineTemplate);
8      static emit newTab()(newTabTemplate);
9  }
10
11 class TemplateTest {
12     templateTest<<!<<=HelperTemplate.newLine()>>
13         <<=HelperTemplate.newTab()>>!>>
14     emit newTab()(templateTest);
15 }
```

Snippet 3-36. Invocation of emitter methods in other classes

An emitter method (Snippet 3-37- Lines 11-12) can refer to templates defined in other classes (Lines 2, 6, and 7), other than its owning class (Line 10).

```
1  class SharedTemplateHelper1 {                                             Umple
2      copyright <<! copyright text !>>
3  }
4
5  class SharedTemplateHelper2 {
6      test1 <<! test 1 !>>
7      end <<! end !>>
8  }
9
10 class GeneratorForLanguage {
11     emit sequence()(SharedTemplateHelper1.copyright,SharedTemplateHelper2.test1,
12         SharedTemplateHelper2.end);
13 }
```

Snippet 3-37. External template references

3.9 Umple-TL and beyond

In the previous sections, we showed how we use Umple-TL to implement the basic features that a developer would expect when using a template generator tool. In this section, we show how Umple-TL can use the powerful features of Umple in a synergistic way when developing templates.

3.9.1 Traits and aspect orientation

Some tools such as JET [35] can help developers easily write template skeletons, which will contain sharable text content across all generated files; a typical use might be to inject a copyright statement in each file. However, the generated files do not necessarily require sharing all text content. In such a case, developers will need to write additional skeletons causing development to be more complicated than necessary. Other commonly-known text

``

emission tools such as Epsilon Generation Language (EGL) [36] and Xtend [37] may require developers to bureaucratically add copyright statements to all template files.

This dilemma mentioned above can be overcome by using the aspect-oriented features Umple provides. Aspect-orientation can be used in conjunction with the trait feature of Umple [10]. Traits allow developers to create external partial definitions that can be used by other classes. In other words, a trait contains attributes and method definitions similarly to classes; when a class uses a trait type, the attributes and methods of this trait are mixed into that class.

Snippet 3-38 shows an example that defines two classes named "Helper" and "Template" in addition to a trait named Extension. Helper has an emitter method used statically to add a copyright statement. The template has two emitter methods, generate1, and generate2; both emitter methods add different template content. As well, Template is a type of the Extension trait. When the Extension trait is used as a part of a class, its definitions will be mixed into that class.

According to Line 7 in Snippet 3-38, Extension looks for all methods starting with the word "generate", using the wildcard symbol "*". This means that for the case of Template, there will be two matches for generate1 and generate2. According to Line 8, for each match, a call to the static helper method, copyrightEmit will be made. This call will be put at the very beginning of the method body of each emitter method in Extension.

Umple

```
1  class Helper {
2      copyright <<//Some copyright>>;
3      static emit copytightEmit()(copyright);
4  }
5
6  trait Extension {
7      before generate* {
8          Helper._copyrightEmit(0, sb);
9      }
10 }
11
12 class Template {
13     isA Extension;
14
15     myContent1 <<! Some content1 !>>
16     myContent2 <<! Some content2 !>>
17
18     emit generate1()(myContent1);
19     emit generate2()(myContent2);
20 }
```

Snippet 3-38. An example encompassing the features of aspect-orientation, traits, and templates

Snippet 3-39 shows a portion of the code added at the beginning of each generated emitter method of Template. This means that a copyright statement will be added before each emitter method.

``

```
1  public StringBuilder _generate1(Integer numSpaces,                    Java
2      StringBuilder sb) {
3      Helper._copyRightEmit(0, sb);
4      ....
5  }
6
7  public StringBuilder _generate2(Integer numSpaces,
8      StringBuilder sb) {
9      Helper._copyRightEmit(0, sb);
10     ....
11 }
```

Snippet 3-39. An example of Java code generated with aspect-orientation, trait, and template features applied

From the example mentioned above, we can see the benefits of using the Umple features of aspect-orientation and traits to alter the behaviour of templates in an easy way. These Umple features maintain the Umple straightforwardness of having a single artifact model, while applying additional behaviour.

3.9.2 Imperative and declarative templates

There are different types of code generators [7], [38]–[41] such as imperative, declarative, and model-based generators.. The purpose of this section is to show that Umple-TL supports all types of generators. A model-based generator relies on modelling features such as classes, state machines, and associations. Since Umple is a model-oriented language, this means this type of generation is supported.

It is up to the user to design their templates imperatively, declaratively or as a mix of both styles. Snippet 3-40 shows an example of an HTML page implemented in with imperative aspects such as a for loop, and declarative aspects such as simple declarations of what the output must look like; this page contains a table consisting of a single column. The number of rows is determined by the attribute, "name".

``

```
1  class HtmlTemplate {
2
3      htmlTemplate<<!
4          <html>
5              <body>
6                  <table>
7                      <<# for  (String name: names) {#>>
8                      <tr>
9                          <td>
10                             <<=name>>
11                         </td>
12                     </tr>
13                     <<#}#>>
14                 </table>
15             </body>
16         </html>
17     !>>
18
19     emit printHTML(List<String> names)(htmlTemplate);
20 }
```

Umple

Snippet 3-40. An example of HTML template generation that blends imperative and declarative styles

Snippet 3-41 shows an invocation example of the Snippet 3-40 model.

```
1  System.out.println(new HtmlTemplate().printHTML
2  (Arrays.asList(new String[]{"Row1", "Row2", "Row3"}));
```

Java

Snippet 3-41. An invocation example of Snippet 3-40

Snippet 3-42 shows an alternative version of Snippet 3-40 done in a declarative manner.

``

```
1  class HtmlNode{                                                    Umple
2      String tag;
3      String content= "";
4
5      0..1--* HtmlNode children;
6
7      nested<<!<<#for(HtmlNode node: children) {
8          #>><<=node.generate()>><<#
9      }#>>!>>
10
11     print<<!<<<=tag>>>
12     <<=content>><<<=nestedPrint()>>
13     </<<=tag>>>!>>
14
15     emit generate()(print);
16     private emit nestedPrint()(nested);
17 }
```

Snippet 3-42. Another example of HTML template generation

Another example of blending declarative and imperative styles is shown in Snippet 3-42. HtmlNode has two attributes, tag and content. The tag attribute refers to a valid html tag such as html or body. The "content" attribute is optional and has an empty value by default; it refers to the text content of an html node. There is an association attribute; "children". Associations are one of the key features that Umple provides. Umple provides all different types and variations of associations. In Snippet 3-42, in Line 5, the type of association used is optional unbound self-reflexive; this means that it refers to an unlimited number of children of the same class, HtmlNode (or its subclasses). In addition, this list of children can be empty, which means that it is fine for an HtmlNode to have an empty list of children.

There are two emitter methods for two templates, "print" and "nested". The "print" template is the main template that is used to print out the content of an HtmlNode instance. The content of an HtmlNode instance includes the nested content of its children in nested ways. The "nested" template is used to take care of looping into the children list of an HtmlNode instance; a child node itself can have a list of children. This will continue recursively until a node does not have any children. Snippet 3-43 shows how the model written in Snippet 3-42 can be used to print out similar content to Snippet 3-40.

``

```
1  HtmlNode html = new HtmlNode("html");                                    Java
2  HtmlNode body = new HtmlNode("body");
3  html.addChild(body);
4
5  HtmlNode table = new HtmlNode("table");
6  body.addChild(table);
7
8  for (String label : Arrays.asList(new String[] {
9              "Row1", "Row2", "Row3" })) {
10         HtmlNode row = new HtmlNode("tr");
11         table.addChild(row);
12
13         HtmlNode tableData = new HtmlNode("td");
14         tableData.setContent(label);
15         row.addChild(tableData);
16 }
17
18 System.out.println(html.generate());
```

Snippet 3-43. An invocation example of Snippet 3-42

Selecting between imperative or declarative approaches is up to the users. A user can decide which approach can fit their needs. Imperative approaches are easy and quick to use but do not provide enough options to flexibly change their template content. On the other hand, declarative approaches require more effort to create and some time to get used to; however, eventually, they can provide many options to manage template content. Both approaches are supported by Umple-TL and can be implemented in a model-based manner. Not only this but also a developer has the freedom to manage their models as they require; there are no restrictions on how they design their models. Features of Umple such as associations, traits, and aspect-orientation provide more options.

3.9.3 UML constructs and generation templates

In the previous sections, we gave an idea about using UML constructs in Umple-TL such as associations. In this section, we will give more examples of using other UML constructs such state machines.

Snippet 3-44 shows an Umple model that displays course information based on the number of the registered students. A student has a name and id, while a course has a name and description. According to Line 4, there is an optional unbound association defined between a course and students. This means that a course can have zero students and at the same time does not have an upper limit of the number of students.

A course has four statuses, opened, registered, withdrawn, and closed. Based on a course status, the displayed template information will be changed. If a course status is open, all information including fees will be displayed; otherwise, it will not be shown. The enrolled student information is displayed regardless a course status. If a course is in registered status, information such as last day to withdraw will be shown.

``

Umple

```
class Course {
    String name;
    String description;
    0..1 -- * Student student;
    cr <<!
    !>>

    courseInfo <<!
    Course: <<=getName()>>
    Description: <<=getDescription()>>
    Number of registered students: <<=numberOfStudent()>>
    <<# switch(getStatus()) {#>><<# case Opened:#>>
    <<# switch(getStatusOpened()) {#>><<# case
          WithoutLateRegistrationFees:#>>
    Last day to register without late registration fees XXX
    <<# case WithLateRegistrationFees :#>>
    Last day to register with late registration fees XXX
    <<# } #>>
    <<# case Registered:#>>
    Last day to withdraw from a course XXX
    <<# } #>>
    !>>

    status{
        Opened {
            register -> Registered;
            close -> Closed;
            WithoutLateRegistrationFees {
                register -> Registered;
                deadLinePassed -> WithLateRegistrationFees;
            }
            WithLateRegistrationFees {
                register -> Registered;
                deadLinePassed -> Closed;
            }
        }

        Registered {
            requestToWithdrow -> Withdrawn;
            LastDayToWithdraw {
                requestToWithdrow -> Withdrawn;
                deadLinePassed -> Closed;
            }
        }
        Withdrawn { }
        Closed {
            open -> Opened;
        }
    }

    emit printCourseInfo()(courseInfo, cr);
}
```

``

```
53
54  class Student{
55      String name;
56      Integer id;
57  }
```

Snippet 3-44. An example of using UML constructs to develop templates

Figure 3-1 shows the state machine defined in Snippet 3-44; mainly the logic of state machine is in the Lines 24-49.

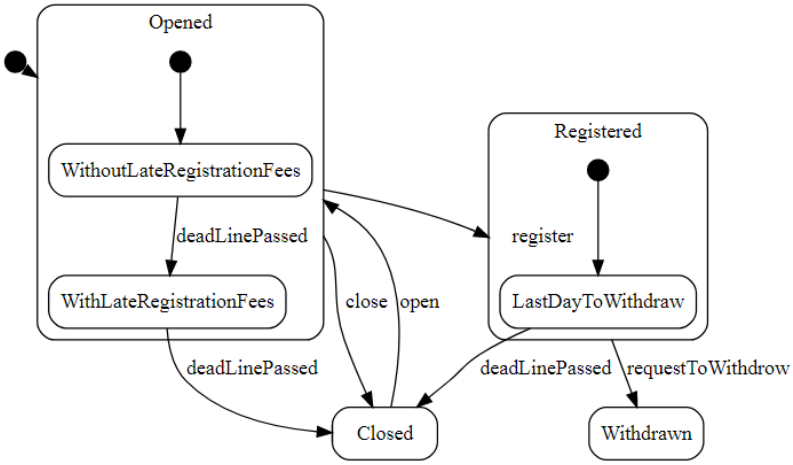


Figure 3-1. The state machine diagram of Snippet 3-44

Snippet 3-45 shows an execution of the generated code of Snippet 3-44. For simplicity, we will show the console output in the same snippet as lines of comments. In Snippet 3-45, we define two students and add both to a course. A course status is "opened" by default. We start by printing out the course information; this means that information will be printed with an assumption that the course status is open. The output in that case is shown in Lines 9-14 in Snippet 3-45. When a course is in open status, all information is displayed. In Line 16 in Snippet 3-45, we change the status of the course to be closed. The output in that case in Lines 18-20 in Snippet 3-45, will only show the basic information about the course and its students.

We change the status back to open (Snippet 3-45 - Line 22); this means that the lines to be printed will be as in Lines 24-29, which are identical to the Lines 10-14. Finally, In Line 31, we change the status to Registered; as compared to the Closed state, the Registered state additionally shows the last day to withdraw (Lines 33-36).

```
1  Student student1 = new Student("Name1", 1234);                    Java
2  Student student2 = new Student("Name2", 432);
3
4  Course course = new Course("Course1", "This is a course");
5  course.addStudent(student1);
6  course.addStudent(student2);
7
8  System.out.println(course.printCourseInfo());
```

``

```
9  //Course: Course1
10 //Description: This is a course
11 //Number of registered students: 2
12 //Last day to register without late registration fees XXX
13 //Last day to register with late registration fees XXX
14 //Last day to withdraw from a course XXX
15
16 course.close();
17 System.out.println(course.printCourseInfo());
18 //Course: Course1
19 //Description: This is a course
20 //Number of registered students: 2
21
22 course.open();
23 System.out.println(course.printCourseInfo());
24 //Course: Course1
25 //Description: This is a course
26 //Number of registered students: 2
27 //Last day to register without late registration fees XXX
28 //Last day to register with late registration fees XXX
29 //Last day to withdraw from a course XXX
30
31 course.register();
32 System.out.println(course.printCourseInfo());
33 //Course: Course1
34 //Description: This is a course
35 //Number of registered students: 2
36 //Last day to withdraw from a course XXX
```

Snippet 3-45. An invocation example of Snippet 3-44

3.10 Challenges and their solutions

Designing a text emission tool gives rise to several challenges. Some of the most important are listed in this section, based on what we faced during the development of Umple-TL. These can also be seen as requirements for an ideal text emission tool.

``

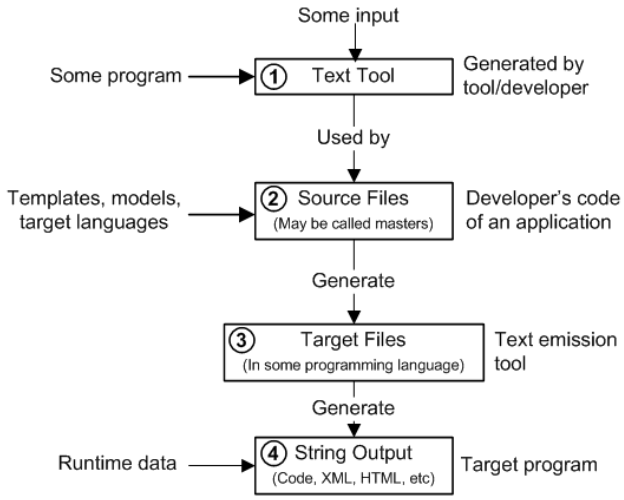


Figure 3-2. The process of text emission

The first group of challenges is requirements for the source language that the developer uses to describe the sort of output to be emitted (Box 1 in Figure 3-2).

- **Constraints and emission flow:** A text emission tool must provide a way to help a developer specify constraints and other forms of execution control in their source files; these will determine how or when target strings will be generated. For example, in C++, the keyword "friend" is reserved; a tool should give a way for developers to set constraints to prevent having this keyword appear improperly in the generated target C++ files. Later in the chapter when discussing this requirement, we will show that Umple-TL has an ability to specify arbitrary control constructs that can serve as constraints; Umple also has a built-in OCL-like constraint mechanism that injects preconditions and postconditions into any method. Together these can be used to ensure detect issues that may result in malformed target files or runtime string output'
- **Transformation rules:** A tool should ideally support rule-based transformation across different schemas. Based on rules specified in the source language, it should be possible to alter the string outputs without the need for a lot of rework to the source. For example, it should be easy in the source to direct the target to switch between generating XML and JSON output.

The second group of challenges relate to both the source and target language.

- **Representation consistency:** The source and generated target files must have a consistent hierarchical representation, which relies on aspects such as structural complexity and generation dynamicity (Section 3.11).
- **Target code efficiency:** A generated target file is expected to be efficient, i.e. it must avoid issues such as unnecessary loops or redundant string manipulations.

``

- **Target code readability:** The generated target code must be properly represented using appropriate spacing, indentation, variable naming and so on. Altering the generated output is something that must be avoided for the same reasons programmers do not modify compiler output. However, for reasons such as debugging and certification, it may be necessary to read the target code, so it must be readable. Also, not all generated output will be 'code'.
- **File structure management:** At both source and target levels, there may be many files of different types; the file structure used, even in the same tool, can differ between the source and target files. For example, a source file can have an extension tailored for a text emission tool, while the target file can have another extension such as Java class or even two extensions as would be the case in C++.
- **Reusability:** When required, a source or generated target file should be reusable by other source or target files. The challenges in that case are mostly related to how the generated target files can be linked together. For example, while a user is developing a source file and they want to reuse another source file, they should not need to think about how this reusability will be enforced at the level of the target files. Such enforcement must be taken care of by the text emission tool.
- **Pattern support:** There is a gap between the source and target files. The main challenge is related to how to enforce the same level of abstraction or transfer patterns between the levels. For example, a user while developing source files can follow patterns such as facade or builder; the target files must follow the same pattern.
- **Comment support:** A tool must provide a way to add comments to the source file, and have them transferred to the target file. In particular, the tool might need to generate comments in the target file to allow traceability back to the source file. Comments in the source files are valuable for maintainability, reusability and all other well-known reasons for commenting source. Comments in the target files are helpful for debugging or investigation purposes.
- **Debugging support:** Debugging ought to be facilitated at the level of source files. When issues occur in generation of target files or runtime generation, the developer ought to be provided with information pointing back to the relevant points in the source. The best approach is to allow full debugging or tracing at the source level so the developer never has to see the target files. But if this is not fully supported, then when running a debugger on the target files, the developer should be easily pointed back to the source files.
- **Additional flexibility aspects:** Sophisticated tools provide several options that help users to handle the content to be emitted; examples include allowing multiple text emission methods and unlimited size of target code and runtime strings. Reusability, which we referred to above, is one aspect of flexibility. Flexibility can also be increased if a tool can incorporate target language expressions; this will be explained later in this section. Some tools may put many restrictions on how text content is emitted. Other tools may have limited solutions, such as the number of emitting methods; this causes the development time to be increased in order to cope with such limitations.

``

The third group of challenges relates to the string output generated by the target code (i.e. the text generated by the generated generator).

- **Consistency of text formatting:** Generation output must be consistent. In this context, we are referring to consistent output regardless of the inputs or applied constraints used. For example, whitespace such as for indentation at the start of lines ought to appear consistently across all output (something not available in many tools). A text emission tool hence ought to provide a way for a user to define whitespace formatting.
- **Content protection:** this refers to preventing end users of the generated systems (Box 4 in Figure 3-2), from accessing the content of text to be emitted. Failing to protect emitable text content can pose major security or privacy issues.

The fourth group of challenges relates to modelling support.

- **Modelling support**: Use of abstract models to represent the source and target languages as well as data in the runtime system can facilitate text emission. Basing emission on models is one of the code generation approaches.
- **Input model restrictions:** A model-based tool usually enforces certain types of models, metamodels, or schemas. This will require tool users to work with the model types that this tool supports. For example, several tools require the use of the Eclipse Modelling Framework (EMF). In addition, obviously, any limitations that may exist in the supported models will also appear in the text emission tool. Some tools such as Epsilon Generation Language (EGL), can overcome such an issue by allowing users to select different types of models [42].

Additional challenges can appear based on the tool used; this is what the fifth group focuses on.

- **Workflow complexity:** Many tools give a large number of options to the developers to handle code generation but unfortunately, this may cause workflow complexity to be increased. For instance, a user may be required to use different editors and wizards, switch back and force between different types of files, and propagate updates across different places.
- **IDE dependencies:** This is a part of the workflow complexity mentioned above. Some tools require developers to use a specific IDE. This poses problems in several contexts: the user may happen to have selected a different IDE, or needs to run the tool outside an IDE (e.g. in a scripting language), Or the user may want to use the tool in a very simple manner, but would be forced to set up the tool in the IDE to get any work done. Many tools only run in Eclipse; an ideal tool should easily run both inside and outside Eclipse.
- **Third-party library independence:** This means that a tool only requires integration in the development environment. The tool should not have to be integrated into a released product (Box 4 in Figure 3-2). For

``

example when using Velocity [43] the Velocity Engine has to be incorporated into the built final product, which can result in problems as Velocity changes over time, and also results in increased system size.

- **Target language restrictions:** A tool may require developers to be familiar with specific languages such as Java. This can be considered a part of workflow complexity, since users will need to learn how to use the language imposed by a tool. An ideal tool would be able to work with several different programming languages.
- **Structural complexity**: This is also a part of workflow complexity. The structural complexity of some sophisticated tools can be high due to requiring users to deal with several types of files; this can be the case even for very simple text output. For instance, a developer may be required first to write a metamodel file. Second, the developer may need to write a model. Third, they might need to write a template file that uses the model in order to specify text content to be emitted. Finally, the developer may need to write configuration and launch files in order to produce results.
- **Syntax complexity:** Typically, a tool has its own script or language. The syntax complexity of a script or language of a tool obviously causes the complexity of a tool to be increased in general. Syntax complexity may increase as well when users are asked to be familiar with specific metamodels or schemas.
- **Target language expressions:** The script or language of tool is not necessarily expected to be as sophisticated as commonly used target programming languages such as Java and C++; otherwise, this will appear as reinventing the wheel. At least the basic features of a target language must be supported such as variable declaration. While using a tool to emit text content, a user may need to use some target language expressions in order to handle the content dynamically. However, relying on a tool's language or script to handle expressions may have limitations. For instance, a tool may support "for" loops but on the other hand, it may not support "while" loops. As a solution, it could be a good idea if a tool allows users to write expressions based on the target language. For instance, if a target language is Java, then a user should be able to write some expressions in Java. It is important to mention that users will still need to use the language provided by the tool.

3.11 A comparison of template development tools

In this section, we compare Umple-TL against other tools commonly used for text emission (Table 3-3). The tools include Java Emitter Templates (JET), Apache Velocity [43], Acceleo [44], Epsilon Generation Language (EGL) [42] [36], Xpand [45], and Xtend [37]. Our selection depends on whether a tool satisfies most of the challenges and requirements listed in the previous section. Not all of the challenges and requirements are shown in Table 3-3. For instance, we did not include Pattern support, and instead focused on generation dynamicity and modelling support. When a tool supports modelling with high generation dynamicity, this may imply that the generation gap patterns will be decreased.

``

Reusability is an open-ended criterion that may not be simple to specify for a tool. Instead, we focus on other criteria such as generation dynamicity, transformation rules, input model restrictions, syntax complexity, structural complexity, and target language expressions. Meeting these criteria mean that the reusability of a tool will be increased.

Comments support is present in all tools; hence, we did not add it to our comparison.

For *file structure management*, we found that this not directly related to a tool itself, but mostly related to the type of text a developer intends to emit. For instance, generating C++ code will require generation of both header and body files. In such a context, our focus will be on that a tool should have lower workflow and structural complexity.

For *target code readability*, we found that other criteria such as formatting complexity and debugging support are enough for our comparison. We did not include *target code efficiency* in or comparison; assessing this aspect can be a completely different topic of discussion.

In terms of *representation consistency*, other aspects such as less structural complexity, high generation dynamicity, and the presence of modelling support could help users to have a consistent representation.

The criterion of the *ability to specify constraints,* is not included as constraints can directly be defined using the tool or target language expressions. In addition, rule-based transformation gives more options to handle constraints.

Consistency of text formatting is maintained when formatting complexity is decreased. Thus, we found that it will be enough to refer to formatting complexity in our comparison.

In Table 3-3, for the criteria, workflow complexity, structural complexity, syntax complexity, and formatting complexity, the rating can be low, medium, or high. A tool is assumed better if it has a low complexity. The same rating scale is used for generation dynamicity. A tool is presumably better if it has high generation dynamicity

For debugging support, this can be applied at the level of source files, runtime, generation, or a combination of them. These criteria are based on the debugging capabilities we found in the tools we studied in the previous section. A tool can support many of those types of debugging, while other tools may not provide any type of debugging.

For other criteria, they are all yes/no answers to determine whether a feature is supported or whether it follows certain restrictions. The desirable features of a tool are set in bold in Table 3-3.

Commonly used programming languages such as PHP and Go can be directly used for text emission. PHP is included in our comparison, since it is a good example on how text emission can be directly handled using a programming language. Technically, when using a tool such as JET, we can say that we use Java for text emission;

``

however, when we refer to PHP in our comparison, we mean that PHP can be used directly for text emission without additional tools or extensions. Any programming language can still be directly used for text emission without additional extensions; however, this will require additional development effort and unfortunately mostly it will be unnecessary or replica of the effort done by other text emission tools.

Table 3-3. A comparison of template development tools
Bold font means better.

	JET	Velocity	Acceleo	Xpand	Xtend	Epsilon	PHP	Umple-TL
Workflow complexity	**Low**	**Low**	Medium	High	**Low**	Medium	Medium	**Low**
IDE dependencies	Eclipse	**None**	Eclipse	Eclipse	Eclipse	Eclipse	**None**	**None**
Third-party library dependencies	Yes	**No**	Yes	Yes	Yes	Yes	**None**	**No**
Target language expressions	**Yes**	No	No	No	No	No	No	**Yes**
Input model restrictions	**No**	**No**	Yes	Yes	**No**	**No**	**No**	**No**
Modelling support	No	No	**Yes**	**Yes**	No	**Yes**	No	**Yes**
Transformation rules	No	No	**Yes**	**Yes**	**Yes**	**Yes**	No	**Yes**
Structural complexity	Medium	**Low**	Medium	High	**Low**	Medium	**Low**	**Low**
Generation dynamicity	Low	Medium	Medium	Medium	**High**	**High**	**High**	**High**
Syntax complexity	**Low**	**Low**	Medium	High	**Low**	Medium	**Low**	**Low**
Debugging support	Runtime	None	**All**	Runtime	Runtime	**All**	Runtime	Runtime
Content protection	**Yes**	No	**Yes**	**Yes**	**Yes**	**Yes**	No	**Yes**
Formatting complexity	Medium	Medium	Medium	Medium	Medium	**Low**	Medium	**Low**
Generator language restrictions	Yes	Yes	Yes	Yes	Yes	**No**	**No**	**No**

3.11.1 Discussion of other tools

In this section, we discuss the results of each table entry except for Umple-TL. The discussion of Umple-TL will come in the next section.

In terms of *workflow complexity*, JET, Velocity, and Xtend are easy to use. Xpand has the highest complexity due to the effort spent to configure its workflow. The workflow of Acceleo and Epsilon is less complex than Xpand. We marked the workflow complexity of Epsilon as medium not as low, since a user is required to be familiar with model-to-text transformation and the Epsilon Model Connectivity (EMC) layer in order to use their metamodels. In addition, Epsilon requires implementing configuration or launch files in order to get an Epsilon file running.

We marked the workflow complexity of PHP as medium not as low, since PHP requires developers to be familiar with additional paradigms such as the client/server architecture. It is important to mention that assessing the

``

workflow complexity of text emission of a programming language is not straightforward. For instance, in C++, developers are not required to be familiar with the client/server architecture; however, C++ requires a lot of effort for compilation and configuration.

All of the entries mentioned have *IDE dependencies* except for Velocity and PHP. However, no tool has *third-party library dependencies* except for Velocity. It is well-known that in PHP, there are no restrictions to specific IDEs or libraries; developers have the liberty to choose development environment as they find convenient. This is the case for all the most widely-used programming languages.

JET is the only tool that controls emission using *target language expressions* (Java in the case of Jet) as opposed to a custom language. As mentioned, relying on custom expressions in a tool can cause limitations as a tool will not be necessary able to handle as comprehensive a set of expressions as compared to a sophisticated language such as Java. The PHP entry is marked as "no", since the concept of target languages does not exist at all when using a programming language directly for text emission.

In terms of *input model restrictions*, only Acceleo and Xpand enforce restrictions to use Ecore to develop models. However, in terms of *modelling support*, they are the tools that directly provide modelling support. Other tools such as Xtend can use additional libraries such Xtext to support modelling. Epsilon overcomes this tradeoff, since a user has the liberty to decide the type of models to be used using the Epsilon Model Connectivity (EMC) layer. In Umple, we managed to overcome this tradeoff as well, since Umple can be used as an object-oriented programming language as well as being a modelling language.

We did not mark for PHP that it supports modelling, since modelling features are not a part of its built-in features or constructs. Although, in fact, PHP or any sophisticated programming language can support some aspects of modelling such as classes and inheritance, we do not consider their level of abstraction to be sufficient to be called modelling languages.

Only model-based tools such as Acceleo, Xpand, and Epsilon can provide a support for t*ransformation rules*. It is important to mention that Xtend can be used with Xtext in order to support model-related features. Transformation rules can be supported in PHP but this will require additional effort in a similar manner to modelling support. Thus, we marked the transformation rules in PHP as "no".

In JET, a template file must use a skeleton file; this adds an additional complexity to its text emission structure, and thus, we marked JET's *structural complexity* as medium. On the other hand, in Xpand, we mentioned that a template file depends on the developed metamodels and models; both add more complexity to its text emission structure and thus we decided to mark their complexity as high.

``

In an Acceleo template, a developer needs to follow the structure specified in their model; this causes the complexity to be increased and become medium. We marked the structural complexity of Epsilon as medium, since users work with different types of files such as Epsilon files, metamodels, launch files, and Epsilon transformations.

Epsilon provides a complete solution for code generation and contains several features represented in different types and syntax. Similarly to any language, a user will need a fair amount of time in order to get used to most features of a language such as Epsilon. [36]. However, we did not mark the structure complexity of Epsilon as high as a user does not necessarily need to learn about all Epsilon features in order to get their text emission working.

Xtend and Velocity directly use a single template file; thus, we marked their structural complexity as low. We also marked the structural complexity of PHP as low, since the main artifact the developers work with is "*.PHP" files. PHP developers still work with other artifacts such as HTML, CSS, and JavaScript files. The structural complexity varies among different programming languages if they are going to be used for text emission directly. For instance, in C and C++, developers must work with two artifact types; headers and bodies, while Java developers only need to write Java files. Typically, in any development process, additional configuration and launch files are required; this is a part of the IDE selected for the development, which we referred to earlier in this section.

In terms of *generation dynamicity*, we found that of the tools that existed prior to this research, Xtend and Epsilon provide the most dynamic solutions. A developer is able to write as many emitter methods as they want with a variable number of parameters; there are no restrictions on how they should implement their text emission content. Velocity, Epsilon, Acceleo, and Xpand provide enough solutions to handle dynamicity but in not in as flexible a way as Xtend and Epsilon; this is why we marked them as medium. On the other hand, we marked JET is low as it does not provide a direct way to have dynamic parameters. We marked the generation dynamicity of PHP as high; this is the case for any sophisticated programming language if it were to be included in our comparison.

The syntax complexity of all pre-existing tools is low except for Xpand and Acceleo. In Xpand, a developer will need to move among different parts of the template file in order to understand what this template does; thus, we marked the syntax complexity of Xpand as high. On the other hand, Acceleo is not as complex as Xpand in terms of syntax complexity; however, it assumes that a developer is familiar with Ecore constructs and methods; this is why we marked its complexity as medium.

We marked the syntax complexity of Epsilon as medium instead of low, since a user needs to use the Epsilon Model Connectivity (EMC) layer to control the metamodel type used.

We marked the syntax complexity of PHP as low, since PHP is a commonly used programming language. This statement may be challenged by others when comparing with different programming languages; this depends on what programming language a developer prefers,

``

We found that only Acceleo and Epsilon support all types of *debugging*. For example, Epsilon provides a traceability feature to debug or audit a transformation process [36].

On the other hand, in JET, debugging is limited but there is at least a workaround to debug the generated classes at runtime; debugging the generated classes at runtime can be sufficient. If there is any syntax in a JET file, there will be direct compiler errors in its generated file.

The same workaround can be applied for Xpand; additionally, in Xpand, the tool support has several features for validation and error highlighting such as the Check language. In Xtend, the debugging is at the runtime level, since the Xtend files are automatically generated. On the other hand, Velocity does not have any type of debugging support even at runtime since the generated files are not produced. Alternatively, a developer can use log statements.

Debugging in PHP depends on the IDE used and the user-written code. Any sophisticated programming language is usually supported by many tools or IDEs, which usually offer debugging as one of its features. Thus, we marked PHP that it supports runtime debugging.

All entries provide support for *content protection* except for Velocity and PHP. In Velocity, template files are required to be a part of the release build in order to have in-memory generation of its template. Such an issue can cause security issues. Content protection in PHP is one of the major discussions that always arise when developing an application or library. When developing a PHP library, it may not be straightforward to protect template content; thus, we did not mark that PHP has content protection. This is not necessarily the case for other programming languages. For instance, in C++ and Java, developers have a lot of options to hide or compile their solutions as binary files.

For *formatting complexity*, we refer to aspects such as spacing and indentation. When a tool enables developers to write their code directly without being wrapped in a method, the spacing and indentation will not be a problem; in other words, what a developer sees in their source files will be what they get in the generation. Those tools include Velocity, Acceleo, and Xpand. Such tools still do not provide APIs to enable a user to shift or control the spacing or indentation of their text content directly. It is important to mention that users can still find a way to handle format. For example, in Xpand, a user can define a method to add some spaces or tabs; however, this does not come as a part of the language semantics, which is why we marked its formatting complexity as medium not low.

We marked the formatting complexity of Epsilon as low, since Epsilon provides a way to set a formatter object. Using a formatter object, a user can make a sequence of calls to format a generated text as they require [36]. However, this formatting feature takes place after the text is already generated, which reduces the developer's control.

``

On the other hand, in both JET and Xtend, the output descriptors are written in an emitter function; a developer will always need to shift their text content and make them aligned with their methods. This will be annoying to the developer as they will need to do that for all methods. As well, like the abovementioned tools, there is no way to control the number of spaces used; thus, a developer will need to do this manually at all places in code. We marked the formatting complexity of PHP as medium, since it is the developers' responsibility to handle text format.

3.11.2 Umple discussion

In terms of *workflow complexity*, a developer will only need to work with a single artifact, an Umple model. An Umple model can encompass the required emission information such as metamodels and models. The model could be in a single file, or several, at the discretion of the developer.

There are no *IDE dependencies* required whatsoever when using Umple-TL. A developer has the liberty to use their IDE of interest; they can even directly use UmpleOnline (try.umple.org) editor and then download the generated files. There are no *third-party library dependencies* for both development and release builds; in fact, Umple is written in itself as an Umple model to be generated without additional tools involved. For the tools that we studied, even those that provide third-party independent solutions, they do not provide independency at the development level.

In terms of *target language expressions*, Umple enables users to write their expressions in the syntax of the target language. A user even has the freedom to switch among different target languages such as C++ and Java; other tools that we listed restrict a developer to use a specific target language.

There are no *input model restrictions* in Umple. A developer can directly write their Umple model and use the appropriate target language that fits their needs; they will not find themselves in a situation where they have to use a specific target language or model schema.

Obviously, there is *modelling support* in Umple, since it is a model-oriented language. Umple incorporates UML constructs into template development. In other words, in Umple, a model-oriented approach is applied on the models being implemented, not at the tool level only. We noticed that sometimes, enforcing a model-oriented paradigm can be turned into a burden of complexity as we showed in Xpand and Acceleo.

Being a model-oriented language means that *transformation rules* can be applied on the written Umple models. In Umple, there are many transformation options such as Java, C++, PHP, Ecore, and XMI. Although a developer has to choose a target language for certain expressions, basic for-loops, while-loops and if-then expressions can be specified the same in Java and C++, allowing a set of templates to be used across target languages.

``

Umple has low *structural complexity*, since it provides a configuration-free solution. As previously mentioned, a model can encompass all information including the configuration information. The process is simple as a user writes a model and then generates output.

Umple-TL provides many solutions to handle generation dynamicity. Developers are not tied to a specific emitter method or number of emitter methods; they are able to define their emitter methods, as they require. The number of parameters to be specified in an emitter method is dynamic; the developer is able to control and define the parameters. Making changes to the parameter signature of an emitter method is straightforward, since all information is represented in a single model artifact. A user is able to invoke and use other template classes within another template class. As well, a template class can be designed to serve as helpers or static classes.

Umple follows the C-family syntax conventions, which is desirable for many developers, since commonly-used languages such C, C++, and Java follow this style. As a result, Umple has a low *syntax complexity*. A developer will not need to worry about learning a new script representation in order to start using Umple-TL.

In terms of *debugging support*, Umple still does *not* provide tool support for debugging at the model level. Alternatively, a user is able to debug the generated files. Line numbers from the source Umple are injected as comments into the generated code, allowing ease of traceability. As well, there is a tracing support in Umple, which enables the developers to write trace statements in their Umple models as a way to debug generated files at runtime.

Content protection is provided by Umple, since developers will only need to use the generated files for their product releases.

In terms of *formatting complexity*, we showed how Umple-TL provides APIs to handle space and string manipulation. The solutions provided help optimize performance; we indicated for example that the string buffer instance is shared across different nested template calls.

Umple-TL follows a similar approach as JET and Xpand in terms of having inline definitions for templates. For example, a user can define a template that will apply tabs or carriage returns. In addition, a feature such as aspect orientation offers flexible options when writing redundant content such as tabs and copyrights.

All existing tools have *generator language restrictions* to Java for all of them except PHP. Being inspired by UML, in Umple-TL, we rely on the multi-language feature of Umple to enable developers to write their models in their target language of interest such as Java, C++, and PHP. We showed template examples in two different target languages, C++ and Java.

Umple formerly had used to use JET before the introduction of Umple-TL[46]. A tool was developed to automatically transform all templates written in JET into Umple-TL, and helped us to get rid of any dependency on

``

third-party transformation tools. Since then, template development in Umple has been based on Umple-TL, which has proven to be powerful and reliable for large projects.

``

Chapter 4 Active Objects

In this chapter, we present the new features related to concurrency we have introduced to Umple. The new features follow the active object pattern. Concurrency features are crucial for the underlying architecture of composite structure, which is the core of this research. The focus is on supporting concurrency for real-time applications, and showing how it can be implemented in Umple using a small number of comprehensive constructs. Finally, we show a comparison between Umple and other specifications that focus on time as a key factor.

4.1 Introduction

System performance can be significantly improved by adding more processors or cores. However, software designs must be able to utilize the extra cores by maximizing concurrent execution. Unfortunately, existing development techniques and languages tend to be limited in how they enable developers to handle concurrency easily.

In a concurrent environment, multiple actors interact and communicate. By 'interactions' we are referring to patterns of communication at the abstract level. The patterns of communication at a more concrete level become synchronous and asynchronous method invocation.

Umple currently supports concurrency in three forms *active*, *do activity*, and *Queued State Machine (QSM)* [47]. Active and do activity spawn a thread, other than their owning class's thread, to execute some action code. *Active* is activated from its class constructor, while do activity is activated from a state machine state.

On the other hand, QSM [47] employs a serial-execution message-passing approach, which uses an event queue to decouple event invocation from execution. Therefore, QSM stores each event invocation as a message containing state, name, and parameters. Then, the QSM dispatches messages in FIFO order, except for a special case, *pooled*, in which QSM delays processing events that cannot be handled in the current state. QSM creates one thread in the owning class to process events and enable thread-safe event execution.

Prior our research, concurrency support in Umple mainly focused on state machines, and had limited support at the model level, such as direct asynchronous method invocation. There was no solid inherent structure to handle complex concurrent programming, and to support different time constructs such as delay, polling, and timeout. For example, when an event in QSM takes longer than anticipated, it halts the entire execution of that QSM. In particular, there are no constructs to handle timeout in such a case.

We extend Umple to support concurrency at the model level, and to enforce 1) data isolation, 2) thread communication through asynchronous messages, 3) processing each task one at a time to satisfy run-to-completion semantics, eliminating concurrency issues, and 4) enabling a generic approach to cover *operations*, *state machine events*, and *actions*.

``

We refer to our extension as the *active* features, as they are derived from the active object pattern [18], which enforces concurrency best practices. We can as well refer to an *active* Umple class as an active object instance, whose behaviour is defined using a state machine. For example, with the new *active* features, QSM can be supported by expressing events as *active* operations in the Umple model level directly, which will not require any changes to the original state machine code.

Our extension distinguishes between *active* and *passive* [48] in terms of their capabilities to execute in their own thread, and to initiate a control activity, such that each method is executed internally and sequentially. An *Active* Umple class means that the class must have at least one active method.

We will discuss how we extend the active object pattern [18] to support additional time constructs and priority-based concurrent execution. We will highlight the possible levels, at which *active* can be applied, such as methods and action code. After that, we will discuss the basic flow of active objects, which consists of several actors such as proxies, messages, and schedulers.

We also incorporate action code with concurrent constructs such as *future* [49] and *promise* [50]. *Future* is a variable that holds the asynchronous response of an active method, while *promise* is an asynchronous provider. Therefore, a *future* result is beneficial to the user while writing their action code, as it enables them to inquire about the status of an asynchronous method such as whether it is ready or not, and to apply various time constructs such as delay.

Future and *promise* are supported in many target languages such as Java and C++ (since C++11). However, we target C++03 code generation, which requires additional effort to support these features. Such an old version is required in contexts such as embedded devices that still do not have the features added in C++11.

We designed *active* to 1) resolve data into a *future* result, 2) provide functions that can notify, wait, inquire status, and handle exceptions, and 3) provide different time constructs within method invocation and a *call/resolve/then* pattern.

The code snippets we are showing in this chapter assume that the target language is C++, meaning that the extra code we write in the Umple models is in C++ as well.

Our contributions related to concurrency can be summarized as follows:

- The implementation of the features related to concurrency and *composite structure* in Umple. This includes code generation for real-time applications.
- Support C++ code generation for real-time systems.

``

- Introducing an active object pattern that extends the one introduced by Lavender [18], [51]. Our pattern aims to enhance communication among active objects as in the points below.
- Simplifying active features at the action code level such as future [49], and other time constructs, using simple Umple keywords.
- Easy handling of complex time constraints related to asynchronous and synchronous method invocation, using simple Umple keywords.
- Introducing a new pattern, call/resolve/then to ease invocation strategies, callback, data resolution, and error handling.

4.2 Active features in Umple

We implement active features at three levels, the Umple constructs, concurrent model, and target language. We designed our implementation to overcome the major challenges of concurrent models we researched (Section 4.2.1).

An Umple construct is a semantic and syntactic extension of Umple that refers to behaviour, keywords, and how they can be used. We discuss keyword usage as scenarios with excerpts from the code generated in C++.

A concurrent model refers to the model used to handle concurrency among operations in terms of communication and synchronization. Concurrent models can be contrasted based on their behaviour patterns and mechanisms applied for inter-process communication such as *shared-state* or *message passing*. Examples of concurrent models include Actor model (AM) [52] and Active Object (AO). The differences between both models are indicated in [53].

We choose the Active Object model as it 1) decouples method execution from invocation, 2) enables invocation using a function call interface or delegate [54], 3) assures data isolation between the caller and receiver, and 4) employs various message-passing mechanisms.

The classic active object pattern typically involves the following elements, 1) interface: defines accessible methods; commonly known as active methods or public interface methods, 2) client: implements the interface, 3) proxy: another simple object internal to the client that the client invokes to access other methods of the system in a thread-safe manner, 4) request: invoked by a client to a proxy, 5) scheduler: organizes how requests execute, 6) response: has different forms such as callbacks, variables, and future objects. Typically, the active object pattern employs a simple FIFO queue with serial execution of the pending requests in the queue. This means that complicated scenarios such as prioritized queues or quasi-concurrency models are not considered. We will show in this chapter how we managed to overcome such limitations. Our work hence extends the classic active object pattern in that a) there is a more sophisticated internal scheduler mechanism; b) there is a more sophisticated form of internal concurrency; c) there is a prioritized FIFO queue; and d) time constraints are allowed to permit deferred or periodic calls.

``

The target language needs to provide a basic support for concurrency or a multithreading environment. Concurrency is supported in Java and C++. However, C++ has additional challenges related to the existence of different compiler vendors, C++ standards, and thread APIs for operating systems and embedded devices. In this thesis, we mainly focus on C++ code generation and real-time systems.

4.2.1 Challenges

In this section, we list the common limitations and challenges expected during the implementation of concurrent programming based on active objects.

- **Cross-platform and embedded devices support:** The generated cross-platform C++ code support is based on main dimensions which are operating system type (Macintosh, Windows and Linux), complier vendor (Microsoft VC++ and GCC), C++ standard (C++03), and multi-threading support.
- **Message-passing protocol:** We need to choose the internal concurrency of processing and handling messages. A message delegates a function pointer, function call interface, or signal event [55]. It can further be extended to handle priority, and other constraints at the internal state of a concurrent model.
- **Priority-based message scheduling:** The most common concurrency implementations use the common message processing, First-In-First-Out (FIFO), which directly processes based on the receiving order. However, this lacks a constraint-based and prioritized ordering mechanism, as well as exception and recovery handling. We extend active objects, such that logical and time constraints can be handled at the scheduler level, in which messages are prioritized, and safely interruptible using constructs such as timeout. We also extend the capabilities of active objects to spawn another thread in order to handle time constraints that require repetitions such as periodic check.
- **Message constraints:** These can be:
  - **Specialized constructs:** A specialized construct refers to a way to handle or execute certain operation after a period based on particular rules or conditions, and may involve prioritization. These include Umple keywords to specify period, and message prioritization. These constructs can be specified at the action code level, which require wrapping the result into a read-only holder, which is *future*. Without such constructs, a developer will not find flexible solutions to organize communication and scheduling processes.

    For example, an active object response for given method invocation may not be ready yet, as it being placed on the pending list. Therefore, using a construct such as *future* enables the user to enquire about the status of a response, which can be useful in such a situation; if the future construct is not a part of the language, the developer may try to either follow a different coding technique or implement their APIs to do similar functions.

``

- **Guard constructs:** A logical construct refers to conditions a developer specify to prevent method execution, or to keep the method in the notification list until those conditions are satisfied. We handle the guard conditions on two cases, *operation* and *invocation*. Guard conditions in the *operation* case request the scheduler to prevent a method from being invoked at all if conditions are not met. On the other hand, in *method invocation*, the scheduler invokes the method, but a guard conditions defined at the beginning of the method body serves as a precondition, which if false, causes the method to return without executing the body.

- **Data sharing:** A concurrent model applies concurrency control mechanism, such as mutual exclusion, to prevent race conditions, and to protect shared resources from concurrent method invocations.

- **Performance:** a common way to improve concurrent object dispatching is by using a thread pool. A thread pool mechanism can improve performance by managing the threads used. In a thread pool, when a connection is open, a number of threads keeps accumulating based on some conditions such as mutual exclusion and atomic access. There are no restrictions on a thread pool size; it is up to a developer to set their own restrictions such as the number of threads to run, queue length, and so on.

- **Incorporation of concurrency constructs at the model level:** This requires separation of concerns between model and operating system levels. In other words, features such as message scheduling and guard conditions must be separated in terms of implementation and development.

  For example, removing or adding guard conditions should not cause the logic of message scheduling to be broken; any changes should only be reflected as a set of conditions that handle the scheduling process. Assume the following a mailbox example; there are some messages; they are ordered in the mailbox as per their priority. Thus, reading a message must be done first before starting to read a next message. Reading a message grants an exclusion access for the content of that message.

  The process of reading a message cannot be interrupted as it will cause the exclusion access to be terminated; especially if there are other processes waiting for this process.

- **Different combinations of synchronous and asynchronous execution:** Having different combinations of execution and active objects can cause development to be more complicated.

- **Inheritance anomaly:** There are several ways to handle inheritance for concurrency such as history variables and redefinition. In history variables, upon communication, variables are created for threads at different inheritance levels; those variables must be updated when thread states change such as joining or interruption [56].

``

The approach of history variables is not easy to follow. On the other hand, in redefinition, active objects are redefined entirely based on the last level of inheritance. We use a redefinition approach as it is easy and supported by UML [57].

- **Fault tolerance:** A developer must be able to debug and trace their active objects; those tasks are tricky because of having multiple threads to deal with.
- **Data recovery:** The system must be able to recover data upon failures or unhandled exceptions.

4.3 Active object as a concurrent model

We use Template Meta-Programming (TMP), and preprocessor to support cross-platform code, and to reduce boilerplate code. We implemented a lightweight multithreading library (Figure 4-1) to support concurrent active object features in Umple.

As we target the C++03 standard, we needed to write additional macros to enable variadic template arguments. By default, the maximum number of arguments is 10, which can be updated in a constant, ARGUMENT_UPPER_LIMIT. We use a naming convention for macro methods similar to the keywords used in Java and C# such as Thread, Runnable, instanceof, synchronized, RefPointer, and delegates.

We will suppress the code related to threads and active object implementation from the snippets shown in this chapter, as they are boilerplate code. UmpleOnline (try.umple.org) has some examples of active objects. Selecting any of these examples and setting the target language as C++ will display generated code.

``

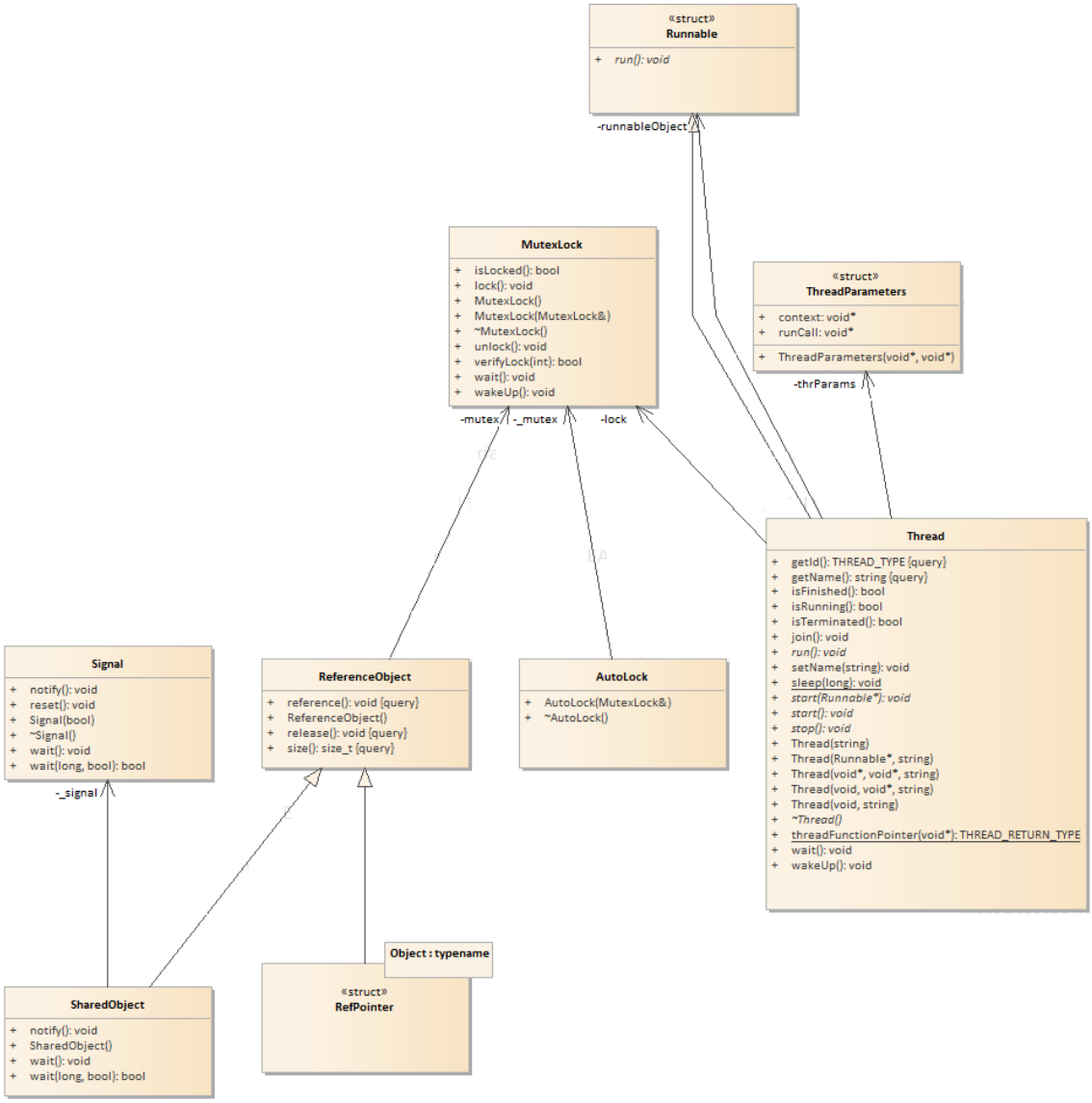


Figure 4-1. Overview of our lightweight multithreading library

4.3.1 Structure

We use Template Meta-Programming (TMP) and Curiously Recurring Template Pattern (CRTP) to enable reusability through traits and mixins [58]. This approach allows to generically define active features, determine the traits needed, and attach traits to a method call back (delegate) within a class object. A delegate or alternatively a function pointer is a variable used to invoke a callback method [55]. It can be either unicast or multicast, based on the number of callbacks to be dispatched.

``

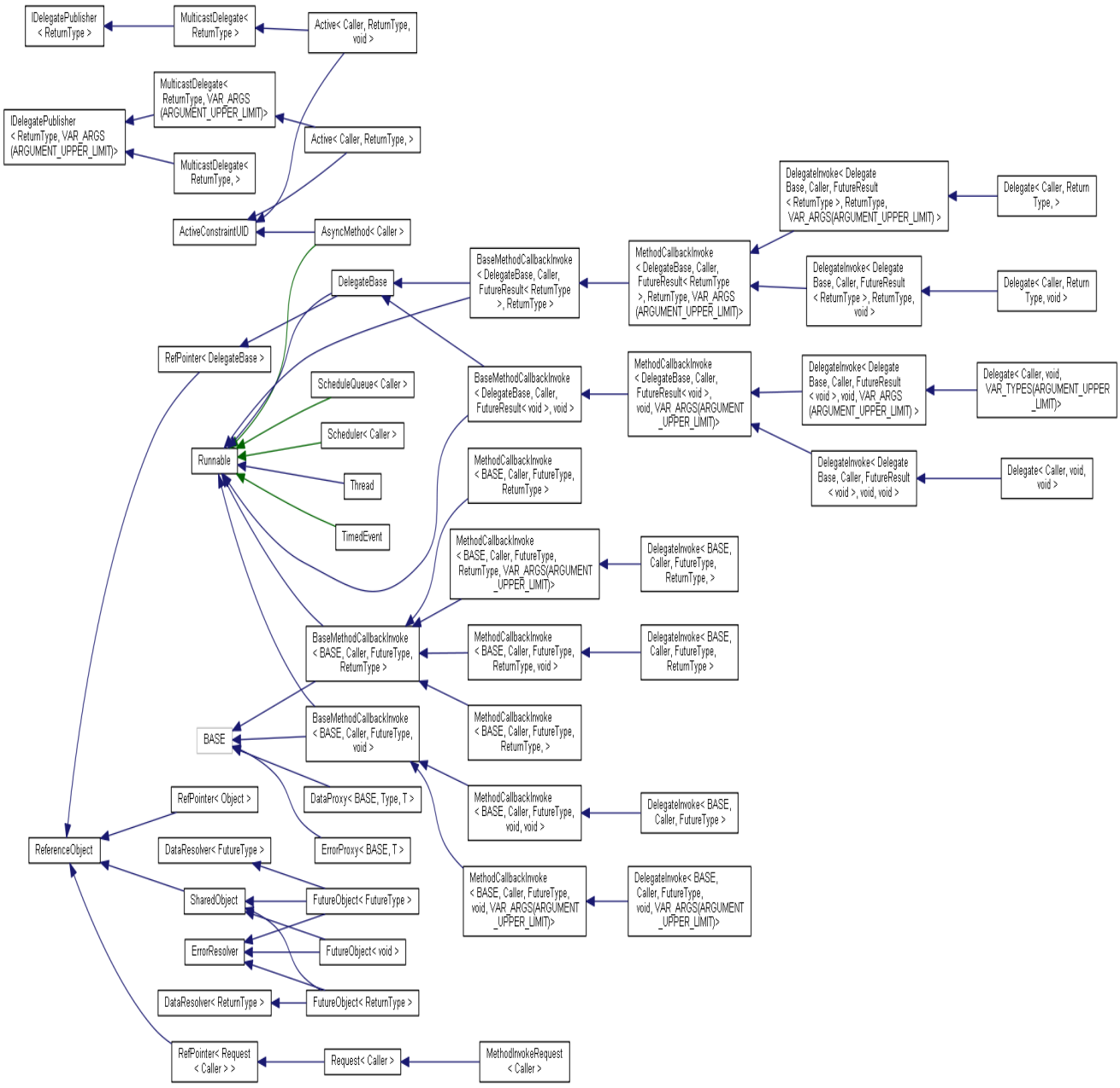


Figure 4-2. The class diagram of active objects in C++ code generated from Umple
This diagram was generated by Doxygen.

We use our variadic template arguments macro to make asynchronous function call parameters matching the exact number of original delegate method parameters. Therefore, it will be in sense similar to invoking the original method, but it will instead defer the execution on the active object.

``

Figure 4-2 shows the underlying class implementation of our active object pattern, including active object, public interface, message, scheduler, multicast delegate, and future object.

Key classes in Figure 4-2 are:

- **DelegatePublisher:** There are two kinds with different template parameters. These notify objects with a response. Active objects are key subclasses of these.
- **Runnable:** There are many kinds, including Thread, Scheduler and many kinds of CallbackInvoke with different template parameters. These are classes whose instances have their own thread of control.
- **ReferenceObject:** A smart pointer to safely remove objects from memory, following the reference counter design pattern.

4.3.1.1 Public interfaces

We use an *Active* template-based class to define public interface methods to decouple method invocation from execution [18], [51]. We employ *delegate* and TMP to preprocess a public interface function to have the same signature of a delegated method, in addition to defaulted additional arguments such as priority and delay. Snippet 4-1 is a basic example to define an active method.

Table 4-1. Basic APIs used in the action code for time constructs
p1 up to p3 refers to a dynamic number of parameters; i.e 0 to *

API	**Example**
Active	*methodCall(p1, p2, ..., pn, priority, delay, timeout)*
AsyncMethod	*methodCall (p1, p2, ..., pn, priority, period, delay, timeout)*

We have two internal types of public interface methods, *active* and *async*, for each we generate API to handle certain time constructs (Table 4-1) They differ based on their ability to spawn a thread; in particular, *async* has its own concurrent thread, similarly to the behaviour of do activity. *Async* is mainly used with repetition or periodic constructs.

Active and *Async* extend the original methods, such that additional parameters are added to handle time constructs(Table 4-1); the default parameter values are zero. The methods that require asynchronous execution rely on the async Method, while those that do not require rely on the Active (Table 4-3). Both Active and Async rely on the Scheduler API (Table 4-3).

It is important to mention that in Umple, for language usability purposes, we allow developers to define their main functions in a way similar to Java (Snippet 4-1 - Line 5).

``

