Documenting Software Architecture: Documenting Behavior

F. Bachman, L. Bass, P. Clements, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford
This work is part of a book on how to produce *high-quality documentation* for software architectures.

- The book is intended to be a *language-independent* guidance.
- Two essential steps:
  1. documenting the set of *relevant views* of that architecture
  2. documenting information that *transcends any single view*.
- This third part describes ways to document the *behavior* of a system.
Beyond structure

- The classical approach to organize the documentation of an architecture uses a collection of *views*.

- A view is described mostly in terms of *structural relationships* among the elements it contains.

- However, architecture *extends beyond structure*.

- To be sure that the system will work as intended, the *behavior of the elements of a view* has to be taken into account.

- Add more *semantic detail* to elements and their interactions that have time-related characteristics.
Beyond structure: what do we need?

- What do we need? A sort of “time-line”.
- Structural relationships provide a view of the system that reflects all potential interactions.
- Only a limited number of these interactions will be actually active during any instant of the system execution.
- It is the system behavior that describes how element interactions may affect one another at any point in time in a given system state.
- Every view is associated with a description that document the behavior of its elements.
Beyond structure: what do we have to describe?

Some system attributes can be analyzed *entirely using a structural description*.

For example: anomalies like inputs for which there is no source available.

On the other hand, properties like potential deadlock contain information about *both the behavior of elements and interactions* among them.

A behavioral description adds information about

- ordering of interactions among the elements,
- opportunities for concurrency,
- time dependencies of interactions

Example: statecharts defined by the Unified Modeling Language (UML).
Where to document behavior

Behavior can be shown in a number of places, depending on what is being shown:

- In a view’s supporting documentation:
  - Behavior has *its own section* in the element catalog.
  - Behavior can be part of an *element’s interface documentation*. For example in the “usage guide” section of an interface document.
  - “Design Background” section, where the results of the analysis are shown.

- In the documentation that applies across views, the rationale for why the architecture satisfies its requirements can include behavioral specifications.
Why document behavior?

- System analysis.
- Driving development activities.
System analysis

With a behavioral description we can reason about the system’s *completeness*, *correctness*, and *quality attributes*.

- It is possible to *simulate the behavior* of the proposed system in order to reason about the architecture’s ability to support system requirements.
- Documenting system behavior provides support for *exploring the quality attributes* of a system very early in the development process.
- Compositional-reasoning techniques that are available today require information about both the internal behavior of a system elements and interactions among them.
Driving development activities

- Behavioral documentation play the role of a *vehicle of communication* among stakeholders during system-development activities.

- **System decomposition.** The behavior of an element has an important influence on the structure of its decomposition.

- Implementing a system using a defined architecture is a continuous process of *decomposition in smaller and more detailed elements* until it is possible to describe the behavior in a programming language.

- **Simulation** can be used during the development of the system.
At a minimum, we have to model

- the *stimulation of actions*,
- the *transfer of information* from an element to another,
- the *time-related and ordering constraints* of these interactions.

For example:

- Types of lines interconnecting two elements: flow of data or flow of control.
- Types of communication: sync, async.
- Types of support for communication: channel, shared memory.
- Types of reactions of an element to its input: requires all of just some of its input to be present before it begins calculating.
Any notation for documenting system behavior must include constructs for describing *sequences of interactions*.

Sequences of interactions are displayed as a set of stimuli and the triggered activities ordered by some means, for example:

- *a line,*
- *numbering,*
- *ordering from top to bottom.*

As a description of behavior refers to structure, the structural elements can be part of the notation.

There are *two type* behavioral documentation tools:

- *static views,*
- *traces.*

Often tools of both types can be used together to support the design process.
This kind of behavioral documentation, shows the *complete behavior* of a structural element or set of elements.

- Often tools of this type are *state based*.
- It is referred as “static” because it is possible to *infer all possible traces through a system*.
- The tools of this type need to support the *description of alternatives and repetitions* to provide the *complete computing power*.
- This type of documentation should be chosen when *simulation is required* or when *static-analysis techniques will be used*.
- The authors describe four tools of this kind:
  - *statecharts*
  - *ROOMcharts*
  - *Specification Description Language (SDL)*
  - *Z language (pronounced “zed”)*
Statecharts

- Formalism developed by David Harel in the 80s for describing *reactive systems*.

- Powerful graphic notation that allow us to trace the behavior of a system given specific inputs.

