# Modeling in Software Architecture

**University of Ottawa SITE Technical Report TR-2009-02**

**Jenya Levin**

Ottawa-Carleton Institute for Computer Science
Ottawa, Ontario, Canada
jlevi053@uottawa.ca

**Abstract**

*Architectural modeling notations are varied based on goals, extensibility, application domain, and other factors. In this paper, multiple graphical and textual modeling notations are reviewed illustrated by examples, with discussion of their advantages and shortcomings.*

## 1. Introduction

According to Taylor et al, "An architectural model is an artifact that captures some or all of the design decisions that comprise a system's architecture. Architectural modeling is the reification and documentation of those design decisions... An architectural modeling notation is a language or means of capturing design decisions." [1]

Without a model, an architecture is inscrutable. Modeling notations can be rich and ambiguous (for example, natural language), highly formal (such as the Rapide notation), semantically weak (such as Power-Point diagrams), or semantically formal. Several notations can be combined to describe an architecture - for example, a UML class diagram can be annotated with natural language.

Section 2 of this paper deals with basic architectural concepts and design decisions that need to be modelled, as well as modeling views, viewpoints, and visualizations. Section 3 covers various modeling notations: natural language (3.1); informal graphical styles (3.2); Unified Modeling Language (3.3); early Architecture Description Languages (3.4) – Darwin (3.4.1), Rapide (3.4.2), and Wright (3.4.3); domain-specific and style-specific Architecture Description Languages (3.5) – Koala (3.5.1), Weaves (3.5.2), and AADL (3.5.3); extensible Architecture Description Languages (3.6) – Acme (3.6.1), ADML (3.6.2), and xADL (3.6.3); User Requirements Notation (3.7); and Dialog Flow Notation (3.8). Section 4 contains discussion of the presented modeling notations.

## 2. Architectural modeling

To choose a notation, one has to decide upon what architectural decisions and concepts should be modeled, the level of detail to include in the models, and how formal the models should be. The choices depend on the cost-benefit analysis of creating and maintaining the models. The most important aspects of the system are modeled with

the greatest detail. Modeling goals include communication, bug finding, quality analysis, generation of needed artifacts, etc. [1]

### 2.1. Basic architectural concepts that need to be modelled

1. Components - subsets of system's functionality and data, exposed via explicitly defined interfaces.
2. Connectors - building blocks regulating interactions among components.
3. Interfaces - points at which components and connectors interact with the outside world.
4. Configurations - specific associations between the components and connectors in the system.
5. Rationale - documentation of architectural decisions.

According to Taylor et al, “Architectural style is a collection of design decisions that are applicable in a given development context, constrain design decisions within that context, and elicit beneficial qualities in the resulting system.” [1]

### 2.2. Design decisions that might be captured in a style model

Various elements and constraints can be captured in a style model. Listed below are types of design decisions detailed by style models.

1. Specific elements - particular components, interfaces and connectors to be used in a particular way in specific situations. Modeling templates or base models can be thus created.
2. Component, connector, and interface types - kinds of elements permitted, required, or prohibited in the architecture.
3. Interaction constraints - temporal (call one before another), topological (can be invoked only by components belonging to a certain layer), protocol (one or a group) constraints on interaction between components and connectors.
4. Behavioural constraints - from simple rules to complete finite state automata of the system.
5. Concurrency constraints - rules for concurrency, synchronization, access to shared resources. [1]

### 2.3. Modeling other aspects of the system

Static aspects are easier to model as they do not include changes over time. Dynamic aspects of a system are harder to represent. This includes interaction models, data flow models, and behavioural models. In cases when the system is usually stable but might have to change behaviour if one of the components fails, both the static and dynamic models might be needed.

Modeling dynamic aspects of a system can be done using static modeling (for example, interaction diagrams). Dynamic models can be employed to visualize the behaviour of a running system, and are updated on-the-fly.

Functional (what a system does) and non-functional (constraints on what a system does) aspects of the system can be captured through modeling. Functional aspects are easier to model as they are complete. Most modeling notations are designed to capture primarily functional aspects. Non-functional aspects are often captured less rigorously, as they are qualitative and subjective. [1]

## 2.4. Mixed content modeling and multiple views

Taylor et al present the following definitions for a view and a viewpoint [1]:

- A view is a set of design decisions related by a common concern (or set of concerns).
- A viewpoint is the perspective from which a view is taken.
- A view (what you see) is an instance of a viewpoint (filter) for a specific system.

### 2.4.1. Traditional viewpoints

Viewpoints provide a way to limit presented information to a cognitively manageable subset of the architecture, display related concepts simultaneously, can be tailored to the needs of specific stakeholders, can be used to display the same data at various levels of abstraction. [1]

Listed below are several traditional viewpoints frequently used in modeling:

1. Logical - logical (often software) entities in the system and how they are interconnected.
2. Physical - physical (often hardware) entities in the system and how they are interconnected.
3. Deployment - how logical entities are mapped onto physical entities.
4. Concurrency - how concurrency and threading are managed in the system
5. Behavioural - expected behaviour of the system or its parts.

### 2.4.2. Inconsistencies among views

The following inconsistencies may exist among views:

1. Direct inconsistencies - two views assert directly contradictory propositions, automatically detectable.
2. Refinement inconsistencies - two views at different levels of detail assert contradictory propositions; can be automatically detected with appropriate consistency rules.
3. Static versus dynamic aspect inconsistencies - view of a static aspect conflicts with a view of the dynamic aspect of the system, harder to automatically detect, depending on how explicit the dynamic aspect's specification is.

4. Dynamic aspect inconsistencies - two views of dynamic aspects of the system conflict, often extremely difficult to detect automatically as it requires extensive state exploration or simulations.
5. Functional versus non-functional aspect inconsistencies - non-functional property prescribed by a non-functional view is not met by the design expressed in functional views; the most difficult to detect because of the general and abstract nature of non-functional properties. [1]

## 2.5. Model visualizations

An architectural visualization defines how architectural models are depicted, and how stakeholders interact with those depictions.

Model is organized information. Visualizations are ways to represent the organized information, usually graphical or textual. Visualizations can be used to hide differences in back-end modeling notations. Many modeling notations have one or more canonical visualizations. Visualizations are associated with particular viewpoints, not views, since the same visualization can be used to visualize many different architectures. [1]

### 2.5.1 Types of visualizations:

**Textual visualizations** – visualizations that depict architectures using ordinary text files. Textual visualizations have several advantages, such as ability to depict the entirety of the architecture in a single file. Models are easily accessible and editable via multiple available text editors. Parsing, processing, and editing structured text are mature technologies: when the syntax is defined according to a particular meta-language (for instance, Backus-Naur form), program libraries can be generated to parse and check the syntax of the documents. Additional features are often available, such as auto-completion and syntax checking.

Disadvantages of textual visualizations stem from the fact that models are usually presented linearly in text editors, without a possibility to reorganize the information in multiple ways. While good at depicting data linearly and hierarchically, they are not as good at graph-like structure depiction.

**Graphical visualizations** – visualizations that depict architecture primarily using graphical symbols, they either confirm to a syntax or are free-form. These notations give stakeholders access to information about architectures in many ways that textual representations cannot: using colours, symbols, and other decorations that are more easily distinguishable than structured text elements. Non-hierarchical relationships can be more effectively presented. Spatial relationships can be used to express relationships among elements. Animation can be employed to show transitions. Interaction mechanisms are superior, allowing for scrolling, zooming, drilling down, showing and hiding elements, as well as allowing direct manipulation with a mouse. [1]

Disadvantages of graphical visualizations are due to graphical notations requiring costly development of tools to support the interactions, as well as proprietary file formats of the tools making connection to a more architecture-centric representation difficult [2, 3].

**Hybrid visualizations** are achieved by a combination of graphical and textual elements (such as UML combined with OCL).

### 2.5.2. Common issues in visualization

There are issues that affect visualizations, due to differences in semantic meaning of symbols and decorations. Listed below are several possible situations that might result in semantic and visual inconsistencies:

1. Same symbol - different meaning: consistency in meaning is lacking when using a particular symbol multiple times.
2. Differences without meaning - using different symbols to express the same meaning.
3. Decorations without meaning - "spicing up" diagrams for aesthetic values with meaningless decorations
4. Borrowed symbol, different meaning – not adhering to a rough meaning of a symbol that is borrowed from a different visualization. [1]

# 3. Modeling notations

## 3.1. Natural language

Natural (human) languages, such as English, are expressive, but ambiguous, non formal, and non rigorous. They cannot be effectively processed and understood by machines, and so can only be inspected by humans. Natural languages are easily accessible to stakeholders and can be handled using common word-processing tools. Due to the abstract and qualitative nature of non-functional requirements, natural languages are best in capturing those.

Sometimes, a restricted form of a language is used instead of a pure language. This is achieved by users employing a particular dictionary of terms, to avoid ambiguity. Extensive use of this technique is not effective, as it results in a domain-specific language without the advantages of having a flexibility of a natural language.

Due to the ambiguity of natural languages, they do not lend themselves to automated processing or verification, and so it cannot be tied to implementation of the models. Natural languages can be used across domains and can express a high degree of abstraction. Natural language models use textual visualization.

*Example 1* contains a model of the Lunar Lander system in natural language (in this case, American English) [1]. The Lunar Lander system is presented by Taylor et al [1] and is used as an example throughout this paper to illustrate various modeling notations.

The Lunar Lander application consists of three components: a **data store** component, a **calculation** component, and a **user interface** component.

The job of the **data store** component is to store and allow other components access to the height, velocity, and

fuel of the lander, as well as the current simulator time.

The job of the **calculation** component is to, upon receipt of a burn rate quantity, retrieve current values of height, velocity, and fuel from the **data store** component, update them with respect to the input burn rate, and store the new values back. It also retrieves, increments, and stores back the simulator time. It is also responsible for notifying the calling component of whether the simulator has terminated, and with what state (landed safely, crashed, and so on).

The job of the **user interface** component is to display the current status of the lander using information from both the **calculation** and the **data store** components. While the simulator is running, it retrieves the new burn rate value from the user, and invokes the **calculation** component.

*Example 1. "Lunar Lander in natural language" (adapted from [1])*

## 3.2. Informal graphical styles

### 3.2.1. Power-Point notation

Informal Power-Point-style modeling allows for creation of decorative diagrams of interconnected shapes. They are easily accessible to stakeholders, assuming the tools for their manipulation are available. The slide nature of these presentations limits the diagram size, which ensures suitable abstraction.

Similarly to natural languages, these diagrams facilitate expressive, flexible ways of capturing ideas. They are useful in early prototypes and exploration, but lack the formality and rigour. Due to the lack of semantics, there is no distinction in modeling static and dynamic aspects of the system. Non-functional aspects are modeled using natural-language diagram annotations. Understanding of the diagrams depends on the correct interpretation of the symbols used. In some ways, due to their visual nature, the diagrams are easier to understand than natural language descriptions.

As the natural language models, Power-Point-style models cannot be processed automatically or tied to implementation. Consistency and correctness have to be checked through manual inspection. This modeling technique can be used across domains and can express any degree of abstraction.

Power-Point-style models use graphical visualization manipulated with the help of informal graphical editors. *Figure 1* shows an example of a Power-Point-style diagram [1]. The resulting models are generally straightforward and aesthetically

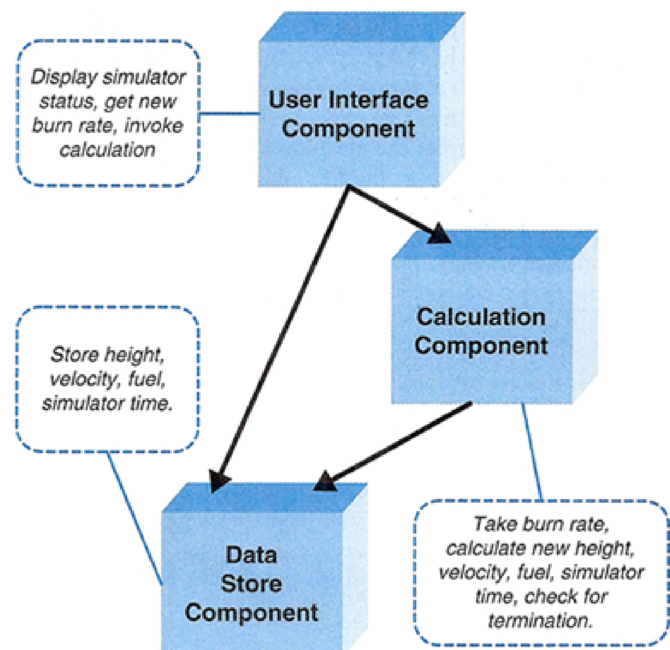


*Figure 1. Lunar Lander in PowerPoint (adapted from [1])*

pleasing; but have no connection with architectural concepts and are very difficult to coordinate with other visualizations. The editors often feature powerful user interfaces and limited animation capabilities. [1]

### 3.2.2 Diagram notation presented in [1]

A notation similar to the Power-Point model notation is used in a few examples in [1]. A legend is provided to denote semantic meaning of the elements involved, yet the models do not appear to be expressed in a notation used by any of the ADLs described later in the book. The notation allows for depicting components, connectors, connections between interfaces and the interaction flow requests and notifications. *Figure 2* shows an example of this one-of-a-kind notation [1].