```
1  class Test {                                                    Umple
2      int active call (int val1, int val2) {
3          return val1*val2;
4      }
5      public static void main(int argc, char * argv[]) {
6          Test test;
7          FutureResult<int> mul = test.call(2,2);
8      }
9  }
```

Snippet 4-1. Basic Active objects in Umple

In Snippet 4-2, the generated code for Snippet 4-1, Line 7 defines a delegate to a function that has two parameters of the type integer, and its return type is integer. The initialization of an active method takes a place in the constructor Line 5, which specifies a callback method and an Active Object (AO) scheduler. There will be no difference in method execution, except that it returns a future response (FutureResult). FutureResult can be represented as a proxy that communicates with the AO and holds response about result, status, and errors.

```
1  class Test {                                                    C++
2      ....
3  public:
4          Test ():
5              call(this,&_internalScheduler, &Test:: callImpl){}
6
7          Active< Test , int, int, int> call;
8
9  protected:
10         int_call(int val1, int val2) {
11                 returnval1 * val2;
12         }
13 private:
14         Scheduler _internalScheduler;
15             ....
16         };
17     ....
18 int main(int argc, char * argv[]) {
19         Test test;
20         FutureResult<int> mul = test.call(2,2);
21     }
```

Snippet 4-2. An example of generated code of an active object

In generated Umple code, we follow the same structure shown in Snippet 4-2. In Line 6, a public method is created for the active method call. The implementation of call is defined in an internal method _callas in Line 10. The visibility of call is protected instead of private in order to give the ability for subclasses to use or inherit it. Therefore, call acts as a public interface or client, while call acts as a servant. There is no need to use history variables or other mechanism to handle inheritance anomaly.

``

4.3.1.2 Future

Future is a shared-object proxy that provides a channel between client and AO. It provides a wait-set (such as wait, notify and wait for a specific time) functionality and a set of functions to inquire about an asynchronous response of an active method containing availability, content and/or errors. *Future* in Umple is an instance of FutureResult.

FutureResult shows a simple use of the wait-set functions (Snippet 4-3 - Lines 8-10). There is one active method with a single integer parameter (Line 2), and it is invoked once (Line 7). There are other optional parameters of the values 1 and 10 referring to the priority and delay (Line 7). The wait function has an optional argument to specify the expected waiting time (Line 8), otherwise it will throw a timeout exception. Each method returns a response describing a status (Line 10).

Umple

```
1  class Test {
2      int active call (int intValue) {
3          return intValue*2;
4      }
5      public int main(int argc,  char* argv[]) {
6          Test test;
7          FutureResult<int> result = test.call(1, 10);
8          result.wait(9000);
9          assert(result.ready());
10         cout<<result.data();
11     }
12 }
```

Snippet 4-3. Wait-set example

Table 4-2 shows the possible statuses, each of which is wrapped in an instance of FutureResult.

Table 4-2. Status types of active object execution

Status	Description
Pending	Not yet processed or activated due to queue requirements such as its order and priority in the queue.
Waiting	Processed and ready for execution in a queue.
Deferred	Postponed from being executed for not satisfying guard constraints, and added to deferred list to be recalled when constraints satisfied.
Done	Executed and completed without errors.
Error	Completed with errors.

FutureResult has error and data resolving functions. They can be used to throw exceptions such as timeout.

``

4.3.1.3 Scheduler

Scheduler handles mutual exclusion among queues and message requests. The process of message queuing may delay some tasks even if trivial such as read or status check tasks. For example, when there is a method request to update a value, it will put a mutex on some variables.

There are three common mechanism to implement the internal concurrency of an AO, which are *serial*, *quasi-concurrent*, and *full-concurrent* [59]–[61]. *Serial* or *sequential* creates only one thread, which uses a First-In-First-Out (FIFO) queue to process messages and executes only one message at a time while other messages are waiting in a queue. Q*uasi-concurrent* extends *serial* to have an auxiliary queue to enable simultaneous messages processing, but only one message can be in execution state at a time.

Full-concurrent takes a different direction by creating multiple threads and enabling simultaneous message executions. However, there will be a need to control message execution, by guarding the shared-state and using wait-set features to pause and resume threads. The messages as well, need to be separated into different independent containers to guarantee message orders and run-to-complete semantics.

We extend *quasi-concurrent* to have three priority-based double-ended queues, requests, pending, and deferred executions. A scheduler can be linked with a spawned thread such as a *async* method to control concurrency of the owning objects.

4.3.1.4 Messages

Message refers to invocation information, which contains the method delegate and arguments passed. We extend the message to include optional information, such as priority, delay, and guards.

A guard is an anonymous function with a Boolean operator that checks satisfiability. It is mainly used by the scheduler to make a decision to filter, execute, or, defer. A defer decision adds a message to a deferred list of messages, such that deferred messages of higher priorities are executed first.

4.3.1.5 Time constraints

Table 4-3 shows the set of time constraints we introduced into Umple. In an active method, a time constraint can be set at the operation level, action code level, or both of them. The operation level refers to the active method definition, while the action code level refers to the user-code written in that active method body.

``

Table 4-3. Time-based constructs

Constructs	Description	API	Allowance
Priority	Sets a priority to determine the order of invocation in a queue	Both	Action code
Timeout	Sets the maximum waiting time for a task to be completed.	Both	Both
Delay	Causes intentional delay	Both	Both
Period	Determines the polling time for rechecking a method	Async	Both

Typically, action code of an active method executed sequentially within their owning active object's thread. Nevertheless, some specialized Umple time constructs can be used to enable asynchronous execution, such that will need to spawn a thread to run concurrently; refer to *period* in the API column, in Table 4-3.

```
Grammar ::= activeMethodDefinition*
activeMethodDefinition ::= (portWatch)? (modifier)? (type)? (activeType)? 'active' activeMethodDeclarator (activeMethodBody)+
modifier ::= 'public' | 'protected' | 'private'
activeType ::= 'atomic' | 'synchronous' | 'intercept'
activeMethodDeclarator ::= methodName (parameterList)?
activeMethodBody ::= (activeDirectionHandler)? '{' ( activeMethodBodyContent )* '}'
activeMethodBodyContent ::= comment | activeTrigger | code
activeDirectionHandler ::= activeDirection ( ',' activeDirection )*
activeDirection ::= 'forward' | 'inverse'
activeTrigger ::= (hitchConstraint)? (constraintList)? '/' activeTriggerBody (thenDefinition)? (resolveDefinition)?
activeTriggerBody ::= deferredList | activeTriggerDefinition
deferredList ::= '[' activeTriggerDefinition ( ',' activeTriggerDefinition )* ']'
activeTriggerDefinition ::= anonymousTriggerBody | invoke
thenDefinition ::= '.then' '(' (anonymousTriggerBody)? ')'
resolveDefinition ::= '.resolve' '(' (anonymousTriggerBody)? ')'
hitchConstraint ::= hitchType ( timer )
hitchType ::= 'delay' | 'poll'
constraintList ::= '[' basicConstraint ( ',' basicConstraint )* ']'
basicConstraint ::= timeConstraint | messageConstraint | constraint
timeConstraint ::= timeConstraintType '(' timer ')'
timeConstraintType ::= 'latency' | 'period' | 'timeout'
messageConstraint ::= 'priority' '(' priorityValue ')'
invoke ::= ( classname'.' )? name '(' ( parameter ( ',' parameter )* )? ')'
anonymousTriggerBody ::= '{' code '}'
portWatch ::= ( constraintList | comment )*
```

Grammar 2 Active objects. Details are in Appendix A

4.4 Active Objects in Umple

Figure 4-3 shows a RailRoad diagram describing the syntax used to describe active objects; this class diagram also shows a part of the implementation of composite structure, since the implementation of both active objects and

``

composite structure are correlated in Umple. In this chapter, we do not aim to discuss composite structure in detail; we leave that for the next chapter.

Grammar 2 shows the grammar definition of active objects in Umple in Railroad syntax.

``

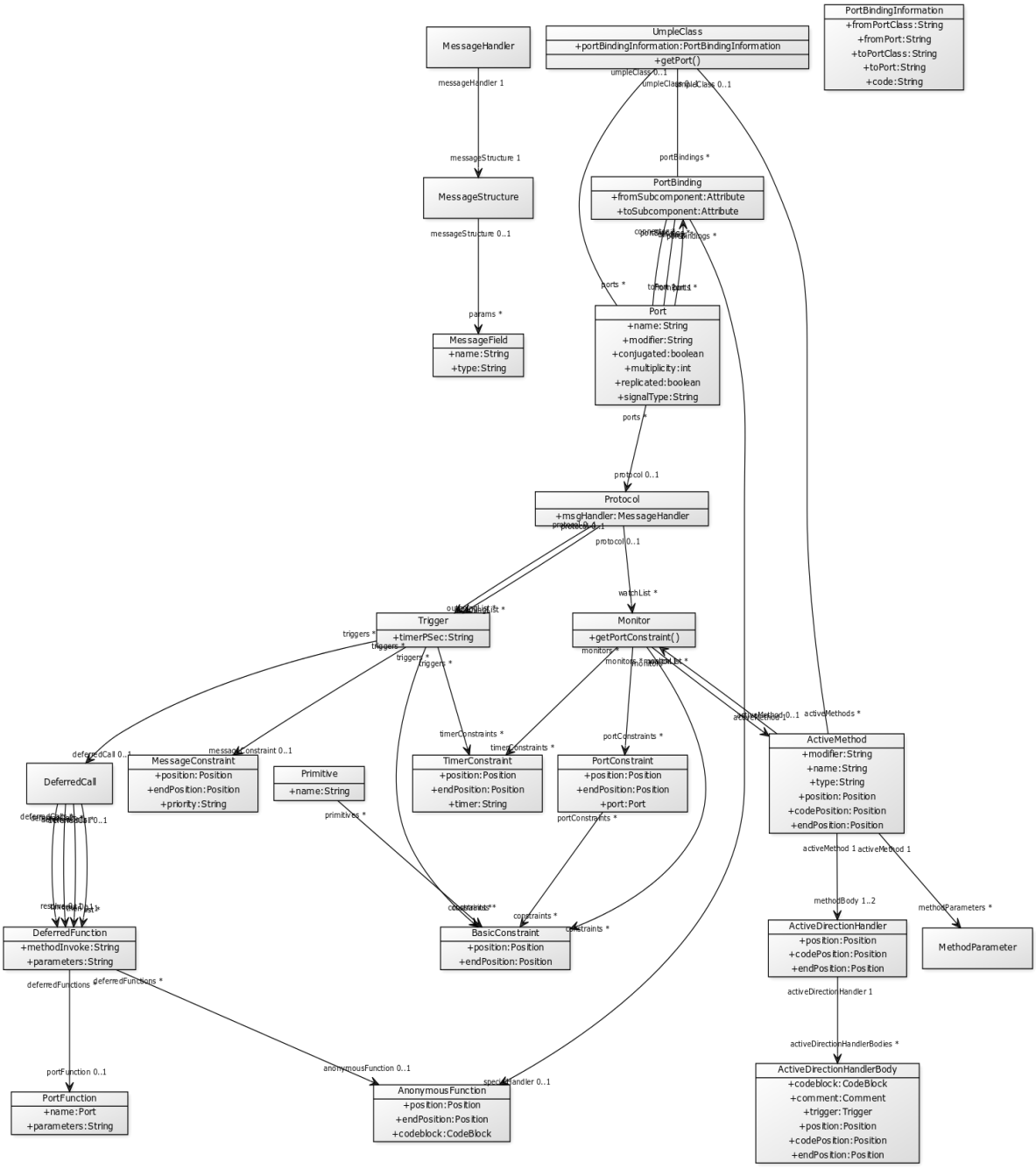


Figure 4-3. The class diagram of the portion of the Umple metamodel relating to active objects and composite structure

``

4.5 Active method declaration

The *active* keyword is used to declare active methods (Snippet 4-4 - Line 2). An active method has the same constraints of regular methods such as having a unique name, return type, and signature. Once a class has at least one active method, it will be generated and designed to be an active class.

An active method is invoked in a similar manner to regular methods (Snippet 4-4 - Line 8).

Umple

```
1  class ActiveMethodDeclaration {
2      String active activeMethodExample (String param1, Integer param2) {
3          return "This is an active method:"+ param1 <<","<< param2;
4      }
5
6      int main(int argc, char *argv[]){
7          ActiveMethodDeclaration activeMethodDeclaration;
8          activeMethodDeclaration.activeMethodExample("Some string", 99);
9      }
10 }
```

Snippet 4-4. A simple active method declaration

In terms of code generation, we rely on the Active API (Table 4-1 and Table 4-3), which we implemented as a part of this research. An Active instance receives two parameters, the class that an active object belongs to and the return type of that active object, in addition to the types of the parameters defined in the active method (Snippet 4-5 - Line 2). Additional parameters are string and integer in the case of the model we are discussing (Snippet 4-5 - Line 2). After that, this Active instance is initialized as a member of the class constructor (Line 7).

C++

```
1  //This portion is from the ActiveMethodDeclaration.h file
2  Active<ActiveMethodDeclaration, string, string, int> activeMethodExample;
3  Scheduler _internalScheduler;
4
5  //This portion is from the ActiveMethodDeclaration.cpp file
6  ActiveMethodDeclaration::ActiveMethodDeclaration() :
7     activeMethodExample(this, &_internalScheduler, &ActiveMethodDeclaration::_activeMethodExample)
8  }
9  .....
10 string _activeMethodExample(string param1){
11    return "This is an active method:"+ param1 << "," << param2;
12 }
```

Snippet 4-5. Portions of the generated code for Snippet 4-4

The name of the *Active* instance is the same as the active method defined in the Umple model (Snippet 4-4- Line 2). The content of the active method is placed in a private method in the generated code, and its name is prefixed with an underscore (Snippet 4-5 - Line 10). The Active instance refers to this private method (Snippet 4-5 - Line 7).

``

The *Scheduler* API is used to handle the execution queue of active methods based on their order of invocation and priorities. When an active object exists in a class, we generate an internal Scheduler (Snippet 4-5 - Line 3) that will be used by all *Active* instances (Line 7 for instance)

An active method without a return type will be assumed void. If an active method does not have parameters, a user will not need to worry about bureaucratically passing empty parentheses.

Snippet 4-6 shows an example of an active method with null parameters and a null return type. Both methods in (Snippet 4-6 - Lines 2 and 6**)** will be considered to have the same signature.

Umple

```
1  class DefaultActiveObject {
2      active activeMethodExample1{
3          cout << "This is an active method without parameter, parentheses, nor a return type";
4      }
5
6      void active activeMethodExample2() {
7          cout << "This is equivalent to activeMethodExample1";
8      }
9  }
```

Snippet 4-6. An example of an active method with null parameters and return type

4.6 Method invocation

In this section, we discuss the features we implemented to improve writing the action code of an active method.

Writing action code in the target language may have limitations. For instance, the C++ 03 standard does not provide an easy way to define anonymous functions. At the model level, we need to have a way to regulate the process of handling error exceptions or then calls.

Although the content of an active method runs in a separate thread, this may still have some limitations. For instance, within the action code of the same active method, we may find it important to invoke other methods, which could be regular methods.

4.6.1 Case 1: Trigger nonactive methods

The trigger operator, "/" is used to invoke a method, either active or nonactive, or an anonymous body. It is used to enforce active behaviour on a method even if it is not defined active. For instance, the call in (Snippet 4-7 - Line 7) will have its own thread, while the call in (Line 8) will not.

``

```
1  class MethodInvocationTest{                                          Umple
2      void regularMethod(){
3              cout << "I am a regular method but can be invoked actively";
4      }
5
6      void active someActiveMethod(){
7        /regularMethod();
8        regularMethod();
9      }
10
11     int main(int argc, char *argv[]){
12             MethodInvocationTest methodInvocationTest;
13             methodInvocationTest.someActiveMethod();
14     }
15 }
```

Snippet 4-7. Active and nonactive method invocation

In terms of the code generation, an *Active* instance will be created to wrap the regular method within active execution (Snippet 4-8 - Lines 3-4).

If a method is invoked directly (Snippet 4-7 - Line 8), it will be as well invoked directly in the code generated (Snippet 4-8 - Line 6).

```
1  //This portion is from the MethodInvocationTest.cpp file                C++
2  void _someActiveMethod(){
3      Active<MethodInvocationTest, void> active_someActiveMethod_regularMethod_1(this,
4          &_internalScheduler, &MethodInvocationTest::regularMethod);
5      active_someActiveMethod_regularMethod_1();
6      regularMethod();
7  }
8
9  void MethodInvocationTest::regularMethod(){
10     cout <<" I am a regular method but can be invoked actively";
11 }
```

Snippet 4-8. Portions of the generated code for Snippet 4-7

4.6.2 Case 2 Trigger nonactive methods multiple times

A new *Active* instance will be created when triggering a nonactive method. Hence, we have to make sure to avoid variable redefinition at the generated code by adding a counter to each new *Active* (Snippet 4-10). We do not use the same Active instance for all triggers, since we need to ensure that each trigger will have its own thread.

``

```
1  class MethodInvocationTest{                                          Umple
2      void regularMethod1(String value){
3              cout << "Print" << value;
4      }
5
6      void active someActiveMethod(){
7              /regularMethod1("Value1");
8              /regularMethod1("Value2");
9              /regularMethod1("Value3");
10     }
11
12     int main(int argc, char *argv[]){
13             MethodInvocationTest methodInvocationTest();
14             methodInvocationTest.someActiveMethod();
15     }
16 }
```

Snippet 4-9. Multiple nonactive method invocation

```
1  //This portion is from the MethodInvocationTest.cpp file                C++
2  Active<MethodInvocationTest, void> active_someActiveMethod_regularMethod1_1(this,
3  &_internalScheduler, &MethodInvocationTest::regularMethod1);
4  active_someActiveMethod_regularMethod1_1("Value1");
5
6  Active<MethodInvocationTest, void> active_someActiveMethod_regularMethod1_2(this,
7  &_internalScheduler, &MethodInvocationTest::regularMethod1);
8  active_someActiveMethod_regularMethod1_2("Value2");
9
10 Active<MethodInvocationTest, void> active_someActiveMethod_regularMethod1_3(this,
11 &_internalScheduler, &MethodInvocationTest::regularMethod1);
12 active_someActiveMethod_regularMethod1_3("Value3");
```

Snippet 4-10. Portions of the generated code for Snippet 4-9

4.6.3 Case 3 Trigger anonymous functions

We allow users to trigger anonymous functions (Snippet 4-11 - Lines 3-9). In the generated code, we create a new method that has the content of the anonymous function (Snippet 4-12 - Lines 14-16 and 18-20). We make sure that this anonymous method has a unique name after the active method name and nonactive method being invoked, in addition to a counter.

``

```
1  class AnonymousFunctionTest {                                          Umple
2      void active someActiveMethod(){
3        /{
4           cout << "I am an anonymous function" ;
5        }
6
7        /{
8           cout << "I am another anonymous function" ;
9        }
10     }
11
12     int main(int argc, char *argv[]){
13             AnonymousFunctionTest anonymousFunctionTest;
14             AnonymousFunctionTest.someActiveMethod();
15     }
16 }
```

Snippet 4-11. Anonymous function invocation

```
1  //This portion is from the AnonymousFunctionTest.cpp file                C++
2  void _someActiveMethod(){
3     Active<AnonymousFunctionTest, void>
4           active_someActiveMethod_someActiveMethod_anonymous_part1(this, &_internalScheduler,
5  &AnonymousFunctionTest::someActiveMethod_anonymous_part1);
6    active_someActiveMethod_someActiveMethod_anonymous_part1();
7
8  Active<AnonymousFunctionTest, void>
9         active_someActiveMethod_someActiveMethod_anonymous_part2(this, &_internalScheduler,
10 &AnonymousFunctionTest::someActiveMethod_anonymous_part2);
11    active_someActiveMethod_someActiveMethod_anonymous_part2();
12 }
13
14 void AnonymousFunctionTest::someActiveMethod_anonymous_part1(){
15       cout <<"I am an anonymous function" ;
16 }
17
18 void AnonymousFunctionTest::someActiveMethod_anonymous_part2(){
19     cout <<"I am another anonymous function" ;
20 }
```

Snippet 4-12. Portions of the generated code for Snippet 4-11

4.6.4 Case 4 Call/then pattern

The call/then pattern is similar to the try/finally pattern that exists in common programming languages such as C++ and Java, but it differs in that it works asynchronously. Simply, we wrap method invocation within an anonymous body (Snippet 4-13 - Lines 7-17). We can directly write code (Lines 8 and 10), or invoke other methods (Line 14). The code generated will make a call to the then body (Snippet 4-14 - Lines 14 and 19) after the call body (Lines 13 and 18).

``

```
1  class CallThenTest{
2     void active activeMethod(){
3          cout <<"do something";
4     }
5
6      void active thenTest(){
7        /{
8             cout <<"I am an anonymous function" ;
9        }.then({
10            cout <<"Anonymous function done";
11       })
12
13       /{
14            this->activeMethod();
15       }.then({
16            cout <<"Active method done";
17       })
18    }
19
20     int main(int argc, char *argv[]){
21        CallThenTest callThenTest;
22        callThenTest.thenTest();
23    }
24 }
```

Umple

Snippet 4-13. A call/then pattern example

```
1  //This portion is from the CallThenTest.cpp file
2  void _thenTest(){
3     Active<CallThenTest, void> active_thenTest_thenTest_anonymous_part1(this, &_internalScheduler,
4  &CallThenTest::thenTest_anonymous_part1);
5     active_thenTest_thenTest_anonymous_part1();
6
7     Active<CallThenTest, void> active_thenTest_thenTest_anonymous_part2(this, &_internalScheduler,
8  &CallThenTest::thenTest_anonymous_part2);
9     active_thenTest_thenTest_anonymous_part2();
10 }
11
12 void CallThenTest::thenTest_anonymous_part1(){
13    cout << "I am an anonymous function" ;
14    cout << "Anonymous function done";
15 }
16
17 void CallThenTest::thenTest_anonymous_part2(){
18    this->activeMethod();
19    cout << "Active method done";
20 }
21
22 void _activeMethod(){
23    cout << "do something";
24 }
```

C++

Snippet 4-14. Portions of the generated code for Snippet 4-13

``

4.6.5 Case 5 Call/resolve and call/then/resolve patterns

The call/resolve pattern is similar to the try/catch pattern. A resolve body is only called when something goes wrong with a call body (Snippet 4-15 - Lines 10 and 12), resolve body (Lines 29, and 31); otherwise, it will not (Lines 4, 6, and 20).

```
1  class CallResolveThenTest{                                          Umple
2     void active resolveTest(){
3        /{
4           cout <<"Do something" ;
5        }.resolve({
6           cout <<"Will not get here";
7        })
8
9        /{
10          throw std::invalid_argument( "An error" );
11       }.resolve({
12          cout <<"An error has occurred";
13       })
14    }
15
16    void active resolveAndThenTest(){
17       /{
18          throw std::invalid_argument( "An error" );
19       }.then({
20          cout <<"Will not get there";
21       }).resolve({
22          cout <<"Something went wrong at the call body";
23       })
24
25       /{
26          cout <<"I am safe";
27       }.then({
28          cout <<"We got there but let try throwing an error";
29          throw std::invalid_argument( "An error" );
30       }).resolve({
31          cout <<"Something went wrong at the then body";
32       })
33    }
34
35    int main(int argc, char *argv[]){
36          CallResolveTest callResolveTest;
37          callResolveTest.resolveTest();
38    }
39 }
```

Snippet 4-15. Examples of call/resolve and call/then/resolve patterns

In terms of the code generation, a resolve body is wrapped within a catch block (Snippet 4-16 - Lines 28, 36, 45, and 55).

``

```C++
//This portion is from the CallResolveThenTest.cpp file
void _resolveTest(){
   Active<CallResolveThenTest, void> active_resolveTest_resolveTest_anonymous_part1(this,
&_internalScheduler, &CallResolveThenTest::resolveTest_anonymous_part1);
   active_resolveTest_resolveTest_anonymous_part1();

  Active<CallResolveThenTest, void> active_resolveTest_resolveTest_anonymous_part2(this,
&_internalScheduler, &CallResolveThenTest::resolveTest_anonymous_part2);
  active_resolveTest_resolveTest_anonymous_part2();
}

void _resolveAndThenTest(){
  Active<CallResolveThenTest, void>
      active_resolveAndThenTest_resolveAndThenTest_anonymous_part1(this, &_internalScheduler,
&CallResolveThenTest::resolveAndThenTest_anonymous_part1);
 active_resolveAndThenTest_resolveAndThenTest_anonymous_part1();

  Active<CallResolveThenTest, void>
      active_resolveAndThenTest_resolveAndThenTest_anonymous_part2(this, &_internalScheduler,
&CallResolveThenTest::resolveAndThenTest_anonymous_part2);
   active_resolveAndThenTest_resolveAndThenTest_anonymous_part2();
}

void CallResolveThenTest::resolveTest_anonymous_part1(){
    try {
      cout <<"Do something" ;
   }catch (...){
      cout <<"Will not get here";
   }
}

void CallResolveThenTest::resolveTest_anonymous_part2(){
    try {
      throw std::invalid_argument( "An error" );
   }catch (...){
      cout <<"An error has occurred";
   }
}

void CallResolveThenTest::resolveAndThenTest_anonymous_part1(){
    try {
      throw std::invalid_argument( "An error" );
      cout <<"Will not get there";
   }catch (...){
      cout <<"Something went wrong at the call body";
   }
}

void CallResolveThenTest::resolveAndThenTest_anonymous_part2(){
    try {
      cout <<"I am safe";
      cout <<"We got there but let try throwing an error";
```

``

```
53        throw std::invalid_argument( "An error" );
54      }catch (...){
55         cout <<"Something went wrong at the then body";
56      }
57  }
```

Snippet 4-16. Portions of the generated code for Snippet 4-28

4.6.6 Case 6: Deferred list

A resolve body may seem trivial with simple method invocation, but it is more useful when it comes to deferred lists. After the execution of a deferred list, the then or resolve bodies are invoked one time (Snippet 4-17 - Lines 14 and 16, and Snippet 4-18).

Umple

```
1   class DeferredListTest{
2      void active someActiveMethod(){
3           cout << "do another thing";
4      }
5
6      void active deferredTest(){
7        [/{
8           cout << "do something";
9        },/{
10       this->someActiveMethod();
11       },/{
12          cout << "do something more";
13       }].then({
14          cout << "Will be invoked once after executing the entire list";
15       }).resolve({
16         cout << "Will print only upon exceptions";
17       })
18     }
19
20     int main(int argc, char *argv[]){
21        DeferredListTest deferredListTest;
22        deferredListTest.deferredTest();
23    }
24  }
```

Snippet 4-17. A deferred list example

``

```
1  //This portion is from the DeferredListTest.cpp file                                    C++
2  void _someActiveMethod(){
3     cout << "do another thing";
4  }
5
6  void _deferredTest(){
7     Active<DeferredListTest, void> active_deferredTest_deferredTest_anonymous_part1(this,
8  &_internalScheduler, &DeferredListTest::deferredTest_anonymous_part1);
9     active_deferredTest_deferredTest_anonymous_part1();
10 }
11
12 void DeferredListTest::deferredTest_anonymous_part1(){
13       try {
14          cout << "do something";
15          this->someActiveMethod();
16          cout << "do something more";
17          cout << "Will be invoked once after executing the entire list";
18      }catch (...){
19          cout << "Will print only upon exceptions";
20     }
21 }
```

Snippet 4-18. Portions of the generated code for Snippet 4-17

4.7 Logical constraints

4.7.1 Case 1: Logical constraints at the operation level

An active method will not execute if it has any unsatisfied logical conditions. For instance, in Snippet 4-19, the logical condition set in Line 4 will work for the first invocation (Line 20), but not for the second one (Line 22) after the logical condition has become unsatisfied (Line 23).

``

```
1  class Test {                                                    Umple
2      Integer increment=0;
3
4      [increment<10]
5      void active activeMethod{
6          cout<< increment;
7      }
8
9      [increment>=10]
10     void active activeMethod2{
11         cout<< increment;
12     }
13
14     void activeactiveMethod3{
15         cout<< increment;
16     }
17
18     public int main(int argc, char *argv[]) {
19         Test test;
20         test.activeMethod();    //Print increment
21         test.setIncrement(10);
22         test.activeMethod();   //Will do nothing
23     }
24 }
```

Snippet 4-19. An example of constraints set on an active method

In terms of code generation, active method guards are applied at two places, the scheduler (Snippet 4-20 - Lines 2-14) and active methods (Lines 17 and 24). At the scheduler level, an active method is not called at all if a condition is unsatisfied. For that, we create a method that checks only for active methods that have logical conditions defined (Lines 6-14). If an active method does not have any logical constraints, this method will return true (Line 13).

The check at an active method is an extra precaution against bad user calls to that method.

``

```
1  //This portion is from the Test.cpp file                                    C++
2  Test::Test(const int& aIncrement) :
3  _internalScheduler(this, &Test::schedulerGuards_),
4  ...
5  }
6  bool Test::schedulerGuards_(int guardId){
7      if(guardId == activeMethod.getConditionUID()) {
8          return (increment < 10);
9      }
10     if(guardId == activeMethod2.getConditionUID()) {
11         return (increment >= 10);
12     }
13     return true;
14 }
15
16 void Test::_activeMethod(){
17     if(!(increment < 10)){
18         throw"Please provide a valid increment";
19     }
20         cout<< increment;
21 }
22
23 void Test::_activeMethod2(){
24     if(!(increment >= 10)){
25         throw"Please provide a valid increment";
26     }
27     cout<< increment;
28 }
29
30 void Test::_activeMethod2(){
31     cout<< increment;
32 }
```

Snippet 4-20. Portions of the generated code for Snippet 4-19

4.7.2 Case 2: Logical constraints at the action code level

A logical constraint at the action code level is used to make the action code execute if the constraints are satisfied, otherwise it will be deferred. For instance, the action code in (Snippet 4-21 - Line 5) will execute after setting a flag variable to true (Line 16).

``

```
1  class LogicalConstraintsTest{                                          Umple
2      Boolean flag= false;
3      void active activeMethod{
4      [flag==true]/{
5              cout <<"Flag is true" ;
6          }
7      }
8
9      public int main(int argc, char *argv[]) {
10         LogicalConstraintsTest logicalConstraintsTest;
11
12         /A listener will be added instead, since flag is still false
13         logicalConstraintsTest.activeMethod();
14
15         //will print "Flag is true"
16         logicalConstraintsTest.setFlag(true);
17     }
18 }
```

Snippet 4-21. An example of constraints set on action code

In terms of code generation, when a code block has a logical condition defined, we externalize the active object declaration of this code block into a field (Snippet 4-22 - Line 2), such that the scheduler can keep track of its logical conditions (Line 14). Otherwise, an active object is defined locally (Snippet 4-18 - Lines 7 and 8) .

```
1  //This portion is from the Test.h file                                 C++
2  Active<Test, void> active_activeMethod_activeMethod_anonymous_part1_1;
3
4  //This portion is from the Test.cpp file
5  void Test::_activeMethod(){
6     active_activeMethod_activeMethod_anonymous_part1_1();
7  }
8
9  bool Test::schedulerGuards_(int guardId){
10    if(guardId == active_activeMethod_activeMethod_anonymous_part1_1.getConditionUID()) {
11       return (flag);
12    }
13    return true;
14 }
```

Snippet 4-22. Portions of the generated code for Snippet 4-21

4.8 Time constraints

In this section, we discuss different variations of using time constraints described in Table 4-3.

``

4.8.1 Constraints at the operation level

4.8.1.1 Case 1: Timeout at the operation level

The active method in (Snippet 4-23 - Lines 2-9) has timeout of 44 milliseconds. The first call to the active method will result in a single for loop, which mostly will not exceed the timeout specified (Line 13). However, the second call will require 100 iterations, which violate the timeout (Line 14).

```
1  class TimeConstraintsTest{                                          Umple
2      [timeout(44)]
3      void active printWithTimeout(String print, Integer max){
4      int count= 0;
5      while(count< max){
6              cout << "Print:" << print << "," << count;
7              Thread::sleep(1);
8              count++;
9          }
10     }
11     int main(int argc, char *argv[]){
12        TimeConstraintsTest timeConstraintsTest;
13        timeConstraintsTest.printWithTimeout ("Some string1", 1);
14        timeConstraintsTest.printWithTimeout ("Some string1", 100);
15     }
16 }
```

Snippet 4-23. A timeout example at the operation level

In the code generated (Snippet 4-24), an instance of *Active* is declared, given that the return type is void, and there are two parameters, string and integer (Snippet 4-23 - Line 3). After that, this Active instance is initialized as a member of the class constructor with the timeout value, 44 (Line 6). Timeout comes third in the time constraint arguments (Table 4-1), which means that we will need to pass a value of zero for the arguments before in order to enforce the default behaviour for other unused time constraints.

``

```
1  //This portion is from the TimeConstraintsTest.h file                    C++
2  Active<TimeConstraintsTest, void, string, int> timeoutOnly;
3
4  //This portion is from the TimeConstraintsTest.cpp file
5  TimeConstraintsTest::TimeConstraintsTest() :
6        timeoutOnly(this, &_internalScheduler, &TimeConstraintsTest::_timeoutOnly, 0, 0, 44){
7  }
8
9  void _timeoutOnly(string print, int max){
10    int count= 0;
11   while(count< max){
12       cout << "Print:" << print << "," << count;
13       count++;
14    }
15 }
```

Snippet 4-24. Portions of the generated code for Snippet 4-23

4.8.1.2 Case 2 Period at the operation level

A period time constraint is specified in (Snippet 4-25 - Line 2). Once the class is constructed, the interval will start right away, meaning that the active method in (Lines 3-5) will be called every one second.

```
1  class TimeConstraintsTest{                                              Umple
2      [period(1000)]
3      void active periodMethod(){
4          cout <<"Will be called repeatedly each one second";
5       }
6
7      int main(int argc, char *argv[]){
8          TimeConstraintsTest* timeConstraintsTest= new TimeConstraintsTest();
9       }
10 }
```

Snippet 4-25. A period example at the operation level

In terms of code generation, the *async* method is used once a period time constraint is applied (Table 4-3).

First, an instance of a*sync* is created (Snippet 4-27 - Line 2). After that, this instance is initialized as a member of TimeConstraintsTest constructor, with the value of the period defined, 1000 (Snippet 4-27 - Line 6 and Snippet 4-25- Line 2). A start call is made at the constructor in order to let the periodic execution starts (Snippet 4-27 - Line 7).

The users still have the capabilities to stop an interval manually, or start it later in the action code (Snippet 4-26 - Lines 5 and 6).

``

```
1  class TimeConstraintsTest{                                                    Umple
2      .....
3      int main(int argc, char *argv[]){
4         TimeConstraintsTest* timeConstraintsTest= new TimeConstraintsTest();
5         timeConstraintsTest.stop();
6         timeConstraintsTest.start();
7      }
8  }
```

Snippet 4-26. Start and stop APIs of AsyncMethod

```
1  //This portion is from TimeConstraintsTest.h file                            C++
2  AsyncMethod<TimeConstraintsTest> listen_periodic_periodMethod;
3
4  //This portion is from TimeConstraintsTest.cpp file
5  TimeConstraintsTest::TimeConstraintsTest() :
6  listen_periodic_periodMethod(this, &_internalScheduler, &TimeConstraintsTest::_periodMethod, 1000){
7      listen_periodic_periodMethod.start();
8  }
9  .....
10 void _periodMethod(){
11     cout <<"Will be called repeatedly each one second ";
12 }
```

Snippet 4-27. Portions of the generated code for Snippet 4-25

4.8.1.3 Case 3 Async versus Active

Both terms are used interchangeably based on the defined time constraints. In terms of code generation, the *Async* API must be used when the defined constraints contain period, and the delegated function must have void parameters. Otherwise, the *Active* API will be used (Snippet 4-28).

In terms of models, users still only need to use the keyword active, and based on their defined time constraints, we use the appropriate API, Active or Async when generating models.

``

```
1   class TimeConstraintsTest{                                                    Umple
2
3       [period(10000), delay(1000)]
4       void active someMethod1(){
5            cout <<"Will be asynchronous";
6       }
10
11      [period(5000), timeout(44)]
12      void active someMethod2(){
13           cout <<"Will be asynchronous as well";
14      }
15
16      [period(5000), timeout(44), delay(1000)]
17      void active someMethod3(){
18           cout <<"Will be asynchronous too";
19      }
20
21      [period(5000), timeout(44), delay(1000)]
22      void active someMethod4(String param){
23           cout <<"Will not be asynchronous, since there is a parameter used.";
24           cout <<"In terms of code generation, period will be ignored. ";
25           cout <<"At the compiler level, a warning will be shown. ";
26      }
27  }
```

Snippet 4-28. Period and method without parameters affect the API used

4.8.1.4 Case 4 Delay at the operation level

There is one active method in (Snippet 4-29 - Lines 3-5). It is invoked (Line 11), and will execute after 10 seconds specified (Line 2).

```
1   class TimeConstraintsTest{                                                    Umple
2       [delay(10000)]
3       void active delayTest(String value){
4           cout <<"I will be printed after 10 seconds:" << value;
5       }
6       int main(int argc, char *argv[]){
10          TimeConstraintsTest timeConstraintsTest;
11          timeConstraintsTest.delayTest("print something");
12      }
13  }
```

Snippet 4-29. A Delay at the operation level example

In the code generation, the delay value is passed when initializing the active method (Snippet 4-30 - Line 3).

``

```
1  //This portion is from TimeConstraintsTest.cpp file                    C++
2  TimeConstraintsTest::TimeConstraintsTest() :
3        delayTest(this, &_internalScheduler, &TimeConstraintsTest::_delayTest, 0, 10000){
4  }
```

Snippet 4-30. Portions of the generated code for Snippet 4-29

4.8.1.5 Case 6: Prioritized method invocation

A priority cannot be set as a time constraint at the operation level, yet an active method can be invoked with a priority using the active or async method (Table 4-1).