- A lot of useful *extensions* to traditional state diagrams.
  - *Nesting of states*: when the superstate is entered, all substates are initiated at their respective default start state and they remain active until the superstate is exited.
  - *Orthogonal regions*: substates in orthogonal regions are activated concurrently upon entry to the superstate.
    . . .

- Expressive power to model *abstraction and concurrency*.
Sequence is represented by a single-headed arrow leading from the source state to the target state.

An arrow is annotated with a pair (cause event/generated event).

Concurrency is represented by grouping sets of states into a superstate.
Limitations of statecharts

Several *simplifying assumptions* are incorporated into the statechart model.

- All transitions take *zero time*. This allows a set of transitions within a substate to replace a single transition at the superstate level.
- All transitions are *reliable*.

Therefore, there is not any built-in support for *modeling protocols*. 
As its name says, the *Real-time Object-Oriented Modeling* (ROOM) environment is an approach to developing real-time software using *object-oriented design techniques*. So ROOM supports the use of *data abstraction*, *encapsulation*, and *inheritance*. In ROOM descriptions there are *actors* that communicate by *exchanging messages*. The behavior of an actor is documented as a *hierarchical state machine* and is incorporated into a ROOMchart.
A ROOMchart is a graphical notation that is a **cousin to the statechart**

The concept in ROOMcharts are very close to the ones in **languages** thus allowing a smooth transition from the high-level design down to the implementation.

Unlike statecharts ROOMcharts **support the modeling of protocols**.

Unlike statecharts ROOMcharts **do not support AND states**.

However there are features for supporting concurrency but with more effort.

Support for **inheritance** and **encapsulation**.

**Simulation is widely supported**: ROOMcharts run on a **virtual machine** provided by the ROOM environment.
**Specification Description Language** (SDL) is an object-oriented, formal language defined by the International Telecommunication Union (ITU).

It is intended for the specification of complex, *event-driven*, *real-time*, and *interactive* applications.

The most commonly application is in the *telephony area* and, in general, in *communication systems*.

The **strength** of SDL is in describing *what happens within a system*.

The **weakness** of SDL is in describing *the interactions between sub-systems*. 
In SDL, structure is described in terms of a **hierarchy of blocks** that can be refined into **sets of processes**.

The flow of data and stimulation among blocks and processes is described as **signals over named channels**.

- Communication is **asynchronous** and specified textually as an annotation attached to a communication channel.
- Signal are visible to other blocks/processes at **lower levels in the hierarchy**.
The internal behavior of a process is described using a *flow-chart notation*.

Processes run *concurrently* and *independently*; concurrent processes have no knowledge of each other's state.

SDL supports *user-defined data types* and provides several predefined types (integers, real, natural, Boolean... ) with the usual meanings.

Once both structural view and system behavior have been documented, *it is possible to simulate the system* and observe control and data flow.
Z (pronounced “zed”) is a mathematical language based on predicate logic and set theory. The Z language focuses on data and its transformations. Systems are specified as sets of schemas. Schemas can be combined using the schema calculus to create a complete behavior. The schema calculus allows type checking. The Z language is particularly useful when you desire to prove properties based on the specification. Last but not least, a lot of commercial tools is available to support developing system with Z language.
The ScheduleClass schema defines what it means to add a class to a schedule.

Above the center horizontal line there are the variable definitions.

The letter $D$ signifies that a schema named Schedule exists and that all of its variables are available to ScheduleClass.

The variables that end in a question mark are input variables.

Below the center horizontal line there are first the pre-conditions for an operation and then the promised results of the transformations.

In this example the class will be added to the schedule and will be associated with the specified time.
Trace-oriented representations consist of *sequences of activities or interactions* that describe the system’s response to a *specific stimulus*.

It is documented the trace of activities through a system described in terms of *its structural elements* and *their interactions*.

Trace descriptions can be used when we have to analyze specific difficult situation that the system has to deal with.

There are different techniques that emphasize particular aspects.

- **Message-oriented** techniques focus on describing the message exchange between instances.
- **Component-oriented** techniques focus on describing which behavioral features an element has to have to accommodate the system performing its functions.
- **Activity-oriented** techniques focus on describing which activities have to be performed in order to achieve the purpose of the system.
- **Flow-oriented** techniques focus on describing the sequencing of responsibilities of elements for a specific scenario or trace.
Specific trace-based techniques examined.

- **Use-case diagrams**: activity-oriented.
- **Use-case maps** (UCM): flow-oriented.
- **Sequence diagrams and procedural sequence diagrams**: respectively message-oriented and component-oriented.
- **Collaboration diagrams**: component-oriented.
- **MSC**: message-oriented.
Use-case diagrams show *how users interact with use cases* and how the latter are interrelated.