## 3.3. Unified Modeling Language (UML)

UML [4] combines concepts from earlier notations: Booch diagrams [5], OMT [6], OOSE [7], and Statecharts [8]. UML provides a wide variety of modeling constructs and viewpoints. There are many tools that provide varying degrees of support for UML (from diagram construction to generation of code from UML models).

UML is an extensive notation that traditionally employs graphical symbols with textual annotations to represent the system from thirteen different viewpoints (as of UML 2.0). Early versions of UML [9] focused on detailed design modeling. UML 2.0 has added support for higher-level architectural constructs. [1]

One of the limitations of UML is its primary focus on design. Architectural modeling affects all other stages of the software lifecycle, such as maintenance, and additional activities, such as project management. This limitation requires other notations to be used in combination with UML to fully-model the architecture of the system. [1]

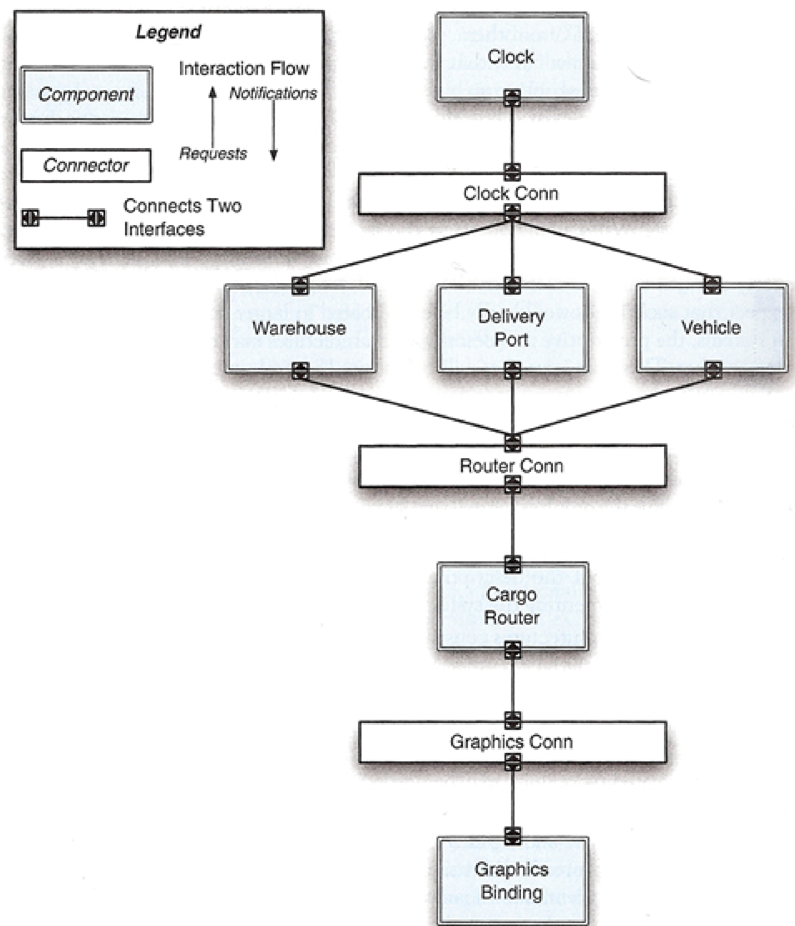


*Figure 2. One-of-a-kind notation (adapted from [1])*

Another feature of UML is the ambiguity of the modeling constructs. This allows for increase in the generality of the notation and its flexibility, but requires additional information to understand the semantics of a particular diagram. For example, a dashed open-headed dependency arrow could mean a function call, a pointer or reference to

an instance of one class from another, instance of one class as a parameter to a method in another, or messages passing from one class to another. [1]

Advantages of UML include:

- A multitude of classes, associations, states, activities, and other constructs
- Ability to model multiple viewpoints
- Support for static (for instance, class diagrams, object diagrams) and dynamic (state diagrams, activity diagrams) modeling
- A variety of available tools
- UML's widespread adoption

Constraints relevant to a particular architectural style can be expressed in OCL (Object Constraint Language), whereas non-functional requirements are loosely supported by allowing natural language annotations. The combination of constraints, stereotypes and tagged values (constituting a UML profile) helps limit the ambiguity of the models.

UML expressed graphically does not lend itself to automated processing. However, UML textual modeling strives to fill that gap. For example, the Umple language [10] allows expressing class diagrams in a textual notation, which is more compact than graphical notation, and can be compiled to produce an implementation of the model.

Through the available viewpoints and a wide variety of modeling constructs, UML allows for multiple levels of abstraction. Different UML models of a system can capture "the big picture" as well as varied levels of design detail.

UML visualization is associated with a canonical depiction, not a canonical user interface. One of the advantages of UML visualization is the consistent use of symbols across diagrams and projects. It should be noted however that consistency in the visualization does not mean semantic precision at the notational level: if a concept is semantically ambiguous, the consistent use of the same shape to represent it does not repair the ambiguity. [1]

Below is an overview of the key UML elements (3.3.1), as well as a few words on Umple notation and UMLet tool for UML diagram creation (3.3.2).

### 3.3.1. UML elements

UML building blocks include things, relationships, and diagrams. Things are abstractions representing elements in the model. Relationships tie things together. Diagrams group relevant collections of things. [9]

#### Things

**Structural things** – nouns in UML models (see *Figure 3*):

- **Class** – describes a set of objects that share same operations, relationships, attributes, and behaviour. A class implements one or multiple interfaces.
- **Interface** – describes externally visible behaviour of an element by specifying services of classes or components.
- **Collaboration** – denotes a structural and behavioural collaboration of a set of elements working together to provide particular cooperative behaviour.
- **Use case** – an element that represents a set of actions available to a particular actor.
- **Active class** – a class that can initiate control activity, due to owning one or more processes or threads. Active classes are used to represent elements that exhibit concurrent behaviour.
- **Component** – a physical and replaceable part of a system that adheres to a set of interfaces.
- **Node** – a computational resource that exists at run time. A node might contain a set of components.

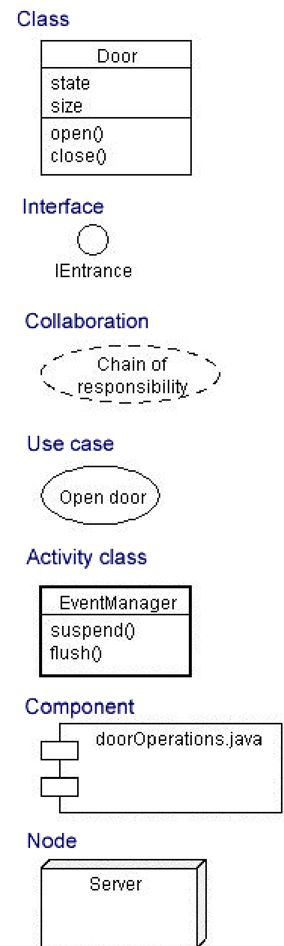


*Figure 3. UML structural things*

**Behavioural things** – verbs in UML models, representing behaviour over time and space (see *Figure 4*):

- **Interaction** – denotes a set of messages exchanged between objects to accomplish a specific purpose within a particular context.
- **State machine** – denotes a set of states of an object, object's response to events depending on the state, and the transitions accompanying each event.

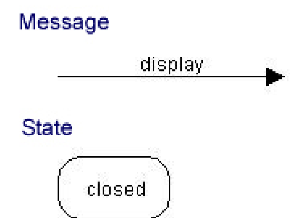


*Figure 4. UML behavioural things*

**Grouping things** – elements assisting in grouping other elements within a diagram (see *Figure 5*):

- **Package** – primary grouping mechanism to envelop structural, behavioural, and other grouping things. This grouping exists in development time, not run time.

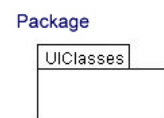


*Figure 5. UML grouping things*

**Annotational things** – elements assisting in explaining other elements (see *Figure 6*):

- **Note** – element that renders comments and constraints related to an element or a collection of elements. Notes can be expressed in informal or formal text.

## Relationships

See *Figure 7* for visualization of the relationships listed below.

**Dependency** – a relationship between two objects where modifications to one object may affect the semantics of the other one.

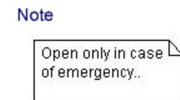


*Figure 6. UML annotational things*

**Association** – a relationship that specifies a connection between objects, such as aggregation, for instance (where link is between a whole and its parts).

**Generalization** – a relationship where child objects (specialized elements) can be substituted for the parent object (a generalized element).

**Realization** – a relationship between classifiers where one specifies a contract and another one guarantees to carry it out. This relationship is encountered between interfaces and the classes or components that realize them, as well as between use cases and the collaborations that realize them.

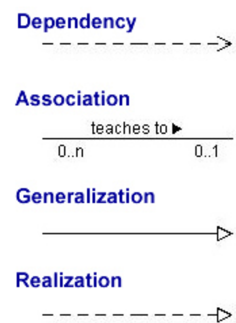


*Figure 7. UML relationships*

## Diagrams

**Class diagram** is a static view of the system that shows classes, interfaces, and collaborations together with the relationships among them.

**Object diagram** shows a static snapshot of objects and their relationships within a running system. This is useful to explore system prototypes.

**Use case diagram** shows a static view of actors and a set of available actions (use cases) for each actor, to assist in modeling the behaviour of the system.

**Sequence diagram** is a type of interaction diagram that emphasizes time-ordering of messages within the system.

**Collaboration diagram** is a type of interaction diagram that emphasizes the structural organization of the objects that act as senders or receivers of the messages within the system.

**Statechart diagram** is a dynamic view of the system, showing a state machine of all available states and transitions within the system.

**Activity diagram** shows the dynamic flow of actions within the system, from activity to activity.

**Component diagram** shows dependency and static organizational structure of a set of components.

**Deployment diagram** shows a static configuration of run-time processing nodes and the component(s) that run on each node.

There are additional variations on the elements listed above, to accommodate a larger set of possible scenarios and system organizations that UML is suited to model.

### 3.3.2. Umple and UMLet

The Umple language [10] is a compact textual notation for UML. At the time of this writing, Umple supports textual modeling of class diagrams, and the state machine modeling is currently in the works. Umple combines the power of UML with the advantages of textual notations, allowing for automated analysis of the models. Umple compiler comes as an Eclipse or Rational Software Modeler plug-in and allows for generation of Java or PHP structures from class models expressed in Umple.

UMLet [11] is an Open Source UML visualization tool that runs stand-alone or as Eclipse plug-in. UMLet models are saved in a notation that utilizes XML and can be exported to a variety of formats.