```
1   class TimeConstraintsTest{                                            Umple
2       void active someMethod1(){
3            Thread::sleep(3000);
4            cout <<"Method 1";
5       }
6
10      void active someMethod2(){
11           cout <<"Method 2";
12      }
13
14      int main(int argc, char *argv[]){
15         TimeConstraintsTest timeConstraintsTest;
16
17         timeConstraintsTest.someMethod1(1); //Priority of 1
18         timeConstraintsTest.someMethod1(1); //Priority of 1
19         timeConstraintsTest.someMethod2(2); //Priority of 2
20     }
21  }
```

Snippet 4-31. An active method invocation with a priority value

When invoking an active method, it is added to the pending queue of the scheduler. However, this does not mean that it is executed or finished execution yet. The invoked method is added to the pending queue based on its priority. If it has priority higher than existing entries in the queue, it will come before them even if they are invoked earlier.

In Snippet 4-31, there are two invocations to an active method (Lines 17-18), which forces a sleep for 3 seconds (Line 3). Then, there is invocation to another active method, but with higher priority, 2 (Line 19). We will refer to the first two invocations as M1_1 and M1_2, and the last invocation as M2_1. Table 4-5 shows the pending and execution queues during the lifecycle of the main function of Snippet 4-31, along with the description.

Based on the logic above, users do not need to worry about suspending or postponing the executing of an active method. Simply, an active method will execute according to the priority set.

``

We do not provide a public API, although it is already implemented internally, to allow users to directly suspend or interrupt active methods. This is in order to ensure the integrity of an application's lifecycle. However, we may consider providing such an API if we find a potential need for them in the future.

Table 4-4. Sequence of execution of Snippet 4-31

Pending	Execution	Description
[]	[M1_1]	Executing M1_1 since there is no pending entries
[M1_2]	[M1_1]	Add M1_2 to the pending queue, since M1_1 is still executing; 3 seconds at least are required
[M2_1, M1_2]	[M1_1]	Add M2_1 to the pending queue but before M1_2, since M2_1 has higher priority. M1_1 is still in process
[M1_2]	[M2_1]	After finishing the execution of M1_1, will start executing M2_1, the next in the queue.
[]	[M1_2]	Finally, M1_2 will execute.

4.8.1.6 Case 6: Delay, and timeout upon method invocation

In a similar manner to priority, delay and timeout constraints can be passed dynamically when invoking an active method (Table 4-1). Snippet 4-32 shows some examples; explanation is in the snippet comments.

``

```
1  class TimeConstraintsTest{                                                    Umple
2      void active activeMethodWithoutParameters(){
3          cout <<"Print";
4      }
5
6      void active activeMethodWithParameters(String param1, String param2){
10         cout <<"Print:"<< param1 <<","<< param2;
11     }
12
13     int main(int argc, char *argv[]){
14         TimeConstraintsTest* timeConstraintsTest= new TimeConstraintsTest();
15
16         //No priority nor latency are passed, so they both are assumed zero
17         timeConstraintsTest->activeMethodWithoutParameters();
18
19         //Will have the same results as above, since default values are zeros
20         timeConstraintsTest->activeMethodWithoutParameters(0, 0, 0);
21
22         //Only priority is passed
23         timeConstraintsTest->activeMethodWithoutParameters(1);
24
25         //If we only have a delay value, we need to put zero for priority, which is the default
26         timeConstraintsTest->activeMethodWithoutParameters(0, 1);
27
28         //Parameters of an active method must be passed first if exist.
29         //Below, there are time constraints passed, so they are all assumed zero
30         timeConstraintsTest->activeMethodWithParameters("a", "b");
31
32         //The values of priority, delay, timeout are 1, 5000, and 9000 respectively.
33         timeConstraintsTest->activeMethodWithParameters("a", "b", 1, 5000, 9000);
34     }
35 }
```

Snippet 4-32. Different parameter combinations when invoking an active method

4.8.1.7 Case 7: Logical constraints at the operation level

A logical constraint can be set directly with any time constraints. In terms of behaviour, simply, an active method will not be invoked if a logical constraint is unsatisfied. For instance, in Snippet 4-33, the periodic update in Line 4 will halt after 100 invocations (Lines 4 and 10). The other periodic call in Line 16 will make the logical condition satisfying again every minute (Line 16).

``

```
1  class LogicalTimeConstraintsTest {                                    Umple
2      Integer increment=0;
3
4      [period(1), increment <100]
5      active activeMethod{
6          cout<< increment;
10         increment++;
11     }
12
13     [period(60000)]
14     active activeMethod2{
15         //Will reset increment after a minute
16         increment= 0;
17     }
18
19     int main(int argc, char *argv[]) {
20         LogicalTimeConstraintsTest logicalTimeConstraintsTest();
21     }
21 }
```

Snippet 4-33. An example of using time constraints with logical conditions

4.8.2 Constraints at the action code level

4.8.2.1 Case 1: delay, priority, and timeout time constraints

Time constraints are placed before the trigger operator (Snippet 4-34 - Lines 14-16). They can be applied on active methods (Line 14), nonactive methods (Line 15), and anonymous functions (Line 16).

Priority can be passed as a time constraint at the action code level, as opposed to the operation level (Table 4-3).

``

```
1  class ActionCodeTimeConstraintsTest{                                    Umple
2      void active activeMethod(String value){
3          cout <<"Active print:"<< value;
4       }
5
6       void regularMethod(String value){
10         cout <<"I am not active but will be called actively. Print:"<< value;
11      }
12
13     void active constraintTest(print){
14         [delay(100), priority(1), timeout(9000)]/activeMethod(print);
15         [delay(10), priority(9), timeout(12000)]/regularMethod(print);
16         [priority(2)]/ {
17             cout <<"Anonymous print:"<< print ;
18         }
19      }
20
21     int main(int argc, char *argv[]){
22         ActionCodeTimeConstraintsTest actionCodeTimeConstraintsTest;
23         actionCodeTimeConstraintsTest.constraintTest("something");
24     }
25 }
```

Snippet 4-34. Time constraints at the action code level

In terms of code generation, the active method is triggered using the time constraints passed, as well as the method parameters (Snippet 4-35 - Line 3). *Active* instances are created to wrap anonymous and regular functions (Lines 5-12); more details were in Section 4.6.

``

```
1  //This portion is from ActionCodeTimeConstraintsTest.cpp file                    C++
2  void _constraintTest(string print){
3       activeMethod(print,1,100,9000);
4
5      Active<ActionCodeTimeConstraintsTest, void, string> active_constraintTest_regularMethod_1(this,
6           &_internalScheduler, &ActionCodeTimeConstraintsTest::regularMethod);
7      active_constraintTest_regularMethod_1(print);
8
9      Active<ActionCodeTimeConstraintsTest, void, string>
10         active_constraintTest_constraintTest_anonymous_part1(this, &_internalScheduler,
11          &ActionCodeTimeConstraintsTest::constraintTest_anonymous_part1);
12     active_constraintTest_constraintTest_anonymous_part1(print, 2);
13 }
14
15 void _activeMethod(string value){
16      cout << "Print:" << value;
17 }
18
19 void _regularMethod(string value){
20    cout << "I am not active but will be called actively. Print:" << value;
21 }
22
23 void ActionCodeTimeConstraintsTest::constraintTest_anonymous_part1(string print){
24    cout << "Print:" << print ;
25 }
```

Snippet 4-35. Portions of the generated code for Snippet 4-34

4.8.2.2 Case 2: Period time constraints

Snippet 4-36 shows examples of using the period time constraint at the action code level, on an action method (Line 18), nonactive method (Line 19), and anonymous function (Line 20). Period can be used with other time constraints, but a method must not have parameters similarly to the time constraints at the operation level; otherwise, it will be useless (Line 25).

``

```
1  class ActionCodeTimeConstraintsTest{                                    Umple
2      void active someActiveMethod(){
3          cout << "Active interval";
4      }
5
6      void active someActiveMethodWithParameters(String s1){
10         cout << "Interval will not work";
11     }
12
13     void someMethod(){
14         cout << "Interval";
15     }
16
17     void active periodTest(){
18         [period(1000)]/someMethod();
19         [period(1000), delay(1000)]/ someActiveMethod ();
20         [period(2000), delay(2000)]/{
21             cout << "anonymous function";
22         }
23
24         //Period is useless, since the active method has parameters
25         [period(1000), delay(1000)]/ someActiveMethodWithParameters ("test");
26     }
27
28     int main(int argc, char *argv[]){
29         ActionCodeTimeConstraintsTest actionCodeTimeConstraintsTest;
30         actionCodeTimeConstraintsTest.periodTest();
31     }
32 }
```

Snippet 4-36. Period at the action code level

Period has special code generation handling, since it relies on the *AsyncMethod* API. We create an AsyncMethod instance as a field (Snippet 4-37 - Lines 2-5), such that upon invoking the method, a previous interval will be terminated in order to avoid memory leaks (Lines 24-27); a new call for the start API terminates a previous start call.

Since the active method in (Snippet 4-36 - Line 25) has parameters, we call it directly and ignore the period set (Snippet 4-37 - Line 27); a warning is also shown by the compiler.

``

```
1  //This portion is from ActionCodeTimeConstraintsTest.h file                    C++
2  AsyncMethod<ActionCodeTimeConstraintsTest> listen_periodic_periodTest_someMethod_1;
3  AsyncMethod<ActionCodeTimeConstraintsTest> listen_periodic_periodTest_someActiveMethod_2;
4  AsyncMethod<ActionCodeTimeConstraintsTest> listen_periodic_periodTest_periodTest_anonymous_part1;
5  AsyncMethod<ActionCodeTimeConstraintsTest> listen_periodic_periodTest_someActiveMethodWithParameters_4;
6
7  //This portion is from ActionCodeTimeConstraintsTest.cpp file
8  ActionCodeTimeConstraintsTest::ActionCodeTimeConstraintsTest() :
9      someActiveMethod(this, &_internalScheduler,
10 &ActionCodeTimeConstraintsTest::_someActiveMethod),
11     someActiveMethodWithParameters(this, &_internalScheduler,
12 &ActionCodeTimeConstraintsTest::_someActiveMethodWithParameters),
13    periodTest(this, &_internalScheduler, &ActionCodeTimeConstraintsTest::_periodTest),
14    listen_periodic_periodTest_someMethod_1(this, &_internalScheduler,
15 &ActionCodeTimeConstraintsTest::someMethod, 1000),
16    listen_periodic_periodTest_someActiveMethod_2(this, &_internalScheduler,
17 &ActionCodeTimeConstraintsTest::someActiveMethod, 1000, 1000),
18    listen_periodic_periodTest_periodTest_anonymous_part1(this, &_internalScheduler,
19 &ActionCodeTimeConstraintsTest::periodTest_anonymous_part1, 2000),
20   someActiveMethodWithParameters(this, &_internalScheduler,
21 &ActionCodeTimeConstraintsTest::_someActiveMethodWithParameters),{
22 }
23 void _periodTest(){
24      listen_periodic_periodTest_someMethod_1.start();
25      listen_periodic_periodTest_someActiveMethod_2.start();
26      listen_periodic_periodTest_periodTest_anonymous_part1.start();
27      someActiveMethodWithParameters("test",0,1000);
28 }
```

Snippet 4-37. Portions of the generated code for Snippet 4-36

4.8.2.3 Case 3 Logical constraints at the action code level

Logical constraints at the code level means the AO will periodically check if constraints are satisfied, and then execute the action code (Section 4.7.2). For instance, in Snippet 4-38, there are two anonymous triggers (Lines 6 and 13). The second one has higher priority (Line 12), so it will be called first once the flag is set to true (Line 23).

``

```
1  class LogicalTimeConstraintsTest {                                    Umple
2      Boolean flag= false;
3
4       void active activeMethod{
5           [flag==true, priority(1)]/{
6               cout << "I will be called second" ;
10          }
11
12          [flag==true, priority(2)]/{
13              cout << "I will be called first" ;
14          }
15          [flag==true, period(1)]/{
16              cout << "I will be called repeatedly when flag is true" ;
17          }
18      }
19
20      int main(int argc, char *argv[]) {
21              LogicalConstraintsTest logicalConstraintsTest;
21              logicalConstraintsTest.activeMethod();
22
23              logicalConstraintsTest.setFlag(true);
24      }
25 }
```

Snippet 4-38. An example of using time constraints with logical conditions at the action code level

Logical conditions are applied on both active method and scheduler levels (Section 4.7). However, when a period constraint is present, logical constraints are applied on the active method only (Snippet 4-39 - Lines 12-14 and Lines 22-24). Simply, there is no need to add an unnecessary overhead to the scheduler, which is ideally used for tasks that execute once. Second, since a periodic method always has a void return type, we can simply wrap a code block within logical conditions (Line 13).

``

```
1  //This portion is from LogicalTimeConstraintsTest.h file                    C++
2  AsyncMethod<LogicalTimeConstraintsTest> listen_periodic_activeMethod_activeMethod_anonymous_part1;
3
4  //This portion is from LogicalTimeConstraintsTest.cpp file
5  LogicalTimeConstraintsTest::LogicalTimeConstraintsTest(const bool& aFlag) :
6  listen_periodic_activeMethod_activeMethod_anonymous_part1(this, &_internalScheduler,
7  &LogicalTimeConstraintsTest::activeMethod_anonymous_part1_logic, 1),
8  ...
9  }
10
11 void LogicalTimeConstraintsTest::activeMethod_anonymous_part1_logic(){
12     if(flag) {
13         activeMethod_anonymous_part1();
14     }
15 }
16
17 void LogicalTimeConstraintsTest::_activeMethod(){
18    ...
19     listen_periodic_activeMethod_activeMethod_anonymous_part1.start();
20 }
21
22 void LogicalTimeConstraintsTest::activeMethod_anonymous_part1(){
23     cout <<"I will be calledrepeatedly when flag is true" ;
24 }
```

Snippet 4-39. Portions of the generated code for Snippet 4-38

4.9 Comparison among different specifications

In this section, we give a comparison among common specifications used to manage time. We aim in this comparison to show how our implementation of concurrency and active objects can cover the core requirements specified in those specifications.

The specifications that we are going to highlight include UML, AUTOSAR, EAST Architecture Design Language (ADL), and Modelling and Analysis of Real Time and Embedded systems (MARTE).

EAST-ADL is aligned with AUTOSAR specifications [62], and it describes automotive and electronic systems via information models that capture information as standardized forms. MARTE is an OMG standard for embedded applications and real-time modelling [14]; it is aligned with UML specifications.

We will only focus on the latest specification documents of each of the specifications mentioned above; UML 2.4.1 [12], AUTOSAR 4.X [13], MARTE 1.X [14], and EAST-ADL 2.1.X [15]. MARTE extends UML in order to have better handling for embedded systems. EAST-ADL extends Systems Modelling Language (SysML).

The comparison is shown in Table 4-5, in which there are two levels of comparison, operating systems and modelling levels. The comparison criteria at the operating system level are timing extensions, activation events, and scheduling. The comparison criteria at the modelling level are *composition*, *synchronization semantics*, and *trigger*.

``

The *timing extension* criterion consists of subcriteria, *timing models*, *time constraints*, *time expressions*, and *synchronization*. By a timing model, we refer to the common way that an item in the table relies on or constrained by. In UML, the root package used to handle time is SimpleTime [12].

In terms of a *timing model* for AUTOSAR, AUTOSAR started to have timing extensions since its 4.X release [63]. In EAST-ADL, Time Augmented Description Language (TADL), a timing model language for time constraints is used [64]. In MARTE, temporal properties are handled using the "Time" package [65]; this package is often used with a non-normative annex of MARTE, Clock Constraint Specification Language (CCSL). On the other hand, a timing model in Umple is handled directly using Umple constructs.

In terms of *time constraints*, UML uses Object Constraint Language (OCL) [66]. In AUTOSAR, the timing extensions provide some possible constraints or variables to be included. For example, a timing unit can be included to determine if time will be handled in seconds, milliseconds, or nanoseconds. Examples of variables include angular position, wheel movement, and engine variables. In EAST-ADL, timing requirements are handled using the metaclass TimingRestriction, which is used to handle time constraints.

In MARTE, there are physical and logical constraints. Both are handled using a clock model, which is handled at the model level, mainly using the abovementioned CCSL. In Umple, we follow OCL semantics when defining physical or logical constraints. We already showed how we define logical and time constraints in Umple code (Sections 4.7 and 4.8).

By *time expressions*, we refer to constructs provided by a language or specification to create time expressions; e.g. to define time constraints. Time expressions are limited in UML, but a specialized type of sequence diagrams, a *timing diagram* is used to handle time constraints and expressions [66].

As well, time expressions are limited in AUTOSAR but some time expressions can be used and added to event chains. Examples include actuators, and minimum and maximum rates of delay, jitter, and repetition [67]. As clarified before, EAST-ADL is aligned with AUTOSAR specifications; thus, time expressions are handled in a similar manner. MARTE provides direct ways to define several time expressions such as conditional assertions and jitter.

We showed before that we have three levels to handle time expressions, task, queue, and scheduler levels; constructs used to define time expressions were summarized in Section 4.8. Those expressions are used to support an end-to-end flow; such a flow will be explained in the discussion of *scheduling*, which will come later in this section.

Synchronization refers to a way used to enforce timing requirements and data flow among channels and events. Examples of synchronization include defining a maximum data rate between input and output events, maximum and minimum jitter, time interval, and absolute and relative duration. An absolute duration refers to hard real-time

``

requirements, which do not accept any sort of delays. A relative duration refers to soft hardware requirements, which accept delays using concepts such as jitter and latency. Latency refers to amount of time taken for transmission between source and target; e.g. response. Jitter varies over time, since it refers to the variation of latency over time, such as in milliseconds. Stable connections have less jitter [68].

Synchronization can be either enforced on input or output events. Output synchronization is supported by all items in our comparison. In UML, synchronization is handled via activity and sequence diagrams. Synchronization is handled as a chain of events in AUTOSAR using a synchronized time-base manager [11], [67].

In EAST-ADL, the InputSynchronization and OutputSynchronization functions are used to handle synchronization requirements. In MARTE, the package TimedConstraint handles both input and output synchronization. In Umple, we handle synchronization using time constructs and the call/then pattern (Section 4.8).

Event management is the way method invocation is handled. Method invocation is temporal and event-oriented so we prefer to refer to the whole process as event management. The subcriteria of our comparison include *repetition, reaction, delay, periodic*, and *other*.

Repetition means making the same calls or invocation several times over a period. However, a repetition rate does not necessarily refer to a repeated sequence of events; it also refers to receiving events from different places such as ports, at the same time. In such a case, the appropriate guards and logical conditions must be applied in order to ensure data acceptance.

A clock port is a good example to describe handling repetition rates. For example, every two seconds a port can receive multiple signals at the same time. In such a case, the port must provide a way to recognize these signals incoming from different places, and properly process them in the right sequence based on the logical and physical constraints.

Reaction is self-explanatory as it refers to the reaction to events or method invocation; this reaction can also be as sending new signals or making new method invocation. Predefined constraints or guards are important to manage reactions.

Delay refers to how to handle delays that are either unintentional or intentional. By intentional delay, we mean that a developer intentionally wants a delay to occur. Unintentional delay refers to delays that occur because of unexpected circumstances such as networking; examples of handling related to this context include jitter, latency, and timeout. *Periodic* refers to the appropriate ways used to handle delays and repetitions such as jitter and latency. By *other*, we refer to any other general terms or additional keywords provided by specifications.

``

UML does not provide a direct way to handle the abovementioned concepts of event management. A developer will need to implement their event management mechanisms manually. For example, they will need to implement a clock port manually. Event signals are done using diagrams such as state machine, sequence, and composite structure. Concepts such as repetition will be done manually such as using for loops.

AUTOSAR provides a way to handle event management as event chains. There are additional ways to handle a task as a burst, sporadic, concrete, or arbitrary task. EAST-ADL provides functions such as RepetitionRate, "Reaction", "Execution", and "Periodic". EAST-DL relies on SysML as we previously mentioned. Additionally, SysML and MARTE CCSL can be used to execute EAST-ADL timing requirements [69].

In MARTE, a base class, TimedConstraint is used to handle event management. MARTE provides several options to manage events as compared to AUTOSAR, using additional concepts such as burst, aperiodic, sporadic, time intervals, and workload generator.

We already explained in Section 4.8, how we manage events in Umple. In terms of reactions, we rely on the call/resolve/then patterns explained in Section 4.6.5. Generally, we rely on OCL constructs to build guard conditions.

Scheduling refers to the way that events are scheduled for a period. Scheduling is important to handle timing constraints. In UML, sequence and activity diagrams can be used to handle the sequence of events. Concepts such as join and fork can be used to enable creation or merging of multiple paths of execution. AUTOSAR handles scheduling as chains of events; similarly, fork and join are used to manage the number of tasks used.

On the other hand, both EAST-ADL and MARTE have end-to-end flows to handle scheduling. An end-to-end flow is a way that enables method invocation from different places or diagrams according to a sequence. A sequence in this context means what method(s) to be invoked next upon method execution. As well, both EAST-ADL and MARTE use the fork and join approaches.

Similarly, in Umple, we support end-to-end flows. For instance, we can invoke a state machine method from an active method. Joining and forking are as well supported in Umple. For example, we can define multiple code blocks in an active method, or multiple regions in a state machine.

At the modelling level, we focus in our comparison is on the context of platform, analysis, resources, and workflow behaviour. This is summed up to three comparison items, *composition, synchronization semantics*, and *trigger*. Generally, *composition* is handled using composite structure diagrams. All items in our comparison rely on interfaces, mainly as provide or require ports.

``

AUTOSAR follows the same concepts as Umple that we discussed in Chapter 2 using R-Ports and P-Ports. As well, in AUTOSAR, communication is done as chains of events. Composition rules and references in AUTOSAR can be defined as runnable entities, tasks, and operations. The common ports used in MARTE include FlowPort and MessagePort [70]. Additionally in Umple, developers are able define composition rules such constraints, guards, and mutexes.

Synchronization semantics refers to the semantics followed for synchronization processes; we mentioned earlier in this section what we mean by synchronization. In UML, the default semantics is Run-to-Completion (RTC), since active objects are not a part of the UML constructs. For example, direct calls for state machines will have RTC behaviour.

In AUTOSAR, the basic semantics of behaviours are either reaction or data age [67]. Reaction semantics can be first-to-first, which means that each data item is processed on reception, or first-to-last, which means that data items are all received and then processed as a group. Data age semantics refers to the last-to-first behaviour, which means that we wait for expected data up to a maximum delay [67] and if it is not received, we then process the earlier-received data anyway. In EAST-ADL and MARTE, synchronization depends on notifications occurring from read and/or write operations. Such operations require using an appropriate locking mechanism.

We support all types of semantics including RTC, since we support the four types of communication. For example, we can invoke state machine methods via asynchronous methods. As well, we support the AUTOSAR semantics "first reaction" and "data age". This is mainly because in Umple, semantics can be written directly at the code level. As well, synchronization can be applied on events, which can be aligned in a prioritized queue.

A *trigger* is a well-known concept that refers to a method or procedure to be invoked upon a condition or event. In UML, triggers are defined at the level of classes and state machines [71]. In AUTOSAR, triggers are represented as a number of RTEEvents. In EAST-ADL, trigger behaviour is defined as a trigger function that receives an event at a specific time; an example of this function is shown in Table 4-5. In MARTE, triggers are defined as "Trigger" objects.

Triggers in Umple can be defined using call/'then/resolve patterns or state machines. The process of triggering has more variations in distributed environment such as Communication Area Networks (CANs) and FlexRay protocols [72].

``

Table 4-5. A comparison of time management specification

			UML	AUTOSAR	EAST-ADL	MARTE	Umple
Operating system level	Timing extensions	Timing model	SimpleTime	Timing extensions	TADL	Time Package	Language constructs
		Time constraints	OCL	Time units and other vehicle variables	Timing requirements (TimingRestriction metaclass)	Logical and physical constraints	Logical and physical OCL constraints Time units and other accessible variables
		Time expressions	Limited (timing diagrams)	Actuators, minimum and maximum rates of delay, jitters, and repetitions		Conditional assertions and Jitters	Supported on end-to-end flows and action code
		Synchronization: Input	Limited (sequence and activity diagrams)	Supported	OutputSynchronization	Supported (TimedConstraint)	Language constructs
		Synchronization: Output			InputSynchronization		
	Event management	Repetition	Limited	Supported	RepetitionRate	TimedConstraint	Poll
		Reaction			Reaction		call/then/resolve
		Delay			Execution		Delay
		Periodic			Periodic		Timeout, period, and priority
		Other		Burst, sporadic, concrete, arbitrary	SysML and CCSL	Burst, aperiodic, sporadic, time intervals, and workload generator	Delay, guards, and constraints
	Scheduling		Sequence and activity diagrams	Event chains	End-to-end flows		End-to-end flows Support other behaviour Umple models. Active blocks
Modelling level	Composition		Interfaces (provides and requires) Runnable entities, takes, operation	Event chains composition rules	Interfaces (provides and requires)		Interfaces (Provides and Requires) Support composition rules through active invoke blocks.
	Synchronization semantics		RTC	First Reaction (first-to-first , first-to-last) Data age (last-to-first, last-to-last)	Depends on synchronization (read/write)events		Depends on synchronization (read/write) events and the precedence of their logical relationship Supports RTC Supports the four types of communication
	Trigger		Classes and state machines	RTE Event	Trigger Behavior:FunctionTrigger {TIME, EVENT}	Trigger	Trigger Blocks Call/'then/resolve patterns

``

Chapter 5 Composite Structure

In this chapter, we discuss the core contribution of this research, the introduction of composite structure into Umple as well as distributed development support. Existing modelling tools provide weak or no support for the rich semantics of composite structure, such as enforcing connection constraints or referential integrity, which require more effort to manage in distributed applications than in single-node applications. Tools that generate code from composite structure typically depend on excessive and complex internal class representations (Chapter 2). In this chapter, we present a compact syntax describing composite structure in Umple. We describe a novel protocol-free approach that dynamically extracts communication protocols as a way to simplify development, leading to concise and comprehensive generated code. Our composite-structure implementation is designed to work with distributed applications and can handle different levels of complexity and networking paradigms such as client/server and peer-to-peer (P2P). We outline Umple composite structure features, as well as related code generation patterns that resolve difficulties around connections and the integrity of multiplicity constraints.

5.1 Introduction

Composite structure features are introduced to Umple as a part of this research, as a major step towards the development of connected embedded devices. Composite structure development refers to the implementation of concurrent components that interact and communicate via ports and connectors.

In UML, interaction and communication among components are handled as *messages* and *signals*. The UML metamodel provides two message passing actions, *one-way* call (asynchronous call) and *Remote Procedure Call* (synchronous call). An asynchronous call is referred to as a one-way message passing, since it does not support a scheduling mechanism to receive results. Typically, events are triggered when receiving messages by invoking a corresponding method. Event handling is normally done in a component's state machine.

Open-source tools such as eTrice and ArgoUml do not support all the major features of composite structure [73]. On the other hand, commercial tools that provide strong support to composite structure typically restrict users to certain libraries, such as Connexis, which is tightly integrated with RSARTE [74].

Handling message flow among ports and components usually depends on protocols. Typically, defining a protocol involves repetitive steps with redundant information to define how in and out events are managed. Hence, unnecessary complexity is added to the development process.

Motivated by the above limitations, we show in this chapter how Umple, an open-source tool, supports composite structure and overcomes the limitations.

Our contributions in this chapter can be summarized as follows:

``

- We support major composite structure features using compact keywords and shortened syntax.
- We provide generic extensible communication and transport definitions to support distributed examples. We implemented a support for the TCP/UDP communication protocol and JSON as a message interchange format to be the default medium for communication among distributed components.
- We incorporate Active Object to extend and enable specifying the request-scheduling mechanism.
- We implemented a protocol-free approach, in which protocols are generated from ports, connectors, and components.
- Our implementation works for distributed applications, and does not restrict users to certain network paradigms or transport data format.

In this chapter, we assume that the target language selected is C++.

5.2 Composite structure

Components, ports, and connectors are used to describe structural implementation of an object, while state machines are used to define its behaviour. A component is viewed as an active object entity with a Unique Identifier (UID) that handles its own thread of execution and encapsulates its well-defined behaviour.

The implementation of composite structure in Umple mainly depends on the active object pattern (Chapter 4). In Umple, an active object class is referred to as a component. Communication among components is established via ports, protocols, and connectors.

Each component has a public interface, UID, *internal router*, connection type, and transport data format. An internal router is used to manage components in a distributed system.

We use Template Meta-Programming (TMP) to avoid stub generation, and to provide an extensible communication stack (discussed in Section 4.3).

The public interface refers to the methods exposed for communication, which can be *synchronous*, *asynchronous*, or *future asynchronous*. They are internally represented as a generic template proxy that enables a publish-subscribe mechanism, and uses the internal router of their owning class. Standard public methods are *synchronous*, while port public interface methods are *asynchronous*. A special case is *future asynchronous*, which represents an active method with a return type (Section 5.4.6).

A component handles both Inter-Process Communication (IPC) and Remote method invocation (RMI). The internal routing structure provides the essential blocks to ease the building of communicable components. The internal routing table is used to handle send-reply and request-scheduling mechanisms between component's internal

``

methods and respondents. A component can be either in local or distributable state. Being in distributable state means that either the component is listening (acting as a server) or initiating communication (acting as a client). There is no restriction to a specific network paradigm or multi-party communication.

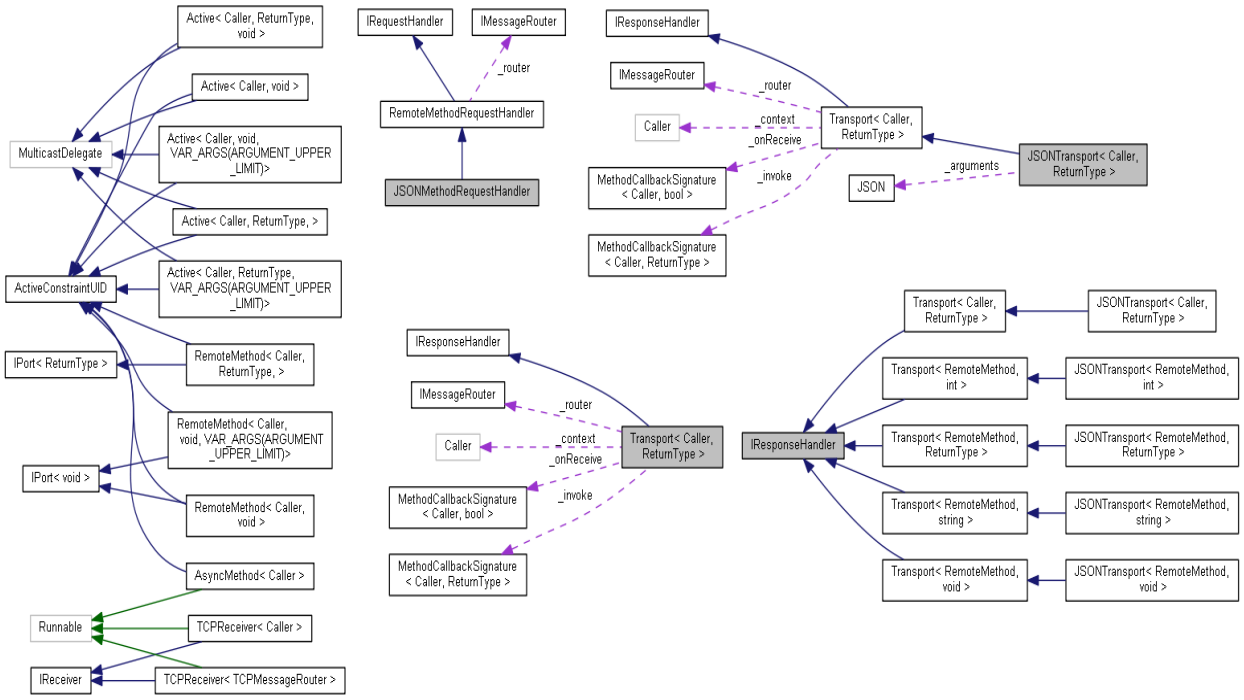


Figure 5-1. The class diagram of connection mechanisms and transport formatter

This diagram was generated by Doxygen: Solid lines are generalizations; dashed lines are compositions.

A communication stack provides abstract definitions for *connection* mechanisms and data interchange *transport* formatters. By default, we support TCP/IP as a connection mechanism, and JSON as a data interexchange format. The code can be easily extended to support different connection mechanisms such as UDP and Bluetooth, and other formats such as XML.

5.3 Composite Structure in Umple

Grammar 3 shows a RailRoad snippet to describe the composite structure syntax, which completes Grammar 2 (Chapter 4).

Figure 5-2 shows a class diagram of the metamodel of Umple composite structure.

``

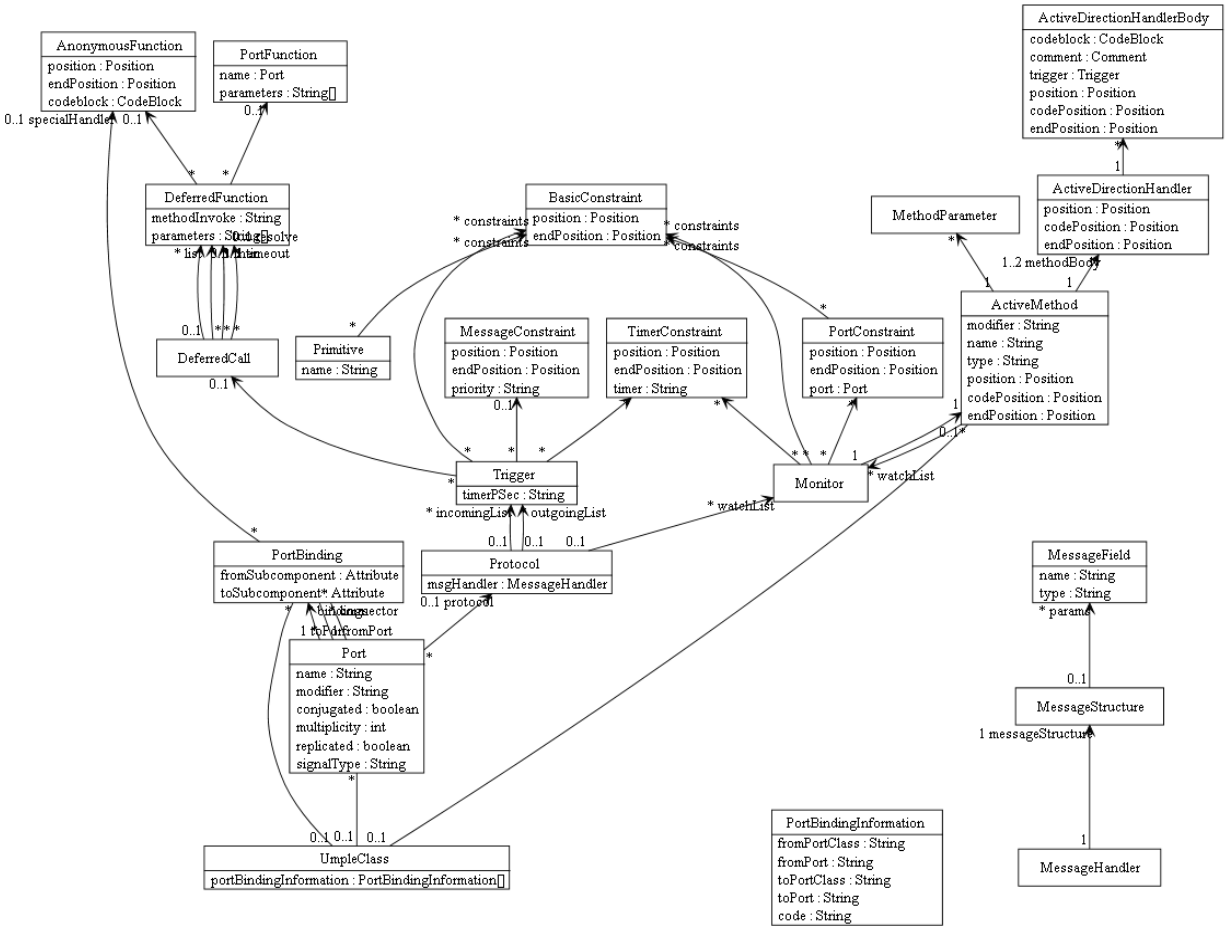


Figure 5-2. The class diagram of Umple composite structure

Grammar ::= (portDefinition | portBindingDefinition)*
// Port Definition
portDefinition ::= (modifier)? (inverse)? (portDirection) activeMethodDeclarator
modifier ::= 'public' | 'protected' | 'private'
portDirection ::= 'in' | 'out' | 'port'
inverse ::= 'conjugated'
typedPortName ::= (type)? portName (portMultiplicity)?
portMultiplicity ::= [0-9]+ | '*'
// Port Connector
portBindingDefinition ::= (fromPort) '->' (toPort)

Grammar 3 Composite structure. Details are in Appendix A

The snippets we show in the next subsections are used to clarify the keywords used in Umple in general. Most of these snippets do not show the entire code as this will take a massive amount of space.

``

5.3.1 Components

A class becomes a component if it has at least one active method, port, or connector. An active method is defined as a regular method proceeded by the *active* keyword. In Snippet 5-1, there is a component defined with two active methods (Lines 2 and 5). When invoking an active method, it executes asynchronously, since it has its own thread (Chapter 4).

Umple

```
1  class ComponentExample {
2      active method {
3            cout <<"Method without parameters"<< endl;
4      }
5      active parameterizedMethod(int someParam){
6           cout << " Parameter value"" << someParam <<endl;
7       }
8  }
```

Snippet 5-1. An example of a component definition

5.3.2 Parts (subcomponents)

A *part* is an instance of a component, and it is owned by the component structure of some component. The instance type of a part, or subcomponent, can be the same as its owning component; this is similar to the programming patterns, in which an instance of a class is created in that class's definition in places such as constructors, attributes, or methods. When a component owns multiple parts, it is referred to as a composite component. A subcomponent can possibly be composite. A component has a composition relationship to each class typed by its owned parts.

In Figure 5-3, the parts "a" and "b" are instances of A and B respectively, and they are owned by a component "c" of the type C. The Umple code is in Snippet 5-2. This means that C has composition relationships to A and B.

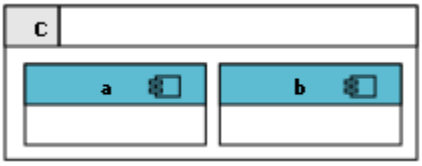


Figure 5-3.Multiple instances of different components

Umple

```
1  class A { // A component
2     active method1 {/* Empty */}
3  }
4  class B {
5     active method2 {/* Empty */}
6  }
7  class C {
8    A a;
9    B b;
10 }
```

Snippet 5-2. Multiple instances of different components

``

5.3.3 Ports

A *port* in Umple is defined as an attribute, and additionally has a direction, which can be *in*, *out*, or both (specified using the keyword *port*). A port attribute is *lazy*. In Umple, lazy attributes are not initialized through their owning class's constructor.

The keywords in, out, and port are used to set a port direction (Snippet 5-3 - Lines 2-4).

Umple

```
1  class Component{
2  in Integer inPort;
3  out Integer outPort;
4  port Integer dualPort;
5  internal in Integer privatePort;
6  in SomeClasssomeComplexPort;
7  port CompoundPort compoundPort;
8     CompoundPort active compoundPortWithActiveMethods() {
9        [someInPort]
10       active void handleDefaultDirection(){}
11       ->handleConjugatedDirection(){
12             //The inversed version CompoundPort will be as below
13             //out Integer someInPort;
14             //int Integer someOutPort;
15       }
16 }
17
18 void active someMethod(){
19     stateEvent(pIn1 + 1);
20   }
21
22   pIn1Statemachine{
23     receive{
24       stateEvent(int val) /{cout<< val;} ->done;
25     }
26     done {}
27   }
28 }
29 class SomeClass{}
30 class CompoundPort{
31    in Integer someInPort;
32    out Integer someOutPort;
33 }
```

Snippet 5-3. **Port examples**

A port has visibility since it is defined as an attribute. A private attribute in Umple is defined using the internal keyword (Snippet 5-3- Line 5); by default an attribute is public (Snippet 5-3- Lines 2-4). Private ports can only be accessed by their owning component.

``

A port can be simple, complex, or compound. A port is *simple* if its attribute type is simple such as string, integer, or double (Snippet 5-3- Lines 2-5); otherwise, it is considered *complex* (Line 6). A *compound* port consists of a number of subports, as a way to encompass a number of events for transmission (Lines 7, 8, and 30-33).

A port attribute type has no restrictions. For instance, a port attribute can be typed by a component.

5.3.4 Connectors

A connector associates between two ports in order to establish a communication channel for data transmission. A class is considered a component if it has a connector defined, even if this class does not own active methods or ports.

The operator "->" is used to define a connector, such that the port on the left is the source, and the port on the right is the target. A connector can associate between ports in the same component (Snippet 5-4 - Line 8) or different components (Lines 24and 25). The composite structure of "D" defined in Snippet 5-4 is visualized in Figure 5-4 (this is generated by UmpleOnline).

A connector can only connect between ports if they have opposite directions; i.e. an in port versus out port, dual port versus in port, dual port versus out port, and dual port versus dual port.

Umple

```
1  class A{
2     out Integer outPort;
3  }
4
5  class B{
6     in Integer inPort ;
7     out Integer outPort;
8     inPort -> outPort;
9  }
10
11 class C{
12    in Integer inPort;
13 }
14
15 class D{
16   A a;
17   B b;
18   C c;
19
20 void active someMethod(){
21      a->outPort(120, 10); // C++ code; the priority is set to 10
22 }
23
24   a.outPort->b.inPort;
25   b.outPort->c.inPort;
26 }
```

Snippet 5-4. **Connector examples**

``

The notion of "->" that we use to define associations and connectors can be confusing to C/C++ developers, since it is exactly the same as using pointers (Snippet 5-4 - Line 21). For future research, we will look into trying to reduce the need to having to use pointers in the users' extra code.

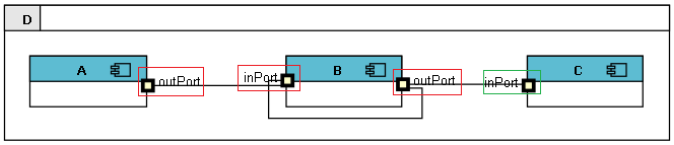


Figure 5-4. Composite structure of connected components

End ports are coloured in green, and relay ports are coloured in red.

5.3.5 Protocols

Typically, an active method uses port *attaches* to listen to port events (Snippet 5-8 - Line 6). In our protocol-free approach, a port attach holds the information about incoming and outgoing ports, which makes it possible to generate protocols based on such information.

In terms of code generation, a protocol class is generated to handle communication for each component via its ports. When a port type is complex, we apply an appropriate serialization/deserialization technique. A port value is serialized into an intermediary object transmitted in the form of messages. When messages transmitted are received, they are deserialized back to the original object form.

Data is sent through connectors as signals. A port is expected to receive signals simultaneously from different connectors. This means that there must be a queue mechanism to handle signals appropriately based on their priority and/or receiving order.

In our implementation, we have a priority FIFO queue, in which requests are ordered based on their priorities (i.e. Snippet 5-4 - Line 21), and then based on their receiving order.

At the level of the generated code, queuing depends on an internally generated helper class, MessageService. The utility class MessageService is also generated when processing any Umple model that uses composite structure features.

Communicating components can exist in different applications or locations. Hence, it was important to support buffered message transmission. This is handled using a generic API MessageDescriptor we implemented, which is also generated as needed. MessageDescriptor follows a publisher- subscriber pattern. MessageDescriptor and MessageService APIs are available in UmpleOnline when generating an Umple model that has ports or connectors defined.

``

By default, the maximum size of a transfer request is 512 Kilobytes. If a message to be transmitted exceeds this maximum size, it will be divided into a number of smaller chunks, such that each chunk size will not exceed that size. When all chunks are received, they will be assembled into a message, which will be deserialized into the form of the object data originally sent. We selected a small maximum size in order to make sure that it will not exceed the Maximum Transmission Unit (MTU) of a network, such that it will take less memory and process fast. For future work, we will experiment with other values to see which can be better, and investigate whether we can adjust the value at the model level.

MessageService works closely with the publisher-subscriber API existing in MessageDescriptor. Upon receiving incoming events, new messages will be created based on a subscriber list, which contains the active methods subscribed. The created messages will be added to the message queue using the MessageService API.

5.3.6 Port types

A port is defined as an attribute, meaning it can be simple such as string and integer, or complex. A port itself has a type that can be either *conjugated* or *base*. By default, a port is base (Snippet 5-3- Lines 2-6). A conjugated port (Line 11) can be alternatively referred to as an invert port. For instance, a conjugated in port also acts as an out port, and a conjugated out port also acts as an in port. Conjugation is only used with compound ports. When a port is compound, its conjugated version will have all of the subports inverted, as commented in (Snippet 5-3- Lines 11-15).

An in or dual port can additionally be a *relay* or *end* port [75]. Relay and end ports are called *external* ports. Any out port is a relay port (Snippet 5-4 - Lines 2 and 7). Ports that propagate signals to other ports are considered relay ports (Line 6). Signal propagation stops at *end* ports (Lines 12).

The process of signal propagation changes whether a port is a *service* or *nonservice* port, and whether it is a *behaviour* or *nonbehaviour* port.

A *service* port expects to receive inputs from or send outputs to its environment. Service ports are drawn on the boundary of its owning part. In Umple, public ports are considered service ports (i.e. Snippet 5-3- Lines 1-4).

On the other hand, *nonservice* ports are only visible within its part, and thus they are drawn within the internal region of its part. In Umple, private ports are considered nonservice (Snippet 5-3- Line 5).

A nonservice port can still receive or send signals to or from other components via another relay port, which will act as an intermediary port.

When a port triggers a state machine event, it is considered a behaviour port (i.e. Snippet 5-3- Lines 17-27).

``

A port cannot be nonservice and nonbehaviour at the same time. On the other hand, a service port can possibly be a behaviour or nonbehaviour port.

Associations are typically used to manage the number of port instances in a class (Section 5.4.14). A replicated port means that this port can have multiple instance. When a port is connected to other ports, it is referred to as a wired port. An unwired port can still connect dynamically to other ports during runtime.

Table 5-1 summarises the different types a port can have.

Table 5-1. Port types

Type	Description
Behaviour	Triggers state machine events
Nonbehaviour	Does not propagates signals via state machines
Complex	Encompasses a number of attributes rather than a single attribute as in simple ports
Base	A port designed to send out signals
Conjugated	A port that also defines an inverse port that operates in the reverse manner
Service	Used to communicate between ports in its environment and ports in other environments; i.e. public or external
Nonservice	Only visible within its part; i.e. private
Replicated	Can have multiple instances.
In	Provides a service for other ports. It can be conjugated, replicated, or service
Out	Requires a service from other ports. It can be conjugated, replicated, or service
Wired	Means that a port is connected to other ports at the model level
Unwired	Means that a port is not connected to other ports at the model level, but possibly can still connect at runtime

Figure 5-5 shows ports of different types visualized using UmpleOnline. We follow the notations in specifications such as UML [12] and AUTOSAR [13].

When a port is a service port, it is drawn on the boundary of the composite structure; otherwise, it is drawn within the composite structure (shown previously in Section 2.2.4). Hence, all ports in Figure 5-5 are service ports, since they are all drawn on the boundary of their owning component.

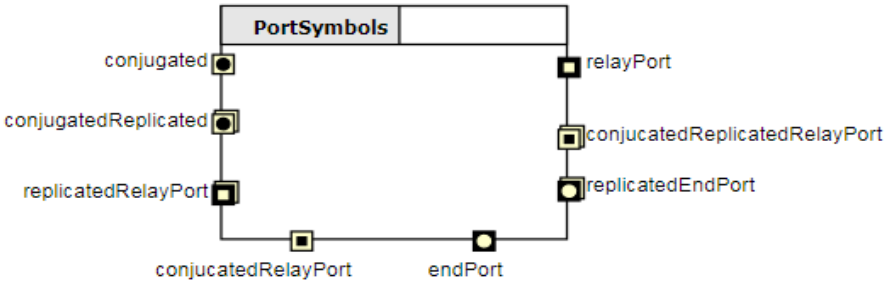


Figure 5-5. Visualization of different port types using Umple

We recognize wired ports if they are connected to other ports via connectors; Figure 5-4 for instance.

``

We distinguish between in and out ports using the crescent symbol (Figure 2-4 for instance), such that the open end of the crescent refers to the out port (Figure 2-3 - Component2), and the closed end refers to the in port (Component 1).

At the moment, we do not visually distinguish between behaviour and nonbehaviour ports, since this will require parsing users' code to check whether there are invocations to state events (Snippet 5-3- Line 19). For future work, we will assess the necessity of supporting this feature.

5.3.7 Port multiplicity

Within a component-based application, communication is established among a number of components instances. The boundary of instances created is managed using class associations, similarly to any normal application (Snippet 5-5- Line 7). Messages propagated will be received by all instances associated (Line 24).

Umple

```
1  class C{
2    in String cp;
3  }
4
5  class S{
6    out String sp;
7    * -- * C;  //Many to many association
8  }
9
10 class Sys{
11   C c;
12   S s;
13   s.sp-> c.cp;
14
15 public static void main(int argc, char *argv[]){
16     S* s= new S();
17     C* c1= new C();
18     C* c2= new C();
19     C* c3= new C();
20     s->addC(c1);
21     s->addC(c2);
22     s->addC(c3);
23     s->sp("Broadcast a message to all instances");
24   }
25 }
```

Snippet 5-5. Broadcast examples

5.4 Use cases

In this section, we show a number of use cases to handle common scenarios. We will show the important code generation elements.

``

5.4.1 Case 1: Port events

In terms of code generation, we create an event for each port, such that this event is generated as an active method (Section 4.4). This active object method is named after the port name (Snippet 5-6 - Line 1 and Snippet 5-4 - Line 2), and it has a void return type and a single parameter of the same type of the port attribute.

A user can directly send a port event by invoking the active method of a port (Snippet 5-4 - Line 22). Since, we rely on the Active API, port events can be executed simultaneously, and will benefit from the API's numerous features (Section 4.3).

```
1  Active<A, void, int> outPort;                                    C++
2     void _outPort(int data);
3     ....
4     A::A():
5        outPort(this, &_internalScheduler, &A::_outPort),
6     .....){
7  ....
8  }
9  ....
```

Snippet 5-6.Code portions of Snippet 5-4

5.4.2 Case 2: Connectors

Connectors can be established between ports at the same component (Snippet 5-4 - Line 8) or different components (Snippet 5-4 - Lines 24 and 25). Connection binding is fairly simple at the code generation level using our Active API, which applies an operator overloading mechanism (Snippet 5-7 - Lines 2 and 6-7). The connection binding is terminated automatically upon a object destruction.