- The purpose of a use case is to define *a piece of the behavior of an element*.
- A use case does not take count of the internal structure of the element.
- Each use case represents *a specific way of using an element* (a service).
- The service is initiated by an user and is composed by *a sequence of interactions* between the user and the element.
- Use cases *cannot be decomposed* thus, in a system decomposition, the use cases for the children elements are the basis for the use cases of the parent element.
- Use-case diagrams *do not have means to document concurrency*.
Use-case diagrams: example

The top portion shows how phone terminals interact with the “Establish Point-to-Point Connection” use case.

Users (in this case phone terminals) are represented by **actors. An actor is a set of roles** that external entities assume when interacting with use cases.

Lines between actors and use cases represent the possible interactions.

Use cases **can have relationship with each other**.

In particular, an **extend** relation (dashed arrow) defines that instances of a use case may be extended with additional behavior defined in another use case.

With the **include** relationship we have a tool **like the inheritance of OO languages**.
Use-case maps (UCMs)

- Developed at Carleton University in 1992.
- UCMs allow dynamic behavior and structures to be represented and evaluated.
- The UCMs concentrate on *visualizing execution paths* through a set of elements: this is a *path-centric view* of system functionalities.
- The basis for UCMs is built on the following notation.

  The execution paths describe how elements are ordered according to their responsibilities in the computation.

  Them UCM notation aims *to link behavior* and *structure* in an *explicit* and *visual* way.
Like the other representations, UCMs show instances of structural elements.

In addition UCMs have a notation for “containment” that is normally shown in structural description.

An execution path that enters an element states that now this element has a responsibility in the execution.

The notation include means to represent the decomposition of execution paths.
Sequence diagrams document *a sequence of stimuli exchanges*.

A sequence diagram has *two dimensions*:

1. The *vertical dimension* represents *time*.
2. The *horizontal dimension* represent different *objects*.

Associations among objects are not shown and there is *no significance in the horizontal ordering of the objects*.

Sequence diagrams *do not furnish tools to define precisely the temporal constraint in case of concurrency*.

![Sequence Diagram Example](image)
Instances have a **lifeline** drawn as a vertical line along the time axis.

The arrows between lifelines can describe **creation** or **destruction of instances**.

Any other stimulus is depicted as an arrows between lifelines.

A stimulus can be annotated with a name of a function.

A stimulus represented as **a dotted arrow** describes **a return of control to the sender** of a previous stimulus.
This variation focuses on the *interface interactions of elements* and is *more suitable to show concurrency*, because it has means to show flow of control.

Thin boxes over the lifeline are the *focus of control* and are used to show that some computation is done by the instance in a precise interval of time.

Alternative paths and parallel execution can be described (in the figure respectively for “*Originating Connection*” and “*Destinating Connection*”).
Collaboration diagrams

- Collaboration diagrams show the *relationships among the interfaces of instances*.
- Collaboration diagrams *are very useful* when the task is to verify that a structure design can fulfill the functional requirements.
- Collaboration diagrams *are not useful if the understanding of concurrent actions is important* (for example in performance analysis).
- The relationships among the instances, called *links*, are also shown in collaborations diagram. Links are simple lines and only state that the two involved instances can interact.
- Sequence diagrams and collaboration diagrams *express similar informations*.
The stimuli are shown as **arrows attached to a link** between instances.

**Sequence numbers** can be added to stimuli to show which stimulus follow which.

**Sub-numbering** can be used to show nested stimuli and/or parallelism.

With numbering, the **information contained in this kind of diagrams is analogous to that obtainable from a sequence diagram** but with the complexity the diagrams become unreadable.
MSCs

An MSC is a *message-oriented representation* that contains the description of the *asynchronous communication* between instances.

Simple MSCs can look like *sequence diagrams*, but they have *a more specific definition and a richer notation*.

Their main area of application is in *telecommunication switching systems*.

A big advantage of MSCs is that in addition to graphical representations, they have *a textual specification language* defined for them.

MSCs can often be seen in conjunction with the SDL (they are both defined by the International Telecommunication Union).

- *MSCs* focus to represent the message exchange *between* instances (systems, processes).
- *SDL* is used to describe what happen *in* a system or process.
MSCs: example

- Instances are boxes with a vertical line.
- The message flow is presented by arrows that can be:
  - horizontal,
  - with a downward slope with respect to the direction to indicate the flow of time.
- A total ordering of the described communication events is assumed along each instance axis.
- The frame represents the scope of the diagram.