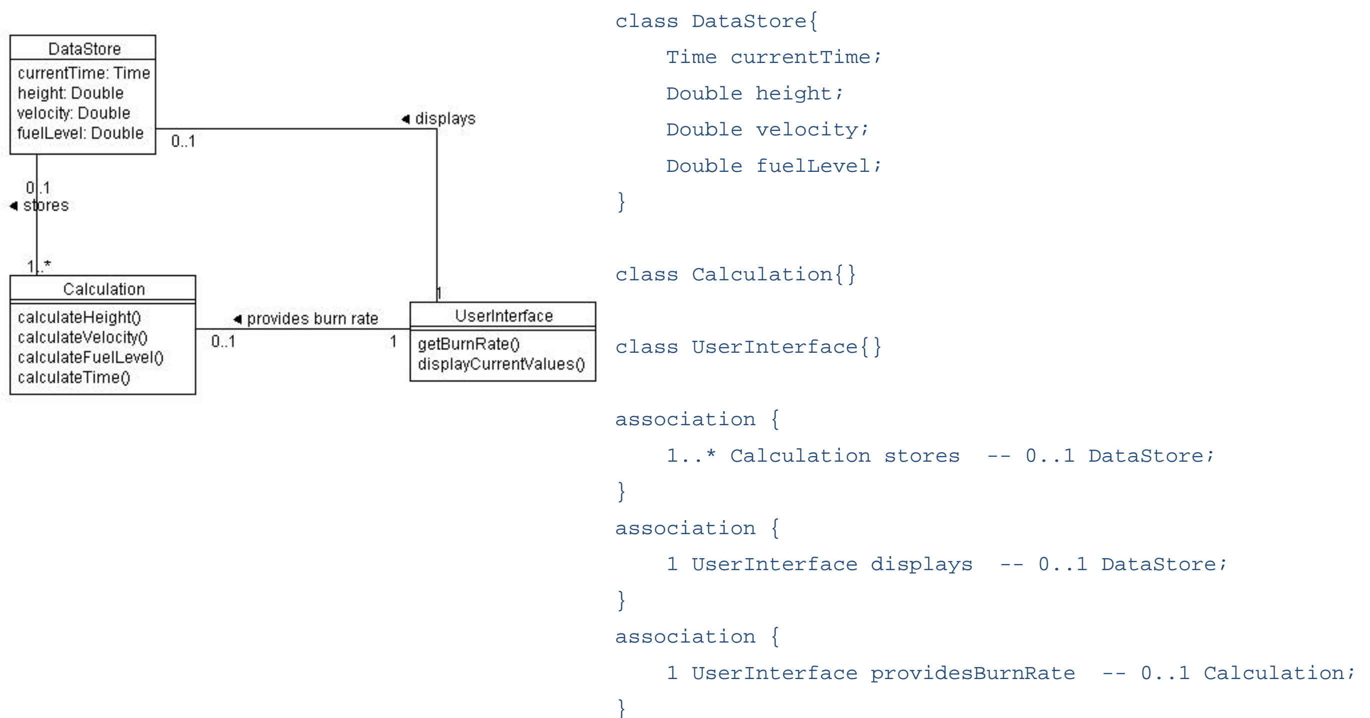


*Figure 8. Lunar Lander in UML graphical notation and Umple textual notation*

*Figure 8* contains the UML model of the Lunar Lander system described previously, both in standard graphical UML notation modelled using the UMLet tool, and in textual Umple notation. Please note that Umple notation models classes, their attributes, and the relationships between classes. Class method specifications are left to be checked by a native language compiler.

## 3.4. Early Architecture Description Languages (ADLs)

There is no single definition of what constitutes an architecture description language. The uncertainty regarding this issue hinges on disagreement on what should and should not be included in the concept of a software architecture. Looking at architecture as a set of principal design decisions about a system, there are several languages that fall into the category of ADLs. The early ADLs did not go past the research project stage and are not used in practice. However, they are an important part of architectural modeling development.

A brief feature overview is provided below for three early ADLs: Darwin (3.4.1), Rapide (3.4.2), and Wright (3.4.3). Each notation is illustrated by an example using the Lunar Lander application described in [1].

### 3.4.1. Darwin

Darwin [12] is a declarative language that describes systems as collections of components interacting via explicit interfaces. The notion of a software connector is not present, but a component that facilitates interactions can be considered a connector. Darwin has a notion of portals, which expose architectural interfaces. Portals can have four possible directions: provide, require, export, and import. *Provide* declarations are provided to encapsulating comments by the defining component; whereas *require* declarations define portals that are being provided by external or encapsulating components. *Export* portals are similar to *provide* ones, with the difference that they are provided to an external trader or nameserver. Exported portals can be bound by *import* declarations, which are similar to *require* declarations. *Import* declarations can be used to bind portals to Darwin portals and non-Darwin programs. {{488 Department of Computing, Imperial College of Science, Technology and Medicine 1997}} Bindings between interfaces define the configuration. Darwin provides a separation of concerns by separating the descriptions of structure, computation, and interaction. [13]

Darwin includes dynamic binding, where a new anonymous instance of a component is created each time the component is accessed by a bound portal. The parameters are supplied by the invoking portal. Method of invocation is implementation. Darwin also includes tagging – ability to include non-structural information into the specification, and assert declarations that assist in component integrity checks. {{488 Department of Computing, Imperial College of Science, Technology and Medicine 1997}}

Darwin is a textual notation with equivalent less precise graphical visualization. It supports hierarchical composition, where components can have internal structure that itself might include components, services and bindings. These can be used to provide deployment viewpoints. Darwin supports static viewpoints as well as dynamic ones (through dynamic instantiation and binding). There is no support for modeling non-functional aspects.

The Darwin language is not domain-specific. Darwin models can be expressed in pi-calculus, allowing for automated processing. It is most appropriate for modeling structural viewpoints. See *Figure 9* for an example of the Lunar Lander application modeled in Darwin [1].

```
component DataStore{
  provide landerValues;
}

component Calculation{
  require landerValues;
  provide calculationService;
}

component UserInterface{
  require claculationService;
  require landerValues;
}

component LunarLander{
  inst
    U: UserInterface;
    C: Calculation;
    D: DataStore;
  bind
    C.landerValues -- D.landerValues;
    U.landerValues -- D.landerValues;
    U.calcualtionService -- C.calculationService;
}
```

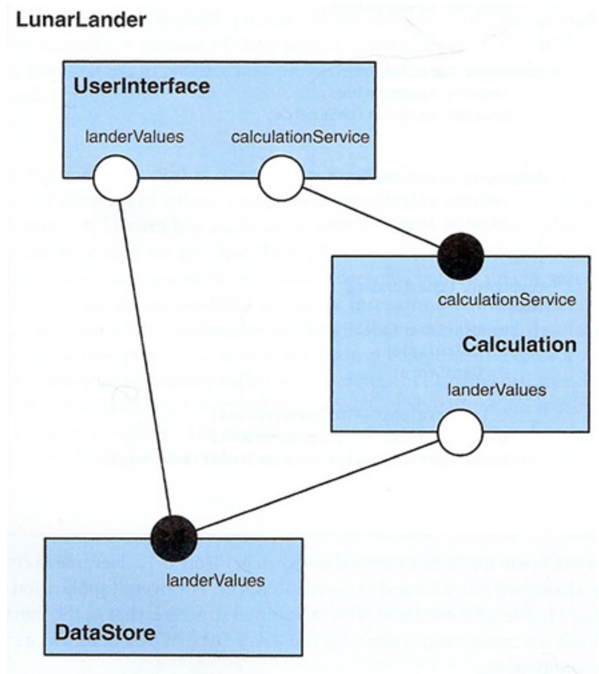


*Figure 9. Lunar Lander in Darwin Notation (adapted from [1])*

### 3.4.2. Rapide

Rapide [14] is a language that focuses on modeling systems that consist of components communicating via events. Procedure calls are supported by using two events: one for the call and one for the return value. Events in Rapide are combined into partially-ordered sets. Components in the system work concurrently. Rapide's textual description is similar to Darwin's, except it also includes behavioural specifications.

Events can be subject to causal relationships. Such relationships exist if:

- both events are generated by the same process;
- process triggered by one event generates the other;
- process triggered by one event assigns a value to a variable and the process triggered by the other event reads it;
- one event triggers a connection that generates the other event;
- or events are related transitively (event A precedes C which precedes B).

Rapide provides a single viewpoint and focuses on modeling dynamic aspects of the architecture. The static aspects are captured by interconnections among components. Rapide models can be executed to simulate the system's operation. The result of execution is a graph showing causal and temporal relationships between events

present in the system. Consistency between multiple models of the same architecture can be checked via manual inspection of simulator outputs.

```
type DataStore is interface
  action  in  SetValues();
          out NotifyNewValues();
  behavior
  begin
          SetValues => NotifyNewValues();;
end DataStore;

type Calculation is interface
  action  in  SetBurnRate();
          out DoSetValues();
  behavior
          action CalcNewState();
  begin
          SetBurnRate => CalcNewState(); DoSetValues();;
end Calculation;

type Player is interface
  action  out DoSetBurnRate();
          in  NotifyNewValues();
  behavior
          TurnsRemaining : var integer := 1;
          action UpdateStatusDisplay();
          action Done();
  begin
          (start or UpdateStatusDisplay) where \
            ($TurnsRemaining > 0) => \
              if(turnsRemaining > 0) then \
                TurnsRemaining := $TurnsRemaining - 1; \
                DoSetBurnRate(); \
              end if;;
          NotifyNewValues => UpdateStatusDisplay();;
          UpdateStatusDisplay where $TurnsRemaining == 0 \
            => Done();;
endUserInterface;

architecture lander() is
  P1, P2 : Player;
  C: Calculation;
  D: DataStore;
connect
  P1.DoSetBurnRate to C.SetBurnRate;
  P2.DoSetBurnRate to C.SetBurnRate;
  C.DoSetValues to D.SetValues;
  D.NotifyNewValues to P1.NotifyNewValues();
  D.NotifyNewValues to P2.NotifyNewValues();
end LunarLander;
```

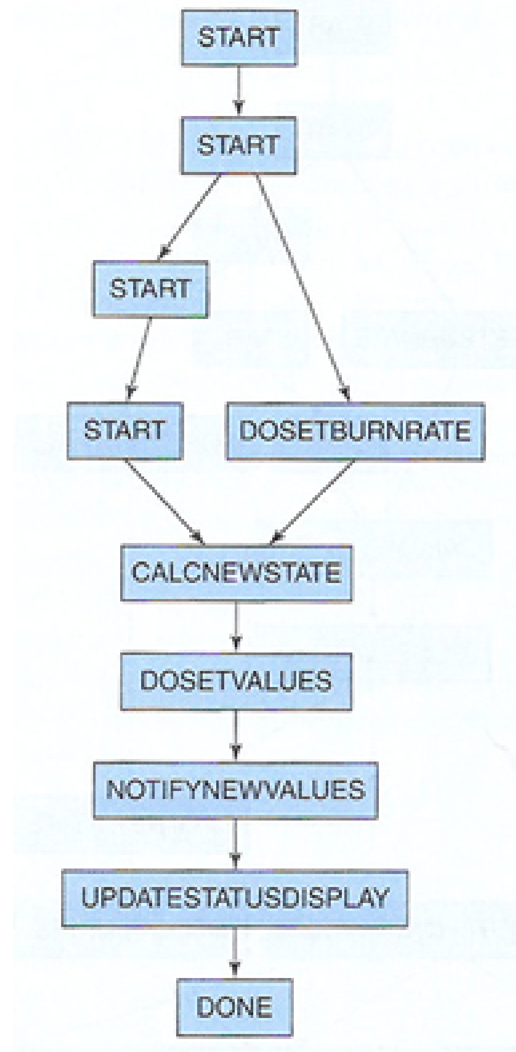


*Figure 10. Two-player Lunar Lander in Rapide notation and one-player Lunar Lander Rapide event graph (adapted from [1])*

Because Rapide is optimized for modeling multi-threaded applications, *Figure 10* contains an example of a slightly modified Lunar Lander application model that includes a game component for 2 players [1]. Using Rapide's simulation capabilities, a one-player event graph shown below can be generated.

Rapide's visualization utilizes a simulator that takes architecture models in Rapide notation as input and simulates the interaction of the various components defined in the model. Repeated simulations can generate different event streams, depending on the simulator's scheduler's allocation of time for the various components. [1]

### 3.4.3. Wright

Wright [15] language deals with checking component interactions through their interfaces. The models include components, connectors, ports, roles (interfaces), attachments, and styles. Wright interfaces are based on notation derived from CSP - Communicating Sequential Process - and can be translated into CSP [16]. The resulting model can be analysed in an automated way to determine compatibility of the interfaces and to identify deadlocks.

Wright supports static structural models with behavioural annotations describing component interactions. There is no support for non-functional aspects of the architecture. Structural, behavioural, and style viewpoints are supported. Automated correctness, consistency, and deadlock checks can be performed using a CSP evaluator.

*Figure 11* shows an example of the Lunar Lander application modelled in Wright [1]. The notation is complicated and takes a long time to learn, even to model a simple system. It is a good notation for modeling safety-critical systems where freedom from deadlock is important. The analysis capabilities are powerful but limited to a small set of properties. There is no support for generating an implementation from the model.

```
Component DataStore
  Port getValues (behavior specification)
  Port storeValues (behavior specification)
  Computation (behavior specification)

Component Calculation
  Port getValues (behavior specification)
  Port storeValues (behavior specification)
  Port calculate (behavior specification)
  Computation (behavior specification)

Component UserInterface
  Port getValues (behavior specification)
  Port calculate (behavior specification)
  Computation (behavior specification)

Connector Call
  Role Caller = call → return → Caller[] §
  Role Callee = call → return → Callee[] §
    Caller.call → Callee.call → Glue
  Glue = []Callee.return → Caller.return → Glue
    [] §

Configuration LunarLander
  Instances
    DS : DataStore
    C : Calcualtion
    UI : UserInterface
    CtoUIgetValues, CtoUIstoreValues, UItoC, UItoDS : Call

  Attachments
    C.getValues as CtoUIgetValues.Caller
    DS.getValues as CtoUIgetValues.Callee

    C.storeValues as CtoUIstoreValues.Caller
    DS.storeValues as CtoUIstoreValues.Callee

    UI.calculate as UItoC.Caller
    C.calculate as UItoC.Callee

    UI.getValues as UItoDS.Caller
    DS.getValues as UItoDS.Callee
End LunarLander
```

*Figure 11. Lunar Lander in Wright notation with some behavioural specifications omitted (adapted from [1])*

### 3.5. Domain-specific and style-specific ADLs

Some architectural languages are created to support a particular architectural style or to model applications within a particular domain. Below three such ADLs are presented: Koala notation for modeling products in Philips TV product line (3.5.1), Weaves notation for a modified pipe-and-filter architectural style (3.5.2), and AADL notation used for embedded and real-time system modeling (3.5.3).