```
1  void B::initPortConnections(){                                   C++
2      inPort+= &outPort;
3  }
4  ....
5  void D::initPortConnections(){
6      a->outPort+= &b->inPort;
7      b->outPort+= &c->inPort;
8  }
```

Snippet 5-7. Connection binding at the code level

5.4.3 Case 3: Binding an active method to a port

A port can be added as a constraint to an active method (Section 4.7), such that upon receiving data via that port, that active method will be invoked (Snippet 5-8 - Line 6). An active method can listen to multiple ports, such that this method will be invoked upon receiving data via any of those ports (Line 12).

``

Umple

```
1  class PortBinding{
2     in String port1;
3     in String port2;
4     Integer increment;
5
6     [port1]
7     void active someActive1{
8        cout <<"Will be called when receiving data via port1";
9        increment++;
10    }
11
12    [port1, port2, increment>10]
13    void active someActive2{
14       cout <<"Will only be called when increment is more than 10"
15            <<" and upon receiving events via port1 or port2";
16    }
17 }
```

Snippet 5-8. Examples of port binding to active methods

Similarly to connectors (Section 5.4.2), we use our API, which relies on operator overloading, to bind an active method to a port (Snippet 5-9 - Lines 2-4). Constraints of an active method will be used congruently with ports (Lines 8-10).

C++

```
1  void PortBinding::initPortConnections(){
2      port1+= &someActive1;
3      port1+= &someActive2;
4      port2+= &someActive2;
5  }
6  ....
7  void PortBinding::_someActive2(string port1, string port2){
8     if(!(increment > 10)){
9        throw "Please provide a valid increment";
10    }
11
12    cout << "Will only be called when increment is more than 10"
13    << " and upon receiving events via port1 or port2";
14 }
15
16 void PortBinding::_someActive1(string port1){
17    cout << "Will be called when receiving data via port1";
18    increment++;
19 }
```

Snippet 5-9.Code portions of Snippet 5-8

We generate protocols based on the active objects defined as well as port bindings that hold information about incoming and outgoing ports; i.e. a protocol-free approach.

``

5.4.4 Case 4: Serialization/deserialization

In remote communication, obviously data transmission of complex types is not as straightforward as simple types. In our implementation, when a port attribute is typed by a class (Snippet 5-10 - Line 3 and Lines 6-10), we create a serialization method, which serializes the values of associations and public fields of this class (Snippet 5-11 - Lines 1-16 and Lines 19-24) into an object of a standard media format (we currently only use JSON). We as well create a deserialization object that receives such an object, and transforms it into a shallow clone of the original instance (Snippet 5-11- Lines 27-43and Lines 45-52).

We make sure that we properly pass the right parameters when constructing a clone (Snippet 5-11 - Lines 2-12, 29-30, 34-35, and 39). Then, we assign values to lazy attributes (Lines 14, 31, 36, and 40). Serialization and deserialization methods have two versions, which allow passing by reference (Lines 27-51) or pointer (Lines 1-25).

Umple

```
1  distributable;
2  class Component1 {
3       out ComplexType outPort;
4  }
5
6  class Component2 {
7      in ComplexType inPort;
8      Component1 cmp1;
9      cmp1.outPort -> inPort;
10 }
11
12 class ComplexType {
13     String att1;
14     String att2;
15     lazy String att3;
16     1 --1 SomeClass;
17     internal String att4;
18 }
19
20 class SomeClass{}
```

Snippet 5-10. Simple composite structure communication

C++

```
1  JSON::operator ComplexType*(){
2     ComplexType* aComplexType;
3     if( (*this).contains("port") ) {
4        aComplexType= new ComplexType((unsigned int)(*this)["port"], (*this)["att1"], (*this)["att2"],
5         (*this)["att4"], (*this)["someClass"]);
6     }elseif ( (*this).contains("endpoint") ) {
7        aComplexType= new ComplexType((Endpoint)(*this)["endpoint"], (*this)["att1"], (*this)["att2"],
8         (*this)["att4"], (*this)["someClass"]);
9     }else {
10       aComplexType= new ComplexType((*this)["att1"], (*this)["att2"], (*this)["att4"],
11        (*this)["someClass"]);
```

``

```
12      }
13
14      aComplexType->setAtt3((*this)["att3"]);
15      return aComplexType;
16  }
17
18  JSON::JSON(ComplexType* aComplexType){
19       JSON object;
20       object["att1"] = aComplexType->getAtt1();
21       object["att2"] = aComplexType->getAtt2();
22       object["someClass"] = aComplexType.getSomeClass();
23       object["att3"] = aComplexType.getAtt3();
24       swap(object);
25  }
26
27  JSON::operator ComplexType(){
28      if((*this).contains("port")){
29          ComplexType aComplexType((unsigned int)(*this)["port"], (*this)["att1"], (*this)["att2"],
30           (*this)["att4"], (*this)["someClass"]);
31          aComplexType.setAtt3((*this)["att3"]);
32          return aComplexType;
33      }elseif ((*this).contains("endpoint")){
34          ComplexType aComplexType((Endpoint)(*this)["endpoint"], (*this)["att1"], (*this)["att2"],
35           (*this)["att4"], (*this)["someClass"]);
36          aComplexType.setAtt3((*this)["att3"]);
37          return aComplexType;
38      }else {
39      ComplexType aComplexType((*this)["att1"], (*this)["att2"], (*this)["att4"], (*this)["someClass"]);
40      aComplexType.setAtt3((*this)["att3"]);
41      return aComplexType;
42      }
43  }
44
45  JSON::JSON(ComplexType& aComplexType){
46       JSON object;
47       object["att1"] = aComplexType.getAtt1();
48       object["att2"] = aComplexType.getAtt2();
49       object["someClass "] = aComplexType. getSomeClass ();
50       object["att3"] = aComplexType.getAtt3();
51       swap(object);
52  }
```

Snippet 5-11. An example of serialization and deserialization

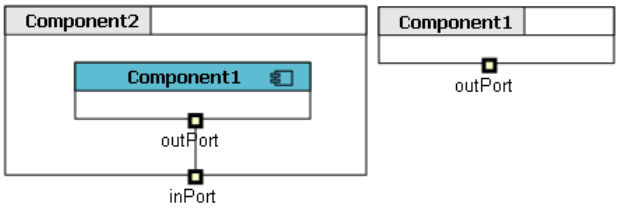


Figure 5-6. The composite structure diagram of Snippet 5-10

``

```
1  classDistributeTest{                                                    Umple
2     distributable;
3     att1;
4     att2;
5     in String somePort;
6
7     void someMethod(){
8        cout <<"Some value";
9     }
10
11    void active someActive(){
12       cout <<"Some other value";
13    }
14
15    int main(int argc, char *argv[]) {
16       DistributeTest distributeTest("Test1", "Test2");
17       distributeTest.someMethod();
18       distributeTest.someActive();
19    }
20 }
```

Snippet 5-12.A distributable keyword example

5.4.5 Case 5: Distributable execution

Distributable execution, as the name suggests, means that a class (Snippet 5-12 - Line 2) will execute in a distributed environment. When the distribute keyword is used at the model level, all classes will become distributable (Snippet 5-5 - Line 1).

In a distributed class, several changes are applied at the generation level, since we have to make sure that all of its public methods are available for distributed access. For that, we use the Remote API, which requires all of class's public methods to be private and prefixed by an underscore (Snippet 5-13 - Lines 14-25), such that remote objects will be created (Lines 3-10) and linked to these formerly public methods (Lines 33-40).

The Remote object follows the same pattern of Active, such that the return type as well as the parameter types of a linked method are specified in a field declaration (Lines 2-10). Even active objects, including port events, are associated to Remote fields (Lines 4-5 and 19-20).

```
1  //This portion is from DistributeTest.h file                            C++
2  public:
3     RemoteMethod<DistributeTest, void> someActive;
4     RemoteMethod<DistributeTest, void, string> somePort;
5     RemoteMethod<DistributeTest, void> someMethod;
6     RemoteMethod<DistributeTest, size_t, void> hashCode;
7     RemoteMethod<DistributeTest, string, void> getAtt1;
8     RemoteMethod<DistributeTest, string, void> getAtt2;
9     RemoteMethod<DistributeTest, bool, const string&> setAtt1;
```

``

```
10      RemoteMethod<DistributeTest, bool, const string&> setAtt2;
11
12  private:
13     TCPMessageRouter _messageRouter;
14      bool _setAtt1(const string& newAtt1);
15      bool _setAtt2(const string& newAtt2);
16      string _getAtt1(void);
17      string _getAtt2(void);
18      Scheduler<DistributeTest> _internalScheduler;
19      Active<DistributeTest, void> internal_someActive;
20      Active<DistributeTest, void, string> internal_somePort;
21      void _someActive();
22      void _somePort(string data);
23      void initPortConnections();
24      void _someMethod();
25      virtual size_t _hashCode(void);
26      TCPMessageRouter getMessageRouter(void);
27      ......
28      //This portion is from DistributeTest.cpp file
29      DistributeTest::DistributeTest(const string& aAtt1, const string& aAtt2):
30          internal_someActive(this, &_internalScheduler, &DistributeTest::_someActive),
31          _internalScheduler(this),
32      _messageRouter(),
33      internal_somePort(this, &_internalScheduler, &DistributeTest::_somePort),
34      someMethod(this, &DistributeTest::_someMethod, &_internalScheduler, "someMethod",
35  &_messageRouter),
36      hashCode(this, &DistributeTest::_hashCode, &_internalScheduler, "hashCode", &_messageRouter),
37      getAtt1(this, &DistributeTest::_getAtt1, &_internalScheduler, "getAtt1", &_messageRouter),
38      getAtt2(this, &DistributeTest::_getAtt2, &_internalScheduler, "getAtt2", &_messageRouter),
39      setAtt1(this, &DistributeTest::_setAtt1, &_internalScheduler, "setAtt1", &_messageRouter),
40      setAtt2(this, &DistributeTest::_setAtt2, &_internalScheduler, "setAtt2", &_messageRouter),
41      someActive(DistributeTest::internal_someActive, "someActive", &_messageRouter),
42      somePort(DistributeTest::internal_somePort, "somePort", &_messageRouter){
43                  ....
44  }
```

Snippet 5-13. Portions of the generated code for Snippet 5-12

The Remote API delegates the Active API's features, such as sending prioritized messaging (Snippet 5-23- Line 40), or leaning on the constraint features.

5.4.6 Case 6: Synchronous versus asynchronous

Distributed tools, such as RMI, do not straight forwardly support asynchronous behaviour. Specifications such as UML only assume asynchronous behaviour. Alternatively, a workaround is applied by sending a void message as acknowledgment of receiving a message.

In our implementation, which is a major advantage, we support both synchronous and asynchronous behaviours. Simply, nonactive methods are executed synchronously (Snippet 5-12 - Line 17) and active methods are executed asynchronously(Line 18).

``

5.4.7 Case 7: Communication

Classes can access a distributable class by having knowledge of its IP address and port number, which we refer to together as an *endpoint*. In a generated distributable class, we create two additional constructors, the first one receives an endpoint (Snippet 5-14 - Line 7) and the second one only receives a port number (Line 11). Typically, a server or super peer uses the latter constructor, since it is not required to provide an IP address of a machine to access. On the other hand, an endpoint constructor is used by clients who need to be aware of the IP and port of the machine they need to access. When using the default constructor, a class instance will not be available for distribute access.

```
1  //This portion is from DistributeTest.cpp file                                    C++
2  DistributeTest::DistributeTest(const string& aAtt1, const string& aAtt2):
3        .....
4  }
5
6
7  DistributeTest::DistributeTest(Endpoint ep, const string& aAtt1, const string& aAtt2):
8        .....
9  }
10
11 DistributeTest::DistributeTest(unsigned int _portValue, const string& aAtt1, const string& aAtt2):
12       .....
13 }
```

Snippet 5-14. Constructor portions of the generated code for Snippet 5-12

We designed the Endpoint API to be extendable, such that clients can support additional media types.

5.4.8 Case 8: Complex and conjugated ports

A complex port is typically used as a template encompassing a number of attributes (Section 5.3.3) as well as active methods. Hence, it can only connect to ports of its type, since they can understand each other. At this point, it becomes important that an active method will have different behaviours upon sending or receiving signals. For instance, in Snippet 5-15, we define a complex port (Lines 1-26)used by two classes, Teacher (Lines 28-30) and Student (Lines 32-34), such that Teacher acts as a sender and Student acts as a receiver (Line 39). A sender will have the forward implementations of active objects (Lines 9-11 and Line 22), and a receiver will have the conjugated implementations, which come after the -> operator (Lines 13-16 and Line 24).Teacher and Student have one-to-many associations (Line 40).

An active method is generated with a number of parameters that match the ports it listens to (Snippet 5-15 - Lines 7 and 20, and Snippet 5-16 - Lines 24, 33, 74, and 86). For instance, a teacher can ask a simple parameterized math question (Snippet 5-15 - Line 45), which consists of two numbers and an operator (Lines 2-4). A student has to answer the question (Line 5). Each parameter value is transmitted via its subport accordingly. Once parameter

``

signals are received, the inverse method will be invoked (Line 13-16). A new signal is sent via the solution port (Lines 14 and 16), which is handled forwardly by Teacher (Line 22) and inversely by Student (24).

Umple

```
1  class MathQuestion{
2     out Integer num1;
3     out Integer num2;
4     out String op;
5     in Integer solution;
6
7     [num1, num2, op]
8     void active ask(int num1, int num2, String op) {
9        cout<< num1;
10       cout<< num2;
11       cout<< op;
12       }->answer {
13          if(op== "+"){
14             solution(num1+num2);
15          }elseif(op== "-"){
16             solution(num1-num2);
17       }
18    }
19
20    [solution]
21    void active receive{
22       cout<<"Answer provided is " + solution;
23    }-> logOut{
24       cout<<"Answer sent to out port is " + solution;
25    }
26 }
27
28 class Teacher{
29    port MathQuestionquestion;
30 }
31
32 class Student{
33    port MathQuestion answer;
34 }
35
36 class System{
37    Student s;
38    Teacher t;
39    t.question -> s.answer;
40    0..1 -- * Student student;
41    public static void main(int argc, char *argv[]){
42       Student* student= new Student();
43       Teacher* teacher= new Teacher();
44       teacher->addStudent(student);
45       teacher->ask(4, 11, "+");
46    }
47 }
48 association { * Student students -- * Teacher teachers;}
```

``

Snippet 5-15. Math Question: a conjugated port example

In terms of the code generation, we inject the forward and inverse implementations into Teacher (Snippet 5-16 - Lines 24-31 and 33-43) and Student (Lines 78-84 and Lines 86-89) respectively. We make sure to propagate *forward* signals to all associated instances right after executing the base active content (Lines 26-29 and 38-42). We do not do the same for inverse active methods, since they do not need to notify the forward active methods; otherwise, we will end to infinite notifications.

We as well notify associated instances when receiving signals via a port (Snippet 5-16 - Lines 45-67 and 86-89). This time, notifications are defined at the out ports as well, which are assumingly used by inverse methods, since we allow for the assumption that a signal may start directly from a port in case a user decided not to invoke an active method directly (Snippet 5-15 - Lines 14 and 16).

Notifying an association depends on its ends, which can either be one or many. For the first case, we directly propagate signals to the single instance (Snippet 5-16 - Lines 87-88). If an association end is optional, we make sure to check for null. For the latter case, we loop through each instance, and notify it (Lines 46-50 for instance).

```
1  //This portion is from Teacher.cpp file                                    C++
2  void Teacher::logOut(int solution, vector<Student*>* copyOfStudents){
3     for (unsigned int index=0; index<copyOfStudents->size(); index++) {
4       Student* aStudent= (*(copyOfStudents))[index];
5       aStudent->logOut(solution);
6     }
7  }
8
9  void Teacher::logOut(int solution, Student* aStudent){
10    aStudent->logOut(solution);
11 }
12
13 void Teacher::answer(int num1, int num2, string op, vector<Student*>* copyOfStudents){
14    for (unsigned int index=0; index<copyOfStudents->size(); index++) {
15      Student* aStudent= (*(copyOfStudents))[index];
16      aStudent->answer(num1, num2, op);
17    }
18 }
19
20 void Teacher::answer(int num1, int num2, string op, Student* aStudent){
21    aStudent->answer(num1, num2, op);
22 }
23
24 void Teacher::_receive(int solution){
25    cout<<"Answer provided is " + solution;
26    vector<Student*>* copyOfStudents = this->getStudents();
27    for (unsigned int index=0; index<copyOfStudents->size(); index++) {
28      Student* aStudent= (*(copyOfStudents))[index];
29      aStudent->logOut(solution);
30    }
```

``

```
}

void Teacher::_ask(int num1, int num2, string op){
   cout<< num1;
   cout<<num2;
   cout<<op;

   vector<Student*>* copyOfStudents = this->getStudents();
   for (unsigned int index=0; index<copyOfStudents->size(); index++) {
     Student* aStudent= (*(copyOfStudents))[index];
     aStudent->answer(num1, num2, op);
   }
}

void Teacher::_op(string data){
   vector<Student*>* copyOfStudents = this->getStudents();
   for (unsigned int index=0; index<copyOfStudents->size(); index++) {
     Student* aStudent= (*(copyOfStudents))[index];
     aStudent->op(data);
   }
}

void Teacher::_num2(int data){
   vector<Student*>* copyOfStudents = this->getStudents();
   for (unsigned int index=0; index<copyOfStudents->size(); index++) {
     Student* aStudent= (*(copyOfStudents))[index];
     aStudent->num2(data);
   }
}

void Teacher::_num1(int data){
   vector<Student*>* copyOfStudents = this->getStudents();
   for (unsigned int index=0; index<copyOfStudents->size(); index++) {
     Student* aStudent= (*(copyOfStudents))[index];
     aStudent->num1(data);
   }
}

void Teacher::initPortConnections(){
   this->solution+= &this->receive;
}

//This portion is from Student.cpp file
void Student::_logOut(int solution){
   cout<<"Answer sent to out port is " + solution;
}

void Student::_answer(int num1, int num2, string op){
   if(op== "+"){
      solution(num1+num2);
   }elseif(op== "-"){
      solution(num1-num2);
```

``

```
83      }
84  }
85
86  void Student::_solution(int data){
87      Teacher* aTeacher= this->getTeacher();
88      aTeacher->solution(data);
89  }
90
91  void Student::initPortConnections(){
92      this->solution+= &this->logOut;
93  }
```

Snippet 5-16 Portions of the generated code for Snippet 5-15

5.4.9 Case 9: Forward versus inverse based on port directions

Utilizing forward or inverse implementation depends on port directions as well as the connectors defined. By default, if no connector exists, we assume that a class will use the forward implementation. A user can use the keyword *conjugated* to use the inverse implementation instead. Table 5-2 shows the common variations involved to decide whether to use forward implementation, inverse implementation, or both of them.

``

Table 5-2. Forward versus inverse implementations

Legend: There is a ComplexPort used by two classes, Source and Target, such that a number of variations exist when defining connectors among ports or using the conjugated keyword. The forward implementation, inverse implementation, or both are used based on such variations

	Defined ports at source	**Defined ports at target**	**Connectors**	**Source**	**Target**
1	port ComplexPort source;	port ComplexPort target;	None	Forward	Forward
	By default, the forward implementations are used when no connectors are defined				
2	port ComplexPort source;	conjugated port ComplexPort target;	None	Forward	Inverse
	If no connector exists, and a port is defined with the conjugated keyword, the inverse implementation is used (see target). Otherwise, the forward implementation is used (see source).				
3	port ComplexPort source; port ComplexPort otherSource;	port ComplexPort target; conjugated port ComplexPort target;	None	Forward	Both
	Multiple instances of the same complex port can be defined in a class. If there is at least a base and conjugated ports defined, both forward and inverse implementations are used (see target). If the multiple ports are all defined as base or conjugated, only the forward or inverse implementations are used respectively (see source).				
4	port ComplexPort source;	port ComplexPort target;	source-> target	Forward	Inverse
	If a connector exists from a source to target, and both ports to not use the conjugated keywords, the source will use the forward implementations, and the target will use the inverse implementation.				
5	conjugated port ComplexPort source;	conjugated port ComplexPort target;	source-> target	Both	Inverse
	A connector guarantees that a source will have the forward implementation, and a target will use the inverse implementation. If a target uses the conjugated keyword, nothing will change, since it is already assigned to use the inverse implementation (see target). If a source uses the conjugated keyword, it will use the inverse implementation as well as the forward implementation it is already assigned (see source).				
6	port ComplexPort source;	port ComplexPort target;	source-> target target-> source	Both	Both
	Multiple connectors can exist between a source and target. If a connector exists from the source to target, we mentioned that the source and target will be assigned forward and inverse implementations respectively. If another connector exists from the target to source, inverse and forward implementations will be assigned to source and target respectively. This means that both source and target will use both implementations.				

5.4.10 Case 10: Conjugated ports and constraints

The parameters of a conjugated active method are the same as those in its forward active method. Parameters are derived from the ports listened to (Snippet 5-15 - Line 7 and Snippet 5-16 - Line 33 and 78). We can set constraints at a forward active method (Snippet 5-17- Line 1). Such constraints are not applied at the inverse method, which can still have its own set of constraints (Line 6).

``

```
1   [num1, num2, op, flag]                                          Umple
2    active ask {
3       num1= num1;
4       num2= num2;
5       op= op;
6    }-> [someOtherFlag] answer {
7        if(op== "+"){
8          solution(num1+num2);
9       }elseif(op== "-"){
10         solution(num1-num2);
11      }
12   }
```

Snippet 5-17. An updated portion of Snippet 5-15— with constraints at forward/inverse methods —

5.4.11 Case 11: Complex port redefinition

The implementations of forward and inverse active methods at a complex port have default implementations. A component that uses a complex port can redefine such methods (Snippet 5-18 - Lines 3-5 and Lines 11-13). In terms of the code generation, a component implementation is placed after a default complex port implementation (Snippet 5-19 - Lines 6 and 20).

```
1   class Student{                                                  Umple
2       port MathQuestion answer;
3       void receive(int num1, int num2, string op){
4           cout <<"Answering a question";
5       }
6   }
7
8   class Teacher{
9       port MathQuestion question;
10      1 -- * Student;
11      void ask(int num1, int num2, string op){
12          cout <<"Asking a question";
13      }
14  }
```

Snippet 5-18. An updated portion of Snippet 5-15 — with method redefinition —

``

```C++
//This portion is from Teacher.cpp file
void Teacher::_ask(int num1, int num2, string op){
  cout<< num1;
  cout<<num2;
  cout<<op;
  cout <<"Asking a question";
  vector<Student*>* copyOfStudents = this->getStudents();
  for (unsigned int index=0; index<copyOfStudents->size(); index++) {
       Student* aStudent= (*(copyOfStudents))[index];
       aStudent->answer(num1, num2, op);
      }
  }
  //This portion is from Student.cpp file
  void Student::_receive(int num1, int num2, string op){
  if(op== "+"){
        solution(num1+num2);
  }else if(op== "-"){
        solution(num1-num2);
  }
  cout << "Answering a question";
}
```

Snippet 5-19 Portions of the generated code for Snippet 5-18

5.4.12 Case 12: Multiple association of different roles

Within a class, Umple allows defining multiple associations between two classes as long as they have unique roles (Snippet 5-20 - Lines 6 and 7). In terms of the code generation with the assumption that a connector exists, we make sure to propagate signals to all of these associations' instances accordingly (Snippet 5-21).

```Umple
class Student{
    port MathQuestion answer;
    ...
}

association { 0..1 Teacher elementaryTeacher -- * Student elementaryStudents;}
association { 0..1 Teacher hightschoolTeacher -- * Student hightschoolStudents;}
class Teacher{
    port MathQuestion question;
 ...
}
```

Snippet 5-20. An updated portion of Snippet 5-15— with multiple associations defined —

```C++
//This portion is from Teacher.cpp file
void Teacher::_receive(int solution){
  ....
  vector<Student*>* copyOfElementaryStudents = this->getElementaryStudents();
  for (unsigned int index=0; index<copyOfElementaryStudents->size(); index++) {
    Student* aStudent= (*(copyOfElementaryStudents))[index];
    aStudent->logOut(solution);
  }

```

``

```
10    vector<Student*>* copyOfHightschoolStudents = this->getHightschoolStudents();
11    for (unsigned int index=0; index<copyOfHightschoolStudents->size(); index++) {
12      Student* aStudent= (*(copyOfHightschoolStudents))[index];
13      aStudent->logOut(solution);
14    }
15  }
16
17  void Teacher::_ask(int num1, int num2, string op){
18    ....
19    vector<Student*>* copyOfElementaryStudents = this->getElementaryStudents();
20    for (unsigned int index=0; index<copyOfElementaryStudents->size(); index++) {
21      Student* aStudent= (*(copyOfElementaryStudents))[index];
22      aStudent->answer(num1, num2, op);
23    }
24
25    vector<Student*>* copyOfHightschoolStudents = this->getHightschoolStudents();
26    for (unsigned int index=0; index<copyOfHightschoolStudents->size(); index++){
27      Student* aStudent= (*(copyOfHightschoolStudents))[index];
28      aStudent->answer(num1, num2, op);
29    }
30  }
31
32  //This portion is from Student.cpp file
33  void Student::_solution(int data){
34    Teacher* aElementaryTeacher= this->getElementaryTeacher();
35    aElementaryTeacher->solution(data);
36
37    Teacher* aHightschoolTeacher= this->getHightschoolTeacher();
38    aHightschoolTeacher->solution(data);
39  }
```

Snippet 5-21 Portions of the generated code for Snippet 5-18

5.4.13 Case 13: Ports with interfaces

Ports are usually defined as interfaces specifying the communication design among protocol layers [76]. In our implementation, we do not restrict users to certain interfaces as a way to improve usability and code readability. However, users can still enforce certain specifications if required. Snippet 5-22 shows an updated version of Snippet 5-15, where we define two interfaces, ISender (Lines 1-4) and IReceiver (Lines 6-9), specifying the methods to be implemented; i.e. communication specifications. Senders (Line 28) and receivers (Line 16) will need to implement both interfaces in order to initiate an appropriate communication.

``

```
1  interface ISender{                                                    Umple
2     ask(Integer num1, Integer num2, String op);
3     receive(Integer solution);
4  }
5
6  interface IReceiver{
7     answer(Integer num1, Integer num2, String op);
8     logOut(Integer solution);
9  }
10
11 class MathQuestion{
12    .....
13 }
14
15 class Student{
16    isA IReceiver;
17    port MathQuestion answer;
18    void answer(int num1, int num2, string op){
19       cout << "Answering a question";
20    }
21
22    void logOut(int solution){
23       cout << "Question answered";
24    }
25 }
26
27 class Teacher{
28    isA ISender;
29    port MathQuestion question;
30    void ask(int num1, int num2, string op){
31       cout << "Asking a question";
32    }
33
34    void receive(int solution){
35       cout << "Answer received";
36    }
37 }
```

Snippet 5-22. An updated portion of Snippet 5-15— with interfaces —

5.4.14 Case 14: Unicast, multicast, and broadcast

The Remote API provides the required methods to handle *broadcast* (Snippet 5-23- Lines 36 and 40), *unicast* (Lines 43and 46), and *multicast* (Lines 49and 52).

```
1  classBaseClient  {                                                    Umple
2   distributable;
3   in String cMessage;
4  }
5
6  classBaseServer{
7   distributable;
8
```

``

```
9    out String sMessage;
10   * -- *BaseClient clients;
11  }
12
13  class Sys{
14   BaseClient  client;
15   BaseServer server;
16   server.sMessage->client.cMessage;
17
18  public static void main(int argc, char *argv[]){
19   BaseServer* server= newBaseServer(9441);    //Will be attached to the port 9441
20
21   //Below is an example of a remote connection but it is somehow useless, since all is on the same machine
22    Endpoint endpoint("192.168.1.1", 9441);
23    BaseClient * client1= newBaseClient(endpoint);
24    BaseClient * client2= newBaseClient(endpoint);
25    BaseClient * client3= newBaseClient(endpoint);
26    BaseClient * client4= newBaseClient(endpoint);
27
28    server->addClient (client1);
29    server->addClient (client2);
30    server->addClient (client3);
31    server->addClient (client4);
31
32    vector<BaseClient*>* group1= new vector<BaseClient*>();
33    group1->push_back(client1);
34    group1->push_back(client2);
35    group1->push_back(client3);
36
37    //Broadcast a message of a default priority, 0
38    server->sMessage("This message will be sent to all clients");
39
40    //Broadcast a message of a priority 10
41    server->sMessage("This message will be sent to all clients", 10);
42
43    //Unicast a message of a default priority
44     server->sMessage(client1, "This message will be sent to client1");
45
46     //Unicast a message of a priority 5
47     server->sMessage(client2, "This message will be sent to client2", 5);
48
49    //Multicast a message of a default priority
50    server->sMessage(group1, "This message will be sent to client1, client2, and client3");
51
52    //Multicast a message of a priority 5
53    server->sMessage(group1, "This message will be sent to client1, client2, and client3", 5);
54   }
    }
```

Snippet 5-23. Broadcast, multicast, and unicast

``

For association ends with many cardinality, we create two additional methods for each port event to handle unicast (Snippet 5-16- Lines 9-11 for instance), broadcast, and multicast (Lines 2-7 for example). For association ends of one cardinality, we create a single method to handle unicast.

5.4.15 Case 15: Multiplicity and wired connection

Associations enforce referential integrity on all connectors. Considering the following client/server example, where a server needs to limit the number of clients to 1000. A client can be either a paying or a non-paying member. We need to divide the allowed instances between these types of members (Snippet 5-24 - Lines 20 and 21), such that, for example, paying members can be up to 900 (Line 7), and non-paying members cannot exceed 100 (Line 12).

Umple

```
1  class Client{
2    in String cMessage;
3  }
4
5  class PayingClient{
6    isA Client;
7    0..900 -- 0..1 Server;
8  }
9
10 class NonPayingClient {
11   isA Client;
12   0..100 -- 0..1 Server;
13 }
14
15 class Server{
16   out String sMessage;
17 }
18
19 class Sys{
20   PayingClient payingClient;
21   NonPayingClient nonpayingClient;
22   Server server;
23
24   server.sMessage-> payingClient.cMessage;
25   server.sMessage-> nonpayingClient.cMessage;
26
27 public static void main(int argc, char *argv[]){
28     Server* server= new Server();
29     PayingClient * payingClient1= new PayingClient ();
30     NonPayingClient * nonPayingClient1= new NonPayingClient();
31     server->addNonPayingClient(nonPayingClient1);
32     server->addPayingClient(payingClient1);
33     }
34 }
```

Snippet 5-24. **Applying boundary at the connector level**

``

In many modelling tools, port multiplicity is used for design purposes while giving the possibility to dynamically change size. For instance, in UML-RT, the method resize is used to allocate more space for incoming connections [26].

5.4.16 Case 16: Incarnated components

If a mandatory association exists between two components, a part at the mandatory end of a connector between those components is referred to as *incarnated* (Snippet 5-25- Line 14). A mandatory association in Umple enforces passing an instance or instances of the class type of the mandatory end as a part of the constructor of the other end (Line 18).

Umple

```
1  class Department {
2    in String dMessage;
3    * -- 1 School; //A school must be assigned to any department
4  }
5
6  class School{
7    out String sMessage;
8  }
9
10 class Sys{
11   School school;
12   Department department;
13   //School is incarnated
14   school.sMessage->department.dMessage;
15
16 public static void main(int argc, char *argv[]){
17     School* school= new School();
18     Department* department= newDepartment(school);
19     school->sMessage("Send a message ");
20   }
21 }
```

Snippet 5-25.An incarnated part example

5.4.17 Case 17: Unwired communication

Connectors among ports and components are referred to as *wired* connections. Our underlying Active framework allows for unwired port communication during runtime (Table 5-1. Simply, we use the plus operator to bind active methods (Snippet 5-26 - Line 19), such that an active method of the left-hand is forward, the method on the right-hand side is inverse. Similarly, port events can be bounded (Line 17). As well, we can let an active method listen to port events (Line 21). The unwired communication is an additional design flavour we give to users, since it is provided by tools such as RSARTE and Connexis [74], which are closed-source.

``

```
1  class Base{                                                    Umple
2     out String sMsg;
3     in String rMsg;
4
5     void active send(String message){
6        cout <<"Send a message: "<< message;
7     }
8
9     void active receive(String message){
10       cout <<"receive a message: "<< message;
11    }
12 }
13
14 class System{
15    public static void main(int argc, char *argv[]){
16       Base* base= new Base();
17       base->sMsg+= base->rMsg;
18
19       base->send+= base->receive;
20
21       base->sMsg+= base->send;
22    }
23 }
```

Snippet 5-26. Unwired communication

5.4.18 Case 18: Networking paradigms: P2P and client/server

Users are able to select the networking paradigms or a hybrid of paradigms when writing and using their distributed models. For instance, (Snippet 5-15 - Lines 42-45) enforces P2P communication between teachers and students.

On the other hand, Snippet 5-27, an extended snippet, assumes that there is a single virtual room that any student or teacher must join (Lines 3 and 4). A student can have up to three teachers (Line 2). The communication between a student and a teacher is still P2P (Lines 17-18, 21-22, and 25-26). However, a virtual room acts as a server to students and teachers; i.e. client/server paradigm.

```
1  ....                                                           Umple
2  association { 0..3 Teacher teachers -- * Student students;}
3  association { 1 VirtualRoom room -- * Teacher teachers;}
4  association { 1 VirtualRoom room -- * Student students;}
5
6  class VirtualRoom{
7
8  public static void main(int argc, char *argv[]){
9        VirtualRoom* virtualRoom = new VirtualRoom(9441);   //The room is on port 9441
10
11       Endpoint endpoint("192.168.1.1", 9441);//Local machine access
12       Student* student1= new Student(endpoint, virtualRoom);
13       Student* student2= new Student(endpoint, virtualRoom);
14       Student* student3= new Student(endpoint, virtualRoom);
```

``

```
15
16        Teacher* teacher1= new Teacher(endpoint, virtualRoom);
17        teacher1->addStudent(student1);
18        teacher1->addStudent(student2);
19
20        Teacher* teacher2= new Teacher(endpoint, virtualRoom);
21        teacher2->addStudent(student2);
22        teacher2->addStudent(student3);
23
24        Teacher* teacher3= new Teacher(endpoint, virtualRoom);
25        teacher2->addStudent(student2);
26        teacher2->addStudent(student3);
27    }
28 }
```

Snippet 5-27. An updated portion of Snippet 5-15 — with different network paradigms —

5.5 An earlier approach

In this section, we show one of the alternative approaches that we adopted for a while before replacing it with the current approach, which we found to be better in terms of lines of code and generation complexity. We will refer to the old approach as the message-based approach. The old approach was a part of Umple builds for a while, and can be accessed from old repositories.

In terms of code generation of the message-based approach, we create a protocol class for each component. In a similar manner to the current approach, we still create an active method named after the port name.

Snippet 5-28 shows a portion of the protocol class created for the component class defined in Snippet 5-3, and more specifically for the ports defined in Lines 2-4. The core methods are sendMessage (Line 8) and receiveMessage (Line 21). The pattern used is publisher-subscriber, in which each port is registered as a map of events. Different places of the code can subscribe to listen to port events. Upon port events, notifications will be published to subscribers (Line 27, 36, and 45).

C++

```
1  void Component_PortProtocol::inPort(int data){
2      component_Port = Component_Port::inPort;
3      component_Event = Component_Event::IN_inPort;
4      sendMessage(Component_Port::inPort,
5          Component_Event::IN_inPort, sizeof(int), &data);
6  }
7  ....
8  void Component_PortProtocol::sendMessage(shortint portId,shortint evtId, int size, void* data) {
9      MessageHeader* msg = service->getBufferedMessage();
10     if (msg!=NULL) {
11         msg->portId = portId;
12         msg->eventId = evtId;
13         if (size>0 && data!=NULL) {
14             msg->data = malloc(size);
15             memcpy(msg->data, data, size);
```

``

```
16          }
17          service->push(msg);
18      }
19  }
20
21  void Component_PortProtocol::receiveMessage(const
22          MessageHeader* msg){
23              switch(msg->eventId){
24                  case Component_Event::IN_inPort:
25                      if(sizeof(msg->data) == sizeof(int)) {
26                          int data = *((int*) msg->data);
27                              inPort_event.publish(data);
28                          } else {
29                          throw"Bad port data";
30                      }
31                      break;
32
33                  case Component_Event::IN_outPort:
34                      if(sizeof(msg->data) == sizeof(int)) {
35                          int data = *((int*) msg->data);
36                          outPort_event.publish(data);
37                      } else {
38                          throw"Bad port data";
39                      }
40                      break;
41
42              case Component_Event::IN_dualPort:
43                  if(sizeof(msg->data) == sizeof(int)) {
44                      int data = *((int*) msg->data);
45                      dualPort_event.publish(data);
46                  } else {
47                      throw"Bad port data";
48                  }
49                  break;
50                  ....
51              default:
52                  break;
53          }
54  }
```

Snippet 5-28.A protocol class generation example

The implementation of each port event method delegates the protocol class (Snippet 5-29 - Lines 2, 6, and 10).

``

```
1  void Component::inPort(int data){                                    C++
2      component_PortProtocol.inPort(data);
3  }
4
5  void Component::outPort(int data){
6      component_PortProtocol.outPort(data);
7  }
8
9  void Component::dualPort(int data){
10     component_PortProtocol.dualPort(data);
11 }
```

Snippet 5-29. **Protocol event methods**

For each port, an internal receive method with an empty body (Snippet 5-30- Lines 27-29). This method is designed to be listened to by subscribers such as the protocol class (Lines 14-18). Upon receiving an event, the receive method of the receiving port will be invoked, and thus will publish notifications to the subscribers.

```
1  void Component::inPort(int data){                                    C++
2  //Constructor
3  Component::Component(){
4  this->initPortConnections();
5  }
6  .....
7  //Destructor
8  Component::~Component(){
9  .....
10 this->disconnectPortConnections();
11 }
12 .....
13 void Component::initPortConnections(){
14     inPort_Handle=component_PortProtocol.inPort_event.subscribe(this, Component::receive_inPort_Data);
15     outPort_Handle=component_PortProtocol.outPort_event.subscribe(this,
16 &Component::receive_outPort_Data);
17     dualPort_Handle=component_PortProtocol.dualPort_event.subscribe(this,
18 &Component::receive_dualPort_Data);
19 }
20 ......
21 void Component::disconnectPortConnections(){
22     component_PortProtocol.inPort_event.disconnect(inPort_Handle);
23    component_PortProtocol.outPort_event.disconnect(outPort_Handle);
24     component_PortProtocol.dualPort_event.disconnect(dualPort_Handle);
25 }
26 .....
27 void Component::receive_inPort_Data(int inPort){}
28 void Component::receive_outPort_Data(int outPort){}
29 void Component::receive_dualPort_Data(int dualPort){}
```

Snippet 5-30.**Event publisher-subscriber handlers**

At the model level, developers are only entitled to work with the event methods of ports; they do not need to worry about how the publishers-subscribers are implemented.

``

In the generated code, ports of a component have an enumeration that defines a way to make them distinct (Snippet 5-31). The publisher-subscriber framework uses this enumeration to determine what port to notify events about (Lines 16-19).

C++

```
1  class _Component_Port{
2  public:
3        typedef enum{ NIL=0,  inPort=1000,  outPort=1001,
4              dualPort=1002 } enum_type;
5              _Component_Port(enum_type val = NIL): _val(val){
6              assert(val <= dualPort);
7        }
8        operator enum_type() const {
9              return _val;
10       }
11       operator int() {
12             return static_cast<int>(_val);
13        }
14       operator string() {
15             switch (_val){
16                 case inPort: return "inPort";
17                 case outPort: return "outPort";
18                 case dualPort: return "dualPort";
19                 default:  return "[Unknown port Type]";
20           }
21        }
22 private:
23       enum_type _val;
24 };
```

Snippet 5-31.Port enumeration

The message-based approach used to create two additional classes, protocol and message classes, as opposed to the current approach that does not require any additional classes. Managing the serialization and deserialization in the old approach did not consider associations. Needless to mention that the amount of code in the new approach is tremendously reduced (>65%).

Handling the logic and time constraints of active methods was not as straightforward as the current approach, which easily amalgamates several ports, active methods, constraints, and connectors together.

5.6 Summary

In this chapter, we discussed the backbone of our research, composite structure and distributed system support in Umple. We managed to eliminate the use of protocols at the model level using simple Umple constructs. The models written in Umple can be easily used in distributed systems without any additional design requirements. A user can apply any networking paradigm such as P2P, client/server, or a hybrid of them. We support unicast, broadcast, and multicast, and provide the required methods that ease the usability of the applications generated.

``

We support both wired and unwired types of communication, which give more design options to users. We use the association features to support incarnated components, multiplicity, and multiple client access.

Our implementation fully supports synchronous and asynchronous access as opposed to many commercial products and standards, which usually support synchronous behaviour only. We support multiple parameters when defining an active method, which is usually limited to void parameters in UML specifications and some modelling tools.

``

Chapter 6 Discussion and Case Studies

In this chapter, we discuss the testing and assessment of the code we have generated, both for snippets presented previously and also two additional case studies, each of which is prototype of a complete running system. In the set of models, we try to make full use of Umple capabilities with a focus on our newly introduced features related to composite structure, distributed system support, and generation templates. We aim to show that Umple models have a much reduced volume user code and complexity, as compared to the equivalent C++.

Our work is ongoing research. As Umple progresses, more composite structure, embedded and real-time examples will be added to UmpleOnline, as is the case with class diagrams and state machines. We are going to regularly update, maintain, and add new features to composite structure.

6.1 Generated code testing

Testing and running generated C++ code by simply compiling the .cpp and .h files would not be straightforward as it requires to be tested on different operating systems and different compilers, each having their own peculiarities. In order to overcome this issue, every time Umple generates C++ from an Umple model it also generates a CMake (cmake.org). With the CMake file, a developer can directly open a C++ project using a development environment of their choice, or run it from the console.

Snippet 6-2 shows the CMake file generated along with the generated files for the model in Snippet 6-1. The CMake maintains the same folder structure defined in a model namespace (Snippet 6-1 - Line 1 and Snippet 6-2 - Lines 10-17).

Umple

```
1  namespace org.test;
2  class Class1{}
3  class Class2{}
4  class Class3{}
5  class Class4{}
```

Snippet 6-1. A simple example to illustrate CMake generation

cMake

```
1  cmake_minimum_required(VERSION 3.0)
2  project(model)
3
4  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=CMAKE_CXX_STANDARD 11")
5  include_directories(${CMAKE_SOURCE_DIR})
6  include_directories(org/test)
7
8  set(SOURCE_FILES
9     model_Model.h
10 org/test/Class1.h
11 org/test/Class1.cpp
```

``

```
12  org/test/Class2.h
13  org/test/Class2.cpp
14  org/test/Class3.h
15  org/test/Class3.cpp
16  org/test/Class4.h
17  org/test/Class4.cpp
18  model_Main.cpp)
19
20  add_executable(model ${SOURCE_FILES})
```

Snippet 6-2.The CMake file for Snippet 6-1

In order to run a CMake file, we can either use the CMake GUI (Figure 6-1) or console (Snippet 6-3).

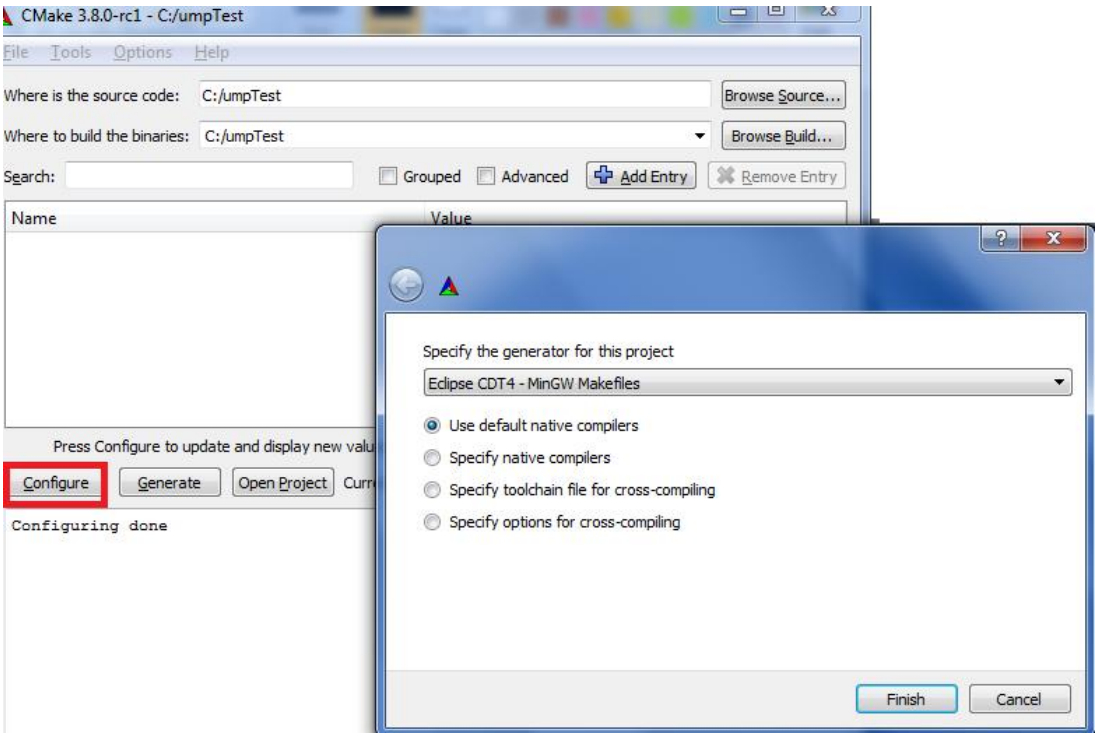


Figure 6-1. An overview of the CMake GUI

Batch

```
1   @echo off
2   Mkdirprj
3   cdprj
4
5   cmake -G "Visual Studio 10 2010"  c:\build\prj
6
7   msbuildprj.sln /t:Rebuild /p:Configuration=Debug
8   set BUILD_STATUS=%ERRORLEVEL%
9   if %BUILD_STATUS%==0 echo $$(project) Build success
10  if not %BUILD_STATUS%==0  echo $$(project) Build failed!!!!!!
```

Snippet 6-3.A simple batch script to run a CMake file

We typically use the console in order to configure, build, and run multiple projects at a time. This allows us to find syntax errors and perform semantic tests, and compute metrics in a batch manner.

``

6.2 Test-driven development

We follow the widely-used test-driven environment process and test models every time we build the system.. We validate models both syntactically (checking that the model can be parsed by the compiler) and semantically (checking that the generated code runs as expected).We apply the tests against 50 Umple models consisting of more than 1000 lines in total. A test run has the following steps:

1) Generate code from models. This will include generating a CMake file for each model (Section 6.1).
2) Create a C++ project for each model using its CMake file.
3) Compile each C++ project and validate that it does not have any syntax error.
4) Run each C++ project, and based on some CppUnit tests, validate the model semantically.

The Umple compiler runs quickly, but the CMake tool runs slowly. When it is applied to all our models sequentially, it takes a long time. In order to overcome this, we run projects in parallel using a Java application that notifies us of the output of each model project.

CMake allows for incremental compilation, which means that regenerating code that has been modified within an existing CMake project will not take as much time as for its original creation. Hence, during an intensive development session we do not regenerate CMake projects. However, in a build test, we make sure to create new CMake projects every time.

6.3 Virtual room case study

This is an updated case study (Snippet 6-4) of the virtual room model we discussed in many snippets in the previous chapter; i.e. Snippet 5-27. In this model, we try to encompass all major composite structural features (Chapter 5).

A VirtualRoom instance acts as a server that communicates among teachers and students, and provides a discovery service. Once a teacher has registered a course in a virtual room, other students become able to register for the course. A virtual room is an endpoint that has a persistent IP address (Snippet 6-4- Line 109), which is known to students and teachers. Communication between students and teachers is P2P. Each class has a main function that sets its connection configuration (Lines 51-55, 69-74 and 109-110), with the assumption that the virtual room is at the port 9441 (Line 109). An enrol operation (Lines 87 and 95) of a student or teacher is synchronous, to wait for a server response for a successful enrolment. On the other hand, the P2P communication between students and teachers is asynchronous.

``

Umple

```
interface ISender{
void ask(Integer num1, Integer num2, String op);
void receive(Integer solution);
}

interface IReceiver{
void answer(Integer num1, Integer num2, String op);
void logOut(Integer solution);
}

class MathQuestion{
   out Integer num1;
   out Integer num2;
   out String op;
   in Integer solution;

   [num1, num2, op]
   active void ask(int num1, int num2, String op) {
         cout<< num1;
         cout<< num2;
         cout<< op;
      }->void answer {
         if(op== "+"){
            solution(num1+num2);
         }elseif(op== "-"){
            solution(num1-num2);
         }
   }

   [solution]
   active void receive{
      cout<<"Answer provided is " + solution;
   }->inverse logOut{
      cout<<"Answer sent to out port is " + solution;
   }
}

class Teacher {
  distributable;
  isA ISender;
  port MathQuestion question;

  0..1 -- * Student;
void ask(int num1, int num2, string op){
    cout <<"Asking a question";
  }
void receive(int solution){
    cout <<"Answer received";
  }
public int main(int argc, char* argv[]) {
     Endpoint serverAddress("127.0.0.1", 9441);
     VirtualRoom* room= new VirtualRoom(endpoint);
```

``

```
53      Teacher* teacher= new Teacher();
54      room->enrollTeacher(teacher);
55      cout << room.getStatusConnected();
56     for (;;);
57    }
58  }
59  class Student {
60   distributable;
61   isA IReceiver;
62   port MathQuestion answer;
63  void answer(int num1, int num2, string op){
64      cout <<"Answering a question";
65   }
66  void logOut(int solution){
67      cout <<"Question answered";
68   }
69  public int main(int argc, char* argv[]) {
70      Endpoint serverAddress("127.0.0.1", 9441);
71      VirtualRoom* room= new VirtualRoom(endpoint);
72      Student* student= new Student();
73      room->enrollStudent(student);
74      cout << room.getStatusConnected();
75      for (;;);
76   }
77  }
78
79  class VirtualRoom
80  {
81   distributable;
82   lazy Teacher admin;
83   status
84   {
85     Open
86     {
87      closeRoom -> Closed;
88      enrollTeacher(Teacher teacher) / {
89                setAdmin(teacher);
90      }-> TeacherEnrolled;
91     }
92     Connected {
93      leaveRoom -> Disconnected;
94      TeacherEnrolled
95      {
96       enrollStudent(Student student) / {
97                admin.addStudent(student);
98       } -> Connected;
99       disconnect -> Disconnected;
100      studentSizeExceedsMinimum -> Closed;
101     }
102    }
103    Disconnected { }
104    Closed {
```

``

```
105       close -> Disconnected;
106     }
107   }
108
109   public int main(int argc, char* argv[]) {
110     VirtualRoom* room= new VirtualRoom(9441);
111     for (;;);
112   }
113 }
```

Snippet 6-4.Virtual room case study

6.4 Ping-pong case study

In Section 2.5, we showed a motivating ping-pong example, which used simple attributes to define ports. In this section, we show a more comprehensive ping-pong example (Snippet 6-5) that uses the complex port features (Lines 9-26). There are two interfaces, IPinger (Lines 1-3) and IPonger (Lines 5-7). The communication paradigm in this example is almost solely P2P, except for the fact that Pinger starts communication, hence it could be considered a super peer. The communication is initialized between a single Pinger and Ponger.

Pinger does not redefine the ping port, meaning that the basic implementation defined in the complex port will be used (Line 15). On the other hand, Ponger redefines the pong port (Lines 37-39), by adding a constraint to ensure that the message propagation will go back and forth between Pinger and Ponger instances, until the count reaches 10 (Line 36)

Umple

```
1  interface IPinger{
2  void ping(int pIn);
3  }
4
5  interface IPonger{
6  void pong(int pOut);
7  }
8
9  class PingPongPort{
10 public out Integer pingPort;  // require port
11 public in Integer pongPort;  // provide port
12
13   [pingPort]
14   active void ping(int num) {
15       pongPort(num + 1);
16   }->void logPortData {
17       cout <<"CMP 1 : Ping Out data = "<< pOut1 << endl;
18   }
19
20   [pongPort]
21   active void pong(int num) {
22       pingPort(num + 1);
23   }->void logPortData {
24       cout <<"CMP 1 : Pong Out data = "<< pOut1 << endl;
```

``