### 3.5.1 Koala

Koala [17] is a language developed by Philips to describe the architecture of consumer electronic devices. Koala models include components that communicate via explicit provided and required interfaces. *Figure 12* shows an example of a TV platform expressed in the Koala notation [17].

Koala combines the concept of portals found in Darwin with constructs that support product-line architectures. Thus multiple products can be described with a single model representing a structural viewpoint where differences are encoded as variation points.

There is no support for modeling the dynamic or non-functional aspects of the system. The variation points are described statically, but the selection can be changed at runtime. Since Koala models are closely tied to implementation, the models can be verified for correctness and completeness through analysis of compilation results.

Interface definition:

```
interface ITuner{
  void SetFrequency(int f);
  int GetFrequency(void);
}
```

Component description:

```
component CTunerDriver{
  provides ITuner ptun;
  IInit pini;
  requires II2c ri2c;
}
```

Compound component example:

```
component CTvPlatform{
  provides IProgram pprg;
  requires II2c slow, fast;
  contains
    component CFrontEnd cfre;
    component CTunerDriver ctun;
  connects
    pprg = cfre.pprg;
    cfre.rtun = ctun.ptun;
    ctun.ri2c = fast;
}
```

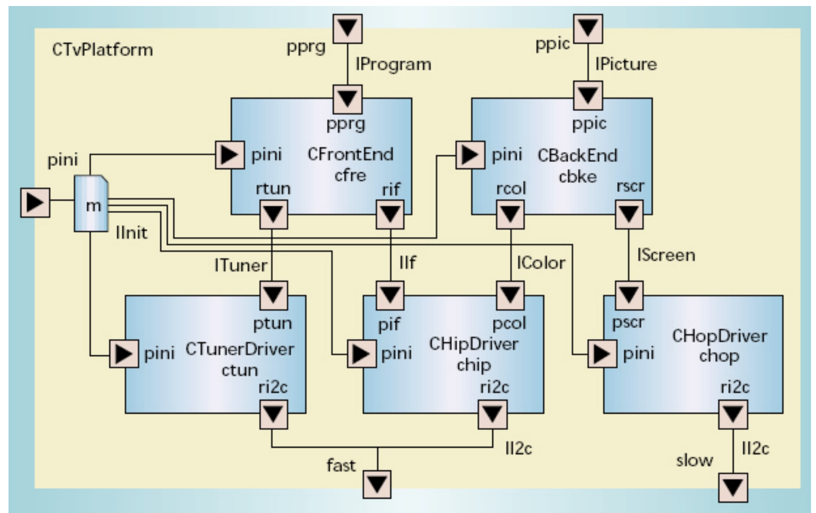


*Figure 12. TV platform in Koala notation (adapted from [17])*

A software system is modeled and implemented as a collection of interacting components, each one with a well-defined connector interfaces (those it provides and those it requires), forming a building block of the application. A given assembly of components can be treated as a compound component that can be used as a single unit.

Variability in Koala is managed by:

- diversity interfaces - mechanisms for parameterizing a component;
- switches - connecting elements that allow a single component to interact with one of a set of components, depending on the value of a given run-time parameter;
- and optional interfaces: component provides or requires particular functionality that may be specialized for certain, but not all, products within a family.

Combinations of components from different families in novel ways comprise product populations.

In addition to allowing for reuse and rapid development, Koala is the concrete manifestation of the company's corporate experience, knowledge and competitive advantage [1].

### 3.5.2 Weaves

Weaves [18] is an architectural style supported by a graphical modeling notation. Weaves can be seen as a variation of pipe-filter systems with three differences:

1. Instead of byte streams, Weaves deals with object streams.
2. Instead of implicit pipes, Weaves connectors are object queues of explicit size.
3. Instead of a single input and a single output, Weaves tools allows for multiple inputs and outputs. [1]

Models allow for components, connectors (queues) and directed interconnections. *Figure 13* shows an example of the Lunar Lander application expressed in Weaves notation [1]. There is no support for dynamic aspect modeling. The architectural style imposes certain non-functional aspects on the system, but these are not explicitly defined. Weaves models provide information about the connections but not about how those connections are used. Natural languages or other notations can be used to elaborate on the protocols used in the connections, objects that are exchanged, etc.

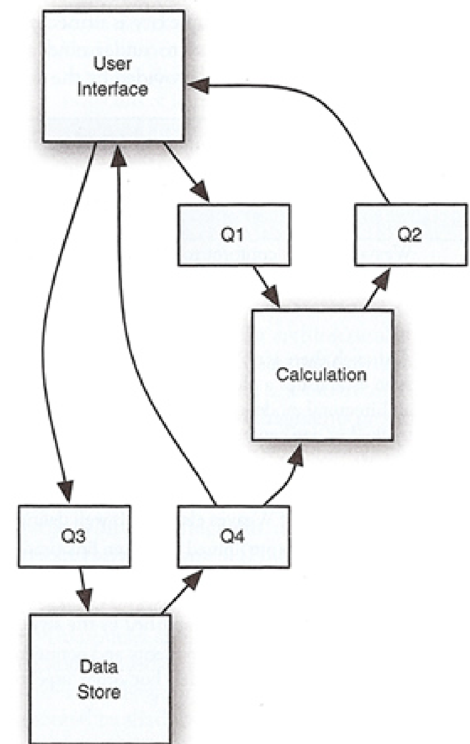


*Figure 13. Lunar Lander in Weaves notation (adapted from [1])*

It is easy to identify errors in models, as they are limited to broken links and interconnection problems. The implementations are closely tied to the model by the architectural style, so it should be possible to check their correctness. [1]

### 3.5.3 Architectural Analysis and Design Language (AADL)

AADL [19] was initially created to model avionics systems. The notation is not specifically bound to that domain, however, and can be used in embedded and real-time system modeling.

AADL is a textual language that can be expressed graphically and is accompanied by a UML profile to capture the information in several ways. AADL includes constructs that describe both software and hardware components, as well as mapping between them. The components available include those for networks, buses, ports, threads, processes, and a large variety of others. AADL allows for specification of interfaces for flow of control and data. There exists limited support for dynamic aspect modeling. There is also support for some non-functional properties, such as timing, reliability, and safety, although they cannot be automatically analysed. The models include a high level of detail.

The basic building block of this notation is a component, defined by its category (hardware, software, or composite), type (how the component interacts with the outside world) and its implementation (an instance of the component type). There can be many instances for one component type. The category of a component defines what properties the component might have. *Figure 14* shows a partial Lunar Lander model in AADL [1].

Automated consistency checks are possible. There are tools that support editing and import/export of AADL models, as well as tools that allow one to analyse different aspects of the model - correctness of the connections, component resource usage being within limits, etc.

AADL is a complicated notation to learn due to extensive capabilities and types of analyses that need to be performed once the model is complete. The specifications are complex even for simple systems. Within embedded real-time system domain the high costs might be justified during the safety-critical and expensive nature of the systems. [1]

```
data lander_state data
end lander_state data;

bus lan_bus_type
end lan_bus_type;

bus implementation lan_bus_type.ethernet
properties
  Transmission_Time => 1 ms .. 5 ms;
  Allowed_Message_Size => 1 b .. 1 kb;
end lan_bus_type.ethernet;

system calculation_type
features
  network         : requires bus access lan_bus_calculation_to_datastore;
  request_get     : out event port;
  response_get    : in event data port lander_state_data;
  request_store   : out event port lander_state_data;
  response_store  : in event port;
end calculation_type;

system implementation calculation_type.calculation
subcomponents
  the_calculation_processor : processor calculation_processor_type;
  the_calculation_process : process calculation_process_type.one_thread;
connections
  bus access network -> the_calculation_processor.network;
  event data port response_get -> the_calculation_process.response_get;
  event port the_calculation_process.request_get -> request_get;
  event data port response_store -> the_calculation_process.response_store;
properties
  Actual_Processor_Binding => reference
    the_calculation_processor applies to the_calculation_process;
end calculation type.calculation;
```

*Figure 14. Partial model of Lunar Lander in AADL Notation (adapted from [1])*

### 3.6. Extensible ADLs

Modeling involves a trade-off between being able to describe a variety of systems, and taking advantage of semantically powerful features of more specialized languages, such as being able to automate model analysis and code generation. One way to resolve this is to use a combination of several languages to describe the system. This entails keeping models in sync, which requires knowledge of multiple notations. This approach might also result in inability to describe certain concerns in any of the notations.

Another way to deal with the trade-off is to use extensible ADLs that strive to combine the flexibility of generic languages with precision and analyzability of semantically rich languages. Extensible ADLs normally support the common architectural constructs (such as components and connectors) and allow the user to extend the syntax to support user-defined constructs [1]. Below three extensible ADLs are presented: Acme (3.6.1), ADML (3.6.2), and xADL (3.6.3).

### 3.6.1 Acme

Acme [20] provides a set of seven constructs: connectors, components, ports (interfaces on components), roles (interfaces on connectors), attachments (links), systems (configurations), and representations (inner architectures for components and connectors).

To allow for extensibility, properties are used. Properties are decorations (expressed as optionally typed name-value pairs), to the seven language constructs described above. Simple property types (integers, Booleans, etc.) are defined in Acme, in addition to allowed custom type definitions. Acme does not parse these values, leaving that up to the tools that process the model.

Acme allows describing static aspects of the system natively, as well as dynamic and non-functional aspects through the use of properties. External tools are required to verify the correctness and types of defined properties, as well as the consistency of the model views.

Acme is verbose, firstly due to its domain-neutrality, and secondly due to it being distributed together with AcmeStudio that allows for graphical manipulation of the model and thus takes the burden of dealing with the verbose textual notation off the user. *Figure 15* shows an example of Lunar Lander in Acme notation [1].

Acme's flexibility comes from the use of properties to describe additional information about the system. However, it would be even more useful to allow for the creation of custom first-class constructs, such as connectors and components [1].

```
// Global Types
Property Type returnsValueType = bool;

Connector Type CallType = {
  Roles { callerRole; calleeRole; };
  Property returnsValue : returnsValueType;
};

System LunarLander = {
  // Components
  Component DataStore = {
    Ports { getValues; storeValues; }
  };
  Component Calculation = {
    Ports { calculate; getValues; storeValues; }
  };
  Component UserInterface = {
    Ports { getValues; calculate; }
  };

  // Connectors
  Connector UserInterfaceToCalculation : callType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
  };
  Connector UserInterfaceToDataStore : callType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
  };
  Connector CalculationToDataStoreS : callType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = false;
  };
  Connector CalculationToDataStoreG : callType {
    Roles { callerRole; calleeRole; };
    Property returnsValue : returnsValueType = true;
  };

  Attachments {
    UserInterface.getValues to UserInterfaceToDataStore.callerRole;
    UserInterfaceToDataStore.calleeRole to DataStore.getValues;

    UserInterface.getValues to UserInterfaceToDataStore.callerRole;
    UserInterfaceToDataStore.caleeRole to DataStore.getValues;

    UserInterface.calculate to UserInterfaceToCalculation.callerRole;
    UserInterfaceToCalculation.calleeRole to Calculation.calculate;

    Calculation.storeValues to CalculationToDataStoreS.callerRole;
    CalculationToDataStoreS.calleeRole to DataStore.storeValues;

    Calculation.getValues to CalculationToDataStoreG.callerRole;
    CalculationToDataStoreG.calleeRole to DataStore.getValues;
  };
};
```

*Figure 15. Lunar Lander in Acme notation (adapted from [1])*

Initially, Acme was intended as an interchange language - a common model format to which other formats would be translated for the purpose of transfer among different tools. Garlan et al[20] have used it to translate a model between Wright and Rapide notations in the manner illustrated in *Figure 16* [20]. This has never taken off, however, primarily due to substantial semantic differences between ADLs.

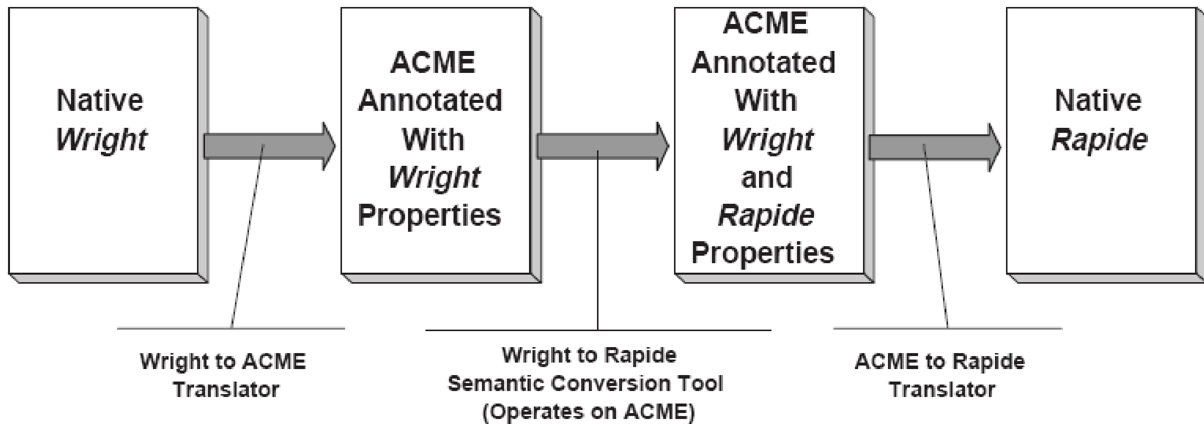


*Figure 16. Wright to Rapide transformation via Acme (adapted from [20])*

### 3.6.2 Architectural Description Markup Language (ADML)

ADML [21] is a language based on XML with its syntax derived from Acme. Semantically ADML is nearly identical to Acme, with the exception of meta-properties, which allow for specification of the properties and property types that should be present on particular elements. ADML's syntax is defined using an XML DTD. *Figure 17* contrasts Acme and ADML modeling notations [1].

```
Extended definition of the DataStore component in Acme:

Property Type StoreType = enum { file, relationalDatabase, objectDatabase };

Component DataStore = {
  Ports {
    getValues; storeValues;
  };

  Property storeType : StoreType = relationalDatabase;
  Property tableName : String = "LanderTable";
  Property numReplicas: int = 0;
};


Description of DataStore component in ADML:

<Component ID="datastore" name="Data Store">
  <ComponentDescription>
    <ComponentBody>
      <Port ID="getValues" name="getValues" />
      <Port ID="storeValues" name="storeValues" />
    </ComponentBody>
  </ComponentDescription>
</Component>
```

*Figure 17. Acme and ADML descriptions of DataStore component of Lunar Lander (adapted from [1])*

Advantages of using XML include the wide variety of tools available for XML document parsing, processing, and visualization, as well as the fact that XML is well-suited to describe hierarchical concepts. ADML does not use XML's extensibility mechanisms - instead extensibility is provided by encoding of the Acme name-value properties into XML. [1]

### 3.6.3 Extensible XML-based ADL (xADL)

xADL [22] is an XML-based domain-neutral extensible ADL that fully leverages XML's extensibility mechanisms to extend the language. The syntax of xADL is defined by XML Schemas. xADL models are well-formed XML documents. Extensibility is achieved through defining derived data types in separate schemas, thus allowing data types declared in one schema to be extended in a different schema. *Figure 18* shows an example architecture for a television in xADL notation [22].

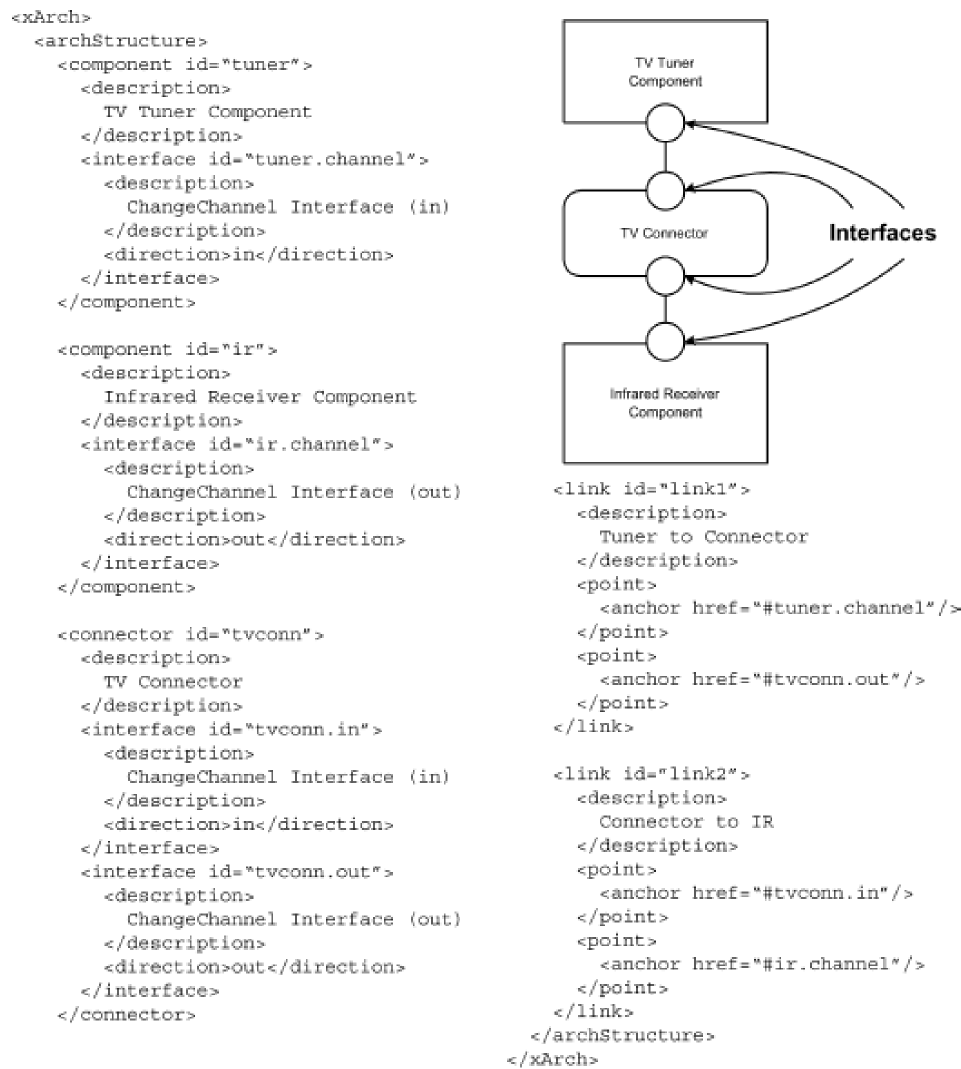


*Figure 18. Diagram of an example architecture for a television accompanied by its xADL 2.0 design time structural description (adapted from [22])*

Syntactically, xADL syntax is defined as the composition of all xADL schemas. This allows for incremental adoption (users can use only what they need), divergent extension (language can be extended in novel or contradictory ways), and feature reuse (schemas can be shared among products). Due to the varying nature of the language, xADL is provided with a set of tools that assist in implementation of parsers, editors, and other tools for a custom xADL-based language. The tools thus created can be used for consistency and correctness checking of the models. [1]

To make the models more readable, a more compact notation exists for xADL, called xADLite, which expresses the models in a less-verbose form. Visualization of xADLite is textual, which allows xADL 2.0 models to be captured in a compact format and can be manipulated using standard text editors.

## 3.7. User Requirements Notation (URN)

"The purpose of URN is to support, in a semi-formal and lightweight way, the modeling and analysis of user requirements in the form of goals and scenarios." [23] User Requirements Notation (URN) consists of two complementary notations: Use Case Map (UCM) notation and Goal-oriented Requirement Language (GRL).

### 3.9.1. Use Case Maps (UCM)

Use case maps focus on representing the behaviour of the elements within a system, in a scenario-based manner. Each scenario in the system can be followed by tracing a path within the system's use case map. Multiple scenarios can be traced concurrently, enabling modeling of concurrent systems. [24]

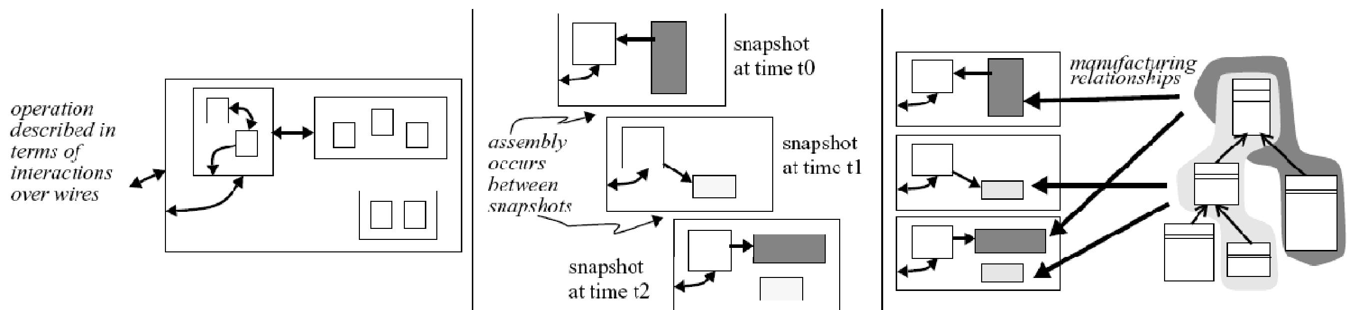


*Figure 19. Operation, Assembly, and Manufacturing complexity factors (adapted from [24])*

Component context diagrams used for a similar purpose consist of components connected by conceptual or actual connections (wires). Buhr [24] calls such diagrams "wirings". He argues that in large complex systems, models should assist with specifying guidelines for creating wirings for the system, instead of dealing with details of individuals wirings. Complexity factors associated with wirings (shown on *Figure 19* [24]) are:

1. **Operation** of a system expressed in terms of interactions over wires between components.
2. **Assembly** of components in different ways during the execution of the system.

3. **Manufacturing** of objects from inheritance relationships in class hierarchies.

These complexity factors contribute to lack of scalability of wiring models, which UCM aims to address. [24]

The complexity factors described above are simplified by UCM in the following manner:

1. **Operation**. Behaviour is represented as cause-effect sequences between components' responsibilities by using paths instead of wires. Responsibilities represent a higher level of interaction than calls, messages, and RPC, therefore reducing need for granularity.
2. **Assembly**. Changing components are modeled using slots that exist independently of their occupancy or occupants. Each slot has responsibilities defined for it. Wires are eliminated from the diagrams.
3. **Manufacturing**. Use case maps become first-class models and may be developed independently of class descriptions, thus eliminating dependence on manufacturing details.

**UCM notation** includes the following elements (illustrated in *Figure 20* [24]):

1. **Paths** trace progression of causes and effects through the system from the points where effect is triggered. Paths documentation includes names and natural language descriptions of scenarios associated with the paths, preconditions and postconditions, path segment labels, and cross-references between large-scale paths and subpaths. Paths traversing a component correspond to methods, messages, and state machines enabling the corresponding functionality.

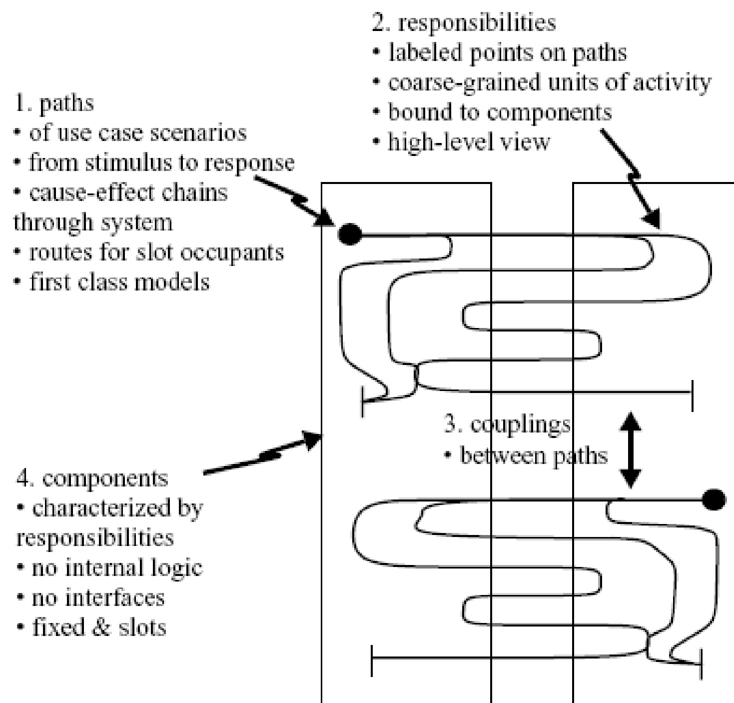


*Figure 20. UCM notation elements (adapted from [24])*

2. **Responsibilities** link paths to components. They act as a direct link between use cases and architecture.
3. **Interpath couplings** connect paths into a larger pattern. They include cases when postcondition of one path is a precondition of another, or when paths share a physical resource.