```
25    }
26  }
27
28  Class Pinger {
29    isA IPinger;
30    port PingPongPort pingPort;
31  }
32
33  Class Ponger {
34    isA IPonger;
35    port PingPongPort pongPort;
36    [pongPort, num < 10]
37    active void pong(int num) {
38        pingPort( num + 1);
39    }
40
41  }
42
43  class PingPong {
44    Pinger cmp1;
45    Ponger cmp2;
46    Integer startValue;
47
48    after constructor {
49      cmp1->ping(startValue); // Initiates communication in the constructor
50    }
51    cmp1.pingPort -> cmp2.pongPort;
52  }
```

Snippet 6-5. **Ping-pong case study**

6.5 Light control case study (an automotive example)

Interior light control is standard in modern cars. When a car door is open, the interior lights will be automatically switched on to warn the driver. The lights will stay on as long as a door is still open.

The example has the following main components (Snippet 6-6), LightActuator , DoorManager, LightSwitch, and Dimmer. DoorSensor directly sends signals to the light circuit in order to either turn the lights on or off.

DoorManager is responsible for sending signals when a car's door has been opened or closed. This depends on the sensors attached to each door. *LightSwitch* simply receives signals when someone, in the car, toggles the interior lights.

The out signals of both DoorManager and LightSwitch are sent to the *Dimmer* component, which handles the communication between those two components and LightActuator. Dimmer is the composite component in this case study (Figure 6-2).

``

When DoorManager sends signals to Dimmer that a door has been opened, Dimmer will send a signal to LightActuator to turn the interior lights on.

When all doors become closed, Dimmer will send a signal to LightActuator to turn the lights off. However, if the lights were already switched on before one the doors were opened, Dimmer will not request the lights to be switched off.

Note that this example does not contain the full action code, nor hardware and sensor configurations. As a result, it is only intended to provide an additional example of the applicability of our work.

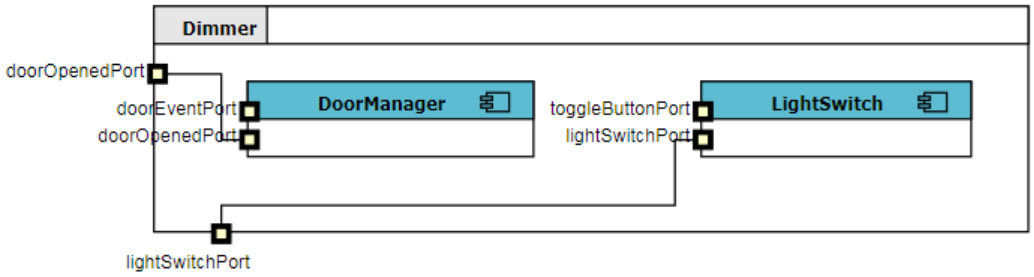


Figure 6-2. The composite structure of Snippet 6-6

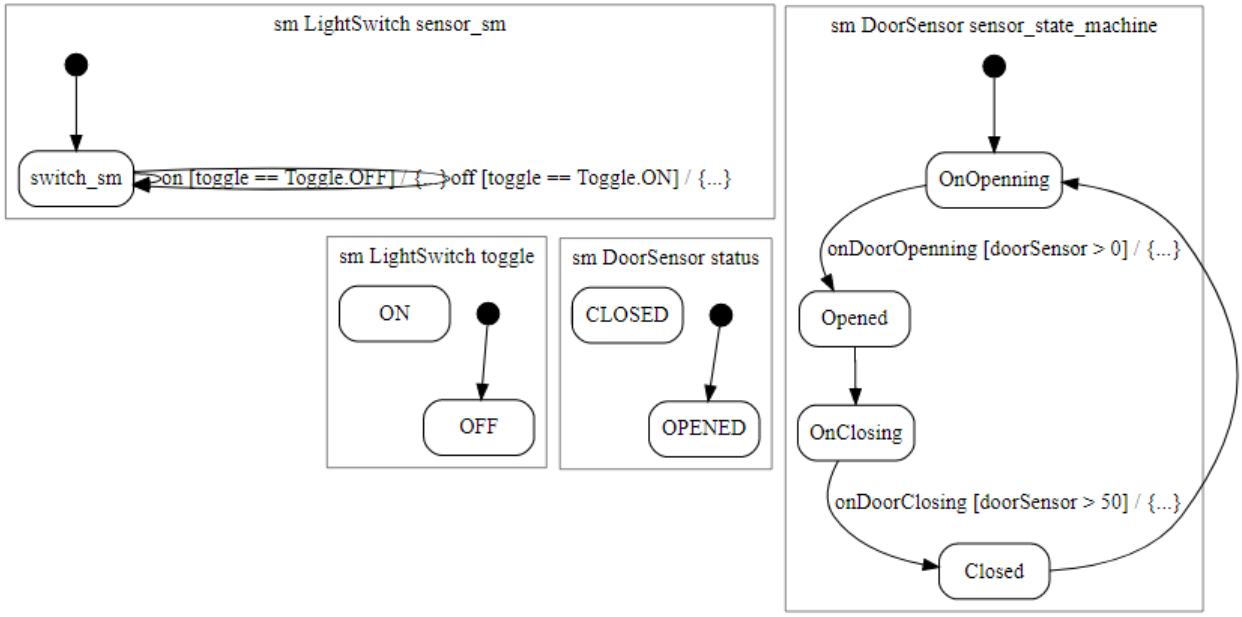


Figure 6-3. The state machines of Snippet 6-6

``

Umple

```
1  class LightSwitch{
2     in Integer toggleButtonPort;
3     out Boolean lightSwitchPort;
4     toggle { OFF{} ON{} }
5
6     [toggleButtonPort]
7     void active receiveToggleButtonPort{
8        if(toggleButtonPort > 0) {
9           on();
10       } else {
11          off();
12       }
13    }
14    void switchOn() {
15       lightSwitchPort(true);
16    }
17    void switchOff() {
18       lightSwitchPort(false);
19    }
20    sensor_sm {
21       switch_sm {
22          on [toggle == Toggle.OFF]-> / {toggle = Toggle.ON; switchOn(); } switch_sm;
23          off [toggle == Toggle.ON]-> / {toggle = Toggle.OFF; switchOff();} switch_sm;
24       }
25    }
26 }
27
28 class DoorSensor {
29    in Integer doorSensorPort;
30    out Integer doorEventPort;
31    status { OPENED{} CLOSED{} }
32
33    [doorSensorPort]
34    void active receiveSensorData{
35       if(doorSensorPort > 10 && doorSensorPort < 40) {
36          onDoorOpenning(doorSensorPort);
37       }else{
38          onDoorClosing(doorSensorPort);
39       }
40    }
41
42    void DoorOpened() {
43       doorEventPort(1);
44    }
45    void DoorClosed() {
46       doorEventPort(0);
47    }
48    sensor_state_machine {
49       OnOpenning {
50          onDoorOpenning[doorSensor > 0] ->/ {status = Status.OPENED;}
51          Opened;
52       }
```

``

```
53          Opened {
54              entry /{ DoorOpened(); }
55              ->OnClosing;
56          }
57          Closed {
58              entry /{ DoorClosed(); }
59              ->OnOpenning;
60          }
61          OnClosing {
62              onDoorClosing[doorSensor > 50] ->/ {status = Status.CLOSED;}
63              Closed;
64          }
65      }
66  }
67
68
69  class DoorManager {
70      in Integer doorEventPort;
71      out Boolean doorOpenedPort;
72      0..1 -- 4 DoorSensor door;
73
74      [doorSensorPort]
75      void active onDoorOpenedEvent{
76          doorOpenedPort(doorSensorPort > 1);
77      }
78
79      door.doorEventPort -> doorEventPort;
80  }
81
82  class Dimmer {
83      in Boolean doorOpenedPort;
84      in Boolean lightSwitchPort;
85      DoorManager dm;
86      LightSwitch ls;
87      Boolean doorOpened;
88      Boolean lightSwitched;
89
90      void dimLight(Boolean light) {
91          //communicate with LightActuator
92      }
93
94      [lightSwitchPort, !isDoorOpened()] // ignore off switch in case of opened door
95      void active onLightSwitchEvent{
96          setLightSwitch(lightSwitchPort);
97          dimLight(lightSwitchPort);
98      }
99
100     [doorOpenedPort]
101     void active onDoorOpenedEvent{
102         setDoorOpened(doorOpenedPort);
103         if(doorOpenedPort) {
104             dimLight(true);
```

``

```
105         } else if( !doorOpenedPort && !isLightSwitched()) {
106             dimLight(false);
107         }
108     }
109
110     dm.doorOpenedPort -> doorOpenedPort;
111     ls.lightSwitchPort -> lightSwitchPort;
112 }
```

Snippet 6-6. Light control case study

6.6 Quantitative evaluation

In this section, we evaluate aspects of our work based on measuring software complexity.

For each Umple model that contains components or ports, we generate over 1000 lines of model-independent code that handles multithreading, remote communication, and JSON serialization. In normal C++ development, a programmer would never need to rewrite such code for every model; they would either write it once and reuse, or else use existing libraries. We do not use existing libraries since we want our generated code to be self-contained, to be platform independent, and to behave (e.g. with regard to multithreading) in a way that we can precisely control.

Hence, in order to avoid bias in terms of the number of lines, we ignore these built-in, always-generated lines. As a result, the evaluation below is based on the generated code that varies from model to model.

We evaluate the cases studies we showed in this chapter in addition to other snippets we selected from previous chapters (Table 6-1). We calculate McCabe Cyclomatic Complexity at the model level based on the Boolean constraints defined, such that each constraint has a weight of two; i.e. two constraints defined will be valued 4. We calculate the complexity ratio as$100 - (Umple\ Mcabe/McCabe) \times 100$. This correspond to the percent reduction of complexity when writing in Umple, as opposed to the generated C++ code. We computed Cyclomatic Complexity using the LocMetrics tool (http://www.locmetrics.com/).

The cyclomatic complexity reduction averages about 71.6% in cases where the Umple model has conditions, and averages 82.5% when cases are included where Umple code has some conditions (i.e. where cases of 100% reduction are included).

Table 6-1 also shows that there is also a big reduction in lines of code between generated C++ and Umple models. This reduction averages 93.7% and is roughly constant, hence independent of model size.

An important threat to validity of this analysis is that the C++ code written by a developer might be rather different from that generated by Umple. It may be possible for a developer to leave out some parts, or find other ways to make the C++ more compact. However, we suggest that writing such compact C++ might in fact make it more obfuscated, and hence add even more to complexity.

``

It seems clear from the above analysis that maintaining an Umple code base for the functionality described in this thesis should be dramatically easier than maintaining raw C++ that attempts to provide the same functionality. We leave assessing this in detail to future work.

Table 6-1. Evaluation of Umple models versus the generated code

Model	Umple Non Blank LOC	C++ Non Blank LOC	C++ Cyclomatic Complexity	Umple Cyclomatic Complexity	LOC reduction %	Cyclomatic Complexity reduction%
Snippet4-15	35	254	10	0	86.2	100
Snippet4-17	22	213	10	0	89.7	100
Snippet4-33	15	241	15	4	93.8	73.3
Snippet4-34	19	225	10	6	91.6	40.0
Snippet4-36	23	252	10	8	90.9	20.0
Snippet4-38	19	280	21	6	93.2	71.4
Snippet5-3	29	508	33	4	94.3	87.9
Snippet5-4	22	846	50	2	97.4	96.0
Snippet5-23	36	826	50	0	95.6	100
Snippet5-24	28	1374	125	0	98.0	100
Snippet5-25	17	838	77	0	98.0	100
PingPong	43	875	44	6	95.1	86.4
VirtualRoom	104	2013	157	4	94.8	97.5

``

Chapter 7 Conclusions and Future Work

In this thesis, we explored how to enable component-based modelling— composite-structure — into Umple, such that these concepts can become easier to use in real-time, distributed, concurrent and multithreaded systems.

7.1 Answers to research questions

At the start of the thesis, we posed several research questions. In the following we summarize how we have answered each of these questions.

RQ1: *How can we improve component-based modelling to enable better and easier development of real-time and distributed systems using Umple?*

We answered this question by improving Umple to have component-based modelling capabilities, as follows:

- **Incorporates the major component-based features supported in the state-of-the-art tools**. In Chapter 2 we showed that after the introduction of composite structure into Umple, Umple has become one of the very few open-source tools that support major composite-structure features.
- **Conforms to component-based implementations, syntactically and semantically, following the existing specifications and standards.** In particular, in Chapter 4 we showed how our composite structure implementation, which follows an enhanced active object pattern, aligns with common standards and specifications such as UML, AUTOSAR, EAST-ADL, and MARTE. We demonstrated by examples (Chapter 4) how complicated time concepts such latency and jitter, which are considered thoroughly in these specifications, can be handled in Umple using simple language constructs.
- **Supports concurrent systems.** In Chapter 4 we showed different Umple constructs that help users manage their multithreaded applications.
- **Enables generic distributable systems development with different networking paradigms and transport.** We demonstrated by examples (Chapter 5 and Chapter 6) that users can write their Umple models, and based on the associations and connectors defined, a network paradigm, such P2P or client/server, can be applied automatically,
- **Keeps Umple users from having to use additional libraries when developing their distributed systems**. We showed in Chapter 5 the implementation details of our generated code, which enforces TCP/IP communication among nodes in a network. Umple users will not be burdened by learning additional libraries when exploring Umple. At the moment, we support JSON as the data interexchange format, but our framework can be easily extended to support other common formats such as XML.

``

- **Enables development with less code when using Umple**. We showed in Chapter 6 how we can write comprehensive Umple models, which are fairly simple in terms of number of lines of code when compared to other languages.
- **Enables easy extension to support component-based standards and different software component descriptions.** In Chapter 3 we showed how we implemented Umple-TL, which allowed us to rapidly create the Umple improvements discussed in the above bullet points. Umple-TL can thus be used to implement additional template-specific stubs.
- **Has a simplified workflow.** In Umple, the only artifacts used are Umple files, which can be used for the whole development process; i.e. designing, coding, deploying, and testing. The Umple workflow is as simple as writing models, generating code, and executing the code generated. In sophisticated commercial tools such as RSARTE or Rhapsody, a user is more prone to errors given that the development process consists of many steps in many isolated views, such as design, action code, and deployment views.
- **Gathers key points into one tool, Umple.** We implemented the core features of composite structure in a single tool, as compared to other common modelling tools which tend to implement various subsets (Chapter 2). Additionally, we are the only tool that supports associations in the generated code, as opposed to the design level only, and at the same time supports distributed environments. Hence, we avoid any dependency on third-party libraries such as Connexis in RSARTE. Hence, this makes Umple easier as compared to these tools in terms of integration and deployment with embedded devices.

RQ2:*What are the major challenges in developing a real-time construct in a textual language?*

The challenges that we addressed during this research can be summarized as follows.

- **Active object as a pattern to handle real-time.** The traditional active object pattern presents challenges such as priority-based message scheduling and constraints (Chapter 4). Hence, we introduced an enhanced active object pattern, which overcomes the list of challenges we listed in Section 4.2.1.
- **Filling the gap between textual and visual notations.** Umple allows users to develop their models both textually and visually. We needed to implement visual representation for most of our implementations in order to follow this philosophy.
- **Propose simple syntax and grammar rules.** We introduced several concepts into Umple, which also required us to be conservative to evaluate the necessity of adding new keywords that might do more harm than good if they became convoluted. For instance, we shortened the time constructs discussed in different time specifications and standards (Section 4.9) into five constructs (Section 4.3.1.5). We showed in Chapter 4how these constructs can be used to fulfil the major features a real-time application requires.

RQ3: How can we evaluate our proposed technology?

``

In Chapter 6, we showed three comprehensive case studies, on which we applied quantitative measurements to evaluate complexity and the amount of code. We demonstrated how we can write sophisticated Umple models, which do not require much code, and have low code complexity.

7.2 Contribution summary

During our discussions in the various chapters of this thesis, we listed contributions related to each chapter (Sections 1.3, 4.1, and 5.1).These are paraphrased in the following, along with a few additional contributions:

Real-time code generation. We created a new C++ code generator for Umple; this includes code generation to support the pre-existing Umple features such as attributes, associations, state machines, mixins, patterns, and aspects. We showed a large number of code snippets for C++ code generation of different Umple models, which use many of core Umple features in addition to our new ones. There are an extensive set of test cases to prove that this works.

Umple as a Template Language (Umple-TL): We created Umple-TL (Chapter 3) to consolidate all template generation in Umple. Our real-time code generation is implemented in Umple-TL; all other Umple generators (e.g. Java, PHP, Ruby) also migrated to Umple-TL. In the thesis we gave a comparison between Umple-TL and other text emission tools, which indicated advantages of using Umple-TL.

Active object implementation and concurrency: We implemented an enhanced active object pattern (Chapter 4), which 1) simplifies active features at the action code level, 2) provides simple constructs to manage time and logical constraints, 3) introduces new invocation patterns such as, call/resolve/then, 4) and handles synchronous and asynchronous behaviours. We showed that our implementation conforms to major specifications and standards that focus on time constraints in real-time applications (Section 4.9).

Composite modelling implementation: We implemented all major composite structure features (Chapter 2) using simple Umple constructs (Chapter 5). We demonstrated how we can define different types of ports (Section 5.3.6), and rely on Umple features such as associations to define the dataflow, and infer the network paradigm dynamically. Our implementation, at the model-level, is protocol-free, which helps users to avoid unnecessary level of complication that can exist in many modelling tools (Chapter 2). Composite structure relies on the active object infrastructure meaning that it takes advantage of all of its features.

Distributed environment support: We showed (Chapter 5 and Chapter 6) that users do not need to apply rigorous design changes to make their applications distributable. We allow both wired and unwired communication. Users are not restricted to specific networking paradigms.

``

Test-driven environment: We implemented a semantic and syntactic test runner to check for errors (Chapter 6), which depends on the CMake files that we generate for and associate with each generated C++ project. With this level of configuration, we managed to reduce the complexity of a language such as C++.

Visualization: We developed a diagramming capability to support our composite structure work. Any Umple code that contains composite structure can be visualized in UmpleOnline (try.umple.org).

7.3 Future work

We suggest the following directions to continue and enhance this work:

Our work could be applied directly in the automotive industry, which was a target of our work. The automotive industry fundamentally relies on composite structure and component-based modelling, so our work would be ideal. To better support automotive development, Umple should be enhanced to generate software descriptors that match standard such as AUTOSAR. As well, Umple needs to support code generation for different RTE and hardware components such as Arduino. We have already created a simple implementation of an autonomous car, which we did not include or discuss in this thesis as it is not complete.

Umple should be enhanced to implement additional communication protocols such as UDP/IP and Bluetooth. It ought also to allow for different serialization mechanisms to support other data interchange transport format such as XML and binary.

Umple needs to be able to emulate multithreading for hardware that does not support threads, such as Arduino.

It would be valuable to provide options to generate component-based models that use other industrially distributed libraries.

It would be beneficial to conduct empirical studies to evaluate the usability of our new Umple features on a number of developers.

``

References

[1] G. P. Maxton and J. Wormald, *Time for a Model Change*. Cambridge: Cambridge University Press, 2004.

[2] J. Schäuffele, T. Zurawka, and L. Mandel, *Automotive Software Engineering*. Wiesbaden: Springer Fachmedien Wiesbaden, 2013.

[3] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software Engineering for Automotive Systems: A Roadmap," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 55–71.

[4] M. Conrad, I. Fey, and S. Sadeghipour, "Systematic Model-Based Testing of Embedded Automotive Software," *Electron. Notes Theor. Comput. Sci.*, vol. 111, pp. 13–26, Jan. 2005.

[5] M. Conrad, H. Dörr, I. Fey, and I. Stürmer, "Using Model and Code Reviews in Model-based Development of ECU Software," in *SAE 2006 World Congress & Exhibition*, 2006.

[6] I. Schieferdecker, E. Bringmann, and J. Großmann, "Continuous TTCN-3: testing of embedded control systems," in *The International Workshop on Software Engineering for Automotive Systems - SEAS '06*, 2006, p. 29.

[7] I. Stürmer, M. Conrad, H. Dörr, and P. Pepper, "Systematic testing of model-based code generators," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 622–634, 2007.

[8] P. Kamga, J., Herrmann, J., Joshi, "Deliverable: D-MINT automotive case study - Daimler, Deliverable 1.1," in *Deployment of model-based technologies to industrial testing, ITEA2 Project*, 2007.

[9] B. Selic and ObjecTime, "Real-Time Object-Oriented Modeling (ROOM)," in *Proceeding RTAS '96 Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, 1996, p. 214.

[10] V. Abdelzad and T. C. Lethbridge, "Promoting traits into model-driven development," *Softw. Syst. Model.*, pp. 1–21, Nov. 2015.

[11] S. Anssi, S. Tucci-Pergiovanni, S. Kuntz, S. Gérard, and F. Terrier, "Enabling scheduling analysis for AUTOSAR systems," in *The 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2011*, 2011, pp. 152–159.

[12] OMG, "UML 2.4.1," 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/. [Accessed: 01-Jan-2015].

[13] AUTOSAR, "Release 4.2 Overview and Revision History," 2014. [Online]. Available: https://www.autosar.org/documents/. [Accessed: 01-Jan-2016].

``

[14] OMG, “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems,” 2011. [Online]. Available: http://www.omg.org/spec/MARTE/1.1/PDF.

[15] EAST-ADL, “EAST-ADL Domain Model Specification- Version V2.1.12,” 2013. [Online]. Available: http://www.east-adl.info/Specification/V2.1.12/EAST-ADL-Specification_V2.1.12.pdf. [Accessed: 01-Jan-2017].

[16] H. Gomaa, *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*, vol. 36, no. 4. Cambridge University Press, 2011.

[17] T. Lethbridge and R. Laganiere, “Chapter 9: Architecting and designing software,” in *Object-Oriented Software Engineering: Practical Software Development using UML and Java*, McGraw-Hill Europe, 2004, pp. 309–368.

[18] R. G. Lavender and D. C. Schmidt, “Active object: an object behavioral pattern for concurrent programming,” in *Pattern languages of program design 2*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996, pp. 483–499.

[19] AUTOSAR, “Specification of the Virtual Functional Bus,” 2012. [Online]. Available: https://www.autosar.org/fileadmin/files/releases/3-2/main/auxiliary/AUTOSAR_SWS_VFB.pdf.

[20] B. Selic, “Protocols and ports: reusable inter-object behavior patterns,” in *2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC’99) (Cat. No.99-61702)*, 1999, pp. 332–339.

[21] P. Vitharana, “Risks and challenges of component-based software development,” *Commun. ACM*, vol. 46, no. 8, pp. 67–72, Aug. 2003.

[22] K.-K. Lau and Z. Wang, “Software Component Models,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007.

[23] S. Wong, “StarUML Tutorial - OpenStax CNX,” 2007. [Online]. Available: http://cnx.org/contents/41030577-6b63-4767-8fc8-326f9638519e@1. [Accessed: 01-Jan-2017].

[24] Eclipse, “eTrice - Real-Time Modeling Tools,” 2016. [Online]. Available: http://www.eclipse.org/etrice/. [Accessed: 01-Jan-2016].

[25] A. Ramirez, P. Vanpeperstraete, A. Rueckert, K. Odutola, J. Bennett, L. Tolke, and M. van der Wulp, “ArgoUML User Manual: A tutorial and reference description,” 2005. [Online]. Available: https://www.tjhsst.edu/~rlatimer/uml/argomanual-0.18.1.pdf.

[26] M. Mohlin, “Rational Software Architect Community: Modeling Real-Time Applications in RSARTE,” 2015. [Online]. Available:

``

https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W0c4a14ff363e_436c_9962_2254bb5cbc60/page/Modeling Real-Time Applications in RSARTE.

[27] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)," in *Integration of Software Specification Techniques for Applications in Engineering*, 2004, pp. 325–354.

[28] Eclipse.org/papyrus, "Papyrus Modeling environment," 2017. [Online]. Available: https://eclipse.org/papyrus/. [Accessed: 01-Jan-2017].

[29] A. Olsen, O. Færgemand, B. Møller-Pedersen, J. R. W. Smith, and R. Reed, *Systems Engineering Using SDL-92*. North Holland, 1994.

[30] O. Badreddin, "A Manifestation of Model-Code Duality: Facilitating the Representation of State Machines in the Umple Model-Oriented Programming Language (PhD Thesis)," University of Ottawa, 2012.

[31] UmpleOnline, "Umple User Manual: Basic Templates," 2016. [Online]. Available: http://cruise.eecs.uottawa.ca/umple/BasicTemplates.html. [Accessed: 01-Jan-2016].

[32] Umple, "Umple: Model-Oriented Programming - embed models in code and vice versa and generate complete systems," 2016. [Online]. Available: https://github.com/umple/umple. [Accessed: 01-Jan-2016].

[33] Cppreference, "Replacing text macros," 2017. [Online]. Available: http://en.cppreference.com/w/c/preprocessor/replace.

[34] Bottlecaps, "Railroad Diagram Generator," 2017. [Online]. Available: http://www.bottlecaps.de/rr/ui. [Accessed: 01-Mar-2017].

[35] Eclipse, "JET Tutorial (Introduction to JET)," 2003. [Online]. Available: https://eclipse.org/articles/Article-JET/jet_tutorial1.html. [Accessed: 01-Jan-2015].

[36] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige, *The Epsilon Book*. Eclipse Public License, 2015.

[37] "Xtend." [Online]. Available: http://eclipse.org/xtend/. [Accessed: 01-Jan-2015].

[38] C. Raistrick, "Model-Driven Engineering for Distributed Real-Time Systems," in *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, 2013.

[39] E. Farchi, A. Hartman, and S. S. Pinter, "Using a model-based test generator to test for standard conformance," *IBM Systems Journal*, vol. 41, no. 1. pp. 89–110, 2002.

[40] D. H. Park and Soo Dong Kim, "XML rule based source code generator for UML CASE tool," in *The 8th Asia-Pacific Software Engineering Conference*, 2001.

``

[41] L. Geiger, C. Schneider, and C. Reckord, “Template- and modelbased code generation for MDA-tools,” in *Proceedings of the Fujaba Days 2005, volum TR-RI-05-259 of Technical Report. University of Paderborn*, 2005.

[42] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack, “The Epsilon Generation Language,” in *4th European Conference, ECMDA-FA 2008, Berlin, Germany*, 2008, pp. 1–16.

[43] J. Carnell, R. Harrop, and K. Mittal, “Velocity Template Engine,” in *Pro Apache Struts with Ajax*, 2006, pp. 317–357.

[44] Acceleo, “Acceleo Eclipse Page.” [Online]. Available: http://www.eclipse.org/acceleo/. [Accessed: 01-Jan-2015].

[45] “Eclipse Modeling Framework (EMF).” [Online]. Available: http://www.eclipse.org/modeling/emf. [Accessed: 01-Jan-2015].

[46] Umple, “JETToUmpleTL,” 2017. [Online]. Available: https://github.com/umple/JETToUmpleTL.

[47] A. Alghamdi, “Queued and Pooled Semantics for State Machines in the Umple Model-Oriented Programming Language (Master’s thesis),” University of Ottawa, 2010.

[48] I. Ober and I. Stan, “On the Concurrent Object Model of UML*,” in *5th International Euro-Par Conference Toulouse, France, August 31 – September 3, 1999 Proceedings*, 1999, pp. 1377–1384.

[49] Cplusplus.com, “Future,” 2016. [Online]. Available: http://www.cplusplus.com/reference/future/. [Accessed: 20-Jun-2001].

[50] Cplusplus.com, “std::promise,” 2016. [Online]. Available: http://www.cplusplus.com/reference/future/promise/. [Accessed: 01-Jan-2016].

[51] P. N. Klein, H. I. Lu, and R. H. B. Netzer, “Detecting race conditions in parallel programs that use semaphores,” *Algorithmica (New York)*, vol. 35, no. 4, pp. 321–345, 2003.

[52] H. Sutter and J. Larus, “Software and the concurrency revolution,” *Queue*, vol. 3, no. 7, p. 54, Sep. 2005.

[53] T. Rouvinez and A. Sobe, “Comparison of Active Objects and the Actor Model, Universite De Neuchatel, Institut D’informatique, Rapport De Recherche, RR-I-AS-14-06.1,” 2014. [Online]. Available: http://members.unine.ch/anita.sobe/res/RR-I-AS-2014.06.1.pdf.

[54] Microsoft.com, “Delegates (C# Programming Guide),” 2015. [Online]. Available: https://msdn.microsoft.com/en-CA/library/ms173171.aspx. [Accessed: 01-Jan-2017].

[55] M. Rahman, “Delegate,” in *Expert C# 5.0*, Berkeley, CA: Apress, 2013, pp. 187–211.

[56] Y. Z. Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, “Reasoning about optimistic concurrency using a

program logic for history," in *CONCUR'10 Proceedings of the 21st international conference on Concurrency theory*, 2010, pp. 388–402.

[57] A. Queralt and E. Teniente, *Conceptual Modeling - ER 2006*, vol. 4215. 2006.

[58] Y. Smaragdakis and D. S. Batory, "Mixin-Based Programming in C++," in *GCSE '00 Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, 2000, pp. 163–177.

[59] P. Wegner, "Concepts and paradigms of object-oriented programming," *ACM SIGPLAN OOPS Messenger*, vol. 1. pp. 7–87, 1990.

[60] B. Meyer, "Systematic concurrent object-oriented programming," *Communications of the ACM*, vol. 36. pp. 56–80, 1993.

[61] O. Fuks, J. S. Ostroff, and R. F. Paige, "SECG: The SCOOP-to-Eiffel code generator," *J. Object Technol.*, vol. 3, pp. 143–160, 2004.

[62] H. L. Hans Blom, F. Hagl, Y. Papadopoulos, M.-O. Reiser, C.-J. Sjöstedt, D.-J. Chen, and R. T. Kolagari, "EAST-ADL- An Architecture Description Language for Automotive Software-Intensive Systems- White Paper Version M2.1.10," 2012.

[63] D. Gunnarsson, "Timing Analysis and Design for System Development," *ATZextra Worldw.*, vol. 18, no. 9, p. 107, 2013.

[64] M. A. Peraldi-Frati, A. Goknil, J. Deantoni, and J. Nordlander, "A timing model for specifying multi clock automotive systems: The timing augmented description language V2," in *IEEE 17th International Conference on Engineering of Complex Computer Systems, ICECCS*, 2012, pp. 230–239.

[65] F. Mallet, "Clock constraint specification language: Specifying clock constraints with UML/MARTE," in *Innovations in Systems and Software Engineering*, 2008, vol. 4, no. 3, pp. 309–314.

[66] A. Gherbi and F. Khendek, "UML Profiles for Real-Time Systems and their Applications," *J. Object Technol.*, vol. 5, no. 4, pp. 149–169, 2006.

[67] S. Anssi, S. Gérard, S. Kuntz, and F. Terrier, "AUTOSAR vs. MARTE for enabling timing analysis of automotive applications," in *The 15th International Conference on Integrating System and Software Modeling*, 2011, vol. 7083, pp. 262–275.

[68] T. S. Rappaport, *Wireless Communications: Principles and Practice*. Prentice Hall; 2 edition, 2001.

[69] F. Mallet, M. A. Peraldi-Frati, and C. André, "Marte CCSL to execute east-ADL timing requirements," in *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*,

ISORC 2009, 2009, pp. 249–253.

[70] H. Espinoza, S. Gérard, H. Lönn, and R. T. Kolagari, "Harmonizing MARTE, EAST-ADL2, and AUTOSAR to Improve the Modelling of Automotive Systems," in *The Workshop STANDRT, Autosar*, 2009.

[71] K. Kaneiwa and K. Satoh, "On the complexities of consistency checking for restricted UML class diagrams," *Theor. Comput. Sci.*, vol. 411, no. 2, pp. 301–323, 2010.

[72] S. Gordon and S. Choosang, "Verification of the FlexRay Transport Protocol for AUTOSAR In-Vehicle Communications," *Int. J. Veh. Technol.*, vol. 2010, pp. 1–23, 2010.

[73] M. Husseini Orabi, A. Husseini Orabi, and T. Lethbridge, "Umple as a component-based language for the development of real-time and embedded applications," in *The 4th International Conference on Model-Driven Engineering and Software Development, Rome, Italy*, 2016, pp. 282–291.

[74] S. K. Lakkimsetti, "Rational Software Architect Community: Connexis User guide," 2014. [Online]. Available: https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/b7da455c-5c51-4706-91c9-dcca9923c303/page/4a572b8d-0db4-4eed-857f-c288470a8acd/attachment/3e87f5cd-51c5-4fa0-aa8f-1635c391af29/media/rsarte-dcs-user guide.pdf.

[75] B. Selic, "Using UML for Modeling Complex Real-Time Systems," *Mueller F., Bestavros A. Lang. Compil. Tools Embed. Syst.*, vol. 1474, LNCS, pp. 250–260, 1998.

[76] B. P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley Professional, 2004.

``

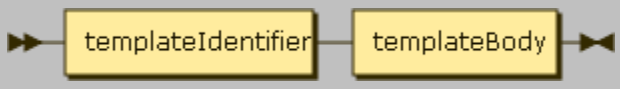
Appendix A

The next figures visualize the Umple-TL grammar (Grammar 1).

(a) Grammar

Grammar ::= templateAttributeDefinition*

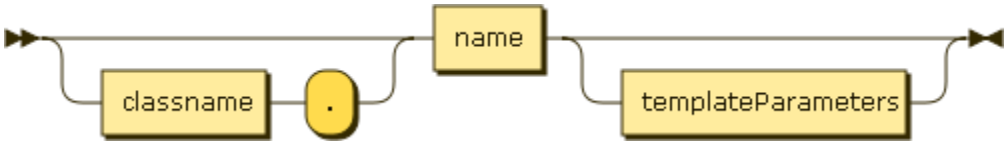
(b)templateAttributeDefinition



templateAttributeDefinition
::= templateIdentifiertemplateBody

Referenced by Grammar

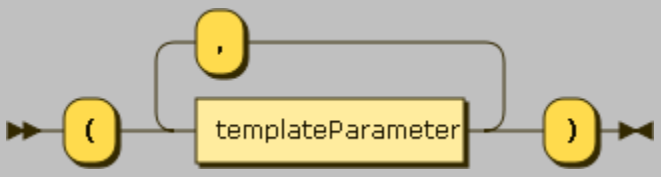
(c) templateIdentifier



templateIdentifier
::= (classname '.')? nametemplateParameters?

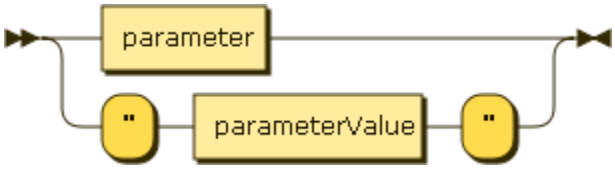
Referenced by templateAttributeDefinition

(d)templateParameters



templateParameters
::= '(' templateParameter (',' templateParameter)* ')'

(e) templateParameter



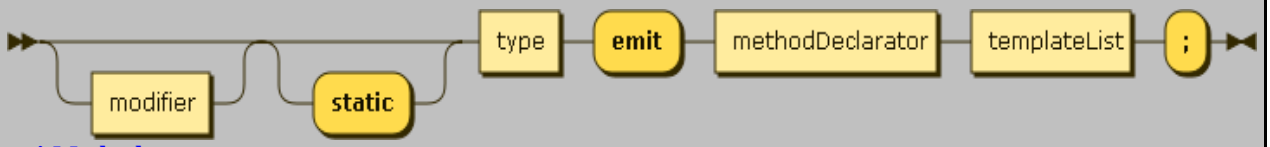
templateParameter
::= parameter

``

| '"' **parameterValue** '"'

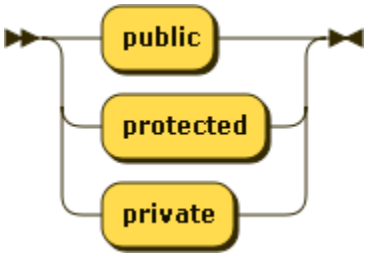
Referenced by templateParameters

(f) emitMethod



emitMethod
::= modifier? 'static'? type 'emit' methodDeclaratortemplateList ';'

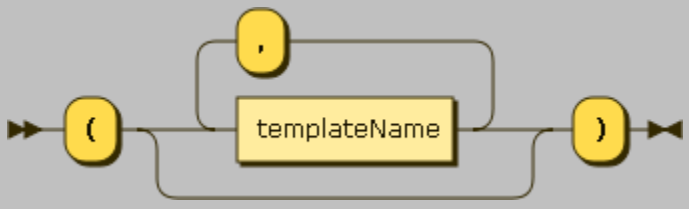
(g) modifier



modifier ::= 'public'
| 'protected'
| 'private'

Referenced by:emitMethod

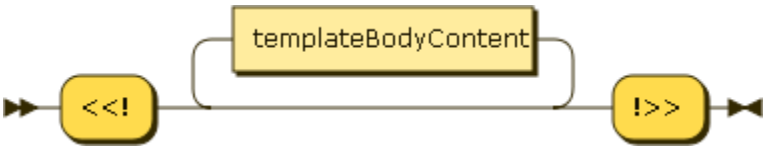
(h) templateList



templateList
::= '(' (templateName (',' templateName)*)? ')'

Referenced by:emitMethod

(i) templateBody

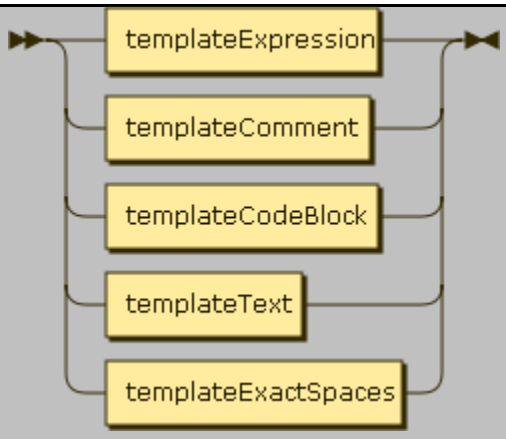


templateBody
::= '<<!' templateBodyContent* '!>>'

Referenced by:templateAttributeDefinition

(j) templateBodyContent

``



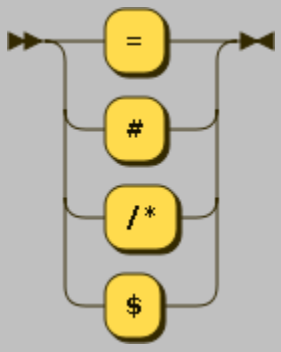
templateBodyContent
::= templateExpression
| templateComment
| templateCodeBlock
| templateText
| templateExactSpaces

Referenced by:templateBody

(k) templateTagContent

templateTagContent
::= '<<' templateTagChoice

(l) templateTagChoice



templateTagChoice
::= '='
| '#'
| '/*'
| '$'

Referenced by:templateTagContent

(m) templateText

``

templateText
::= (text [^templaTgCon])*

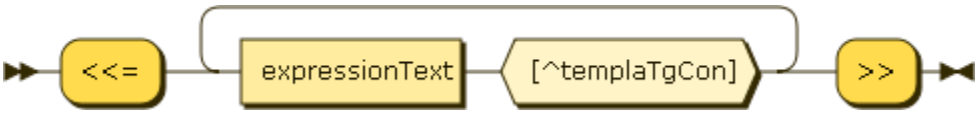
Referenced by:templateBodyContent and templateExactSpaces

(n) templateComment

templateComment
::= '<<//*' commentText '*//>>'

Referenced by:templateBodyContent

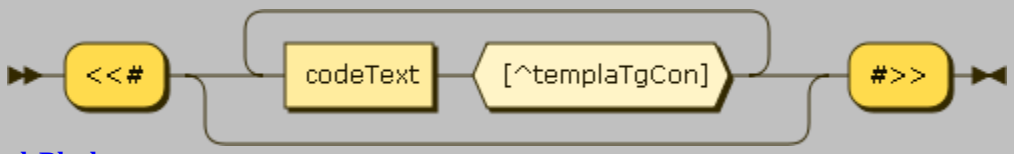
(o) templateExpression



templateExpression
::= '<<=' (expressionText [^templaTgCon])+ '>>'

Referenced by:templateBodyContent

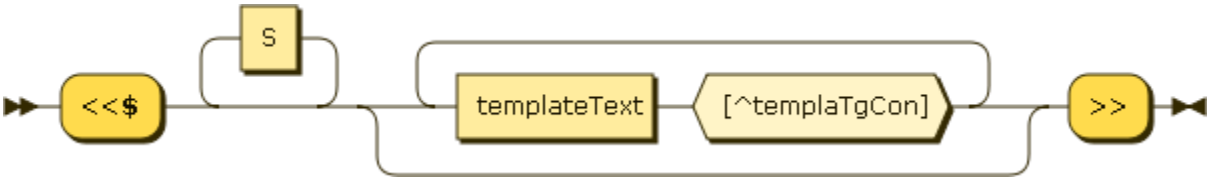
(p) templateCodeBlock



templateCodeBlock
::= '<<#' (codeText [^templaTgCon])* '#>>'

Referenced by:templateBodyContent

(q) templateExactSpaces



templateExactSpaces
::= '<<$' S* (templateText [^templaTgCon])* '>>'

Figure A.1. The diagrams of Grammar 1 (Umple-TL)

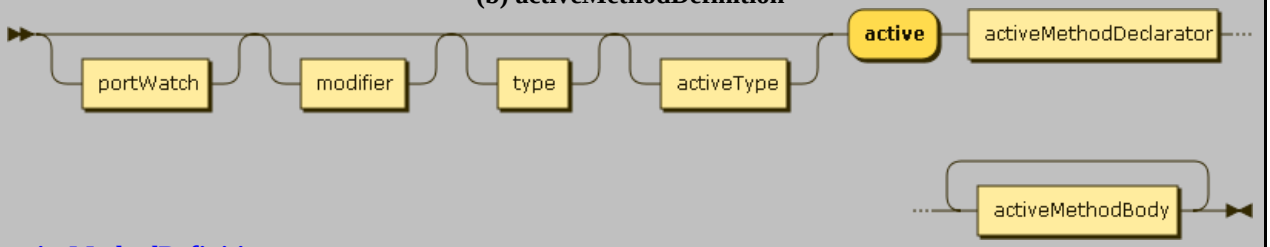
``

The next figures visualize the Active Object grammar (Grammar 2).

(a) Grammar

Grammar ::= activeMethodDefinition*

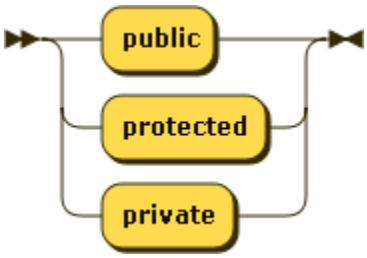
(b) activeMethodDefinition



activeMethodDefinition
::= portWatch? modifier? type? activeType? 'active' activeMethodDeclaratoractiveMethodBody+

Referenced by: Grammar

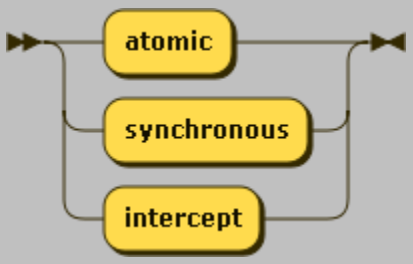
(c) modifier



modifier ::= 'public'
| 'protected'
| 'private'

Referenced by: activeMethodDefinition

(d) activeType



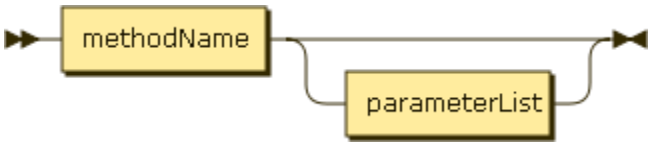
activeType
::= 'atomic'
| 'synchronous'

``

| 'intercept'

Referenced by: activeMethodDefinition

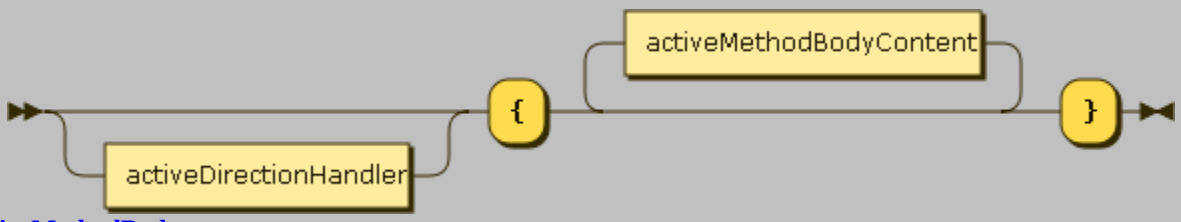
(e) activeMethodDeclarator



activeMethodDeclarator
::= methodNameparameterList?

Referenced by: activeMethodDefinition

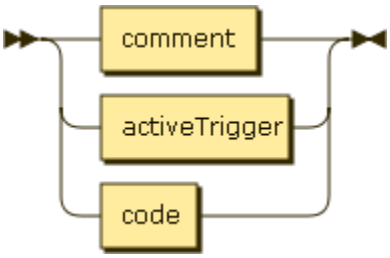
(f) activeMethodBody



activeMethodBody
::= activeDirectionHandler? '{' activeMethodBodyContent* '}'

Referenced by: activeMethodDefinition

(g) activeMethodBodyContent



activeMethodBodyContent
::= comment
| activeTrigger
| code

Referenced by: activeMethodBody

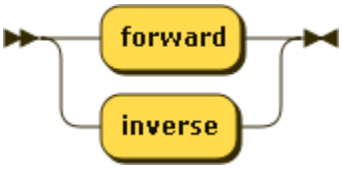
(h) activeDirectionHandler

``

activeDirectionHandler
::= activeDirection (',' activeDirection)*

Referenced by: activeMethodBody

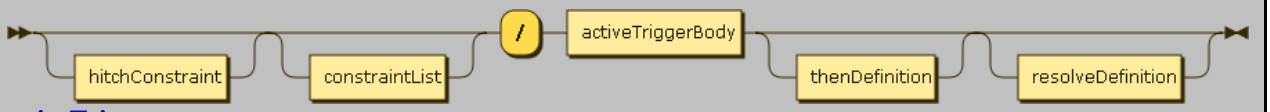
(i) activeDirection



activeDirection
::= 'forward'
| 'inverse'

Referenced by: activeDirectionHandler

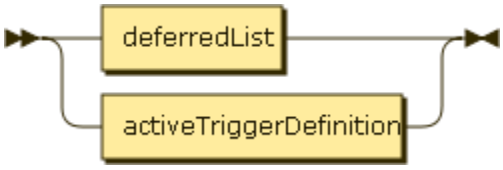
(j) activeTrigger



activeTrigger
::= hitchConstraint? constraintList? '/' activeTriggerBodythenDefinition? resolveDefinition?

Referenced by: activeMethodBodyContent

(k) activeTriggerBody

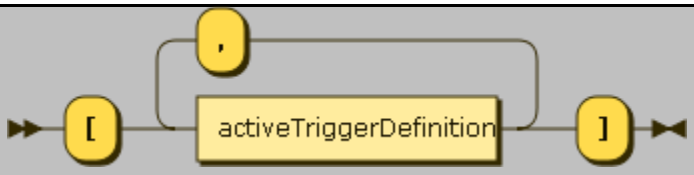


activeTriggerBody
::= deferredList
| activeTriggerDefinition

Referenced by: activeTrigger

(i) deferredList

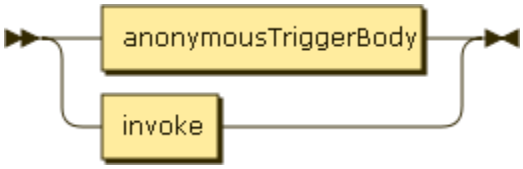
``



deferredList
::= '[' activeTriggerDefinition (',' activeTriggerDefinition)* ']'

Referenced by: activeTriggerBody

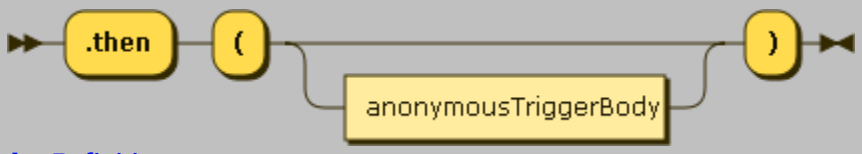
(m) activeTriggerDefinition



activeTriggerDefinition
::= anonymousTriggerBody
| invoke

Referenced by: activeTriggerBody and deferredList

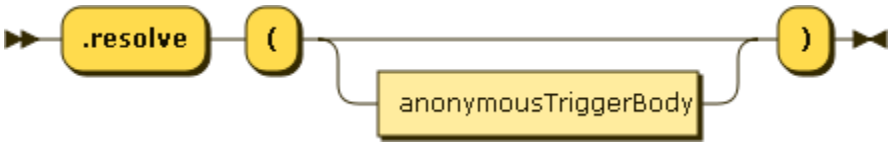
(n) thenDefinition



thenDefinition
::= '.then' '(' anonymousTriggerBody? ')'

Referenced by: activeTrigger

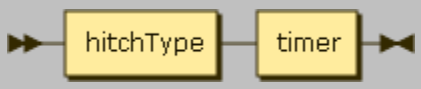
(o) resolveDefinition



resolveDefinition
::= '.resolve' '(' anonymousTriggerBody? ')'

Referenced by: activeTrigger

(p) hitchConstraint



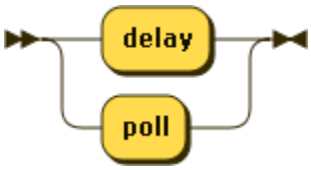
hitchConstraint

``

::= hitchTypetimer

Referenced by: activeTrigger

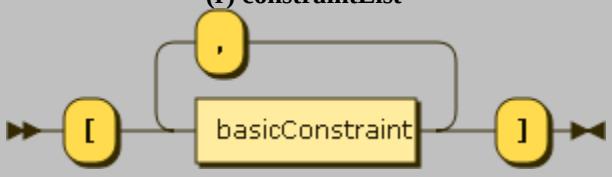
(q) hitchType



hitchType
::= 'delay'
| 'poll'

Referenced by: hitchConstraint

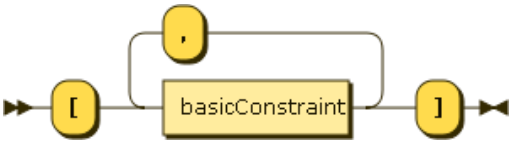
(r) constraintList



constraintList
::= '[' basicConstraint (',' basicConstraint)* ']'

Referenced by: activeTrigger and portWatch

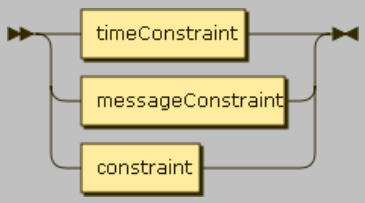
(s) constraintList



constraintList
::= '[' basicConstraint (',' basicConstraint)* ']'

Referenced by: activeTrigger and portWatch

(t) basicConstraint

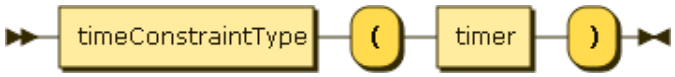


basicConstraint
::= timeConstraint
| messageConstraint
| constraint

``

Referenced by: constraintList

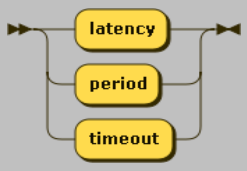
(u) timeConstraint



timeConstraint
::= timeConstraintType '(' timer ')'

Referenced by: basicConstraint

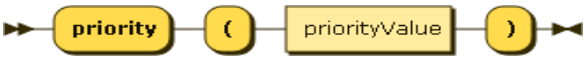
(v) timeConstraintType



timeConstraintType
::= 'latency'
| 'period'
| 'timeout'

Referenced by: timeConstraint

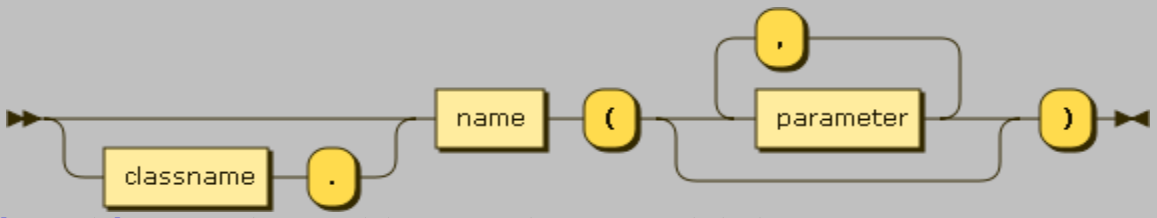
(w) messageConstraint



messageConstraint
::= 'priority' '(' priorityValue ')'

Referenced by: basicConstraint

(x) invoke



invoke ::= (classname '.')? name '(' (parameter (',' parameter)*)? ')'

Referenced by: activeTriggerDefinition

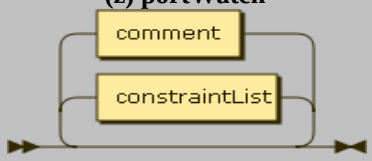
(y) anonymousTriggerBody

anonymousTriggerBody
::= '{' code '}'

``

Referenced by: activeTriggerDefinition, resolveDefinition, and thenDefinition

(z) portWatch



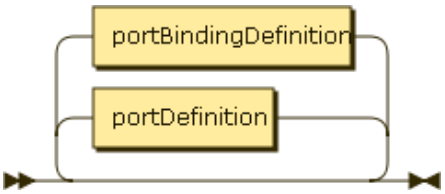
portWatch
::= (constraintList | comment)*
Referenced by: activeMethodDefinition

Figure A.2. The diagrams of Grammar 2

``

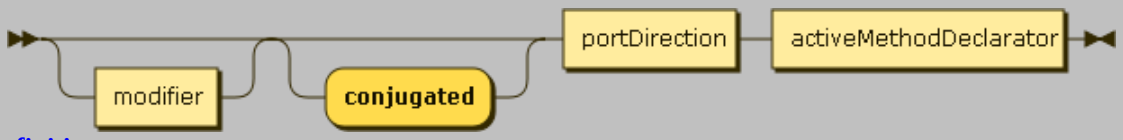
The next figures visualize the composite structure grammar (Grammar 3).

(a) Grammar



Grammar ::= (portDefinition | portBindingDefinition)*

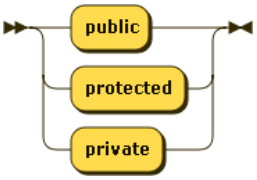
(b) **portDefinition**



portDefinition
::= modifier? 'conjugated'? portDirectionactiveMethodDeclarator

Referenced by:Grammar

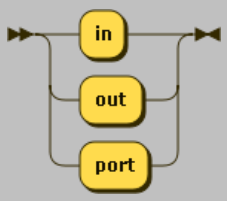
(c) **modifier**



modifier ::= 'public'
| 'protected'
| 'private'

Referenced by:portDefinition

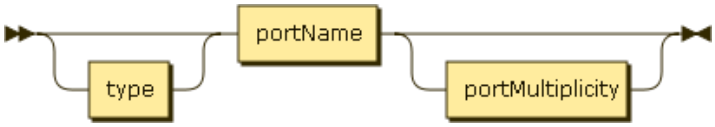
(d) **portDirection**



portDirection
::= 'in'
| 'out'
| 'port'
Referenced by:portDefinition

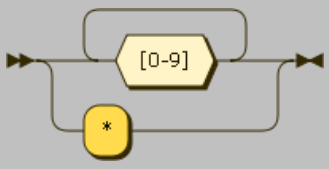
(e) **typedPortName**

``



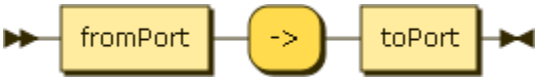
typedPortName
::= type? portNameportMultiplicity?

(f) **portMultiplicity**



portMultiplicity
::= [0-9]+
| '*'
Referenced by:typedPortName

(g) **portBindingDefinition**



portBindingDefinition
::= fromPort '->' toPort
Referenced by:Grammar

Figure A-3. The diagrams of Grammar 3

``

Appendix B

In this appendix, we show almost complete generated code of Snippet 6-5, which is simple in terms of the code at the model level; however, it generates thousands of C++ lines of code shown here.

We preferred to show code in a single column in order to improve readability (there are many single-lined comments and code would have spanned more than two lines because of the narrow space of a column in a two-column page). The exception is PingPong_Model.h, at the end, which we preferred to show in two column as it has the largest amount of code.

Ponger.h