4. **Components** perform responsibilities. UCM components include objects, teams of objects, and slots occupied by objects dynamically as scenarios run. Slots are modelled by indicating points along paths where components are created, moved, or destroyed. [25]

UCM notation focuses on a path-centric view - a higher level of abstraction compared to a component-centric view. Buhr [25] suggests that annotating UML class diagrams with run-time relationships between instances, or superimposing use case paths onto class diagrams, would be stretching the class diagram notation too far. Similarly, extending the message sequence charts by interpreting messages as causal relationships and message processing segments of timelines as causal sequences can approximate use case maps. However, overloading the existing notation leads to cluttered, complicated models, with possibility of misinterpretation.

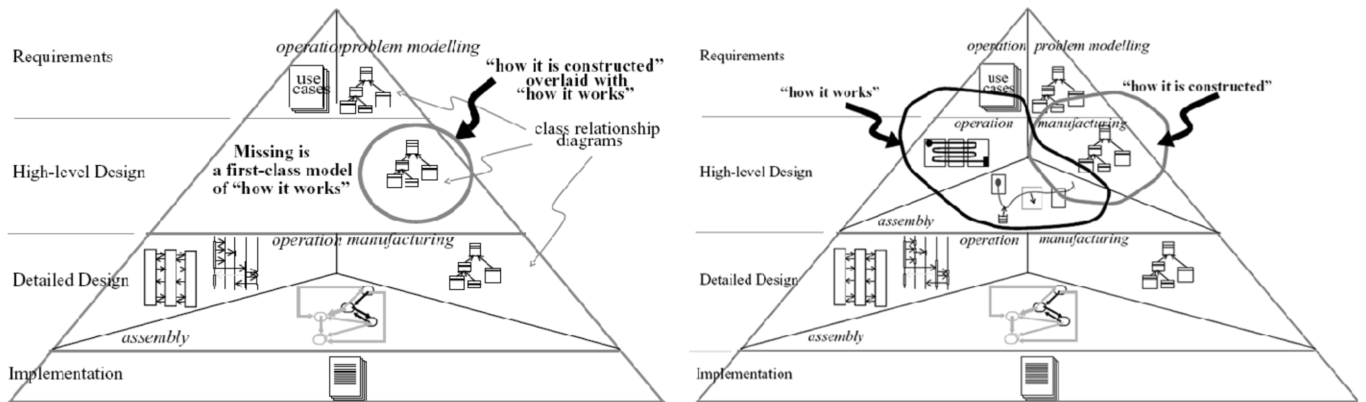


*Figure 21. UCM niche in design (adapted from [25])*

Instead, UCM aims at separating the "how it works" (use case maps) from "how it is constructed" (class relationship diagrams). *Figure 21* shows a contrast between where the gap in design process lies and how it is filled by the inclusion of use case maps [25]. The outer triangle identifies stages of design, from requirements to implementation.

There exists a trade-off between the high-level system behavioural specification expressed in UCM and the presence of finer component interaction details. Because of this omission of low-level details, UCM models are not used to generate application code. Instead, use case maps provide a system-wide view that exploits human pattern-recognition

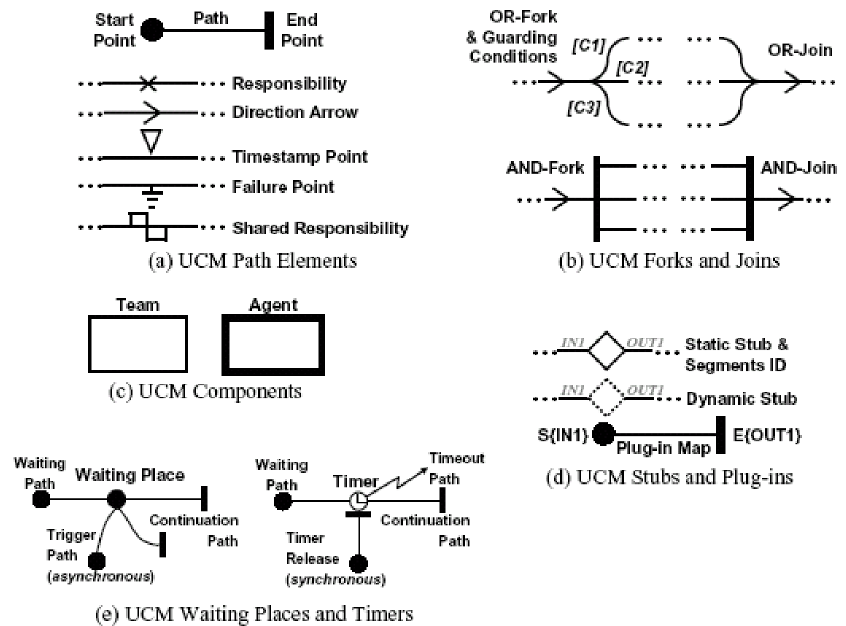


*Figure 22. Summary of the UCM notation (adapted from [23])*

abilities, which enable stakeholders to visualize, think about, discuss and explain the overall behaviour of a system [26]. *Figure 22* summarizes main elements of the UCM notation [23], and *Figure 23* shows an example of a network transaction modelled in UCM [26].

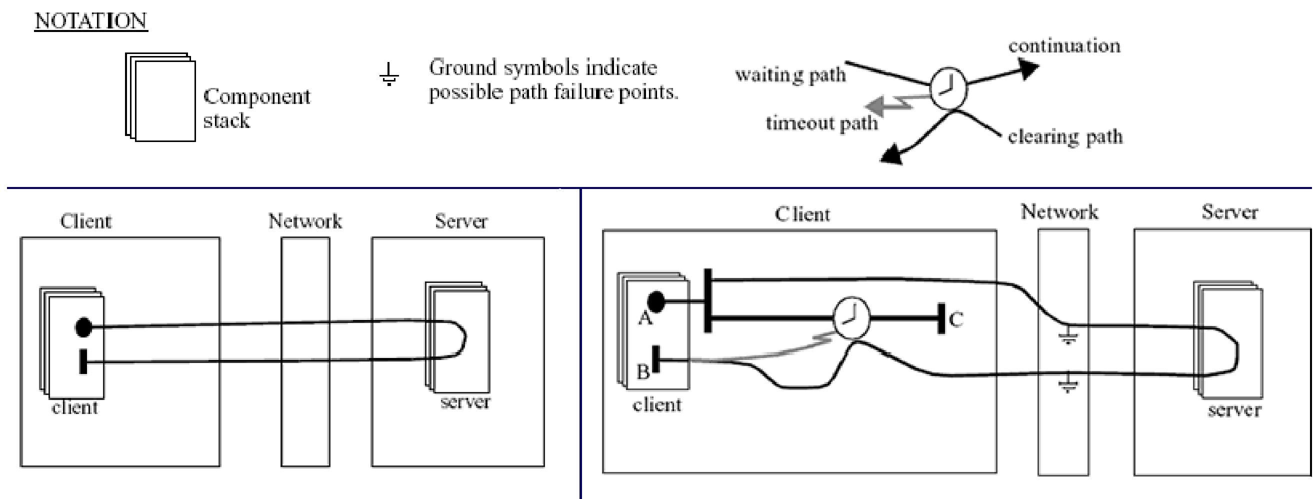


*Figure 23. Use case maps for a simple network transaction and for a generalized one (adapted from [26])*

UCM is technology-independent and domain-independent, which makes it applicable for a variety of system types, from sequential to distributed and parallel systems. Behaviour patterns of an application being modeled can be explored using this notation. [24]

### 3.9.2. Goal-oriented Requirement Language (GRL)

Goal-oriented Requirement Language (GRL) [27] supports goal-oriented modeling and reasoning about requirements, in particular the non-functional requirements. GRL elements are used to represent various concepts that arise during the requirements process. *Figure 24* summarizes main elements of the GRL notation [23].

GRL includes the following concepts:

1. **Actor**
2. **Intentional event**
    1. **Goal** - a state of affairs that stakeholders would like to achieve. This may be a business goal or a system goal.
    2. **Task** - a particular way of doing something, or a solution in the target system which will satisfy a softgoal.
    3. **Softgoal** - similar to goal, a desirable state

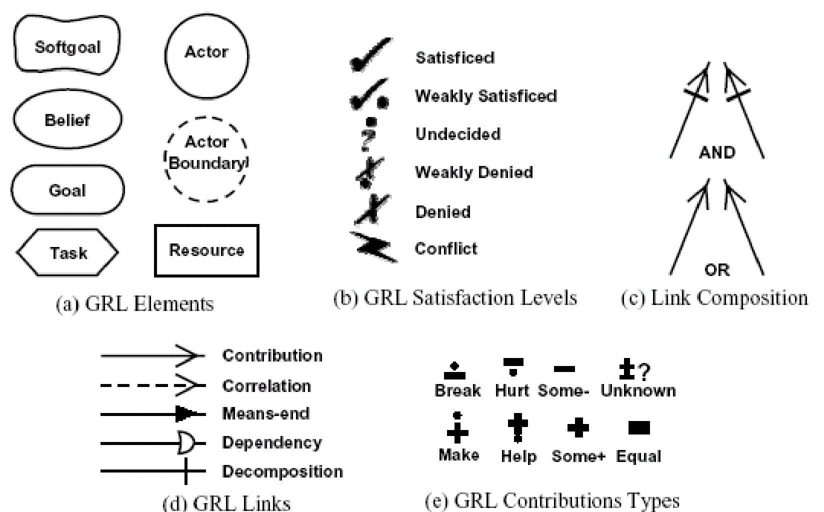


*Figure 24. Summary of the GRL (adapted from [23])*

of affairs, but without clear-cut criteria for whether the state is achieved. This is left to subjective interpretation of the stakeholders.

4. **Belief** - a statement representing design rationale, used to provide justification for decisions. Beliefs facilitate review and enhance traceability.
5. **Resource** - an entity that can either be available or not available.

3. **Intentional relationship**:
    1. **Means-ends** relationship connects the end node with the means node achieving it.
    2. **Decomposition** defines what other elements need to be available or achieved in order to complete a task.
    3. **Contribution** describes how elements contribute to each other.
    4. **Correlation** is similar to contribution, with contribution not being direct, instead being a side-effect.
    5. **Dependency** denotes an intentional relationship between two actors.

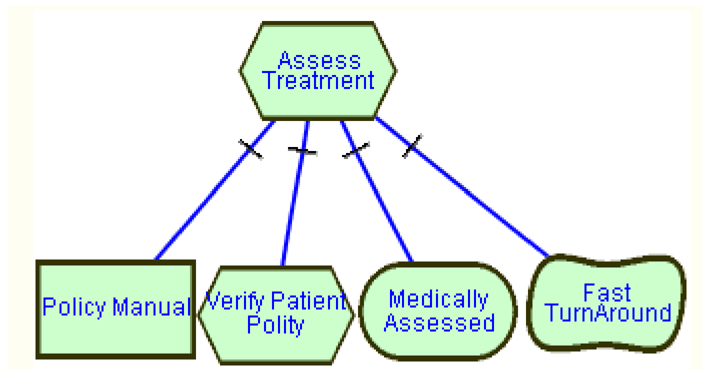


*Figure 25. GRL model of task decomposition (adapted from [27])*

*Figure 25* shows an example of a GRL model that uses decomposition structure for the "Assess Treatment" task using "Policy Manual" resource, "Verify Patient Policy" task, "Medically Assessed" goal, and "Fast TurnAround" softgoal, connected by the "AND" links [27].

GRL enables tracking why certain decisions were made, what alternatives were considered, what criteria and reasoning was used to arrive at the end decision. Operational details of the system are omitted, much like in UCM notation, and the focus is given to the "why" of behavioural and structural choices made. [27]

### 3.9.3. Integration of UCM and GRL

[28] propose to combine the use case maps and GRL models during architectural design. The GRL language can be used for goal-oriented modeling and reasoning, and the UCM notation can be used to capture the design at each stage of development. *Figure 26* illustrates how UCM and GRL work together in modeling [27].