```
#ifndef DEF__PONGER_H
#define DEF__PONGER_H

#ifdef PRAGMA
#pragma once
#ifndef _MSC_VER
#pragma interface "Ponger.h"
#endif
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <Ponger.h>
#include <IPonger.h>


// line 41 "PingPong.ump"
class Ponger : public IPonger{

   private:

      //Ponger Attributes
      TCPMessageRouter _messageRouter;
      Scheduler<Ponger> _internalScheduler;
      Active<Ponger, void, int> internal_pingPort;
      Active<Ponger, void, int> internal_pongPort;
      Active<Ponger, void, int> internal_pong;
      Active<Ponger, void, int> internal_logPortData;
      Active<Ponger, void, int> internal_logPortData2;
      RemoteMethod<Ponger, size_t, void> _hashCode_void;
      //------------------------
      //DESTRUCTOR
      //------------------------
      RemoteMethod<Ponger, void> _deleteAssociatedObjects_void;

      void _pong(int num);
      void _logPortData(int pingPort);
      void _logPortData2(int pongPort);
      void _pingPort(int data);
      void _pongPort(int data);
      void initPortConnections();

   public:
```

``

```
      //Ponger Attributes
      RemoteMethod<Ponger, void, int> pingPort;
      RemoteMethod<Ponger, void, int> pongPort;
      RemoteMethod<Ponger, void, int> logPortData2;
      RemoteMethod<Ponger, void, int> logPortData;
      RemoteMethod<Ponger, void, int> pong;

      //------------------------
      //CONSTRUCTOR
      //------------------------
      Ponger();
      Ponger(Endpoint ep);
      Ponger(unsigned int _portValue);
      Ponger(Ponger& other);

      //------------------------
      //STREAM HELPER GROUPDECLARATION
      //------------------------
      friend ostream& operator<<(ostream& os, const Ponger& dt);

      //------------------------
      //PREDEFINED OPERATORS
      //------------------------
      friend bool operator == (Ponger& Right, Ponger& Left);
      friend bool operator != (Ponger& Right, Ponger& Left){
        return !( Right == Left);
      }
      bool operator == (const Ponger& Right) const{
        return this == &Right;
      }
      bool operator != (const Ponger& Right) const{
        return this != &Right;
      }
      Ponger& operator=(Ponger& other);

      void internalCopy(Ponger& other);
      virtual size_t hashCode(void);
      virtual size_t hashCode_void(void);
      TCPMessageRouter getMessageRouter(void);

      //------------------------
      //DESTRUCTOR
      //------------------------
      virtual ~Ponger();
      void deleteAssociatedObjects(void);
      void deleteAssociatedObjects_void(void);

   protected:

      //------------------------
      //STREAM HELPER GROUPDECLARATION
      //------------------------
      virtual void toOstream(ostream& os) const;
};


//------------------------
//GNU HASH FUNCTION USE
//------------------------
#ifdef __GNUC__
using namespace __gnu_cxx;
namespace __gnu_cxx{
  template<> struct hash<Ponger*>{
```

``


```
    size_t operator()(Ponger* ptr ) const {
        return ptr->hashCode();
    }
  };
}
#include <ext/hash_map>
#else
#include <hash_map>
#endif

#endif
```

``

PingPong_Main.cpp

```
#include <IPinger.h>
#include <IPonger.h>
#include <Pinger.h>
#include <Ponger.h>
#include <PingPong.h>
int main(int argc, char *argv[]){
  return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(PingPong)

set(CMAKE_CXX_STANDARD 11)
include_directories(${CMAKE_SOURCE_DIR})

set(SOURCE_FILES
PingPong_Model.h
PingPong_Main.cpp
IPinger.h
IPinger.cpp
IPonger.h
IPonger.cpp
Pinger.h
Pinger.cpp
Ponger.h
Ponger.cpp
PingPong.h
PingPong.cpp)

add_executable(PingPong ${SOURCE_FILES})
```

Ponger.h

```
#define DEF__PONGER_BODY

#if defined( PRAGMA ) && ! defined( PRAGMA_IMPLEMENTED )
#pragma implementation <Ponger.h>
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <Ponger.h>
#include <IPonger.h>



//------------------------
//CONSTRUCTOR IMPLEMENTATION
//------------------------
Ponger::Ponger():
      _messageRouter(),
      _internalScheduler(this),
      internal_pong(this, &_internalScheduler, &Ponger::_pong),
      internal_pingPort(this, &_internalScheduler, &Ponger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Ponger::_pongPort),
      internal_logPortData(this, &_internalScheduler, &Ponger::_logPortData),
      internal_logPortData2(this, &_internalScheduler, &Ponger::_logPortData2),
```

``

```
      _deleteAssociatedObjects_void(this, &Ponger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _hashCode_void(this, &Ponger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pong(Ponger::internal_pong, "pong", &_messageRouter),
      pingPort(Ponger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Ponger::internal_pongPort, "pongPort", &_messageRouter),
      logPortData(Ponger::internal_logPortData, "logPortData", &_messageRouter),
      logPortData2(Ponger::internal_logPortData2, "logPortData2", &_messageRouter){
  this->initPortConnections();
}

Ponger::Ponger(Endpoint ep):
      _messageRouter(),
      _internalScheduler(this),
      internal_pong(this, &_internalScheduler, &Ponger::_pong),
      internal_pingPort(this, &_internalScheduler, &Ponger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Ponger::_pongPort),
      internal_logPortData(this, &_internalScheduler, &Ponger::_logPortData),
      internal_logPortData2(this, &_internalScheduler, &Ponger::_logPortData2),
      _deleteAssociatedObjects_void(this, &Ponger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _hashCode_void(this, &Ponger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pong(Ponger::internal_pong, "pong", &_messageRouter),
      pingPort(Ponger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Ponger::internal_pongPort, "pongPort", &_messageRouter),
      logPortData(Ponger::internal_logPortData, "logPortData", &_messageRouter),
      logPortData2(Ponger::internal_logPortData2, "logPortData2", &_messageRouter){
  this->initPortConnections();
}

Ponger::Ponger(unsigned int _portValue):
      _messageRouter(),
      _internalScheduler(this),
      internal_pong(this, &_internalScheduler, &Ponger::_pong),
      internal_pingPort(this, &_internalScheduler, &Ponger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Ponger::_pongPort),
      internal_logPortData(this, &_internalScheduler, &Ponger::_logPortData),
      internal_logPortData2(this, &_internalScheduler, &Ponger::_logPortData2),
      _deleteAssociatedObjects_void(this, &Ponger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _hashCode_void(this, &Ponger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pong(Ponger::internal_pong, "pong", &_messageRouter),
      pingPort(Ponger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Ponger::internal_pongPort, "pongPort", &_messageRouter),
      logPortData(Ponger::internal_logPortData, "logPortData", &_messageRouter),
      logPortData2(Ponger::internal_logPortData2, "logPortData2", &_messageRouter){
  this->initPortConnections();
}

Ponger::Ponger(Ponger& other):
      _messageRouter(),
      _internalScheduler(this),
      internal_pong(this, &_internalScheduler, &Ponger::_pong),
      internal_pingPort(this, &_internalScheduler, &Ponger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Ponger::_pongPort),
      internal_logPortData(this, &_internalScheduler, &Ponger::_logPortData),
      internal_logPortData2(this, &_internalScheduler, &Ponger::_logPortData2),
      _deleteAssociatedObjects_void(this, &Ponger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _hashCode_void(this, &Ponger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pong(Ponger::internal_pong, "pong", &_messageRouter),
      pingPort(Ponger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Ponger::internal_pongPort, "pongPort", &_messageRouter),
      logPortData(Ponger::internal_logPortData, "logPortData", &_messageRouter),
```

``


```
      logPortData2(Ponger::internal_logPortData2, "logPortData2", &_messageRouter){
  internalCopy(other);
}

//------------------------
//STREAM HELPER GROUP IMPLEMENTATION
//------------------------
void Ponger::toOstream(ostream& os) const{
  //No Implementation in this context
}

ostream& operator<<(ostream& os, const Ponger& dt){
  dt.toOstream(os);
  return os;
}

//------------------------
//PREDEFINED OPERATORS IMPLEMENTATION
//------------------------
bool operator == (Ponger& Right, Ponger& Left){
  //if (typeid(Right) != typeid(Left)) {
  //  return false;
  //}

  if(Right.hashCode() != Left.hashCode()) {
    return false;
  }
}

Ponger& Ponger::operator=(Ponger& other){
  internalCopy(other);
  return *this;
}

void Ponger::_pong(int num){
  if(!(10 < num)){
    throw "Please provide a valid num";
  }
  pingPort( num + 1);
}

void Ponger::_logPortData(int pingPort){
  cout << "CMP 1 : Ping Out data = " << pingPort << endl;
}

void Ponger::_logPortData2(int pongPort){
  cout << "CMP 1 : Pong Out data = " << pingPort << endl;
}

void Ponger::internalCopy(Ponger& other){
  //No Implementation in this context
}

void Ponger::_pingPort(int data){

}

void Ponger::_pongPort(int data){

}

JSON::JSON(Ponger& aPonger){
  JSON object;
  swap(object);
```

``


```
}

JSON::operator Ponger(){
  if((*this).contains("port")){
    Ponger aPonger((unsigned int)(*this)["port"]);

    return aPonger;
  }else if ((*this).contains("endpoint")){
    Ponger aPonger((Endpoint)(*this)["endpoint"]);

    return aPonger;
  }else {
    Ponger aPonger;

    return aPonger;
  }
}

JSON::JSON(Ponger* aPonger){
  JSON object;
  swap(object);
}

JSON::operator Ponger*(){
  Ponger* aPonger;
  if((*this).contains("port")) {
    aPonger= new Ponger((unsigned int)(*this)["port"]);
  }else if ( (*this).contains("endpoint")) {
    aPonger= new Ponger((Endpoint)(*this)["endpoint"]);
  }else {
    aPonger= new Ponger();
  }

  return aPonger;

}

void Ponger::initPortConnections(){

}

size_t Ponger::hashCode(void){
  return reinterpret_cast<size_t>(this);
}

size_t Ponger::hashCode_void(void){
  return _hashCode_void();
}

TCPMessageRouter Ponger::getMessageRouter(void){
  return this->_messageRouter;
}

//------------------------
//DESTRUCTOR IMPLEMENTATION
//------------------------
Ponger::~Ponger(){
  this->deleteAssociatedObjects();
}

void Ponger::deleteAssociatedObjects(void){
  //No Implementation in this context
}
```

``


```
void Ponger::deleteAssociatedObjects_void(void){
  _deleteAssociatedObjects_void();
}
;
```

``

PingPong.h

```
#ifndef DEF__PINGPONG_H
#define DEF__PINGPONG_H

#ifdef PRAGMA
#pragma once
#ifndef _MSC_VER
#pragma interface "PingPong.h"
#endif
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <PingPong.h>

//------------------------
//LIBRARY INCLUDES
//------------------------
using namespace std;
#include <ostream>

class Pinger;
class Ponger;
// line 52 "PingPong.ump"
class PingPong{

   private:

      //PingPong Attributes
      TCPMessageRouter _messageRouter;
      Scheduler<PingPong> _internalScheduler;
      Pinger* cmp1;
      Ponger* cmp2;
      int startValue;
      RemoteMethod<PingPong, bool, Pinger*> _setCmp1_Pinger;
      RemoteMethod<PingPong, bool, Ponger*> _setCmp2_Ponger;
      RemoteMethod<PingPong, bool, int> _setStartValue_int;
      RemoteMethod<PingPong, Pinger*, void> _getCmp1_void;
      RemoteMethod<PingPong, Ponger*, void> _getCmp2_void;
      RemoteMethod<PingPong, int, void> _getStartValue_void;
      RemoteMethod<PingPong, size_t, void> _hashCode_void;
      //------------------------
      //DESTRUCTOR
      //------------------------
      RemoteMethod<PingPong, void> _deleteAssociatedObjects_void;

      void initPortConnections();

   public:

      //------------------------
      //CONSTRUCTOR
      //------------------------
      PingPong(Pinger* aCmp1, Ponger* aCmp2, const int aStartValue);
      PingPong(Endpoint ep, Pinger* aCmp1, Ponger* aCmp2, const int aStartValue);
      PingPong(unsigned int _portValue, Pinger* aCmp1, Ponger* aCmp2, const int aStartValue);
      PingPong(PingPong& other);

      //------------------------
      //STREAM HELPER GROUPDECLARATION
      //------------------------
      friend ostream& operator<<(ostream& os, const PingPong& dt);
```

``

```
      //------------------------
      //PREDEFINED OPERATORS
      //------------------------
      friend bool operator == (PingPong& Right, PingPong& Left);
      friend bool operator != (PingPong& Right, PingPong& Left){
        return !( Right == Left);
      }
      bool operator == (const PingPong& Right) const{
        return this == &Right;
      }
      bool operator != (const PingPong& Right) const{
        return this != &Right;
      }
      PingPong& operator=(PingPong& other);

      void internalCopy(PingPong& other);
      bool setCmp1(Pinger* aNewCmp1);
      bool setCmp2(Ponger* aNewCmp2);
      bool setStartValue(int aNewStartValue);
      // line 0 ""
      Pinger* getCmp1(void);
      // line 0 ""
      Ponger* getCmp2(void);
      // line 0 ""
      int getStartValue(void);
      bool setCmp1_Pinger(Pinger* aNewCmp1);
      bool setCmp2_Ponger(Ponger* aNewCmp2);
      bool setStartValue_int(int aNewStartValue);
      Pinger* getCmp1_void(void);
      Ponger* getCmp2_void(void);
      int getStartValue_void(void);
      virtual size_t hashCode(void);
      virtual size_t hashCode_void(void);
      TCPMessageRouter getMessageRouter(void);

      //------------------------
      //DESTRUCTOR
      //------------------------
      virtual ~PingPong();
      void deleteAssociatedObjects(void);
      void deleteAssociatedObjects_void(void);

   protected:

      //------------------------
      //STREAM HELPER GROUPDECLARATION
      //------------------------
      virtual void toOstream(ostream& os) const;
};


//------------------------
//GNU HASH FUNCTION USE
//------------------------
#ifdef __GNUC__
using namespace __gnu_cxx;
namespace __gnu_cxx{
  template<> struct hash<PingPong*>{
    size_t operator()(PingPong* ptr ) const {
        return ptr->hashCode();
    }
  };
}
```

``


```
#include <ext/hash_map>
#else
#include <hash_map>
#endif
#endif
```

``

PingPong.cpp

```
#define DEF__PINGPONG_BODY

#if defined( PRAGMA ) && ! defined( PRAGMA_IMPLEMENTED )
#pragma implementation <PingPong.h>
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <PingPong.h>
#include <Pinger.h>
#include <Ponger.h>

//------------------------
//LIBRARY INCLUDES
//------------------------
using namespace std;
#include <ostream>



//------------------------
//CONSTRUCTOR IMPLEMENTATION
//------------------------
PingPong::PingPong(Pinger* aCmp1, Ponger* aCmp2, const int aStartValue):
      _messageRouter(),
      _internalScheduler(this),
      _deleteAssociatedObjects_void(this, &PingPong::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _setCmp1_Pinger(this, &PingPong::setCmp1, &_internalScheduler, "setCmp1", &_messageRouter),
      _setCmp2_Ponger(this, &PingPong::setCmp2, &_internalScheduler, "setCmp2", &_messageRouter),
      _setStartValue_int(this, &PingPong::setStartValue, &_internalScheduler, "setStartValue",
&_messageRouter),
      _getCmp1_void(this, &PingPong::getCmp1, &_internalScheduler, "getCmp1", &_messageRouter),
      _getCmp2_void(this, &PingPong::getCmp2, &_internalScheduler, "getCmp2", &_messageRouter),
      _getStartValue_void(this, &PingPong::getStartValue, &_internalScheduler, "getStartValue",
&_messageRouter),
      _hashCode_void(this, &PingPong::hashCode, &_internalScheduler, "hashCode", &_messageRouter){
  this->cmp1= aCmp1;
  this->cmp2= aCmp2;
  this->startValue= aStartValue;
  this->initPortConnections();
  // line 58 "PingPong.ump"
  cmp1->ping(startValue); // Initiates communication in the constructor
}

PingPong::PingPong(Endpoint ep, Pinger* aCmp1, Ponger* aCmp2, const int aStartValue):
      _messageRouter(),
      _internalScheduler(this),
      _deleteAssociatedObjects_void(this, &PingPong::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _setCmp1_Pinger(this, &PingPong::setCmp1, &_internalScheduler, "setCmp1", &_messageRouter),
      _setCmp2_Ponger(this, &PingPong::setCmp2, &_internalScheduler, "setCmp2", &_messageRouter),
      _setStartValue_int(this, &PingPong::setStartValue, &_internalScheduler, "setStartValue",
&_messageRouter),
      _getCmp1_void(this, &PingPong::getCmp1, &_internalScheduler, "getCmp1", &_messageRouter),
      _getCmp2_void(this, &PingPong::getCmp2, &_internalScheduler, "getCmp2", &_messageRouter),
      _getStartValue_void(this, &PingPong::getStartValue, &_internalScheduler, "getStartValue",
&_messageRouter),
      _hashCode_void(this, &PingPong::hashCode, &_internalScheduler, "hashCode", &_messageRouter){
  this->cmp1= aCmp1;
  this->cmp2= aCmp2;
```

``

```
  this->startValue= aStartValue;
  this->initPortConnections();
}

PingPong::PingPong(unsigned int _portValue, Pinger* aCmp1, Ponger* aCmp2, const int aStartValue):
      _messageRouter(),
      _internalScheduler(this),
      _deleteAssociatedObjects_void(this, &PingPong::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _setCmp1_Pinger(this, &PingPong::setCmp1, &_internalScheduler, "setCmp1", &_messageRouter),
      _setCmp2_Ponger(this, &PingPong::setCmp2, &_internalScheduler, "setCmp2", &_messageRouter),
      _setStartValue_int(this, &PingPong::setStartValue, &_internalScheduler, "setStartValue",
&_messageRouter),
      _getCmp1_void(this, &PingPong::getCmp1, &_internalScheduler, "getCmp1", &_messageRouter),
      _getCmp2_void(this, &PingPong::getCmp2, &_internalScheduler, "getCmp2", &_messageRouter),
      _getStartValue_void(this, &PingPong::getStartValue, &_internalScheduler, "getStartValue",
&_messageRouter),
      _hashCode_void(this, &PingPong::hashCode, &_internalScheduler, "hashCode", &_messageRouter){
  this->cmp1= aCmp1;
  this->cmp2= aCmp2;
  this->startValue= aStartValue;
  this->initPortConnections();
}

PingPong::PingPong(PingPong& other):
      _messageRouter(),
      _internalScheduler(this),
      _deleteAssociatedObjects_void(this, &PingPong::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _setCmp1_Pinger(this, &PingPong::setCmp1, &_internalScheduler, "setCmp1", &_messageRouter),
      _setCmp2_Ponger(this, &PingPong::setCmp2, &_internalScheduler, "setCmp2", &_messageRouter),
      _setStartValue_int(this, &PingPong::setStartValue, &_internalScheduler, "setStartValue",
&_messageRouter),
      _getCmp1_void(this, &PingPong::getCmp1, &_internalScheduler, "getCmp1", &_messageRouter),
      _getCmp2_void(this, &PingPong::getCmp2, &_internalScheduler, "getCmp2", &_messageRouter),
      _getStartValue_void(this, &PingPong::getStartValue, &_internalScheduler, "getStartValue",
&_messageRouter),
      _hashCode_void(this, &PingPong::hashCode, &_internalScheduler, "hashCode", &_messageRouter){
  internalCopy(other);
}

//------------------------
//STREAM HELPER GROUP IMPLEMENTATION
//------------------------
void PingPong::toOstream(ostream& os) const{
  PingPong* thisptr = const_cast<PingPong*>(this);
  os << "[" << "startValue:" << thisptr->getStartValue() << "]" << endl;
  (thisptr->getCmp1()!= NULL ?  os << "cmp1:" << thisptr->getCmp1() : os << "cmp1:" << "NULL")<< endl;
      (thisptr->getCmp2()!= NULL ?  os << "cmp2:" << thisptr->getCmp2() : os << "cmp2:" << "NULL");
}

ostream& operator<<(ostream& os, const PingPong& dt){
  dt.toOstream(os);
  return os;
}

//------------------------
//PREDEFINED OPERATORS IMPLEMENTATION
//------------------------
bool operator == (PingPong& Right, PingPong& Left){
  //if (typeid(Right) != typeid(Left)) {
  //  return false;
  //}
```

``

```
  if(Right.hashCode() != Left.hashCode()) {
    return false;
  }
  if(!compare(Right.cmp1, Left.cmp1, sizeof Right.cmp1)){
    return false;
  }
  if(!compare(Right.cmp2, Left.cmp2, sizeof Right.cmp2)){
    return false;
  }
  if(Right.startValue!= Left.startValue){
     return false;
  }
}

PingPong& PingPong::operator=(PingPong& other){
  internalCopy(other);
  return *this;
}

void PingPong::internalCopy(PingPong& other){
  copyObject(other.cmp1, this->cmp1, sizeof other.cmp1);
  copyObject(other.cmp2, this->cmp2, sizeof other.cmp2);
  this->startValue= other.startValue;
}

bool PingPong::setCmp1(Pinger* aNewCmp1){
  bool wasSet= false;
  this->cmp1 = aNewCmp1;
  wasSet= true;
  return wasSet;
}

bool PingPong::setCmp2(Ponger* aNewCmp2){
  bool wasSet= false;
  this->cmp2 = aNewCmp2;
  wasSet= true;
  return wasSet;
}

bool PingPong::setStartValue(int aNewStartValue){
  bool wasSet= false;
  this->startValue = aNewStartValue;
  wasSet= true;
  return wasSet;
}

// line 0 ""
Pinger* PingPong::getCmp1(void){
  return this->cmp1;
}

// line 0 ""
Ponger* PingPong::getCmp2(void){
  return this->cmp2;
}

// line 0 ""
int PingPong::getStartValue(void){
  return this->startValue;
}

JSON::JSON(PingPong& aPingPong){
  JSON object;
  object["cmp1"] = aPingPong.getCmp1();
```

``

```
  object["cmp2"] = aPingPong.getCmp2();
  object["startValue"] = aPingPong.getStartValue();
  swap(object);
}

JSON::operator PingPong(){
  if((*this).contains("port")){
    PingPong aPingPong((unsigned int)(*this)["port"], (*this)["cmp1"], (*this)["cmp2"],
(*this)["startValue"]);

    return aPingPong;
  }else if ((*this).contains("endpoint")){
    PingPong aPingPong((Endpoint)(*this)["endpoint"], (*this)["cmp1"], (*this)["cmp2"],
(*this)["startValue"]);

    return aPingPong;
  }else {
    PingPong aPingPong((*this)["cmp1"], (*this)["cmp2"], (*this)["startValue"]);

    return aPingPong;
  }
}

JSON::JSON(PingPong* aPingPong){
  JSON object;
  object["cmp1"] = aPingPong->getCmp1();
  object["cmp2"] = aPingPong->getCmp2();
  object["startValue"] = aPingPong->getStartValue();
  swap(object);
}

JSON::operator PingPong*(){
  PingPong* aPingPong;
  if((*this).contains("port")) {
    aPingPong= new PingPong((unsigned int)(*this)["port"], (*this)["cmp1"], (*this)["cmp2"],
(*this)["startValue"]);
  }else if ( (*this).contains("endpoint")) {
    aPingPong= new PingPong((Endpoint)(*this)["endpoint"], (*this)["cmp1"], (*this)["cmp2"],
(*this)["startValue"]);
  }else {
    aPingPong= new PingPong((*this)["cmp1"], (*this)["cmp2"], (*this)["startValue"]);
  }

  return aPingPong;

}

bool PingPong::setCmp1_Pinger(Pinger* aNewCmp1){
  return _setCmp1_Pinger(aNewCmp1);
}

bool PingPong::setCmp2_Ponger(Ponger* aNewCmp2){
  return _setCmp2_Ponger(aNewCmp2);
}

bool PingPong::setStartValue_int(int aNewStartValue){
  return _setStartValue_int(aNewStartValue);
}

Pinger* PingPong::getCmp1_void(void){
  return _getCmp1_void();
}

Ponger* PingPong::getCmp2_void(void){
```

``


```
  return _getCmp2_void();
}

int PingPong::getStartValue_void(void){
  return _getStartValue_void();
}

void PingPong::initPortConnections(){
  cmp1->pingPort+= &cmp2->pongPort;
}

size_t PingPong::hashCode(void){
  return reinterpret_cast<size_t>(this);
}

size_t PingPong::hashCode_void(void){
  return _hashCode_void();
}

TCPMessageRouter PingPong::getMessageRouter(void){
  return this->_messageRouter;
}

//------------------------
//DESTRUCTOR IMPLEMENTATION
//------------------------
PingPong::~PingPong(){
  this->deleteAssociatedObjects();
  delete cmp1;
  delete cmp2;
}

void PingPong::deleteAssociatedObjects(void){
  //No Implementation in this context
}

void PingPong::deleteAssociatedObjects_void(void){
  _deleteAssociatedObjects_void();
}
;
```

``

Pinger.h

```
#ifndef DEF__PINGER_H
#define DEF__PINGER_H

#ifdef PRAGMA
#pragma once
#ifndef _MSC_VER
#pragma interface "Pinger.h"
#endif
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <Pinger.h>
#include <IPinger.h>


// line 35 "PingPong.ump"
class Pinger : public IPinger{

   private:

      //Pinger Attributes
      TCPMessageRouter _messageRouter;
      Scheduler<Pinger> _internalScheduler;
      Active<Pinger, void, int> internal_pingPort;
      Active<Pinger, void, int> internal_pongPort;
      Active<Pinger, void, int, int> internal_ping;
      Active<Pinger, void, int, int> internal_pong;
      RemoteMethod<Pinger, void, int> _ping_int;
      RemoteMethod<Pinger, size_t, void> _hashCode_void;
      //------------------------
      //DESTRUCTOR
      //------------------------
      RemoteMethod<Pinger, void> _deleteAssociatedObjects_void;

      void _ping(int pingPort, int num);
      void _pong(int pongPort, int num);
      void _pingPort(int data);
      void _pongPort(int data);
      void initPortConnections();

   public:

      //Pinger Attributes
      RemoteMethod<Pinger, void, int> pingPort;
      RemoteMethod<Pinger, void, int> pongPort;
      RemoteMethod<Pinger, void, int, int> pong;
      RemoteMethod<Pinger, void, int, int> ping;

      //------------------------
      //CONSTRUCTOR
      //------------------------
      Pinger();
      Pinger(Endpoint ep);
      Pinger(unsigned int _portValue);
      Pinger(Pinger& other);

      //------------------------
      //STREAM HELPER GROUPDECLARATION
      //------------------------
      friend ostream& operator<<(ostream& os, const Pinger& dt);
```

``

```
      //------------------------
      //PREDEFINED OPERATORS
      //------------------------
      friend bool operator == (Pinger& Right, Pinger& Left);
      friend bool operator != (Pinger& Right, Pinger& Left){
        return !( Right == Left);
      }
      bool operator == (const Pinger& Right) const{
        return this == &Right;
      }
      bool operator != (const Pinger& Right) const{
        return this != &Right;
      }
      Pinger& operator=(Pinger& other);

      void internalCopy(Pinger& other);
      void ping_int(int pIn);
      virtual size_t hashCode(void);
      virtual size_t hashCode_void(void);
      TCPMessageRouter getMessageRouter(void);

      //------------------------
      //DESTRUCTOR
      //------------------------
      virtual ~Pinger();
      void deleteAssociatedObjects(void);
      void deleteAssociatedObjects_void(void);

   protected:

      //------------------------
      //STREAM HELPER GROUPDECLARATION
      //------------------------
      virtual void toOstream(ostream& os) const;
};


//------------------------
//GNU HASH FUNCTION USE
//------------------------
#ifdef __GNUC__
using namespace __gnu_cxx;
namespace __gnu_cxx{
  template<> struct hash<Pinger*>{
    size_t operator()(Pinger* ptr ) const {
        return ptr->hashCode();
    }
  };
}
#include <ext/hash_map>
#else
#include <hash_map>
#endif

#endif
```

``

Pinger.cpp

```
#define DEF__PINGER_BODY

#if defined( PRAGMA ) && ! defined( PRAGMA_IMPLEMENTED )
#pragma implementation <Pinger.h>
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <Pinger.h>
#include <IPinger.h>
unsigned int UID::uid = 0;
JSONParser *JSON::_parser = new JSONInternalParser();


//------------------------
//CONSTRUCTOR IMPLEMENTATION
//------------------------
Pinger::Pinger():
      _messageRouter(),
      _internalScheduler(this),
      internal_pingPort(this, &_internalScheduler, &Pinger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Pinger::_pongPort),
      internal_ping(this, &_internalScheduler, &Pinger::_ping),
      internal_pong(this, &_internalScheduler, &Pinger::_pong),
      _deleteAssociatedObjects_void(this, &Pinger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _ping_int(this, &Pinger::ping, &_internalScheduler, "ping", &_messageRouter),
      _hashCode_void(this, &Pinger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pingPort(Pinger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Pinger::internal_pongPort, "pongPort", &_messageRouter),
      ping(Pinger::internal_ping, "ping", &_messageRouter),
      pong(Pinger::internal_pong, "pong", &_messageRouter){
  this->initPortConnections();
}

Pinger::Pinger(Endpoint ep):
      _messageRouter(),
      _internalScheduler(this),
      internal_pingPort(this, &_internalScheduler, &Pinger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Pinger::_pongPort),
      internal_ping(this, &_internalScheduler, &Pinger::_ping),
      internal_pong(this, &_internalScheduler, &Pinger::_pong),
      _deleteAssociatedObjects_void(this, &Pinger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _ping_int(this, &Pinger::ping, &_internalScheduler, "ping", &_messageRouter),
      _hashCode_void(this, &Pinger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pingPort(Pinger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Pinger::internal_pongPort, "pongPort", &_messageRouter),
      ping(Pinger::internal_ping, "ping", &_messageRouter),
      pong(Pinger::internal_pong, "pong", &_messageRouter){
  this->initPortConnections();
}

Pinger::Pinger(unsigned int _portValue):
      _messageRouter(),
      _internalScheduler(this),
      internal_pingPort(this, &_internalScheduler, &Pinger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Pinger::_pongPort),
      internal_ping(this, &_internalScheduler, &Pinger::_ping),
      internal_pong(this, &_internalScheduler, &Pinger::_pong),
```

``

```
      _deleteAssociatedObjects_void(this, &Pinger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _ping_int(this, &Pinger::ping, &_internalScheduler, "ping", &_messageRouter),
      _hashCode_void(this, &Pinger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pingPort(Pinger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Pinger::internal_pongPort, "pongPort", &_messageRouter),
      ping(Pinger::internal_ping, "ping", &_messageRouter),
      pong(Pinger::internal_pong, "pong", &_messageRouter){
  this->initPortConnections();
}

Pinger::Pinger(Pinger& other):
      _messageRouter(),
      _internalScheduler(this),
      internal_pingPort(this, &_internalScheduler, &Pinger::_pingPort),
      internal_pongPort(this, &_internalScheduler, &Pinger::_pongPort),
      internal_ping(this, &_internalScheduler, &Pinger::_ping),
      internal_pong(this, &_internalScheduler, &Pinger::_pong),
      _deleteAssociatedObjects_void(this, &Pinger::deleteAssociatedObjects, &_internalScheduler,
"deleteAssociatedObjects", &_messageRouter),
      _ping_int(this, &Pinger::ping, &_internalScheduler, "ping", &_messageRouter),
      _hashCode_void(this, &Pinger::hashCode, &_internalScheduler, "hashCode", &_messageRouter),
      pingPort(Pinger::internal_pingPort, "pingPort", &_messageRouter),
      pongPort(Pinger::internal_pongPort, "pongPort", &_messageRouter),
      ping(Pinger::internal_ping, "ping", &_messageRouter),
      pong(Pinger::internal_pong, "pong", &_messageRouter){
  internalCopy(other);
}

//------------------------
//STREAM HELPER GROUP IMPLEMENTATION
//------------------------
void Pinger::toOstream(ostream& os) const{
  //No Implementation in this context
}

ostream& operator<<(ostream& os, const Pinger& dt){
  dt.toOstream(os);
  return os;
}

//------------------------
//PREDEFINED OPERATORS IMPLEMENTATION
//------------------------
bool operator == (Pinger& Right, Pinger& Left){
  //if (typeid(Right) != typeid(Left)) {
  //  return false;
  //}

  if(Right.hashCode() != Left.hashCode()) {
    return false;
  }
}

Pinger& Pinger::operator=(Pinger& other){
  internalCopy(other);
  return *this;
}

void Pinger::_ping(int pingPort, int num){
  pongPort(num + 1);
}

void Pinger::_pong(int pongPort, int num){
```

```

```
  pingPort(num + 1);
}

void Pinger::internalCopy(Pinger& other){
  //No Implementation in this context
}

void Pinger::_pingPort(int data){

}

void Pinger::_pongPort(int data){

}

JSON::JSON(Pinger& aPinger){
  JSON object;
  swap(object);
}

JSON::operator Pinger(){
  if((*this).contains("port")){
    Pinger aPinger((unsigned int)(*this)["port"]);

    return aPinger;
  }else if ((*this).contains("endpoint")){
    Pinger aPinger((Endpoint)(*this)["endpoint"]);

    return aPinger;
  }else {
    Pinger aPinger;

    return aPinger;
  }
}

JSON::JSON(Pinger* aPinger){
  JSON object;
  swap(object);
}

JSON::operator Pinger*(){
  Pinger* aPinger;
  if((*this).contains("port")) {
    aPinger= new Pinger((unsigned int)(*this)["port"]);
  }else if ((*this).contains("endpoint")) {
    aPinger= new Pinger((Endpoint)(*this)["endpoint"]);
  }else {
    aPinger= new Pinger();
  }

  return aPinger;

}

void Pinger::ping_int(int pIn){
  _ping_int(pIn);
}

void Pinger::initPortConnections(){
  this->pingPort+= &this->ping;
  this->pingPort+= &this->pong;
  this->pongPort+= &this->pong;
}
```
```

``

```
size_t Pinger::hashCode(void){
  return reinterpret_cast<size_t>(this);
}

size_t Pinger::hashCode_void(void){
  return _hashCode_void();
}

TCPMessageRouter Pinger::getMessageRouter(void){
  return this->_messageRouter;
}

//------------------------
//DESTRUCTOR IMPLEMENTATION
//------------------------
Pinger::~Pinger(){
  this->deleteAssociatedObjects();
}

void Pinger::deleteAssociatedObjects(void){
  //No Implementation in this context
}

void Pinger::deleteAssociatedObjects_void(void){
  _deleteAssociatedObjects_void();
}
;
```

``

IPonger.h

```
#ifndef DEF__IPONGER_H
#define DEF__IPONGER_H

#ifdef PRAGMA
#pragma once
#ifndef _MSC_VER
#pragma interface "IPonger.h"
#endif
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <IPonger.h>


// line 5 "PingPong.ump"
class IPonger{

   public:

      //------------------------
      //CONSTRUCTOR
      //------------------------
      IPonger();
      IPonger(IPonger& other);

      void internalCopy(IPonger& other);
      virtual size_t hashCode(void) = 0;

      //------------------------
      //DESTRUCTOR
      //------------------------
      virtual ~IPonger();
};


//------------------------
//GNU HASH FUNCTION USE
//------------------------
#ifdef __GNUC__
using namespace __gnu_cxx;
namespace __gnu_cxx{
  template<> struct hash<IPonger*>{
    size_t operator()(IPonger* ptr ) const {
        return ptr->hashCode();
    }
  };
}
#include <ext/hash_map>
#else
#include <hash_map>
#endif

#endif
```

``

```
//IPonger.cpp
#define DEF__IPONGER_BODY

#if defined( PRAGMA ) && ! defined( PRAGMA_IMPLEMENTED )
#pragma implementation <IPonger.h>
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <IPonger.h>



//------------------------
//CONSTRUCTOR IMPLEMENTATION
//------------------------
IPonger::IPonger(){
  //No Implementation in this context
}

IPonger::IPonger(IPonger& other){
  //No Implementation in this context
}

void IPonger::internalCopy(IPonger& other){
  //No Implementation in this context
}

size_t IPonger::hashCode(void){
  return reinterpret_cast<size_t>(this);
}

//------------------------
//DESTRUCTOR IMPLEMENTATION
//------------------------
IPonger::~IPonger(){
  //No Implementation in this context
}
;
```

``

```
//IPinger.h
#ifndef DEF__IPINGER_H
#define DEF__IPINGER_H

#ifdef PRAGMA
#pragma once
#ifndef _MSC_VER
#pragma interface "IPinger.h"
#endif
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <IPinger.h>


// line 1 "PingPong.ump"
class IPinger{

   public:

      //------------------------
      //CONSTRUCTOR
      //------------------------
      IPinger();
      IPinger(IPinger& other);

      void internalCopy(IPinger& other);
      virtual size_t hashCode(void) = 0;

      //------------------------
      //DESTRUCTOR
      //------------------------
      virtual ~IPinger();
};


//------------------------
//GNU HASH FUNCTION USE
//------------------------
#ifdef __GNUC__
using namespace __gnu_cxx;
namespace __gnu_cxx{
  template<> struct hash<IPinger*>{
    size_t operator()(IPinger* ptr ) const {
        return ptr->hashCode();
    }
  };
}
#include <ext/hash_map>
#else
#include <hash_map>
#endif

#endif
```

``

```
//IPinger.cpp
#define DEF__IPINGER_BODY

#if defined( PRAGMA ) && ! defined( PRAGMA_IMPLEMENTED )
#pragma implementation <IPinger.h>
#endif

//------------------------
//FILE INCLUDES
//------------------------
#include <PingPong_Model.h>
#include <IPinger.h>



//------------------------
//CONSTRUCTOR IMPLEMENTATION
//------------------------
IPinger::IPinger(){
  //No Implementation in this context
}

IPinger::IPinger(IPinger& other){
  //No Implementation in this context
}

void IPinger::internalCopy(IPinger& other){
  //No Implementation in this context
}

size_t IPinger::hashCode(void){
  return reinterpret_cast<size_t>(this);
}

//------------------------
//DESTRUCTOR IMPLEMENTATION
//------------------------
IPinger::~IPinger(){
  //No Implementation in this context
}
;
```

``

PingPong_Model.h

```
#ifndef DEF__
#define DEF__

#if defined(WIN32) || defined(_WIN32) ||
defined(__WIN32__) || defined(__NT__) ||
defined(_WIN64)
#define WINDOWS_OS
// NO PREPROCESSOR DEFINITION FOR PRAGMA
#if _MSC_VER
#define PRAGMA
#pragma warning( disable : 4290 )
#endif
#elif defined(hpux) || defined(__hpux) ||
defined(__hpux__)
#define HPUX_OS
#elif defined(__APPLE__) || defined(macintosh)
#define MAC_OS
#elif defined(bsdi) || defined(__bsdi__)
#define BSD_OS
#endif

#ifdef PRAGMA
#pragma once
#ifdef _MSC_VER
#pragma include_alias("..\\PingPong_Model.h",
"PingPong_Model.h")
#pragma include_alias(".\\IPinger.h",
"/IPinger.h")
#pragma include_alias(".\\IPonger.h",
"/IPonger.h")
#pragma include_alias(".\\Pinger.h", "/Pinger.h")
#pragma include_alias(".\\Ponger.h", "/Ponger.h")
#pragma include_alias(".\\PingPong.h",
"/PingPong.h")
#else
#pragma interface "PingPong_Model.h"
#endif
#endif

//------------------------
// PACKAGE FILES DECLARATION
//------------------------
#include <sstream>
#include <cmath>
#define NETWORK_BUFFER_SIZE 512
#ifndef DELIMITER
#define DELIMITER char(0x0A)
#endif

#ifdef WINDOWS_OS
#include <windows.h>
#include <process.h>
#else
#include <errno.h>
#include <pthread.h>
#include <unistd.h>
#include <cstring>
#include <signal.h>
#endif

#ifdef HPUX_OS
#include <sys/pstat.h>
#elif defined MAC_OS
#undef DEBUG
#include <CoreServices/CoreServices.h>
#elif defined BSD_OS
#include <mach/mach_types.h>
#include <sys/systm.h>
#include <sys/types.h>
#include <sys/sysctl.h>
#endif
//------------------------
//USED LIBRARIES
//------------------------
using namespace std;

//------------------------
//USED LIBRARIES
//------------------------
#include <vector>
#include <algorithm>
#include <iostream>
#include <cstring>
#include <iostream>
#include <queue>
#include <iostream>
#include <map>
#include <exception>
#include <stdexcept>
#include <cassert>
#include "stdio.h"
#include <queue>
#include <map>
#include <queue>
#include <map>
#include <queue>
#include <map>
#include <queue>
#include <map>

//------------------------------
//NAMESPACES AND PREDEFINITIONS
//------------------------------
#ifdef __cplusplus

#endif


//is_pointer
template <typename T> struct remove_const_type {
typedef T type; };
template <typename T> struct
remove_const_type<const T> { typedef T type; };
template <typename T> struct remove_volatile_type
{ typedef T type; };
template <typename T> struct
remove_volatile_type<volatile T> { typedef T
type; };
template <typename T> struct removeType :
remove_const_type<typename
remove_volatile_type<T>::type> {};
template <typename T> struct is_ptr_type { enum {
value = false }; };
template <typename T> struct is_ptr_type<T*> {
enum { value = true }; };
template <typename T> struct is_ptr :
is_ptr_type<typename removeType<T>::type> {};
```

``

```
#define PLACE_HOLDER      int
#define USECS_PER_MSEC      1000
#define MUSECS_PER_SEC      1000
#define USECS_PER_SEC      1000000

#define INSTANCEOF(object, clazz)
!dynamic_cast<clazz*>(object)
#define ARGUMENT_UPPER_LIMIT      10
#define EMPTY()
#define COMMA() ,
#define SEMICOLON() ;
#define TYPENAME_ARGS(i, value) typename
ArgumentType##i
#define TYPENAME_VALUE_ARGS(i, value) typename
ArgumentType##i=value
#define INIT_VALUE_ARG(i, name)
_##name##i=name##i
#define SER_ARG(i, name) _##name##i=transport[i -
1]
#define DES_ARG(i, name) transport[i -
1]=_##name##i
#define NAMED_ARG(i, name) name##i
#define MEMBER_ARG(i, name) ArgumentType##i
name##i
#define INIT_MEMBER_ARG(i, name)
_##name##i(name##i)
#define VOID_ARG(i, value) void
#define CAT(a, ...) a ## __VA_ARGS__
#define REPEAT_DEC(count ,macro, split, ...)
CAT(REPEAT_DEC_,count)(macro, split, __VA_ARGS__)
#define REPEAT_DEC_1(macro, split, ...)
#define REPEAT_DEC_2(macro, split, ...)  macro(1,
__VA_ARGS__)
#define REPEAT_DEC_3(macro, split, ...)  macro(2,
__VA_ARGS__) split() REPEAT_DEC_2(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_4(macro, split, ...)  macro(3,
__VA_ARGS__) split() REPEAT_DEC_3(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_5(macro, split, ...)  macro(4,
__VA_ARGS__) split() REPEAT_DEC_4(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_6(macro, split, ...)  macro(5,
__VA_ARGS__) split() REPEAT_DEC_5(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_7(macro, split, ...)  macro(6,
__VA_ARGS__) split() REPEAT_DEC_6(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_8(macro, split, ...)  macro(7,
__VA_ARGS__) split() REPEAT_DEC_7(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_9(macro, split, ...)  macro(8,
__VA_ARGS__) split() REPEAT_DEC_8(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_10(macro, split, ...) macro(9,
__VA_ARGS__) split() REPEAT_DEC_9(macro, split,
__VA_ARGS__)
#define REPEAT_DEC_11(macro, split, ...)
macro(10, __VA_ARGS__) split()
REPEAT_DEC_10(macro, split, __VA_ARGS__)
#define REPEAT_DEC_12(macro, split, ...)
macro(11, __VA_ARGS__) split()
REPEAT_DEC_11(macro, split, __VA_ARGS__)
#define REPEAT_DEC_13(macro, split, ...)
macro(12, __VA_ARGS__) split()
REPEAT_DEC_12(macro, split, __VA_ARGS__)
#define REPEAT_DEC_14(macro, split, ...)
macro(13, __VA_ARGS__) split()
REPEAT_DEC_13(macro, split, __VA_ARGS__)
#define REPEAT_DEC_15(macro, split, ...)
macro(14, __VA_ARGS__) split()
REPEAT_DEC_14(macro, split, __VA_ARGS__)
#define REPEAT_DEC_16(macro, split, ...)
macro(15, __VA_ARGS__) split()
REPEAT_DEC_15(macro, split, __VA_ARGS__)