First, GRL models are created, non-functional requirements and business goals are refined, and some concrete design decisions are obtained. These decisions are further expanded into UCM scenarios. Sometimes, it is difficult to capture highly abstract goals without going into further design detail. In such cases, UCM can be used to create scenarios for a hypothetical system, giving more insight into the finer details, and allowing setting more concrete goals. [28]

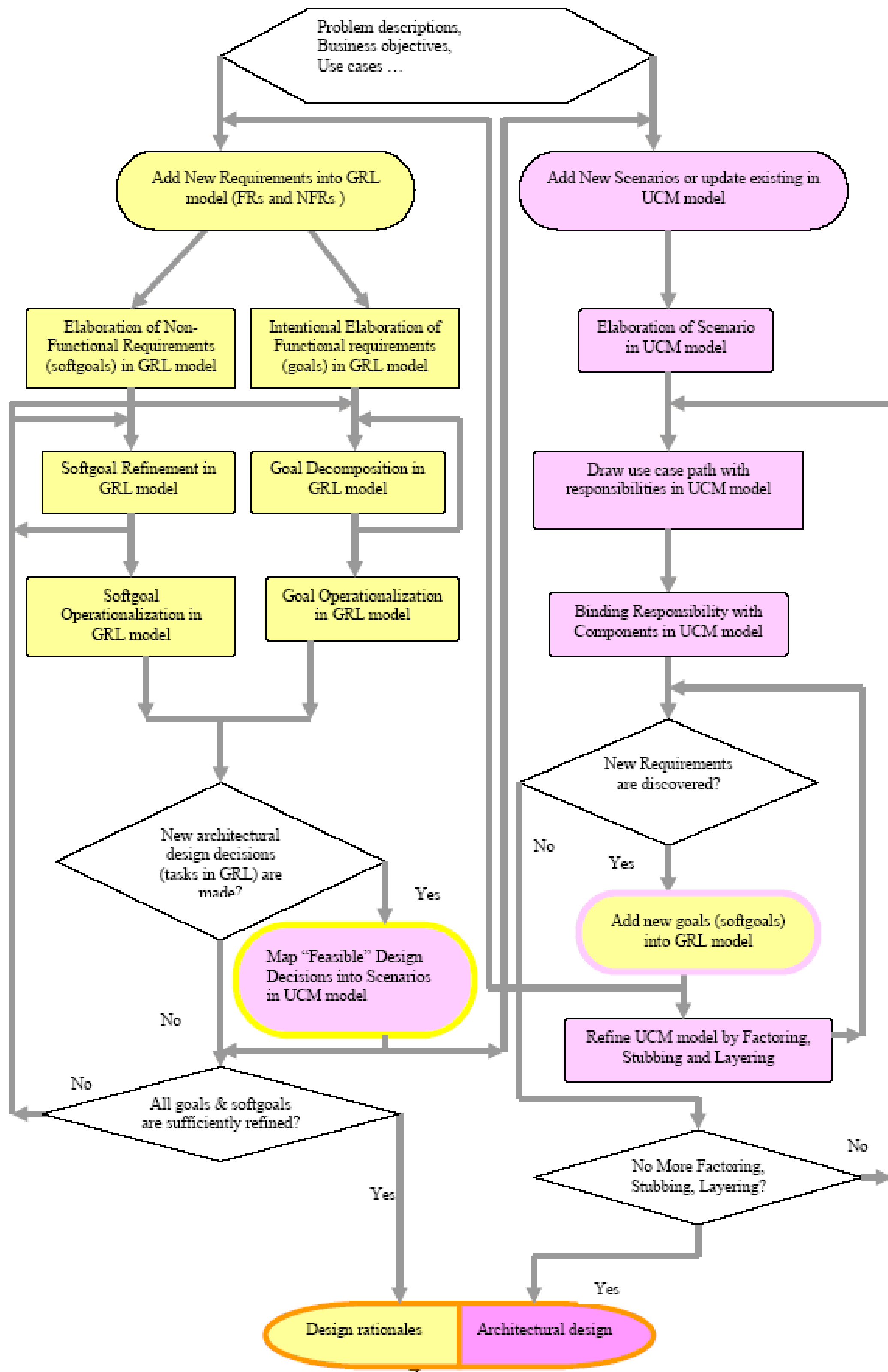


*Figure 26. Integration of goal-oriented and scenario-based modeling (adapted from [27])*

### 3.8. Dialog Flow Notation

There are a number of domain-specific notations. Multiple domains are represented: from a particular product line (such as Koala notation for TV platform) to a particular interaction (such as Rapide for event-based systems). Dialog Flow Notation (DFN) is an example of another domain-specific notation, created by [29] for describing hypertext systems from interaction viewpoint.

In DFN, a dialog step is a step that happens between the submission of a client request to the server and the client's receipt of the server response. A dialog sequence is comprised from multiple consecutive dialog steps. A dialog flow encompasses all possible dialog sequences for a given presentation channel of a given application.

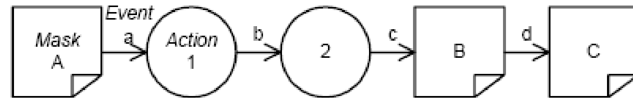


*Figure 27. Dialog graph (adapted from [30])*

A particular dialog flow is modeled as a directed state-transition graph called dialog graph. States correspond to dialog elements, and transitions - to events. Dialog elements can send and receive multiple events. The notation is similar to the UML state diagram notation. An example of a dialog graph [30] is shown in *Figure 27.*

Dialog elements can be either atomic, or modular. Atomic dialog elements are masks (hypertext pages) and application logic operations (actions). Modular elements (dialog modules) are higher-level elements encapsulating dialog graphs, allowing for representation of nested dialog structures. Every module has one entry point (triggered by initial event from a super-module) and one or more exit points (triggering terminal events that are propagated to the super-module). [29] mention that DFN provides additional constructs to specify more complex dialog structures. An example of a user authorization dialog module [29] is shown in *Figure 28*.

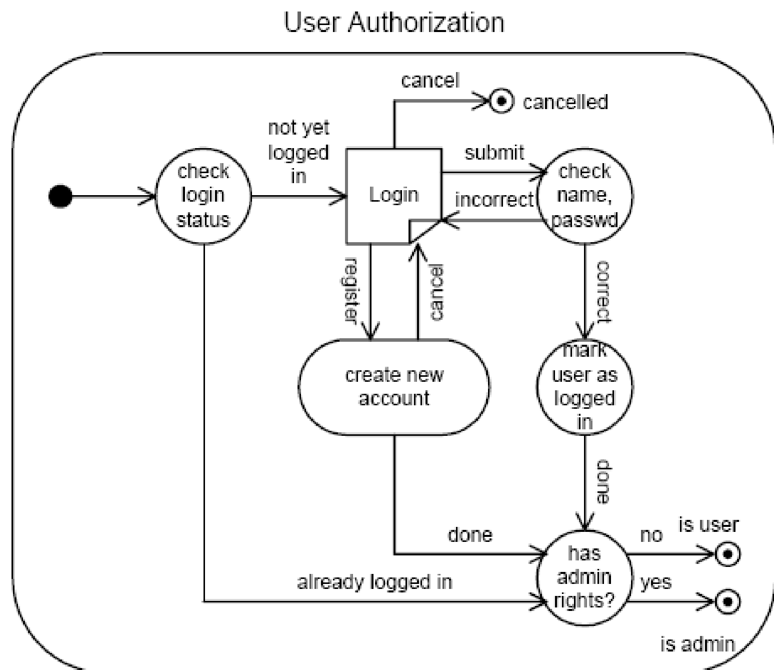


*Figure 28. Dialog module (adapted from [29])*

DFN was designed to address the following differences between page-based applications for the Web and the traditional window-based desktop applications:

1. In Web applications the amount of information presented to the user is restricted (by result set size, screen size, etc.), therefore different user interfaces and different interaction patterns are needed.
2. Actions that do not require application logic involvement in window-based systems (such as closing a window), require it in page-based systems (when an application window is closed, a new page needs to be generated and loaded).
3. Web applications interact via request-response mechanism, as opposed to being able to push data to the client, so they are passive in nature.
4. Due to HTTP being stateless, the application itself has to manage the state of the dialog system.

DFN omits information about server-side code generation, user interface widgets, or client-side scripting, as those may be related to a specific presentation channel, whereas the notation is not channel-specific. To specify channel-specific dialog flows, labels of the module diagrams are appended with a channel name. *Figure 29* shows examples of HTML and WML presentation channels for the same module [29].

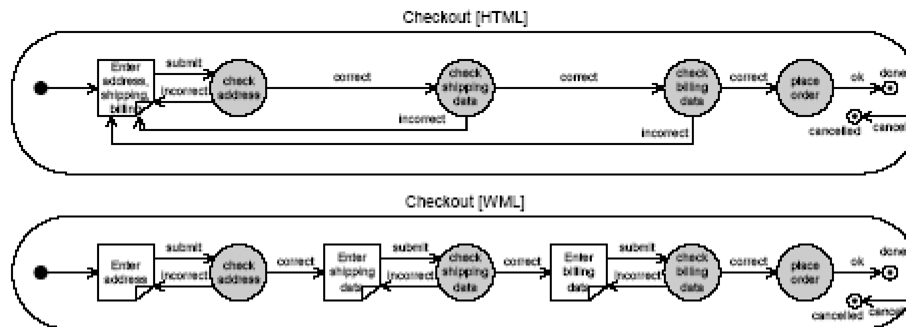


*Figure 29. Separate HTML and WML presentation channel for the same module (adapted from [29])*

DFN diagrams can be transformed into Dialog Flow Specification Language (DFSL). This transformation allows DFN models to serve as input to the dialog control implementation of applications. DFSL is an XML-based language whose elements are based on DFN elements. Dialog Control Framework (DCF) is a reusable framework based on the Model-View-Controller pattern that reads DFSL documents and facilitates development of flow control for multiple presentation channels. *Figure 30* shows architecture of the DCF [29].

DCF provides flexibility, reusability, and maintainability due to strict separation of user interface, application logic, dialog flow specification, and dialog control logic. It allows for building applications targeted for different presentation channels while reusing the application logic. Developers have to provide classes that implement the actions, pages that implement dialog masks, DFSL documents, and channel servlets (if required) for additional presentation channels (HTML and WML channels are provided). [29]

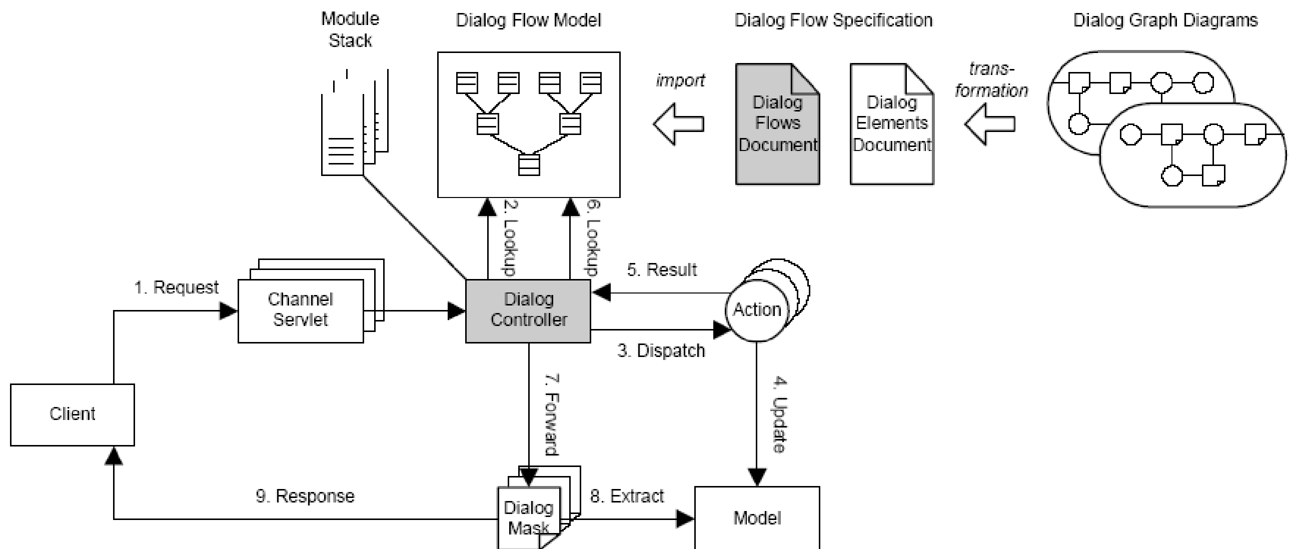


*Figure 30. Dialog Control Framework architecture (adapted from [29])*

One of the weaknesses of DSN is its granularity, which results in higher specification, implementation, and performance overhead. Another issue that was not addressed during DFN creation is ability to revisit previous masks without triggering actions, such as happens on pressing a browser's "back" button. [29]

## 4. Discussion

The many modeling notations presented above are not mutually exclusive, and one of them is not clearly superior to the others. Depending on the domain and architectural style of a system being modelled, there might be several appropriate notations from which to choose. In some cases, the choice is dictated by factors not based on the merits of a modeling notation itself. These can include such external drivers as corporate policies or culture, established processes and tools used within the organization, existing familiarity with a particular notation, whether a notation has been developed in-house, and so on.