#define REPEAT_INC(count, macro, split, ...)
CAT(REPEAT_INC_,count)(macro, split, __VA_ARGS__)
#define REPEAT_INC_1(macro, split, ...)  macro(1,
__VA_ARGS__)
#define REPEAT_INC_2(macro, split, ...)
REPEAT_INC_1(macro, split, __VA_ARGS__) split()
macro(2, __VA_ARGS__)
#define REPEAT_INC_3(macro, split, ...)
REPEAT_INC_2(macro, split, __VA_ARGS__) split()
macro(3, __VA_ARGS__)
#define REPEAT_INC_4(macro, split, ...)
REPEAT_INC_3(macro, split, __VA_ARGS__) split()
macro(4, __VA_ARGS__)
#define REPEAT_INC_5(macro, split, ...)
REPEAT_INC_4(macro, split, __VA_ARGS__) split()
macro(5, __VA_ARGS__)
#define REPEAT_INC_6(macro, split, ...)
REPEAT_INC_5(macro, split, __VA_ARGS__) split()
macro(6, __VA_ARGS__)
#define REPEAT_INC_7(macro, split, ...)
REPEAT_INC_6(macro, split, __VA_ARGS__) split()
macro(7, __VA_ARGS__)
#define REPEAT_INC_8(macro, split, ...)
REPEAT_INC_7(macro, split, __VA_ARGS__) split()
macro(8, __VA_ARGS__)
#define REPEAT_INC_9(macro, split, ...)
REPEAT_INC_8(macro, split, __VA_ARGS__) split()
macro(9, __VA_ARGS__)
#define REPEAT_INC_10(macro, split, ...)
REPEAT_INC_9(macro, split, __VA_ARGS__) split()
macro(10, __VA_ARGS__)
#define REPEAT_INC_11(macro, split, ...)
REPEAT_INC_10(macro, split, __VA_ARGS__) split()
macro(11, __VA_ARGS__)
#define REPEAT_INC_12(macro, split, ...)
REPEAT_INC_11(macro, split, __VA_ARGS__) split()
macro(12, __VA_ARGS__)
#define REPEAT_INC_13(macro, split, ...)
REPEAT_INC_12(macro, split, __VA_ARGS__) split()
macro(13, __VA_ARGS__)
#define REPEAT_INC_14(macro, split, ...)
REPEAT_INC_13(macro, split, __VA_ARGS__) split()
macro(14, __VA_ARGS__)
#define REPEAT_INC_15(macro, split, ...)
REPEAT_INC_14(macro, split, __VA_ARGS__) split()
macro(15, __VA_ARGS__)
```

``

```
#define REPEAT_INC_16(macro, split, ...)
REPEAT_INC_15(macro, split, __VA_ARGS__) split()
macro(16, __VA_ARGS__)

#define VAR_TYPES(N) REPEAT_INC(N, TYPENAME_ARGS,
COMMA)
#define VAR_TYPES_DEFAULT(N,VALUE) REPEAT_INC(N,
TYPENAME_VALUE_ARGS, COMMA, VALUE)
#define VAR_ARGS(N) REPEAT_INC(N, NAMED_ARG,
COMMA, ArgumentType)
#define VAR_NAMED_ARGS(N, name) REPEAT_INC(N,
NAMED_ARG, COMMA, name)
#define VOID_ARGS(N) REPEAT_INC(N, VOID_ARG,
COMMA)

#define VAR_ARGS_MEMBERS(N, name, delim)
REPEAT_INC(N, MEMBER_ARG, delim, name)
#define INIT_VAR_ARGS_MEMBERS(N, name)
REPEAT_INC(N, INIT_MEMBER_ARG, COMMA, name)
#define INIT_VALUE_ARGS(N, name) REPEAT_INC(N,
INIT_VALUE_ARG, SEMICOLON, name)

#define SERIALIZE_ARGS(N, name) REPEAT_INC(N,
SER_ARG, SEMICOLON, name)
#define DESERIALIZE_ARGS(N, name) REPEAT_INC(N,
DES_ARG, SEMICOLON, name)

#define
GENERATE_METHOD_CALLBACK_SIGNATURES_ARGUMENTS(N,
value)      \
template<typename Caller, typename ReturnType,
VAR_TYPES(N)>         \
  struct
MethodCallbackSignature<Caller,ReturnType,
VAR_ARGS(N)> {  \
  typedef
ReturnType(Caller::*Method)(VAR_ARGS(N)); };
\
template<typename Caller, VAR_TYPES(N)>
\
  struct MethodCallbackSignature<Caller, void,
VAR_ARGS(N)> {     \
  typedef void (Caller::*Method)(VAR_ARGS(N)); };

#define
GENERATE_METHOD_CALLBACK_INVOKE_ARGUMENTS(N,
value)                                      \
template <class BASE, class Caller, class
FutureResultType, class ReturnType, VAR_TYPES(N)>
\
class
MethodCallbackInvoke<BASE,Caller,FutureResultType
,ReturnType, VAR_ARGS(N)>
\
    : public BaseMethodCallbackInvoke<BASE,
Caller, FutureResultType, ReturnType> {public:
\
  typedef typename
MethodCallbackSignature<Caller, ReturnType,
VAR_ARGS(N)>::Method Callback;                  \
  MethodCallbackInvoke(Caller* caller, Callback
method, VAR_ARGS_MEMBERS(N, arg, COMMA), const
FutureResultType& result) :  \
    BaseMethodCallbackInvoke(caller, result),
_method(method), INIT_VAR_ARGS_MEMBERS(N, arg) {}
\
protected: VAR_ARGS_MEMBERS(N, _arg, SEMICOLON);
Callback _method; };

#define GENERATE_DELEGATE_INVOKE_ARGUMENTS(N,
value)                                  \
template <class BASE, class Caller, class
FutureType, class ReturnType, VAR_TYPES(N)>
\
class DelegateInvoke<BASE, Caller, FutureType,
ReturnType, VAR_ARGS(N)> :                      \
public MethodCallbackInvoke<BASE, Caller,
FutureType, ReturnType, VAR_ARGS(N)> {public:
\
  DelegateInvoke(Caller* caller, Callback method,
VAR_ARGS_MEMBERS(N, arg, COMMA), const
FutureType& result)  \
    : MethodCallbackInvoke(caller, method,
VAR_NAMED_ARGS(N, arg), result) {}
\
  void invokeMethod() { _result.resolveData(new
ReturnType((_context->*_method)(VAR_NAMED_ARGS(N,
_arg))));}};  \
template <class BASE, class Caller, class
FutureType, VAR_TYPES(N)>
\
class DelegateInvoke<BASE, Caller, FutureType,
void, VAR_ARGS(N)> :                          \
public MethodCallbackInvoke<BASE, Caller,
FutureType, void, VAR_ARGS(N)>{public:
\
  DelegateInvoke(Caller* caller, Callback method,
VAR_ARGS_MEMBERS(N, arg, COMMA), const
FutureType& result)  \
    : MethodCallbackInvoke(caller, method,
VAR_NAMED_ARGS(N, arg), result) {}
\
  void invokeMethod() { (_context-
>*_method)(VAR_NAMED_ARGS(N, _arg));}};

#define GENERATE_DELEGATE_ARGUMENTS(N, value)
\
template <class Caller, class ReturnType,
VAR_TYPES(N)> class Delegate<Caller, ReturnType,
VAR_ARGS(N)> :             \
  public DelegateInvoke < DelegateBase, Caller,
FutureResult<ReturnType>, ReturnType, VAR_ARGS(N)
> {              \
  public:  Delegate(Caller* caller, Callback
method, VAR_ARGS_MEMBERS(N, arg, COMMA), const
FutureResult<ReturnType>& result)  \
    : DelegateInvoke(caller, method,
VAR_NAMED_ARGS(N, arg), result) {} };
\
template <class Caller, VAR_TYPES(N)> class
Delegate<Caller, void, VAR_TYPES(N)> :
\
  public DelegateInvoke < DelegateBase, Caller,
FutureResult<void>, void, VAR_ARGS(N) >{
\
```

``

```
  public:  Delegate(Caller* caller, Callback
method, VAR_ARGS_MEMBERS(N, arg, COMMA), const
FutureResult<void>& result)      \
      : DelegateInvoke(caller, method,
VAR_NAMED_ARGS(N, arg), result) {}};

#define GENERATE_MULTICAST_ARGUMENTS(N, value)
\
template<class ReturnType, VAR_TYPES(N)>
\
class IDelegatePublisher<ReturnType, VAR_ARGS(N)>
{public:
\
  virtual FutureResult<ReturnType>
publish(VAR_ARGS_MEMBERS(N, arg, COMMA), int
priority = 0, long delay = 0, long timeout = 0) =
0;            \
};
\
template <class ReturnType, VAR_TYPES(N)>
\
class MulticastDelegate<ReturnType, VAR_ARGS(N)>
: public IDelegatePublisher<ReturnType,
VAR_ARGS(N)>{private:                        \
  typedef std::vector<
IDelegatePublisher<ReturnType, VAR_ARGS(N)>* >
SubscribersList;
\
  SubscribersList subscribers;
\
public:
\
  MulticastDelegate() {}
\
  MulticastDelegate& operator +=
(IDelegatePublisher<ReturnType, VAR_ARGS(N)>*
method) {                                    \
    subscribers.push_back(method);
\
    return *this;}
\
  FutureResult<ReturnType> operator ()
(VAR_ARGS_MEMBERS(N, arg, COMMA), int priority =
0, long delay = 0, long timeout = 0) {
\
    FutureResult<ReturnType> result =
publish(VAR_NAMED_ARGS(N, arg), priority, delay,
timeout);                              \
    typename SubscribersList::iterator it =
subscribers.begin();
\
    for (; it != subscribers.end(); it++) {
(*it)->publish(VAR_NAMED_ARGS(N, arg), priority,
delay, timeout); }                      \
    return result;}};

#define GENERATE_ACTIVE_ARGUMENTS(N, value)
\
template <class Caller, class ReturnType,
VAR_TYPES(N)> class Active<Caller, ReturnType,
VAR_ARGS(N)> : public ActiveConstraintUID, public
MulticastDelegate<ReturnType, VAR_ARGS(N)> {
public:  \
  typedef Delegate<Caller, ReturnType,
VAR_ARGS(N)> DelegateType;
\
  typedef typename
MethodCallbackSignature<Caller, ReturnType,
VAR_ARGS(N)>::Method Callback;
\
  Active(Caller* caller, Scheduler<Caller>* sch,
Callback method) :_context(caller), _sch(sch),
_method(method) {}
\
  FutureResult<ReturnType>
publish(VAR_ARGS_MEMBERS(N, arg, COMMA), int
priority = 0, long delay = 0, long timeout = 0) {
\
    FutureResult<ReturnType> result(new
FutureObject<ReturnType>());
\
    DelegateBase::Ptr pDelegate(new
DelegateType(_context, _method, VAR_NAMED_ARGS(N,
arg), result));
\
    _sch->schedule(pDelegate,priority,delay,
timeout, _guardId, _conditionId);
\
    return result;} private: Caller* _context;
Scheduler<Caller>* _sch; Callback _method; };

#define GENERATE_METHOD_CALLBACK_SIGNATURES(N)
REPEAT_DEC(N,
GENERATE_METHOD_CALLBACK_SIGNATURES_ARGUMENTS,
EMPTY)
#define GENERATE_METHOD_CALLBACK_INVOKE(N)
REPEAT_DEC(N,
GENERATE_METHOD_CALLBACK_INVOKE_ARGUMENTS, EMPTY)
#define GENERATE_DELEGATE_INVOKE(N)
REPEAT_DEC(N, GENERATE_DELEGATE_INVOKE_ARGUMENTS,
EMPTY)
#define GENERATE_DELEGATE(N)  REPEAT_DEC(N,
GENERATE_DELEGATE_ARGUMENTS, EMPTY)
#define GENERATE_MULTICAST_METHOD(N)
REPEAT_DEC(N, GENERATE_MULTICAST_ARGUMENTS,
EMPTY)
#define GENERATE_ACTIVE_METHOD(N)  REPEAT_DEC(N,
GENERATE_ACTIVE_ARGUMENTS, EMPTY)

#ifdef WINDOWS_OS
#define isnan(x) _isnan(x)
#define isinf(x) (!_finite(x))

#define SOCKET_TYPE            SOCKET
#define CLOSE_SOCKET(arg) \
  closesocket(arg)

#define EVENT_TYPE            HANDLE
#define CONDITION_TYPE          PLACE_HOLDER
#define THREAD_TYPE             HANDLE
#define THREAD_RETURN_TYPE        unsigned WINAPI
#define THREAD_ERROR_INSTANCE(returnValue)
((returnValue) == NULL)
#define THREAD_ERROR_CODE(value)
GetLastError()
```

``

```
#define MUTEX_CRITICAL_SECTION
CRITICAL_SECTION
#define START_MUTEX_FUNCTION(arg) \
  InitializeCriticalSection((arg))

#define TERMINATE_MUTEX_FUNCTION(arg) \
  DeleteCriticalSection((arg))

#define LOCK_MUTEX_FUNCTION(arg) \
  EnterCriticalSection((arg))

#define UNLOCK_MUTEX_FUNCTION(arg) \
  LeaveCriticalSection((arg))

#define START_EVENT_TYPE_FUNCTION(mutex, cond,
reset) \
  mutex = CreateEvent(NULL, reset, FALSE, NULL);
\
  if (!mutex)                          \
    throw ThreadException("mutex signal failed")

#define TERMINATE_EVENT_FUNCTION(mutex, cond) \
  CloseHandle(mutex)

#define WAIT_EVENT_FUNCTION(mutex, cond, wakeup)
\
  switch(WaitForSingleObject(mutex, INFINITE)) {
\
  case WAIT_OBJECT_0:                  \
    return;                       \
  default:                       \
    throw ThreadException("wait event failed");
\
  }

#define WAIT_TIME_EVENT_FUNCTION(mutex, cond,
time, wakeup, reset, status)      \
  switch (WaitForSingleObject(mutex, time + 1))
\
  {                                        \
  case WAIT_OBJECT_0:
\
    status = true;
\
    break;
\
  case WAIT_TIMEOUT:
\
    status = false;
\
    break;
\
  default:                                    \
    throw ThreadException("wait failed");
\
  }

#define WAKEUP_EVENT_FUNCTION(mutex, cond,
wakeup) \
  SetEvent(mutex)

#define THREAD_JOIN_FUNCTION(hdl)
WaitForSingleObject(hdl, INFINITE)

#define THREAD_SLEEP_FUNCTION(ms)
Sleep((ms))

#define THREAD_CREATE_FUNCTION(id, funPtr,
callPtr)  id
=(HANDLE)CreateThread(NULL,0,(LPTHREAD_START_ROUT
INE)funPtr,callPtr,0L,NULL)

#define THREAD_TERMINATE_FUNCTION(hdl)
TerminateThread(hdl, 0)

#define THREAD_CANCEL_FUNCTION(hdl)
TerminateThread(hdl, 0)

#define IS_THREAD_ALIVE_FUNCTION(hdl, isRunning)
\
  DWORD exitCode = 0;                \
  if(GetExitCodeThread(hdl, &exitCode))       \
    isRunning = (exitCode == STILL_ACTIVE)

#define BROADCAST_FUNCTION(arg) 0

#define SET_EVENT_FUNCTION(arg) \
  SetEvent((arg))

#define RESET_EVENT_FUNCTION(arg) \
  ResetEvent((arg))

#define LOCK_MUTEX_EVENT_FUNCTION(arg) 0
#define UNLOCK_MUTEX_EVENT_FUNCTION(arg) 0

#else
typedef int                 BOOL;

#ifndef FALSE
#define FALSE               0
#endif

#ifndef TRUE
#define TRUE                1
#endif
#define SOCKET_TYPE            int
#define CLOSE_SOCKET(arg) \
  close(arg)

#define EVENT_TYPE              pthread_mutex_t
#define CONDITION_TYPE            pthread_cond_t

#define THREAD_TYPE              pthread_t
#define THREAD_RETURN_TYPE          void *

#define THREAD_ERROR_INSTANCE(returnValue)
((returnValue) == NULL)
#define THREAD_ERROR_CODE(value)      errno

#define MUTEX_CRITICAL_SECTION
pthread_mutex_t
#define START_MUTEX_FUNCTION(arg)  \
  pthread_mutex_init ((arg), NULL)

#define TERMINATE_MUTEX_FUNCTION(arg) \
  pthread_mutex_destroy((arg))

#define LOCK_MUTEX_FUNCTION(arg)  \
```

``

```
  pthread_mutex_lock((arg))

#define UNLOCK_MUTEX_FUNCTION(arg) \
  pthread_mutex_unlock((arg))

#define START_EVENT_TYPE_FUNCTION(mutex, cond,
reset) \
   if (pthread_mutex_init(&mutex, NULL))        \
    throw ThreadException("mutex signal failed");
\
   pthread_cond_init(&cond, NULL)

#define TERMINATE_EVENT_FUNCTION(mutex, cond) \
  pthread_cond_destroy(&cond); \
  pthread_mutex_destroy(&mutex)

#define WAIT_EVENT_FUNCTION(mutex, cond, wakeup)
\
  pthread_mutex_lock(&mutex);              \
  int err = 0;                    \
  while (!wakeup) {                        \
    err = pthread_cond_wait(&cond, &mutex);
\
    if (err) {                    \
      pthread_mutex_unlock(&mutex);          \
      throw ThreadException("wait event failed");
\
    }                         \
  }                           \
  wakeup = FALSE;                    \
  pthread_mutex_unlock(&mutex)

#define WAIT_TIME_EVENT_FUNCTION(mutex, cond, ms,
wakeup, reset, status)      \
  struct timeval tv
\
  struct timespec tdif
\
  gettimeofday(&tv, NULL)
\
  tdif.tv_sec  = tv.tv_sec + ms / MUSECS_PER_SEC
\
  tdif.tv_nsec = tv.tv_usec*MUSECS_PER_SEC + (ms
% MUSECS_PER_SEC)*USECS_PER_SEC  \
  if (tdif.tv_nsec >= NSECS_PER_SEC) {
\
    tdif.tv_nsec -= NSECS_PER_SEC
\
    tdif.tv_sec++
\
      }                                   \
  pthread_mutex_lock(&mutex)
\
      while (!wakeup)
\
           {                              \
    status = pthread_cond_timedwait(&cond,
&mutex, &tdif)            \
    if(status) {                                \
      if (status == ETIMEDOUT) break;
\
      pthread_mutex_unlock(&mutex)
\
      throw ThreadException(get_error(status))
\
           }                              \
           }                              \
  wakeup =  status == 0 && reset ? false : wakeup
\
  pthread_mutex_unlock(&mutex)

#define WAKEUP_EVENT_FUNCTION(mutex, cond,
wakeup)  \
  pthread_mutex_lock(&mutex);            \
  wakeup = TRUE;                  \
  pthread_cond_signal(&cond);            \
  pthread_mutex_unlock(&mutex)

#define THREAD_JOIN_FUNCTION(id)
pthread_join(id, NULL)
#define THREAD_SLEEP_FUNCTION(ms) \
  struct timeval tv; \
    tv.tv_usec = (ms % MUSECS_PER_SEC) *
USECS_PER_MSEC; \
    tv.tv_sec = ms / MUSECS_PER_SEC; \
    select(0, NULL, NULL, NULL, &tv)

#define THREAD_CREATE_FUNCTION(id, funPtr,
callPtr)        \
  pthread_attr_t attr;                    \
  pthread_attr_init(&attr);                  \

pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_
DETACHED);  \
  pthread_attr_setinheritsched(&attr,
PTHREAD_INHERIT_SCHED);  \
  pthread_create(&id, &attr, funPtr, callPtr);
\
  pthread_attr_destroy(&attr)

#define THREAD_TERMINATE_FUNCTION(arg)
pthread_exit(arg)
#define THREAD_CANCEL_FUNCTION(Id)          \
  if (pthread_cancel(Id) == 0)          \
    pthread_detach(Id);

#define IS_THREAD_ALIVE_FUNCTION(hdl, isRunning)
\
  isRunning = (pthread_kill(hdl, 0) == 0)

#define BROADCAST_FUNCTION(arg) \
  pthread_cond_broadcast((arg))

#define SET_EVENT_FUNCTION(arg) 1

#define RESET_EVENT_FUNCTION(arg) 1

#define LOCK_MUTEX_EVENT_FUNCTION(arg)  \
  pthread_mutex_lock((arg))

#define UNLOCK_MUTEX_EVENT_FUNCTION(arg) \
  pthread_mutex_unlock((arg))

#endif

#define CREATE_THREAD(id, funPtr, callPtr)
\
```

``

```
  THREAD_CREATE_FUNCTION(id, funPtr, callPtr);
\
  if(THREAD_ERROR_INSTANCE(id))
\
    throw
ThreadException(ErrorMessage(THREAD_ERROR_CODE(id
)))

static std::string ErrorMessage(int errorCode){
  string str = "";
  if (errorCode == 0) {
    return str;
  }

  char const* what = "Error Numer";
  int whlen = strlen(what);
  int ncode = errorCode, dlen = 1;
  while (ncode !=0) {dlen++; ncode/=10;}
  char *buffer = (char *) malloc(sizeof(char) *
(whlen + dlen + 1));
    sprintf(buffer, "%s = %d", what, errorCode);
  str = buffer;
  return str;
}

class Exception : public std::exception {
  friend ostream& operator<<(ostream& output,
const Exception& ex) {
    output << "Exception: " << ex.error;
    return output;
  }
public:
  Exception() throw() :
error(std::string("Exception")) {}
  Exception(std::string err) throw() : error(err)
{}
  Exception(const Exception& source) throw() :
std::exception(source) { error = source.error; }
  virtual ~Exception() throw () {}
  Exception& operator=(const Exception& source)
throw() {
    if (&source != this) {
      error = source.error;
    }
    std::exception::operator= (source);
    return *this;
  }
  void setError(std::string exce) { error = exce;
}
  virtual const char* what() const throw() {
return error.c_str(); }
protected:
  std::string error;
};

struct ThreadException : public Exception{
public:
  ThreadException() : Exception() {}
  ThreadException(char* errorMessage) :
Exception(errorMessage) {}
  ThreadException(std::string errorMessage) :
Exception(errorMessage) {}
};
```

```
template <typename T>
void* ConvertToFunctionPointer(T x) {
  return *reinterpret_cast<void**>(&x);
}

struct Runnable {
  virtual void run() = 0;
};

class AtomicMutex{
  public:
  AtomicMutex() { START_MUTEX_FUNCTION(&section);
}
  ~AtomicMutex() {
TERMINATE_MUTEX_FUNCTION(&section); }
    void lock() { LOCK_MUTEX_FUNCTION(&section);
}
    void unlock() {
UNLOCK_MUTEX_FUNCTION(&section); }

  private:
  MUTEX_CRITICAL_SECTION section;
};

static AtomicMutex atomicMutex;

class MutexLock{
public:

  MutexLock() : m_value(0){
    wakeup = FALSE;
    START_EVENT_TYPE_FUNCTION(mutex, cond,
FALSE);
  }

  MutexLock(const MutexLock& m) {
    this->m_value = m.m_value;
    this->wakeup = m.wakeup;
    this->mutex = m.mutex;
    this->cond = m.cond;
  }

  ~MutexLock(){
    TERMINATE_EVENT_FUNCTION(mutex, cond);
  }

  void wait() {
    WAIT_EVENT_FUNCTION(mutex, cond, wakeup);
  }

  void wakeUp() {
    WAKEUP_EVENT_FUNCTION(mutex, cond, wakeup);
  }

  void lock() {
    bool canBeAccessed = this->verifyLock();
    if (canBeAccessed){
      return;
    }

    atomicMutex.lock();
    canBeAccessed = m_value == 0;
    m_value += 1;
    atomicMutex.unlock();
```

``

```
    if (!canBeAccessed){
      wait();
      atomicMutex.lock();
      m_value -= 1;
      atomicMutex.unlock();
    }

  }

  bool isLocked() {
    atomicMutex.lock();
    bool canBeAccessed = m_value == 0;
    atomicMutex.unlock();
    return !canBeAccessed;
  }

  bool verifyLock(int set = 0) {
    atomicMutex.lock();
    bool canBeAccessed = m_value == 0;
    if (m_value == set) {
      m_value = !set;
      canBeAccessed = true;
    }
    atomicMutex.unlock();
    return canBeAccessed;
  }

  void unlock() {
    if (!verifyLock(1)){
      wakeUp();
    }
  }

private:
  volatile int m_value;
  volatile bool wakeup;

  EVENT_TYPE mutex;
  CONDITION_TYPE  cond;
};

#define synchronized(L)
\
        for(L.lock();L.isLocked();
\
      L.unlock() )

struct ThreadParameters
{
    void* runCall;
    void* context;
    ThreadParameters(void* ctx, void* callPtr) :
context(ctx), runCall(callPtr) {}
};

class Thread: public Runnable {

  public:
    Thread(string threadName = "") :
runnableObject(NULL), name(threadName),
thrParams(this,ConvertToFunctionPointer(&Thread::
run))
      ,
_threadHdl(0),_isRunning(false),_isTerminated(fal
se) {}
    Thread(Runnable *target, string threadName =
"") :
name(threadName),thrParams(this,ConvertToFunction
Pointer(&Thread::run)) , runnableObject(target)
{}
    Thread(void* funcPtr, void* ctx = 0 ,string
threadName = "") : runnableObject(NULL),
name(threadName), thrParams(ctx,funcPtr) {}
    Thread(void (*funcPtr)(void*), void* ctx = 0,
string threadName = "") :
runnableObject(NULL),name(threadName),
thrParams(ctx,ConvertToFunctionPointer(funcPtr))
{}
    Thread(void (*funcPtr)(), string threadName =
"") : runnableObject(NULL),name(threadName),
thrParams(this,ConvertToFunctionPointer(funcPtr))
{}
    template<class T>
    Thread(void (T::*RunnableCall)(), string
threadName = "") : runnableObject(NULL),
name(threadName),
thrParams(this,ConvertToFunctionPointer(RunnableC
all)) {}
    template<class T>
    Thread(void (T::*RunnableCall)(void*), void*
ctx = 0, string threadName = "") :
runnableObject(NULL), name(threadName),
thrParams(ctx,ConvertToFunctionPointer(RunnableCa
ll)) {}
    virtual ~Thread() {
      THREAD_TERMINATE_FUNCTION(0);
    }

    static void sleep(long ms)
throw(ThreadException) {
      THREAD_SLEEP_FUNCTION(ms);
    }

    THREAD_TYPE getId() const {
      return this->_threadHdl;
    }

    static THREAD_RETURN_TYPE
threadFunctionPointer(void* ptr) {
      ThreadParameters* threadParameters =
(ThreadParameters*)ptr;
      if(threadParameters->context != NULL) {
        Runnable* run = (Thread*)
threadParameters->context;
        run->run();
        ///((void (*)(void*)) threadParameters-
>runCall)(threadParameters->context);
      } else {
          ((void (*)(void)) threadParameters-
>runCall)();
      }

      return 0;
    }

    bool isFinished() {
```

``

```
      return !this->isRunning();
    }

    bool isTerminated() {
      synchronized(lock) {
        return this->_isTerminated;
      }
      return 0;
    }

    bool isRunning() {
      synchronized(lock) {
        if(this->_isRunning) {

IS_THREAD_ALIVE_FUNCTION(_threadHdl,_isRunning);
        }
        return this->_isRunning;
      }

      return 0;
    }

    virtual void run() {
      if(this->runnableObject != NULL) {
        runnableObject->run();
      }
    }

    virtual void stop() {
      synchronized(lock) {
        THREAD_CANCEL_FUNCTION(_threadHdl);
        _isTerminated = true;
        _isRunning = false;
      }
    }

    string getName() const {
      return name;
    }

    void setName(string name) {
      this->name = name;
    }

    virtual void start(Runnable *target)
throw(ThreadException) {
      runnableObject = target;
      start();
    }

    virtual void start() throw(ThreadException) {
      synchronized(lock) {
        reset();

CREATE_THREAD(_threadHdl,Thread::threadFunctionPo
inter, &thrParams);
        _isRunning = true;
      }
    }

    void join() throw(ThreadException) {
      THREAD_JOIN_FUNCTION(_threadHdl);
    }

    void wait() throw(ThreadException) {
      lock.wait();
    }

    void wakeUp() throw(ThreadException) {
      lock.wakeUp();
    }

  private:
    mutable MutexLock lock;
    THREAD_TYPE _threadHdl;
    string name;

    ThreadParameters thrParams;
    Runnable* runnableObject;

    bool _isRunning;
    bool _isTerminated;

    void reset() {
      _threadHdl = 0;
      _isRunning = false;
      _isTerminated = false;
    }
};

template <class PT>
class AutoPtr {
public:
  AutoPtr(PT* p = 0, bool shared = false) :
_ptr(p) { if (shared && _ptr) _ptr->reference();
}
  AutoPtr(const AutoPtr& ptr) : _ptr(ptr._ptr) {
if (_ptr) _ptr->reference(); }
  ~AutoPtr() { if (_ptr) _ptr->release(); }
  AutoPtr& operator = (const AutoPtr& ptr) {
    if (&ptr != this) {
      if (_ptr) _ptr->release();
      _ptr = ptr._ptr;
      if (_ptr) _ptr->reference();
    }
    return *this;
  }
  PT* operator -> () {
    if (_ptr)
      return _ptr;
    else
      throw std::exception("Null Pointer
Exception");
  }
  PT* reference() { if (_ptr) _ptr->reference();
return _ptr; }
  PT* value() { return _ptr; }
  operator const PT* () const { return _ptr; }
  bool operator == (const AutoPtr& ptr) const {
return _ptr == ptr._ptr; }
  bool operator != (const AutoPtr& ptr) const {
return _ptr != ptr._ptr; }
private:
  PT* _ptr;
};

class ReferenceObject {
private:
```

``

```
  mutable MutexLock mutex;
  mutable volatile int referenceNumber;
public:
  ReferenceObject() {
    synchronized(mutex) {
      referenceNumber = 1;
    }
  }
  void reference() const {
    synchronized(mutex) {
      ++referenceNumber;
    }
  }
  void release() const {
    synchronized(mutex) {
      --referenceNumber;
    }
    if (referenceNumber == 0) {
      delete this;
    }
  }
  size_t size() const {
    int val = 0;
    synchronized(mutex) {
      val = referenceNumber;
    }
  }
protected:
  virtual ~ReferenceObject() {}
};
template <typename Object> struct RefPointer :
public ReferenceObject { typedef AutoPtr<Object>
Ptr; };

class AutoLock {
public:
  explicit AutoLock(MutexLock& mutex) :
_mutex(mutex) { _mutex.lock(); }
  ~AutoLock() { try{ _mutex.unlock(); } catch
(...) {} }
private:
  MutexLock& _mutex;
};

class Signal {
public:
  Signal(bool manualReset = true) {
    START_EVENT_TYPE_FUNCTION(_mutex, _cond,
manualReset ? FALSE : TRUE);
  }

  ~Signal() {
    TERMINATE_EVENT_FUNCTION(_mutex, _cond);
  }

  void notify() {
    if (LOCK_MUTEX_EVENT_FUNCTION(&_mutex))
      throw ThreadException("cannot notify
lock");
    if (BROADCAST_FUNCTION(&_cond))
    {
      UNLOCK_MUTEX_EVENT_FUNCTION(&_mutex);
      throw ThreadException("cannot notify
lock");
    }
    if (!SET_EVENT_FUNCTION(_mutex))
    {
      throw ThreadException("cannot notify
lock");
    }
    UNLOCK_MUTEX_EVENT_FUNCTION(&_mutex);
  }

  void wait() {
    WAIT_EVENT_FUNCTION(_mutex, _cond, _state);
  }

  bool wait(long ms, bool timeout = false) {
    int status = false;
    WAIT_TIME_EVENT_FUNCTION(_mutex, _cond, ms,
_state, _auto, status)
      if (timeout && !status)
        throw ThreadException("Timeout
Exception");
    return status;
  }

  void reset()
  {
    if (LOCK_MUTEX_EVENT_FUNCTION(&_mutex)) {
      throw ThreadException("reset signal lock");
    }
    if (!RESET_EVENT_FUNCTION(_mutex)) {
      throw ThreadException("reset signal lock");
    }
    UNLOCK_MUTEX_EVENT_FUNCTION(&_mutex);
  }

private:
  EVENT_TYPE _mutex;
  CONDITION_TYPE  _cond;
};

class UID {
public:
  unsigned int _uid;
  UID() { _uid = ++uid; }
  UID(const UID& uid) { _uid = uid._uid; }
  UID& operator=(const UID& uid)  { _uid =
uid._uid; return(*this); }
  operator int() { return _uid; }
  bool operator == (const UID& uid) const {
return _uid == uid._uid; }
  bool operator != (const UID& uid) const {
return _uid != uid._uid; }
  bool operator == (const unsigned int& uid)
const { return _uid == uid; }
  bool operator != (const unsigned int& uid)
const { return _uid != uid; }
protected:
  static unsigned int uid;
};

template <class DataType>
class DataResolver {
public:
  DataResolver() : _data(0) { }
  DataType& data() {
```

``

```
    return *_data;
  }
  void resolveData(DataType* data) {
    delete _data;
    _data = data;
  }
private:
  DataType* _data;
};

class ErrorResolver {
public:
  ErrorResolver() :_error(0) {}
  std::string getErrorMessage() const {
    return (_error) ? _error->what() :
std::string();
  }
  std::exception* getError() const {
    return _error;
  }
  void resolveError(const std::string& msg) {
    delete _error;
    _error = new std::exception(msg.c_str());
  }
  bool hasError() const {
    return _error != 0;
  }
private:
  std::exception*  _error;
};

class SharedObject : public ReferenceObject {
public:
  SharedObject() : _signal(false) {}
  void wait() {
    _signal.wait();
  }
  bool wait(long ms, bool timeout = false) {
    return _signal.wait(ms, timeout);
  }
  void notify() {
    _signal.notify();
  }
private:
  Signal       _signal;
};

template <typename T>
struct SharedObjectProxy
{
  typedef T* DataTypePtr;
public:
  SharedObjectProxy(DataTypePtr data) :
_data(data) {}
  SharedObjectProxy(const SharedObjectProxy&
proxy) {
    _data = proxy._data;
    _data->reference();
  }
  ~SharedObjectProxy() {
    _data->release();
  }
  void wait() {
    _data->wait();
  }
  bool wait(long ms, bool timeout = false) {
    return _data->wait(ms, timeout);
  }
  bool ready() const {
    return _data->wait(0);
  }
  void notify() {
    _data->notify();
  }
protected:
  SharedObjectProxy();
  DataTypePtr  _data;
};

template <typename BASE, typename T = typename
BASE::DataTypePtr>
struct ErrorProxy : public BASE {
public:
  ErrorProxy(T data) : BASE(data) {}
  ErrorProxy(const ErrorProxy& proxy) :
BASE(proxy) {}
  std::string getErrorMessage() const {
    return _data->getErrorMessage();
  }
  std::exception* getError() const {
    return _data->getError();
  }
  void resolveError(const std::string& msg) {
    _data->resolveError(msg);
  }
  bool hasError() const {
    return _data->hasError();
  }
};

template <typename BASE, class Type, typename T =
typename BASE::DataTypePtr>
struct DataProxy : public BASE {
public:
  DataProxy(T data) : BASE(data) {}
  DataProxy(const DataProxy& proxy) : BASE(proxy)
{}
  Type& data() const {
    return _data->data();
  }
  void resolveData(Type* data) {
    _data->resolveData(data);
  }
};

template<typename Caller, typename ReturnType =
void, VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT,
void)> struct MethodCallbackSignature;
template<typename Caller, typename ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)> struct
MethodCallbackSignature { typedef
ReturnType(Caller::*Method)(VAR_ARGS(ARGUMENT_UPP
ER_LIMIT)); };
template<typename Caller, typename ReturnType>
struct MethodCallbackSignature<Caller,
ReturnType, VOID_ARGS(ARGUMENT_UPPER_LIMIT)> {
typedef ReturnType(Caller::*Method)(); };
```

``

```
template<typename Caller,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)> struct
MethodCallbackSignature<Caller, void,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)> { typedef void
(Caller::*Method)(VAR_ARGS(ARGUMENT_UPPER_LIMIT))
; };
template<typename Caller> struct
MethodCallbackSignature<Caller, void,
VOID_ARGS(ARGUMENT_UPPER_LIMIT)> { typedef void
(Caller::*Method)(); };

GENERATE_METHOD_CALLBACK_SIGNATURES(ARGUMENT_UPPE
R_LIMIT)

template <class BASE, class Caller, class
FutureType, class ReturnType>
class BaseMethodCallbackInvoke : public BASE {
public:
  BaseMethodCallbackInvoke(Caller* caller, const
FutureType& result) :
    _result(result), _context(caller) {}
  ~BaseMethodCallbackInvoke() { this->release();
}
  void run(){
    try {
      invokeMethod();
    }
    catch (std::exception& e) {
      _result.resolveError(e.what());
    }
    catch (...) {
      _result.resolveError("Invoke Error");
    }
    _result.notify();
  }
  void resolveError(const std::string& msg) {
    _result.resolveError(msg);
  }
protected:
  virtual void invokeMethod() = 0;
  Caller* _context;
  FutureType _result;
};

template<class BASE, class Caller, class
FutureType, class ReturnType,
VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT, void)>
class MethodCallbackInvoke;

template <class BASE, class Caller, class
FutureType, class ReturnType = void,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)>
class MethodCallbackInvoke : public
BaseMethodCallbackInvoke<BASE, Caller,
FutureType, ReturnType> {
public:
  typedef typename
MethodCallbackSignature<Caller, ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)>::Method Callback;
  MethodCallbackInvoke(Caller* caller, Callback
method, VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT,
arg, COMMA), const FutureType& result) :
    BaseMethodCallbackInvoke(caller, result),
_method(method),
INIT_VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT, arg)
{}
protected:
  VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT, _arg,
SEMICOLON);
  Callback    _method;
};

GENERATE_METHOD_CALLBACK_INVOKE(ARGUMENT_UPPER_LI
MIT)

template <class BASE, class Caller, class
FutureType, class ReturnType>
class MethodCallbackInvoke<BASE, Caller,
FutureType, ReturnType> : public
BaseMethodCallbackInvoke<BASE, Caller,
FutureType, ReturnType>{
public:
  typedef typename
MethodCallbackSignature<Caller,
ReturnType>::Method Callback;
  MethodCallbackInvoke(Caller* caller, Callback
method, const FutureType& result) :
    BaseMethodCallbackInvoke(caller, result),
_method(method) {}
protected:
  Callback    _method;
};

template<class BASE, class Caller, class
FutureType, class ReturnType = void,
VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT, void)>
class DelegateInvoke;

template <class BASE, class Caller, class
FutureType, class ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)>
class DelegateInvoke : public
MethodCallbackInvoke<BASE, Caller, FutureType,
ReturnType, VAR_ARGS(ARGUMENT_UPPER_LIMIT)> {
public:
  DelegateInvoke(Caller* caller, Callback method,
VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT, arg,
COMMA), const FutureType& result)
    : MethodCallbackInvoke(caller, method,
VAR_NAMED_ARGS(N, arg), result) {}
  void invokeMethod() {
    _result.resolveData(new ReturnType((_context-
>*_method)(VAR_ARGS(ARGUMENT_UPPER_LIMIT))));
  }
};

template <class BASE, class Caller, class
FutureType, VAR_TYPES(ARGUMENT_UPPER_LIMIT)>
class DelegateInvoke<BASE, Caller, FutureType,
void, VAR_ARGS(ARGUMENT_UPPER_LIMIT)> : public
MethodCallbackInvoke<BASE, Caller, FutureType,
void, VAR_ARGS(ARGUMENT_UPPER_LIMIT)>{
public:
  DelegateInvoke(Caller* caller, Callback method,
VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT, arg,
COMMA), const FutureType& result)
    : MethodCallbackInvoke(caller, method,
VAR_NAMED_ARGS(N, arg), result) {}
```

``

```
  void invokeMethod() {
    (_context-
>*_method)(VAR_ARGS(ARGUMENT_UPPER_LIMIT));
  }
};

GENERATE_DELEGATE_INVOKE(ARGUMENT_UPPER_LIMIT)

template <class BASE, class Caller, class
FutureType, class ReturnType>
class DelegateInvoke<BASE, Caller, FutureType,
ReturnType> : public MethodCallbackInvoke<BASE,
Caller, FutureType, ReturnType, void>{
public:
  DelegateInvoke(Caller* caller, Callback method,
const FutureType& result) :
MethodCallbackInvoke(caller, method, result) {}
  void invokeMethod() {
    _result.resolveData(new ReturnType((_context-
>*_method)()));
  }
};

template <class BASE, class Caller, class
FutureType>
class DelegateInvoke<BASE, Caller, FutureType> :
public MethodCallbackInvoke<BASE, Caller,
FutureType, void, void>{
public:
  DelegateInvoke(Caller* caller, Callback method,
const FutureType& result) :
MethodCallbackInvoke(caller, method, result) {}
  void invokeMethod() {
    (_context->*_method)();
  }
};

class DelegateBase : public Runnable, public
RefPointer<DelegateBase> {
public:
  virtual void resolveError(const std::string&
msg) = 0;
};

template <class FutureType> class FutureObject :
public SharedObject, public
DataResolver<FutureType>, public ErrorResolver
{};
template <> class FutureObject<void> : public
SharedObject, public ErrorResolver{};
template <class FutureType> class FutureResult :
public DataProxy< ErrorProxy<
SharedObjectProxy<FutureObject<FutureType>>>,
FutureType>{
public:
FutureResult(SharedObjectProxy::DataTypePtr ptr)
:DataProxy(ptr){}
};
template <> class FutureResult<void> : public
ErrorProxy<
SharedObjectProxy<FutureObject<void>>>{
public:
FutureResult(SharedObjectProxy::DataTypePtr ptr)
:ErrorProxy(ptr){}
};

template<class Caller, class ReturnType = void,
VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT, void)>
class Delegate;
template <class Caller, class ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)> class Delegate :
public DelegateInvoke < DelegateBase, Caller,
FutureResult<ReturnType>, ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT) > {
public:  Delegate(Caller* caller, Callback
method, VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT,
arg, COMMA), const FutureResult<ReturnType>&
result) : DelegateInvoke(caller, method,
VAR_NAMED_ARGS(N, arg), result) {}
};
template <class Caller,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)> class
Delegate<Caller, void,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)> : public
DelegateInvoke < DelegateBase, Caller,
FutureResult<void>, void,
VAR_ARGS(ARGUMENT_UPPER_LIMIT) >{
public:  Delegate(Caller* caller, Callback
method, VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT,
arg, COMMA), const FutureResult<void>& result) :
DelegateInvoke(caller, method, VAR_NAMED_ARGS(N,
arg), result) {}
};
template <class Caller, class ReturnType> class
Delegate<Caller, ReturnType, void> : public
DelegateInvoke< DelegateBase, Caller,
FutureResult<ReturnType>, ReturnType, void >{
public:  Delegate(Caller* caller, Callback
method, const FutureResult<ReturnType>& result) :
DelegateInvoke(caller, method, result) {}
};
template <class Caller> class Delegate<Caller,
void, void> : public DelegateInvoke<
DelegateBase, Caller, FutureResult<void>, void,
void >{
public:  Delegate(Caller* caller, Callback
method, const FutureResult<void>& result) :
DelegateInvoke(caller, method, result) {}
};

GENERATE_DELEGATE(ARGUMENT_UPPER_LIMIT)

template <class Caller>
struct Request : public
RefPointer<Request<Caller>> {
public:
  typedef std::deque<Ptr> RequestsQueue;
  typedef typename
MethodCallbackSignature<Caller, bool,
int>::Method RequestGuard;
  Request(int priority = 0, long delay = 0, long
timeout = 0, Caller* caller = 0, RequestGuard
guard = 0, int guardId = 0, int conditionId = 0)
: _priority(priority), _delay(delay),
_timeout(timeout), _context(caller),
_guard(guard), _guardId(guardId),
_conditionId(conditionId) {}
  int getPriority() const { return _priority; }
```

``

```
  int getDelay() const { return _delay; }
  int getTimeout() const { return _timeout; }
  bool filtered() {
    if (_guard){
      return !((_context->*_guard)(_guardId));
    }
    return false;
  }
  bool deferred() {
    if (_guard){
      return !((_context-
>*_guard)(_conditionId));
    }
    return false;
  }
private:
  Caller*    _context;
  RequestGuard _guard;
  int _guardId;
  int _conditionId;
  int _priority;
  long _delay;
  long _timeout;
};

template <class Caller>
struct MethodAccessRequest {
  typename Request<Caller>::Ptr request;
  Signal ready;
  typedef
std::deque<MethodAccessRequest<Caller>*>
MethodAccessRequestQueue;
};

template <class Caller> struct
MethodInvokeRequest : public Request<Caller> {
public:
  MethodInvokeRequest(DelegateBase::Ptr
requestRunnable, int priority = 0, long delay =
0, long timeout = 0, Caller* caller = 0,
RequestGuard guard = 0, int guardId = 0, int
conditionId = 0)
    : _requestRunnable(requestRunnable),
Request(priority, delay, timeout, caller, guard,
guardId, conditionId){}
  DelegateBase::Ptr getRequest() const { return
_requestRunnable; }
private:
  DelegateBase::Ptr _requestRunnable;
};

template <class Caller>
class ScheduleQueue : protected Runnable {
public:
  ScheduleQueue() : deferringRunning(false)  {}
  ~ScheduleQueue() { try { clear(); } catch (...)
{} }
  void request(typename Request<Caller>::Ptr
request, bool urgent = false) {
    synchronized(_lock) {
      deferringRunning = false;
      if (request->deferred()) {
        _deferred.push_back(request);
      }
      else {
        addRequest(request, urgent);
      }
    }
  }

  typename Request<Caller>*
processPendingRequests(){
    Request<Caller>::Ptr request = 0;
    MethodAccessRequest<Caller>* mq = 0;
    processDeferred();
    synchronized(_lock){
      request = getNextRequest();
    }
    if (request) {
      return request.reference();
    }
    else if (_pending.empty() &&
!_deferred.empty() && !deferringRunning) {
      _thread.start(this);
      deferringRunning = true;
    }
    mq = new MethodAccessRequest<Caller>();
    _pending.push_back(mq);
    mq->ready.wait();
    request = mq->request;
    delete mq;
    return request.reference();
  }
  void done() {
    synchronized(_lock) {
      for
(MethodAccessRequest<Caller>::MethodAccessRequest
Queue::iterator it = _pending.begin(); it !=
_pending.end(); ++it) {
        (*it)->ready.notify();
      }
      _pending.clear();
    }
  }
  bool empty() const {
    synchronized(_lock) {
      return _requests.empty() &&
_deferred.empty();
    }
  }
  int size() const {
    synchronized(_lock) {
      return static_cast<int>(_requests.size() +
_deferred.size());
    }
  }
  void clear() {
    synchronized(_lock) {
      _requests.clear();
      _deferred.clear();
    }
    _thread.join();
  }
protected:
  void run() {
    while (deferringRunning) { try {
processDeferred(); } catch (...) {} }
  }
```

``

```
private:
  void addRequest(typename Request<Caller>::Ptr
request, bool urgent = false) {
    if (_pending.empty()) {
      if (urgent) {
        _requests.push_front(request);
      }
      else if (request->getPriority() <= 0) {
        _requests.push_back(request);
      }
      else {
        typename
Request<Caller>::RequestsQueue::iterator it;
        for (it = _requests.begin(); it !=
_requests.end(); ++it) {
          if (request->getPriority() > (*it)-
>getPriority())
            break;
        }
        _requests.insert(it, request);
      }
    }
    else {
      MethodAccessRequest<Caller>* mq =
_pending.front();
      _pending.pop_front();
      mq->request = request;
      mq->ready.notify();
    }
  }
  void processDeferred() {
    Request<Caller>::Ptr deferred = 0;
    if (!_deferred.empty()) {
      deferred = _deferred.front();
      if (!deferred->deferred()) {
        _deferred.pop_front();
        addRequest(deferred);
      }
    }
    else {
      deferringRunning = false;
      return;
    }
  }
  typename Request<Caller>::Ptr getNextRequest()
{
    Request<Caller>::Ptr request;
    if (!_requests.empty()) {
      request = _requests.front();
      _requests.pop_front();
    }
    return request;
  }
private:
  mutable MutexLock  _lock;
  Thread        _thread;
  bool        deferringRunning;
  typename Request<Caller>::RequestsQueue
_requests;
  typename Request<Caller>::RequestsQueue
_deferred;
  typename
MethodAccessRequest<Caller>::MethodAccessRequestQ
ueue  _pending;
};

class TimedEvent : protected Runnable {
public:
  TimedEvent() { }
  virtual ~TimedEvent() { }
  void timeout(DelegateBase::Ptr  method, long
ms) {
    _method = method;
    _thread.start(this);
    try{
      _timeoutEvent.wait(ms, true);
    }
    catch (...) {
      _thread.stop();
      _method->resolveError("Timeout Exception");
      _method = 0;
      _timeoutEvent.notify();
    }
  }
protected:
  void run() {
    _method->reference();
    _method->run();
    _timeoutEvent.notify();
  }
private:
  Thread        _thread;
  DelegateBase::Ptr  _method;
  Signal        _timeoutEvent;
};

template <class Caller>
class Scheduler : protected Runnable {
public:
  typedef typename Request<Caller>::RequestGuard
GuardList;
  Scheduler(Caller* caller = 0, GuardList
guardList = 0) : _context(caller),
_guardList(guardList) {
    _run = true;
    _thread.start(this);
  }
  virtual ~Scheduler() { try { stop(); } catch
(...) {} }
  void schedule(DelegateBase::Ptr pDelegate, int
priority, long delay, long timeout, UID guard,
UID condition) {
    _queue.request(new
MethodInvokeRequest<Caller>(pDelegate, priority,
delay, timeout, _context, _guardList, guard,
condition));
  }
  void cancel() {
    _queue.clear();
  }
protected:
  void run() {
    AutoPtr<Request<Caller>> pendingRequest =
_queue.processPendingRequests();
    while (pendingRequest) {
```

``

```
      MethodInvokeRequest<Caller>* mth =
static_cast<MethodInvokeRequest<Caller>*>(pending
Request.value());
      if (!mth->filtered()) {
        long delay = mth->getDelay();
        if (delay > 0) {
          Thread::sleep(delay);
        }
        if (mth->getTimeout() > 0) {
          _timeEvent.timeout(mth->getRequest(),
mth->getTimeout());
        }
        else {
          DelegateBase::Ptr pDelegate = mth-
>getRequest();
          pDelegate->reference();
          pDelegate->run();
          pDelegate = 0;
        }
      }
      pendingRequest = 0;
      if (_run)
        pendingRequest =
_queue.processPendingRequests();
    }
  }
  void stop() {
    _queue.clear();
    _queue.done();
    _run = false;
    _thread.join();
  }
private:
  Caller*          _context;
  Thread           _thread;
  ScheduleQueue<Caller>  _queue;
  GuardList        _guardList;
  bool          _run;
  TimedEvent        _timeEvent;
};

struct ActiveConstraintUID {
public:
  UID getGuardUID() { return _guardId; }
  UID getConditionUID() { return _conditionId; }
protected:
  UID _guardId;
  UID _conditionId;
};

template<class ReturnType,
VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT, void)>
class IDelegatePublisher;
template<class ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)>
class IDelegatePublisher {
public:
  virtual FutureResult<ReturnType>
publish(VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT,
arg, COMMA), int priority = 0, long delay = 0,
long timeout = 0) = 0;
};
template<class ReturnType>
class IDelegatePublisher<ReturnType> {
public:
  virtual FutureResult<ReturnType> publish(int
priority = 0, long delay = 0, long timeout = 0) =
0;
};

template<class ReturnType,
VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT, void)>
class MulticastDelegate;
template <class ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)>
class MulticastDelegate : public
IDelegatePublisher<ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)> {
private:
  typedef std::vector<
IDelegatePublisher<ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)> * >
SubscribersList;
  SubscribersList subscribers;
public:
  MulticastDelegate() {}
  MulticastDelegate& operator += (const
IDelegatePublisher<ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)>* method) {
    subscribers.push_back(method);
    return *this;
  }
  FutureResult<ReturnType> operator ()
(VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT, arg,
COMMA), int priority = 0, long delay = 0, long
timeout = 0) {
    FutureResult<ReturnType> result =
publish(VAR_NAMED_ARGS(ARGUMENT_UPPER_LIMIT,
arg), priority, delay, timeout);
    typename SubscribersList::iterator it =
subscribers.begin();
    for (; it != subscribers.end(); it++) { it-
>publish(VAR_NAMED_ARGS(ARGUMENT_UPPER_LIMIT,
arg), priority, delay, timeout);}
    return result;
  }
};
template <class ReturnType>
class MulticastDelegate<ReturnType> : public
IDelegatePublisher<ReturnType>{
private:
  typedef std::vector<
IDelegatePublisher<ReturnType> * >
SubscribersList;
  SubscribersList subscribers;
public:
  MulticastDelegate() {}
  MulticastDelegate& operator += (const
IDelegatePublisher<ReturnType>* method) {
    subscribers.push_back(method);
    return *this;
  }
  FutureResult<ReturnType> operator () (int
priority = 0, long delay = 0, long timeout = 0) {
    FutureResult<ReturnType> result =
publish(priority, delay, timeout);
    typename SubscribersList::iterator it =
subscribers.begin();
```

``

```
    for (; it != subscribers.end(); it++) {
(*it)->publish(priority, delay, timeout); }
    return result;
  }
};

GENERATE_MULTICAST_METHOD(ARGUMENT_UPPER_LIMIT)

template<class Caller, class ReturnType,
VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT, void)>
class Active;
template <class Caller, class ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)>
class Active : public ActiveConstraintUID, public
MulticastDelegate<ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)>{
public:
  typedef Delegate<Caller, ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)> DelegateType;
  typedef typename
MethodCallbackSignature<Caller, ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)>::Method Callback;
  Active(Caller* caller, Scheduler<Caller>* sch,
Callback method) :_context(caller), _sch(sch),
_method(method) {}
  FutureResult<ReturnType>
publish(VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT,
arg, COMMA), int priority = 0, long delay = 0,
long timeout = 0) {
    FutureResult<ReturnType> result(new
FutureObject<ReturnType>());
    DelegateBase::Ptr pDelegate(new
DelegateType(_context, _method,
VAR_NAMED_ARGS(ARGUMENT_UPPER_LIMIT, arg),
result));
    _sch->schedule(pDelegate, priority, delay,
timeout, _guardId, _conditionId);
    return result;
  }
private:
  Caller* _context;
  Scheduler<Caller>* _sch;
  Callback   _method;
};

template <class Caller, class ReturnType>
class Active <Caller, ReturnType, void> : public
ActiveConstraintUID, public
MulticastDelegate<ReturnType> {
public:
  typedef Delegate<Caller, ReturnType, void>
DelegateType;
  typedef typename
MethodCallbackSignature<Caller,
ReturnType>::Method Callback;
  Active(Caller* caller, Scheduler<Caller>* sch,
Callback method) :_context(caller), _sch(sch),
_method(method) {}
  FutureResult<ReturnType> publish(int priority =
0, long delay = 0, long timeout = 0) {
    FutureResult<ReturnType>  result(new
FutureObject<ReturnType>());
    DelegateBase::Ptr pDelegate(new
DelegateType(_context, _method, result));
    _sch->schedule(pDelegate, priority, delay,
timeout, _guardId, _conditionId);
    return result;
  }
private:
  Caller* _context;
  Scheduler<Caller>* _sch;
  Callback   _method;
};
GENERATE_ACTIVE_METHOD(ARGUMENT_UPPER_LIMIT)

template <class Caller>
class AsyncMethod : protected Runnable, public
ActiveConstraintUID {
public:
  typedef Delegate<Caller, void, void>
DelegateType;
  typedef Active<Caller, void> ActiveType;
  typedef typename
MethodCallbackSignature<Caller>::Method Callback;
  AsyncMethod(Caller* caller, Callback method,
long interval = 0, long delay = 0, long timeout =
0) :_context(caller), _delegate(new
DelegateType(_context, method, new
FutureObject<void>())), _delay(delay),
_timeout(timeout), _interval(interval),
_active(false){}
  AsyncMethod(Caller* caller, Scheduler<Caller>*
sch, Callback method, long interval = 0, long
delay = 0, long timeout = 0) :_context(caller),
_delegate(new DelegateType(_context, method, new
FutureObject<void>())), _delay(delay),
_timeout(timeout), _interval(interval),
_sch(sch), _active(true) {}
  virtual ~AsyncMethod() { try { stop();
_thread.join(); } catch (...) {} }
  void start() {
    _run = true;
    _thread.start(this);
  }
  void stop() {
    _run = false;
  }
protected:
  void run() {
    while (_run) {
      if (!_active && _delay > 0) {
        Thread::sleep(_delay);
      }
      if (_interval > 0) {
        if (_active) {
          _sch->schedule(_delegate, -1, _delay,
_timeout, _guardId, _conditionId);
        }
        else {
          _delegate->run();
          if (_timeout > 0)
            _timeoutEvent.wait(_timeout, true);
        }
        Thread::sleep(_interval);
      }
    }
  }
private:
```

``

```
  bool        _run;
  long        _interval;
  long        _delay;
  long        _timeout;
  bool        _active;
  Thread        _thread;
  Caller*        _context;
  DelegateBase::Ptr  _delegate;
  Scheduler<Caller>*  _sch;
  Signal        _timeoutEvent;
};
#ifdef WINDOWS_OS
typedef int socklen_t;
#pragma comment(lib, "ws2_32.lib")
struct WindowsSocket {
  WindowsSocket() {
    WSADATA init;
    if (WSAStartup(MAKEWORD(2, 2), &init) != 0) {
      throw Exception("WSAStartup Init Error");
    }
  }
  ~WindowsSocket() {
    if (WSACleanup() != 0) {
      std::cerr << "WSACleanup Error" <<
std::endl;
    }
  }
};
extern struct WindowsSocket windowsSocketInit;
#endif

class Endpoint {
public:
  Endpoint() {}
  Endpoint(string host, string port) :
host(host), port(atoi(port.c_str())) {}
  Endpoint(string host, int port) : host(host),
port(port) {}
  ~Endpoint() {}
  std::string getHost(){ return host; }
  int getPort() { return port; }
private:
  std::string host;
  int port;
};

class PingPongPort;
class Pinger;
class Ponger;
class PingPong;
class JSON;
struct JSONParser {
 virtual JSON Parse(const string &json) = 0;
};
enum JSONType { JSON_Undefined, JSON_String,
JSON_Bool, JSON_Number, JSON_Array, JSON_Object
};
class JSON {
private:
  static JSONParser *_parser;
public:
  JSON(JSONType type = JSON_Undefined) {
    setType(type);
  }
  JSON(PingPongPort& aPingPongPort);
  operator PingPongPort();
  JSON(PingPongPort* aPingPongPort);
  operator PingPongPort*();
  JSON(Pinger& aPinger);
  operator Pinger();
  JSON(Pinger* aPinger);
  operator Pinger*();
  JSON(Ponger& aPonger);
  operator Ponger();
  JSON(Ponger* aPonger);
  operator Ponger*();
  JSON(PingPong& aPingPong);
  operator PingPong();
  JSON(PingPong* aPingPong);
  operator PingPong*();
  JSON(const char *m_char_value) {
    _type = JSON_String;
    _string = new string(m_char_value);
  }
  JSON(const std::string &m_string_value) {
    _type = JSON_String;
    _string = new string(m_string_value.c_str());
  }
  JSON(bool m_bool_value) {
    _type = JSON_Bool;
    _boolean = m_bool_value;
  }
  JSON(float m_number_value) {
    _type = JSON_Number;
    _number = m_number_value;
  }
  JSON(double m_number_value) {
    _type = JSON_Number;
    _number = m_number_value;
  }
  JSON(long m_number_value) {
    _type = JSON_Number;
    _number = m_number_value;
  }
  JSON(int m_integer_value) {
    _type = JSON_Number;
    _number = (double)m_integer_value;
  }
  JSON(unsigned int m_integer_value) {
    _type = JSON_Number;
    _number = (unsigned int)m_integer_value;
  }
  JSON(Endpoint m_ep) {
    (*this)["host"] = m_ep.getHost();
    (*this)["port"] = m_ep.getPort();
  }
  JSON(const JSON &src) {
    swap(src);
  }
  ~JSON() {
    if (_type == JSON_Array) {
      vector<JSON*>::iterator iter;
      for (iter = _array->begin(); iter !=
_array->end(); iter++)
        delete *iter;
      delete _array;
    } else if (_type == JSON_Object) {
```

``

```
      map<string, JSON*>::iterator iter;
      for (iter = _object->begin(); iter !=
_object->end(); iter++)
        delete (*iter).second;
      delete _object;
    }
    else if (_type == JSON_String) { delete
_string; }
  }

  static JSON Parse(const string &json) {
    return _parser->Parse(json);
  }
  bool IsUndefined() const { return _type ==
JSON_Undefined; }
  bool IsString() const { return _type ==
JSON_String; }
  bool IsBoolean() const { return _type ==
JSON_Bool; }
  bool IsNumber() const { return _type ==
JSON_Number; }
  bool IsArray() const { return _type ==
JSON_Array; }
  bool IsObject() const { return _type ==
JSON_Object; }
  operator string() const { return (*_string); }
  operator bool() const { return _boolean; }
  operator double() const { return _number; }
  operator float() const { return _number; }
  operator int() const { return _number; }
  operator long() const { return _number; }
  operator unsigned() const { return _number; }

  #ifdef _WIN64
    JSON(size_t m_size_t) {
      _type = JSONType::JSON_Number;
      _number = m_size_t;
    }
    operator size_t() const { return _number; }
  #endif

  operator Endpoint() {
    string host = (*this)["host"];int port =
(*this)["port"];
    Endpoint p(host, port);
    return p;
  }
  operator const JSON&() const { return *this; }

  void push(JSON* item) {
    setType(JSON_Array);
    _array->push_back(item);
  }
  size_t length() const {
    switch (_type) {
      case JSON_Array:
        return _array->size();
      case JSON_Object:
        return _object->size();
      default:
        return 0;
    }
  }
  bool hasIndex(std::size_t index) const {
    if (_type == JSON_Array) {
      return index < _array->size();
    }
    else {
      return false;
    }
  }
  JSON *at(std::size_t index) {
    if (index < _array->size()) {
      return (*_array)[index];
    } else {
      return NULL;
    }
  }
  bool contains(const char* name) const {
    if (_type == JSON_Object) { return _object-
>find(name) != _object->end();
    } else { return false;}
  }
  JSON *at(const char* name) {
    map<string, JSON*>::const_iterator it =
_object->find(name);
    if (it != _object->end()) { return it-
>second;
    } else { return NULL;}
  }
  std::vector<string> Keys() const {
    std::vector<string> keys;
    if (_type == JSON_Object) {
      map<string, JSON*>::const_iterator iter =
_object->begin();
      while (iter != _object->end()) {
        keys.push_back(iter->first);
        iter++;
      }
    }
    return keys;
  }

  string toString() {
    return jsonize();
  }

  string toString(const JSON *value) {
    string json = "";
    if (value != NULL)
      json = value->jsonize();
    return json;
  }

  JSON& operator=(const JSON& val) {
    swap(val);
    return *this;
  }
  JSON& operator=(JSON *src) {
    swap(*src);
    return *this;
  }

  JSON& operator[](const char *key) {
      return this->operator[](key);
  }
```

``

```
  JSON& operator[](const string &key) {
    setType(JSON_Object);
    JSON* ret = _object->operator[](key);
    if (ret == NULL) {
      ret = new JSON();
      _object->operator[](key) = ret;
    }
    return *ret;
  }

  JSON& operator[](unsigned index) {
    setType(JSON_Array);
    if (index >= _array->size())
      _array->resize(index + 1);
    JSON* ret = _array->operator[](index);
    if (ret == NULL) {
      ret = new JSON();
      _array->operator[](index) = ret;
    }
    return *ret;
  }

protected:
  void swap(const JSON &src) {
    _type = src._type;
    switch (_type) {
      case JSON_String:
        _string = new string(*src._string);
        break;
      case JSON_Bool:
        _boolean = src._boolean;
        break;
      case JSON_Number:
        _number = src._number;
        break;
      case JSON_Array: {
        vector<JSON*> source_array = *src._array;
        vector<JSON*>::iterator iter;
        _array = new vector<JSON*>();
        for (iter = source_array.begin(); iter !=
source_array.end(); iter++)
          _array->push_back(new JSON(**iter));
        break; }
      case JSON_Object: {
        map<string, JSON*> source_object =
*src._object;
        _object = new map<string, JSON*>();
        map<string, JSON*>::iterator iter;
        for (iter = source_object.begin(); iter
!= source_object.end(); iter++)
        {
          string name = (*iter).first;
          (*_object)[name] = new
JSON(*((*iter).second));
        }
        break; }
      case JSON_Undefined:
        break;
    }
  }
  void setType(JSONType type) {
    if (this->_type == type)
      return;
    switch (type) {
      case JSON_Undefined:
        break;
      case JSON_Object:
        _object = new map<string, JSON*>();
        break;
      case JSON_Array:
        _array = new vector<JSON*>();
        break;
      case JSON_String:
        _string = new string();
        break;
      case JSON_Number:
        _number = 0;
        break;
      case JSON_Bool:
        _boolean = false;
        break;
      }
      this->_type = type;
    }

    string escapeString(const string &json) const {
      string escaped = "\"";
      for (unsigned i = 0; i < json.length(); ++i)
{
        if (json[i] == '\"')    escaped += "\\\"";
        else if (json[i] == '\\')  escaped +=
"\\\\";
        else if (json[i] == '\b')  escaped +=
"\\b";
        else if (json[i] == '\f')  escaped +=
"\\f";
        else if (json[i] == '\n')  escaped +=
"\\n";
        else if (json[i] == '\r')  escaped +=
"\\r";
        else if (json[i] == '\t')  escaped +=
"\\t";
        else            escaped += json[i];
      }
      escaped += "\"";
      return escaped;
    }

    string jsonize() const {
      string out;
      switch (_type)
      {
      case JSON_Undefined:
        out = "null";
        break;
      case JSON_String:
        out = escapeString(*_string);
        break;
      case JSON_Bool:
        out = _boolean ? "true" : "false";
        break;
      case JSON_Number:
      {
        if (isinf(_number) || isnan(_number))
          out = "null";
        else {
          stringstream ss;
          ss.precision(15);
```

``

```
        ss << _number;
        out = ss.str();
      }
      break;
    }

    case JSON_Array:
    {
      out = "[";
      vector<JSON*>::const_iterator iter =
_array->begin();
      while (iter != _array->end()) {
        out += (*iter)->jsonize();
        if (++iter != _array->end())
          out += ",";
      }
      out += "]";
      break;
    }

    case JSON_Object: {
      out = "{";
      map<string, JSON*>::const_iterator iter =
_object->begin();
      while (iter != _object->end()) {
        out += escapeString((*iter).first);
        out += ":";
        out += (*iter).second->jsonize();
        if (++iter != _object->end())
          out += ",";
      }
      out += "}";
      break;
    }
    }
    return out;
  }

private:
  JSONType _type;
  union{
    bool _boolean;
    double _number;
    string *_string;
    vector<JSON*> *_array;
    map<string, JSON*> *_object;
  };

};

class JSONInternalParser : public JSONParser {
public:
  JSONInternalParser() {}
  JSON Parse(const string &json) {
    size_t offset = 0;
    JSON value = internal_parse(json, offset);
    return value;
  }
protected:
  void skipWhitespace(const string &json, size_t
&index) { while (isspace(json[index])) ++index; }
  string getString(const string &json, size_t
&index) {
    string str;
    for (char c = json[++index]; c != '\"' &&
index < json.size(); c = json[++index]) {
      if (c == '\\') {
        c = json[++index];
        if (c == '\"')    str += '\"';
        else if (c == '\\') str += '\\';
        else if (c == '/')  str += '/';
        else if (c == 'b')  str += '\b';
        else if (c == 'f')  str += '\f';
        else if (c == 'n')  str += '\n';
        else if (c == 'r')  str += '\r';
        else if (c == 't')  str += '\t';
        else if (c == 'u') {
          str += "\\u";
          for (unsigned i = 1; i <= 4; ++i) {
            c = json[index + i];
            if ((c >= '0' && c <= '9') || (c >=
'a' && c <= 'f') || (c >= 'A' && c <= 'F')) str
+= c;
            else return NULL;
          }
          index += 4;
        }
        else        str += '\\';
      }
      else str += c;
    }
    ++index;
    return str;
  }

  JSON internal_parse(const string &json, size_t
&index) {
    skipWhitespace(json, index);
    char c = json[index];
    if (c == '[' ) {
      JSON Array(JSON_Array);
      skipWhitespace(json, index);
      if (json[++index] == ']') { ++index; return
Array; }
      for (unsigned array_index = 0;; ++index,
array_index++) {
        Array[array_index] = new
JSON(internal_parse(json, index));
        skipWhitespace(json, index);
        if (json[index] == ']') { ++index; break;
}
        else if (json[index] != ',') { return
JSON(); }
      }
      return Array;
    }
    else if (c == '{') {
      JSON jObj(JSON_Object);
      skipWhitespace(json, index);
      if (json[++index] == '}') { ++index; return
JSON_Object; }
      for (string objectKey;; ++index) {
        objectKey = getString(json, index);
        skipWhitespace(json, index);
        if (json[index] != ':') { std::cerr <<
"Missing colon, Error char is'" << json[index] <<
"'\n";break;}
        skipWhitespace(json, ++index);
```

``

```
        JSON *parsedObject = new
JSON(internal_parse(json, index));
        jObj[objectKey] = parsedObject;
        skipWhitespace(json, index);
        if (json[index] == '}') {++index; break;}
        else if (json[index] != ',') {std::cerr
<< "Missing comma, Error char is '" <<
json[index] << "'\n";break;}
      }
      return jObj;
    }
    else if (c == '\"') {
      string val = getString(json, index);
      return JSON(val);
    }
    else if (c == 't' || c == 'f') {
      bool value;
      if (json.substr(index, 4) == "true") {value
= true; index += 4;}
      else if (json.substr(index, 5) == "false")
{value = false; index += 5;}
      else { return JSON();}
      return JSON(value);
    }
    else if (c == 'n') {
      if (json.substr(index, 4) != "null")
{return JSON(); index += 4; }
      else { std::cerr << "Missing Null\n";}
      return JSON();
    }
    else if ((c <= '9' && c >= '0') || c == '-')
{
      double num;
      string parsedNumber;
      char n = json[index++];
      bool isDbl = false;
      long exponential = 0;
      for (; n == '-' || (n >= '0' && n <= '9')
|| n == '.'; parsedNumber += n, n =
json[index++], isDbl = isDbl ? true : n == '.');
      if (n == 'E' || n == 'e') {
        string e;
        n = json[index++];
        if (n == '-'){ ++index; e += '-'; }
        for (; n >= '0' && n <= '9'; n =
json[index++], e += n);
        if (!isspace(n) && n != ',' && n != ']'
&& n != '}') return JSON();
        exponential = std::stol(e);
      }
      --index;
      if (isDbl) num = std::stod(parsedNumber) *
std::pow(10.0, exponential);
      else {
        if (exponential > 0) num =
std::stol(parsedNumber) * std::pow(10.0,
exponential);
        else num = std::stol(parsedNumber);
      }
      return JSON(num);
    }
  }
};
```

```
class ConnectionException : public Exception {
public:
  ConnectionException() : Exception() {}
  ConnectionException(std::string exMessage) :
Exception(exMessage) {}
};
class SocketException : public  Exception {
public:
  SocketException() : Exception() {}
  SocketException(std::string exMessage) :
Exception(exMessage) {}
};
class IConnector {
public:
  virtual ~IConnector(){}
  virtual string Send(const std::string& message)
throw(ConnectionException) = 0;
};
class IReceiver {
public:
  virtual ~IReceiver(){}
  virtual unsigned int Listen()
throw(ConnectionException) = 0;
};

class TCPConnector : public IConnector {
public:
  TCPConnector() {}
  TCPConnector(const string& ip, const unsigned
int &port) :endpoint(ip,port) {}
  TCPConnector(const Endpoint& endpoint)
:endpoint(endpoint) {}
  virtual ~TCPConnector() {}
  virtual string Send(const string& message)
throw (ConnectionException) {
    SOCKET_TYPE socket_fd = openSocketPort();
    string result = transport(socket_fd,
message);
    CLOSE_SOCKET(socket_fd);
    return result;
  }

private:
  string transport(SOCKET_TYPE socket_fd, string
message) throw (ConnectionException) {
    char buffer[NETWORK_BUFFER_SIZE];
    string msg = message;
    unsigned int bytes;
    bool receive = false;
    do {
      bytes = !receive ? send(socket_fd,
msg.c_str(), msg.size(), 0) :
            recv(socket_fd, buffer,
NETWORK_BUFFER_SIZE, 0);
      if (bytes == -1) {
        CLOSE_SOCKET(socket_fd);
        string error = recv ? "Can not receive
data" : "Can not send data";
        throw SocketException(error);
      }
      if (!receive) {
        if (bytes =  bytes - msg.size()) { msg =
msg.substr(bytes + sizeof(char), msg.size() -
bytes); }
```

``

```
      } else {
        msg.append(buffer, bytes);
      }
      if (bytes == 0) {
        if (!receive) {
          msg = ""; receive = true;
shutdown(socket_fd, 1);
        }
        else {
          break;
        }
      }
    } while (receive && msg.find(DELIMITER) ==
string::npos);

    return msg;
  }

  SOCKET_TYPE openSocketPort() throw
(ConnectionException) {
    SOCKET_TYPE sd, rc;
    struct sockaddr_in localAddr, servAddr;
    struct hostent *host;
    string host_str = endpoint.getHost();
    const char *hostStr = host_str.c_str();
    host = gethostbyname(hostStr);
    if (host == NULL) {
      throw SocketException("Unknown Host");
    }
    servAddr.sin_family = host->h_addrtype;
    memcpy((char *)&servAddr.sin_addr.s_addr,
host->h_addr_list[0], host->h_length);
    servAddr.sin_port =
htons(endpoint.getPort());
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd<0) {
      throw SocketException("Cannot open socket
for communication");
    }
    localAddr.sin_family = AF_INET;
    localAddr.sin_addr.s_addr =
htonl(INADDR_ANY);
    localAddr.sin_port = htons(0);
    rc = bind(sd, (struct sockaddr *) &localAddr,
sizeof(localAddr));
    if (rc<0) {
      throw SocketException("Cannot bind to the
port");
    }
    rc = connect(sd, (struct sockaddr *)
&servAddr, sizeof(servAddr));
    if (rc<0) {
      throw SocketException("Cannot connect to
the server");
    }
    return sd;
  }
  Endpoint endpoint;
};

template<class Caller>
class TCPReceiver : public IReceiver, protected
Runnable{
public:
  typedef typename
MethodCallbackSignature<Caller, string,
string>::Method onReceive;
  TCPReceiver(Caller* caller, onReceive method,
unsigned int port = 0) :_port(port),
_context(caller), _onReceive(method),
_running(false) {}
  virtual ~TCPReceiver() { Stop(); }
  virtual unsigned int Listen()
throw(ConnectionException) {
    Stop();
    unsigned int bindPort = _port;
    struct sockaddr_in servAddr;
    socketPort = socket(AF_INET, SOCK_STREAM, 0);
    if (socketPort<0) {
      cerr << "cannot open socket " << endl;
    }
    servAddr.sin_family = AF_INET;
    servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servAddr.sin_port = htons(_port);
    if (bind(socketPort, (struct sockaddr *)
&servAddr, sizeof(servAddr))<0) {
      cerr << "cannot bind port " << endl;
    }
    if (_port == 0) {
      struct sockaddr_in bindAddr;
      socklen_t bindSize = sizeof(bindAddr);
      if (getsockname(socketPort, (struct
sockaddr *)&bindAddr, &bindSize) == -1)
      {
        cerr << "getsockname error" << endl;
      }
      cerr << "Server instance running on port :
" << bindAddr.sin_port << endl;
      bindPort = bindAddr.sin_port;
    }
    else {
      cerr << "Server instance running on port :
" << _port << endl;
    }
    listen(socketPort, 5);
    _thread.start(this);
    _running = true;
    return bindPort;
  }
  void Stop(){
    if (_running) {
      _running = false;
      Thread::sleep(1);
    }
  }

protected:
  void run(){
    SOCKET_TYPE newSd;
    socklen_t cliLen;
    struct sockaddr cliAddr;
    while (1) {
      cliLen = sizeof(cliAddr);
      newSd = accept(socketPort, &cliAddr,
&cliLen);
      if (newSd<0) {
        fprintf(stderr, "cannot accept
connection\n");
```

``

```
        break;
      }
      processRequest(newSd);
    }
    CLOSE_SOCKET(socketPort);
  }

  void processRequest(SOCKET_TYPE sockfd) {
    int n = 0;
    char *line = new char[80];
    string recvMsg;
    do {
      memset(line, 0x00, sizeof(line));
      n = recv(sockfd, line, sizeof(line), 0); /*
wait for data */
      if (n<0) {
        cerr << "cannot receive data" << endl;
        break;
      }
      else if (n == 0) {
        break;
      }
      recvMsg.append(line, n);
    } while (1);
    string sendMsg = (_context-
>*_onReceive)(recvMsg);
    n = send(sockfd, sendMsg.c_str(),
sendMsg.size(), 0);
    if (n<0) {
      cerr << "Cannot send data" << endl;
    }
    CLOSE_SOCKET(sockfd);
  }
private:
  Thread    _thread;
  Caller*    _context;
  onReceive _onReceive;
  unsigned int _port;
  SOCKET_TYPE socketPort;
  bool _running;
};

class Buffer {
public:
  std::string buffer;
};

class RemoteException : public Exception {
public:
  RemoteException() : Exception() {}
  RemoteException(std::string exMessage) :
Exception(exMessage) {}
};

enum RemoteMethodType { Local, Client, Server};

class IResponseHandler {
public:
  virtual ~IResponseHandler(){}
  virtual string handleResponse(string response)
const throw (RemoteException) = 0;
};

class IRequestHandler {
public:
  virtual ~IRequestHandler(){}
  virtual string handleRequest(string request)
const throw (RemoteException) = 0;
};

class IMessageRouter {
public:
  virtual ~IMessageRouter(){}
  virtual void registerMethod(string methodName,
const IResponseHandler* entry) = 0;
  virtual RemoteMethodType getType() = 0;
  virtual IConnector* getConnecter() = 0;
  virtual const IResponseHandler*
getResponseHandler(string methodName) const = 0;
};

class RemoteMethodRequestHandler : public
IRequestHandler {
public:
  RemoteMethodRequestHandler(IMessageRouter*
messageRouter) : _router(messageRouter){}
  virtual ~RemoteMethodRequestHandler(){}
protected:
  IMessageRouter* _router;
};

template<class Caller, class ReturnType>
class Transport : public IResponseHandler {
public:
  typedef typename
MethodCallbackSignature<Caller, bool>::Method
onReceive;
  typedef typename
MethodCallbackSignature<Caller,
ReturnType>::Method Invoke;

  Transport(Caller* caller, onReceive method,
Invoke invoke, string methodName, IMessageRouter*
messageRouter) : _methodName(methodName),
_router(messageRouter), _context(caller),
_onReceive(method), _invoke(invoke){
    _router->registerMethod(methodName, this);
  }
  virtual ~Transport(){}
  virtual size_t length() const = 0;

  virtual ReturnType remote() throw
(RemoteException) = 0;
  IMessageRouter* getRouter() {
    return _router;
  }
protected:
  IMessageRouter* _router;
  string _methodName;
  Caller*    _context;
  onReceive _onReceive;
  Invoke _invoke;
};

class JSONMethodRequestHandler : public
RemoteMethodRequestHandler {
public:
```

``

```
  JSONMethodRequestHandler(IMessageRouter*
messageRouter) :
RemoteMethodRequestHandler(messageRouter){}
  virtual ~JSONMethodRequestHandler(){}
  string handleRequest(string request) const
throw (RemoteException) {
    string result;
    const IResponseHandler* handler = NULL;
    JSON delegateMethod = JSON::Parse(request);
    if (delegateMethod.IsObject() == true) {
      string methodName = "";
      if (delegateMethod.contains("method") &&
delegateMethod["method"].IsString()) {
        methodName = delegateMethod["method"];
      }
      handler = _router-
>getResponseHandler(methodName);
      if (handler == NULL) {
        JSON ex; ex["exception"] = "Unknown
method";
        return ex.toString();
      }
      else {
        if (delegateMethod.contains("values") &&
delegateMethod["values"].IsArray()) {
          JSON values = delegateMethod["values"];
          result = handler-
>handleResponse(values.toString());
        }
      }
    }
    return result;
  }
};

class TCPMessageRouter : public IMessageRouter {
public:
  typedef map<string, const IResponseHandler* >
HandlerMap;
  TCPMessageRouter() : _port(0), _receiver(this,
&TCPMessageRouter::processResponseHandler),
_type(RemoteMethodType::Local), _handler(new
JSONMethodRequestHandler(this)) {}
  TCPMessageRouter(Endpoint endpoint) :
_ep(endpoint), _tcpConnector(_ep), _port(0),
_receiver(this,
&TCPMessageRouter::processResponseHandler),
_type(RemoteMethodType::Client), _handler(new
JSONMethodRequestHandler(this)) {}
  TCPMessageRouter(unsigned int port) :
_port(port), _receiver(this,
&TCPMessageRouter::processResponseHandler, port),
_type(RemoteMethodType::Server), _handler(new
JSONMethodRequestHandler(this)) {
    _port = _receiver.Listen();
  }
  virtual ~TCPMessageRouter() { _entries.clear();
}
  void registerMethod(string methodName, const
IResponseHandler* entry) {

_entries.insert(HandlerMap::value_type(methodName
, entry));
  }
  unsigned int getPort() { return _port; }
  Endpoint getEndPoint() { return _ep; }
  RemoteMethodType getType() { return _type; }
protected:
  virtual IConnector* getConnecter() {
    return &_tcpConnector;
  }
  string processResponseHandler(string response)
throw (RemoteException) {
    return _handler->handleRequest(response);
  }
  const IResponseHandler*
getResponseHandler(string methodName) const {
    HandlerMap::const_iterator it =
_entries.begin();
    it = _entries.find(methodName);
    if (it != _entries.end())
      return it->second;
    return NULL;
  }
private:
  unsigned int _port;
  Endpoint _ep;
  TCPConnector _tcpConnector;
  HandlerMap _entries;
  TCPReceiver<TCPMessageRouter> _receiver;
  RemoteMethodType _type;
  RemoteMethodRequestHandler* _handler;
};

template<class ReturnType>
class IPort {
public:
  virtual ~IPort(){}
  virtual ReturnType send() = 0;
  virtual bool receive() = 0;
};

template<class Caller, class ReturnType>
class JSONTransport : public Transport<Caller,
ReturnType> {
public:
  JSONTransport(Caller* caller, onReceive method,
Invoke invoke, string methodName, IMessageRouter*
messageRouter) : Transport(caller, method,
invoke, methodName, messageRouter),
_arguments(JSON_Array) {
  }
  JSON& operator[](unsigned index) {
    return _arguments[index];
  }
  size_t length() const {
    return _arguments.length();
  }

  template<class T> T& at(unsigned index) {
    return _arguments[index];
  }

  ReturnType remote() throw (RemoteException) {
    string args_json = _arguments.toString();
    RemoteMethodRequestHandler* requester =
dynamic_cast<RemoteMethodRequestHandler
*>(getRouter());
```

``

```
    string returnJson =
invokeRemoteMethod(_methodName, args_json);
    JSON returnObject = JSON::Parse(returnJson);
    return returnObject;
  }

  string invokeRemoteMethod(string methodName,
string arguments) throw (RemoteException) {
    IConnector* connector = _router-
>getConnecter();
    string result;
    if (connector == NULL)
      throw RemoteException("Connector can not be
initialized");
    JSON method;
    method["method"] = methodName;
    JSON jsonValues = JSON::Parse(arguments);
    if (jsonValues.IsArray() == true) {
      method["values"] = jsonValues;
    }
    else { method["values"] = new
JSON(JSON_Array); }
    string methodCall = method.toString();
    string methodReturn = connector-
>Send(methodCall);
    assert(methodReturn.size() > 0);
    JSON delegateMethod =
JSON::Parse(methodReturn);
    if (delegateMethod.IsObject() == true) {
      string methodName = "";
      if (delegateMethod.contains("exception") &&
delegateMethod["exception"].IsString()) {
        string error =
delegateMethod["exception"];
        throw RemoteException(error);
      }
      if (delegateMethod.contains("method") &&
delegateMethod["method"].IsString()) {
        methodName = delegateMethod["method"];
      }
      if (methodName == methodName) {
        if (delegateMethod.contains("return")) {
          result =
delegateMethod["return"].toString();
        }
      }
    }
    return result;
  }

  virtual string handleResponse(string response)
const throw (RemoteException) {
    string result;
    JSON methodArgs = JSON::Parse(response);
    JSON delegateMethod;
    if (methodArgs.IsArray() == true) {
      _arguments = methodArgs;
      if (!(_context->*_onReceive)()) { return
throwException("Invalid Values"); }
      JSON method;
      method["method"] = _methodName;
      method["return"] = (_context->*_invoke)();
      JSON value = method;
      result = value.toString();
    }
    else {
      JSON ex;
      ex["exception"] = "Unknown method";
      JSON value = ex;
      result = value.toString();
      return result;
    }
    return result;
  }

protected:
  mutable JSON _arguments;
private:
  string throwException(string msg) const throw
(RemoteException) {
    JSON ex;
    ex["exception"] = msg;
    JSON value = ex;
    return value.toString();
  }
};

#define GENERATE_REMOTE_METHODS_ARGUMENTS(N,
value)
\
template<class Caller, class ReturnType,
VAR_TYPES(N)>
\
class RemoteMethod<Caller, ReturnType,
VAR_ARGS(N)> : public IPort<ReturnType>, public
ActiveConstraintUID{public:            \
  typedef typename
MethodCallbackSignature<Caller, ReturnType,
VAR_ARGS(N)>::Method Callback;
\
  typedef typename Active<Caller, ReturnType,
VAR_ARGS(N)> ActiveRemoteMethod;
\
  RemoteMethod(Caller* caller, Callback method,
Scheduler<Caller>* sch, string methodName,
IMessageRouter* messageRouter)        \
    :_method(caller, sch, method),
transport(this, &RemoteMethod::receive,
&RemoteMethod::invoke, methodName, messageRouter)
{ }  \
  RemoteMethod(ActiveRemoteMethod method, string
methodName, IMessageRouter* messageRouter)
\
    :_method(method), transport(this,
&RemoteMethod::receive, &RemoteMethod::invoke,
methodName, messageRouter) { }          \
  ReturnType operator () (VAR_ARGS_MEMBERS(N,
arg, COMMA)) {
\
    INIT_VALUE_ARGS(N, arg);
\
    if (transport.getRouter()->getType() ==
RemoteMethodType::Client) { return send(); }
\
    else { return invoke(); }
\
  }
\
```

```

```
private:
\
  JSONTransport<RemoteMethod, ReturnType>
transport;
\
  ActiveRemoteMethod _method;
\
  VAR_ARGS_MEMBERS(N, _arg, SEMICOLON);
\
  virtual bool receive() {
\
    if (transport.length() == N) {
\
      try{ SERIALIZE_ARGS(N, arg); }
\
      catch (...) { return false; }
\
    }
\
    else { return false; }
\
    return true;
\
  }
\
  ReturnType invoke() {
\
    FutureResult<ReturnType> result =
_method(VAR_NAMED_ARGS(N, _arg));
\
    result.wait();
\
    return result.data();
\
  }
\
  virtual ReturnType send() {
\
    DESERIALIZE_ARGS(N, arg);
\
    return transport.remote();
\
  }
\
};
\
template<class Caller, VAR_TYPES(N)>
\
class RemoteMethod<Caller, void, VAR_ARGS(N)> :
public IPort<void>, public
ActiveConstraintUID{public: \
  typedef typename
MethodCallbackSignature<Caller, void,
VAR_ARGS(N)>::Method Callback;
\
  typedef typename Active<Caller, void,
VAR_ARGS(N)> ActiveRemoteMethod;
\
  RemoteMethod(Caller* caller, Callback method,
Scheduler<Caller>* sch, string methodName,
IMessageRouter* messageRouter)   \
    :_method(caller, sch, method),
transport(this, &RemoteMethod::receive,
&RemoteMethod::invoke, methodName, messageRouter)
{ } \
  RemoteMethod(ActiveRemoteMethod method, string
methodName, IMessageRouter* messageRouter)
\
    :_method(method), transport(this,
&RemoteMethod::receive, &RemoteMethod::invoke,
methodName, messageRouter) { }   \
  void operator () (VAR_ARGS_MEMBERS(N, arg,
COMMA)) {
\
    INIT_VALUE_ARGS(N, arg);
\
    if (transport.getRouter()->getType() ==
RemoteMethodType::Client) { send(); }
\
    else { invoke(); }
\
  }
\
private:
\
  JSONTransport<RemoteMethod, string> transport;
\
  ActiveRemoteMethod _method;
\
  VAR_ARGS_MEMBERS(N, _arg, SEMICOLON);
\
  virtual bool receive() {
\
    if (transport.length() == N) {
\
      try{ SERIALIZE_ARGS(N, arg); }
\
      catch (...) { return false; }
\
    }
\
    else { return false; }
\
    return true;
\
  }
\
  string invoke() {
\
    FutureResult<void> result =
_method(VAR_NAMED_ARGS(N, _arg));
\
    result.wait();
\
    return "";
\
  }
\
  virtual void send() {
\
    DESERIALIZE_ARGS(N, arg);
\
    transport.remote();
\
  }};
```
```

``

```
#define GENERATE_REMOTE_METHODS(N)  REPEAT_DEC(N,
GENERATE_REMOTE_METHODS_ARGUMENTS, EMPTY)

template<class Caller, class ReturnType,
VAR_TYPES_DEFAULT(ARGUMENT_UPPER_LIMIT, void)>
class RemoteMethod;
template<class Caller, class ReturnType,
VAR_TYPES(ARGUMENT_UPPER_LIMIT)>
class RemoteMethod : public IPort<ReturnType>,
public ActiveConstraintUID{
public:
  typedef typename
MethodCallbackSignature<Caller, ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)>::Method Callback;
  typedef typename Active<Caller, ReturnType,
VAR_ARGS(ARGUMENT_UPPER_LIMIT)>
ActiveRemoteMethod;
  RemoteMethod(Caller* caller, Callback method,
Scheduler<Caller>* sch, string methodName,
IMessageRouter* messageRouter)
    :_method(caller, sch, method),
transport(this, &RemoteMethod::receive,
&RemoteMethod::invoke, methodName, messageRouter)
{ }
  RemoteMethod(ActiveRemoteMethod method, string
methodName, IMessageRouter* messageRouter)
    :_method(method), transport(this,
&RemoteMethod::receive, &RemoteMethod::invoke,
methodName, messageRouter) { }
  ReturnType operator ()
(VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT, arg,
COMMA)) {
    INIT_VALUE_ARGS(ARGUMENT_UPPER_LIMIT, arg);
    if (transport.getRouter()->getType() ==
RemoteMethodType::Client) { return send(); } else
{ return invoke(); }
  }
private:
  JSONTransport<RemoteMethod, ReturnType>
transport;
  ActiveRemoteMethod _method;
  VAR_ARGS_MEMBERS(ARGUMENT_UPPER_LIMIT, _arg,
SEMICOLON);
  virtual bool receive() {
    if (transport.length() ==
ARGUMENT_UPPER_LIMIT) {
      try{ SERIALIZE_ARGS(ARGUMENT_UPPER_LIMIT,
arg);}catch (...) { return false; }
    } else { return false; }
    return true;
  }
  ReturnType invoke() {
    FutureResult<ReturnType> result =
_method(VAR_NAMED_ARGS(ARGUMENT_UPPER_LIMIT,
_arg));
    result.wait();
    return result.data();
  }
  virtual ReturnType send() {
    DESERIALIZE_ARGS(ARGUMENT_UPPER_LIMIT, arg);
    return transport.remote();
  }
};

template<class Caller, class ReturnType>
class RemoteMethod<Caller, ReturnType> : public
IPort<ReturnType>, public ActiveConstraintUID{
public:
  typedef typename
MethodCallbackSignature<Caller,
ReturnType>::Method Callback;
  typedef typename Active<Caller, ReturnType>
ActiveRemoteMethod;
  RemoteMethod(Caller* caller, Callback method,
Scheduler<Caller>* sch, string methodName,
IMessageRouter* messageRouter)
    :_method(caller, sch, method),
transport(this, &RemoteMethod::receive,
&RemoteMethod::invoke, methodName, messageRouter)
{ }
  RemoteMethod(ActiveRemoteMethod method, string
methodName, IMessageRouter* messageRouter)
    :_method(method), transport(this,
&RemoteMethod::receive, &RemoteMethod::invoke,
methodName, messageRouter) { }
  ReturnType operator () () {
    if (transport.getRouter()->getType() ==
RemoteMethodType::Client) { return send(); }
    else { return invoke(); }
  }
private:
  ActiveRemoteMethod _method;
  JSONTransport<RemoteMethod, ReturnType>
transport;
  virtual bool receive() {
    if (transport.length() == 0) {
    }
    else { return false; }
    return true;
  }
  ReturnType invoke() {
    FutureResult<ReturnType> result = _method();
    result.wait();
    return result.data();
  }
  virtual ReturnType send() {
    return transport.remote();
  }
};

template<class Caller>
class RemoteMethod<Caller, void> : public
IPort<void>, public ActiveConstraintUID{
public:
  typedef typename
MethodCallbackSignature<Caller, void>::Method
Callback;
  typedef typename Active<Caller, void>
ActiveRemoteMethod;
  RemoteMethod(Caller* caller, Callback method,
Scheduler<Caller>* sch, string methodName,
IMessageRouter* messageRouter)
    :_method(caller, sch, method),
transport(this, &RemoteMethod::receive,
&RemoteMethod::invoke, methodName, messageRouter)
{ }
  RemoteMethod(ActiveRemoteMethod method, string
methodName, IMessageRouter* messageRouter)
```

``

```
    :_method(method), transport(this,
&RemoteMethod::receive, &RemoteMethod::invoke,
methodName, messageRouter) { }
  void operator () () {
    if (transport.getRouter()->getType() ==
RemoteMethodType::Client) { send(); }
    else { invoke(); }
  }
private:
  ActiveRemoteMethod _method;
  JSONTransport<RemoteMethod, string> transport;
  virtual bool receive() {
    if (transport.length() == 0) {
    }
    else { return false; }
    return true;
  }
  string invoke() {
    FutureResult<void> result = _method();
    result.wait();
    return "";
  }
  virtual void send() {
    transport.remote();
  }
};

GENERATE_REMOTE_METHODS(ARGUMENT_UPPER_LIMIT)


//A template function to get the index of an item
for a given vector. It returns -1 if the
//item was not found
template <typename InputIterator, typename
EqualityComparable>
  int IndexOf(const InputIterator& begin,
  const InputIterator& end, const
EqualityComparable& item) {
    if(begin == end )
      return -1;

    InputIterator fnd = std::find(begin, end,
item);
    unsigned int index = std::distance(begin,
fnd);
    return fnd != end ? index : -1;
};

//Used as a function pointer to safely destroy
elements or collections of elements
//Example:
//  std::for_each( arr->begin(), arr->end(),
delete_pointer_element());
//  delete arr;
struct deleteElement{
template< typename T >
void operator()( T element ) const{
  delete element;
  }
};

template<typename T>
unsigned int compare(T c1, T c2, unsigned int sz
) {
  if(!is_ptr<T>::value) {
    return std::memcmp(&c1,&c2, sz);
  }
  return std::memcmp((void*)c1, (void*)c2, sz);
};

template<typename T>
void copyObject(T* object1, T* object2, unsigned
int sz ) {
  if(!is_ptr<T>::value) {
    std::memcpy(&object2,&object1, sz);
  }
  std::memcpy((void*)object2, (void*)object1,
sz);
};


//A template function to copy vector
template<typename InputIterator, typename T>
void copyVector(InputIterator& it, InputIterator&
end ,T* vectorObj) {
  for (; it != end; ++it) {
    vectorObj->push_back(*it);
  }
};


typedef char samllInt;
typedef unsigned char unsignedSamllInt;
typedef short int shortInt;
typedef unsigned short int unsignedShortInt;

#define MSG_MAX 80
#define MSG_SIZE 128

typedef int SlotHandle;

template <typename T,typename P>
class IPublisher
{
public:
  virtual T publish(P param) = 0;
};

template <typename L,typename R,typename P>
class Publisher : public IPublisher<R,P>
{
private:
  typedef R (L::*FuncPtr)(P);
  L* _object;
  FuncPtr _functionPointer;

public:
  Publisher(L* object, FuncPtr funcPtr)
    : _object(object), _functionPointer(funcPtr)
  {}

  R publish(P arg)
  {
    return (_object->*_functionPointer)(arg);
  }
};

template <typename R,typename P1>
```

``

```
class Event
{
private:
  typedef std::map<int,IPublisher<R,P1> *>
SubscribersMap;
  SubscribersMap subscribers;
  int subscribersCount;

public:
  Event()
    : subscribersCount(0) {}

  template <typename L>
  SlotHandle subscribe(L* component,R
(L::*func)(P1))
  {
    typedef R (L::*FuncPtr)(P1);
    subscribers[subscribersCount] = (new
Publisher<L,R,P1>(component,func));
    subscribersCount++;
    return subscribersCount-1;
  }

  bool disconnect(SlotHandle id)
  {
    typename SubscribersMap::iterator it =
subscribers.find(id);
    if(it == subscribers.end())
      return false;
    delete it->second;
    subscribers.erase(it);
    return true;
  }

  R publish(P1 arg)
  {
    typename SubscribersMap::iterator it =
subscribers.begin();
    for(; it != subscribers.end(); it++)
    {
      it->second->publish(arg);
    }
  }
};

typedef struct MessageHeader{
  shortInt portId;
  shortInt eventId;
  void* data;
} MessageHeader;



class MessageService {
  private:
    mutable MutexLock lock;

    queue<MessageHeader*>* msgQueue;
    queue<MessageHeader*>* msgPool;
    //Thread* executeThread;

    SlotHandle dispatcherHandleId;
    Event<void,const MessageHeader*>
msgDispatcher;
    unsignedSamllInt buffer[MSG_MAX*MSG_SIZE];
  public:
    template <typename L,typename R,typename P1>
    MessageService(L* component,R (L::*func)(P1))
    {
      dispatcherHandleId =
msgDispatcher.subscribe(component,func);
      msgQueue = new queue<MessageHeader*>();
      msgPool = new queue<MessageHeader*>();
      for (int i=0; i< MSG_MAX; i++){
        MessageHeader* block = (MessageHeader*)
&buffer[i* MSG_SIZE];
        msgPool->push(block);
      }

    }

    virtual ~MessageService() {

msgDispatcher.disconnect(dispatcherHandleId);
      while(!msgQueue->empty()) {
        MessageHeader* msg = msgQueue->front();
        msgQueue->pop();
        delete msg;
      }
      delete msgQueue;

      while(!msgPool->empty()) {
        MessageHeader* msg = msgPool->front();
        msgPool->pop();
        delete msg->data;
        delete msg;
      }
      delete msgPool;
    }

    void push(MessageHeader* msg) {
      {
        msgQueue->push(msg);
        execute();
      }
    }
    MessageHeader* pop() {
      MessageHeader* msg = msgQueue->front();
      msgQueue->pop();
      return msg;
    }

    MessageHeader* getBufferedMessage(){
      {
        if (msgPool->size()>0){
          MessageHeader* msg = msgPool->front();
          msgPool->pop();
          return msg;
        }
      }
      return NULL;
    }

    void poolMessage(MessageHeader* buffer){
      msgPool->push(buffer);
    }
```

``

```
    void execute() {
        while(!msgQueue->empty()) {
          MessageHeader* msg = msgQueue->front();
          msgQueue->pop();
          msgDispatcher.publish(msg);
          poolMessage(msg);
        }
    }
};

#endif
```