### 4.1. Choosing a notation

Some notations are clearly outdated and not used since they have been explored in research literature. Darwin, Wright, and Rapide are examples of such notations. Others are strongly tied to a particular product line (such as Koala), a particular domain (such as Diagram Flow Notation for web-based applications, or AADL for critical real-time applications), or a certain architectural style (such as Weaves that deals with object streams).

Unless one finds oneself within one of the clearly-defined situations when a certain modeling notation is obviously the best (or the only) choice, it makes sense to focus on extensible and flexible notations that can adapt to

a variety of situations. Time spent learning a new notation is a significant factor, and so it would be logical to invest into a notation or a closely-related group of notations that can be used for modeling a variety of projects.

One extensible notation discussed above is xADL. This notation claims to be highly flexible, domain-neutral, XML-based and thus easy to parse. However, it appears that there are multiple steps to learn and multiple tools to adopt to be able to work effectively with this notation. In addition, the notation has come from recent research and is not widely adopted. Getting assistance, support, or tool updates might be difficult.

Another extensible and flexible notation is UML. A variety of diagram types, entities, and relationships, as well as ability to extend the diagrams with annotations, makes it suitable for modeling varying applications in multiple domains. UML is not tied to an architectural style (although class and object diagrams have strong connection with object-oriented architectures) and thus can be used in many contexts.

Due to UML being widely adopted, the number of tools that support modeling and code generation using UML is growing. Multiple Open Source and commercial applications allow modeling using UML graphical notation. One of the strengths of UML is the ability to express it in textual notation (as Umple language demonstrates), which allows for automated processing and code generation. However, transformations between UML models and code generated from those models are not yet widely supported. The Umple compiler allows for generation of PHP or Java code from the textually-encoded class diagrams. This functionality, however, is still in development, and at the time of this writing is limited to generating the code for classes, their attributes, and associations.

In addition to UML, URN appears to be a useful notation, allowing one to work with the requirements to create goal-oriented models and use case maps to model application's behaviour.

Overall, at the time of this writing, UML seems like the most extensible and flexible notation, most widely adopted for a large variety of domains. It is supported by multiple available tools and user communities, and there are multiple books and tutorials available in print and on-line to facilitate learning of the UML notation.

## 4.2. Extending UML

**Appendix A** presents a comparison table between the notations presented above, across a variety of aspects. Natural language, Power-Point-style notation, and one-of-a-kind notations have been excluded from the table, being semantically-weak notations that cannot be automatically analyzed or generated. User Requirement Notation has been omitted as its purpose is to build on UML capabilities.

UML is unique in providing multiple viewpoints, due to a variety of diagram types. Most other notations focus on a single viewpoint or a small number of them. It is not limited by a particular domain (unlike Wright, AADL,

DFN, or Koala) or a particular architectural style (unlike Weaves). It is also extensible (using stereotypes and profiles). The only other extensible notation discussed above is xADL.

Darwin models can be expressed in pi-calculus, Wright models can be translated into Communicating Sequential Processes notation, and AADL syntax is based on the Backus-Naur Form production rules. This connection with rigorous notations allows for automated processing and analysis of the models. UML, however, is not based on any mathematical notation, which limits the automated analysis capabilities of the models. This is a trade-off for UML flexibility and extensibility.

Most notations listed in the table (except for DFN and Weaves) can be expressed textually, which makes them candidates for automated analysis. All of the languages have support for basic architectural concepts: components, connectors, interfaces, and configuration.

However, a few languages have support for modeling non-functional aspects as well as rationale for architectural decisions. UML has a limited way of modeling non-functional aspects through textual annotations and stereotypes. Acme and ADML have a way to specify component properties that describe these aspects. Even though properties cannot be automatically analyzed, they can be captured for documentation purposes and to support future architectural decisions. UML could be extended in a similar way, for instance by specifying classes with non-functional properties. Other classes that are to possess the properties could then extend the class where the properties are specified.

UML supports modeling both static (class, object, package diagrams) and dynamic (state and activity diagrams) aspects of the architecture. However, when it comes to dynamic modeling, the support is tool-dependent and limited: very few systems tie into the UML model directly allowing "running" it and observing the run-time behaviour [1]. State diagrams and activity diagrams would be good candidates for dynamic models, as scenarios can be run in accordance with connections existing within these diagrams.

Rapide simulations of running systems produce event graphs (see *Figure 10*). These graphs differ for multiple simulations of the same system, as events occur in different order. Having similar functionality for UML state diagrams would be of value, allowing for model testing through simulation.

UML allows for modeling applications in multiple domains. However, there is no direct support for real-time deadlock-affected architectures. AADL notation allows for specification of time-critical properties which then can be used to model and test a system with precision, allowing for detailed simulation. This is a useful modeling functionality for embedded real-time systems where precision is important.

To allow for support for critical system design in UML, state or activity diagrams could be extended, allowing for specification of connection properties (such as the time allowed for response or the frequency with which a certain message can be sent) and component properties (such as a possible size of a message). Such a highly-detailed model could then be run to simulate the system and discover deadlock-prone segments.

The last aspect of notation comparison is the tool support. UML is currently supported by multiple tools. However, most of them support either graphical modeling only, or both textual and graphical modeling with a weak connection (or no connection at all) between the two visualizations. Additionally, to provide possible extensions to UML described above (dynamic modeling, system simulations, non-functional aspect modeling), tools these extensions would have to be developed.

## List of Figures


Example 1. Lunar Lander in Natural Language [1] .......... 1

Figure 1. Lunar Lander in PowerPoint (adapted from [1]) .......... 6

Figure 2. One-of-a-kind notation (adapted from [1]) .......... 7

Figure 3. UML structural things .......... 9

Figure 4. UML behavioural things .......... 9

Figure 5. UML grouping things .......... 9

Figure 6. UML annotational things .......... 10

Figure 7. UML relationships .......... 10

Figure 8. Lunar Lander in UML graphical notation and Umple textual notation .......... 11

Figure 9. Lunar Lander in Darwin Notation (adapted from [1]) .......... 13

Figure 11. Lunar Lander in Wright notation with some behavioural specifications omitted (adapted from [1]) .......... 16

Figure 12. TV platform in Koala notation (adapted from [17]) .......... 17

Figure 13. Lunar Lander in Weaves notation (adapted from [1]) .......... 18

Figure 14. Partial model of Lunar Lander in AADL Notation (adapted from [1]) .......... 20

Figure 15. Lunar Lander in Acme notation (adapted from [1]) .......... 22

Figure 16. Wright to Rapide transformation via Acme (adapted from [20]) .......... 23

FIGURE 17. ACME AND ADML DESCRIPTIONS OF DATASTORE COMPONENT OF LUNAR LANDER (ADAPTED FROM [1]) ....23

FIGURE 18. DIAGRAM OF AN EXAMPLE ARCHITECTURE FOR A TELEVISION ACCOMPANIED BY ITS XADL 2.0 DESIGN TIME STRUCTURAL DESCRIPTION (ADAPTED FROM [22]) ....24

FIGURE 19. OPERATION, ASSEMBLY, AND MANUFACTURING COMPLEXITY FACTORS (ADAPTED FROM [24]) ....25

FIGURE 20. UCM NOTATION ELEMENTS (ADAPTED FROM [24]) ....26

FIGURE 21. UCM NICHE IN DESIGN (ADAPTED FROM [25]) ....27

FIGURE 22. SUMMARY OF THE UCM NOTATION (ADAPTED FROM [23]) ....27

FIGURE 24. SUMMARY OF THE GRL (ADAPTED FROM [23]) ....28

FIGURE 25. GRL MODEL OF TASK DECOMPOSITION (ADAPTED FROM [27]) ....29

FIGURE 26. INTEGRATION OF GOAL-ORIENTED AND SCENARIO-BASED MODELING (ADAPTED FROM [27]) ....30

FIGURE 27. DIALOG GRAPH (ADAPTED FROM [30]) ....31

FIGURE 28. DIALOG MODULE (ADAPTED FROM [29]) ....31

FIGURE 29. SEPARATE HTML AND WML PRESENTATION CHANNEL FOR THE SAME MODULE (ADAPTED FROM [29]) ....32

FIGURE 30. DIALOG CONTROL FRAMEWORK ARCHITECTURE (ADAPTED FROM [29]) ....33

## References

[1] Taylor, R.N., Medvidovic, N. and Dashofy, E.M."Software Architecture: Foundations, Theory, and Practice". Wiley Publishing, 2009, pp. 750.

[2] Goldman, N.M. and Balzer, R.M., "The ISI Visual Design Editor Generator". in *Proceedings of the IEEE Symposium on Visual Languages 1999, 1999,* p. 20.

[3] Ren, J. and Taylor, R. N. "Visualizing software architecture with off-the-shelf components." in *Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2003),* 2003, pp. 132-141.

[4] "Object Management Group, Unified Modeling Language (UML), version 2.1.2," accessed 2008, http://www.omg.org/technology/documents/formal/uml.htm

[5] G. Booch. "Object-oriented development". 1986. IEEE Trans.Softw.Eng. vol 12, pp. 211-221.

[6] Rumbaugh, J., Blaha, M., Premerlani,W., Eddy, F. and Lorensen,W., "Object-Oriented Modeling and Design". Prentice-Hall, Inc., 1991, pp. 500.

[7] Jacobson, I., Christerson, M., Jonsson and P., Övergaarg, G., "Object-Oriented Software Engineering: A use Case Driven Approach". Addison-Wesley, 1992, pp. 552.

[8] David Harel. "Statecharts: A visual formalism for complex systems". 1987. Sci.Comput.Program. vol 8, pp. 231-274.

[9] Booch, G., Rumbaugh, J. and Jacobson,I., "The Unified Modeling Language User Guide". Addison Wesley Longman Publishing Co., Inc., 1999, pp. 482.

[10] Forward, A., Lethbridge, T. C. and Brestovansky, D. "Improving program comprehension by enhancing program constructs: An analysis of the umple language," in 2009, pp. 311-312.

[11] Auer, M., Tschurtschenthaler, T. and Biffl, S., "A flyweight UML modeling tool for software development in heterogeneous environments". *Euromicro Conference, 2003. Proceedings. 29th,* pp. 267-272, 2003.

[12] Magee,J., Dulay, N., Eisenbach, S. and Kramer,J. "Specifying Distributed Software Architectures"pp. 137-153, 1995.

[13] Magee, J. and Kramer, J. "Dynamic structure in software architectures", *Dynamic structure in software architectures*, pp. 3-14, 1996.

[14] Luckham, D.C. and Vera. J. "An event-based architecture definition language". 1995. Software Engineering, IEEE Transactions on vol 21, pp. 717-734.

[15] Allen, R. and Garlan, D. "A formal basis for architectural connection". 1997. ACM Trans.Softw.Eng.Methodol. vol 6, pp. 213-249.

[16] Hoare C. A. R. "Communicating sequential processes". 1978. Commun ACM vol 21, pp. 666-677.

[17] Rob van Ommering, Frank van der Linden, Jeff Kramer and Jeff Magee. "The Koala Component Model for Consumer Electronics Software". 2000. Computer vol 33, pp. 78-85.

[18] Gorlick, M. and Razouk, R. " Using Weaves for software construction and analysis". *Using weaves for software construction and analysis*, 1991, pp. 23-34, 1991.

[19] Feiler, P. H., Lewis, B.A. and Vestal, S. "The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems". *Computer Aided Control System Design, 2006 IEEE,* pp. 1206-1211, 2006.

[20] Garlan, D., Monroe, R. and Wile, D. "Acme: an architecture description interchange language". *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, pp. 7, 1997.

[21] "Architecture Description Markup Language - Creating an Open Market for IT Architecture Tools," http://www.opengroup.org/tech/architecture/adml/background.htm

[22] Eric M. Dashofy, André Hoek, van der and Richard N. Taylor. "A comprehensive approach for the development of modular software architecture description languages". 2005. ACM Trans.Softw.Eng.Methodol. vol 14, pp. 199-245.

[23] Weiss, M. and Amyot, D. "Designing and evolving business models with URN," in *Montreal Conference on eTechnologies (MCETECH05,* 2005, pp. 149-162.

[24] Buhr, R. J. A., " Use case maps for attributing behaviour to system architecture". *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, pp. 3, 1996.

[25] Buhr, R. J. A. "Use case maps: A new model to bridge the gap between requirements and design," in *OOPSLA Workshop -- Requirements Engineering: Use Cases and More, Sunday October 15,* 1995,

[26] R. J. A. Buhr. "Use Case Maps as Architectural Entities for Complex Systems". 1998. IEEE Trans.Softw.Eng. vol 24, pp. 1131-1155.

[27] "GRL - Goal-oriented requirement language," accessed 2009, http://www.cs.toronto.edu/km/GRL/

[28] Liu, Lin and Yu, Eric. "From Requirements to Architectural Design – Using Goals and Scenarios". *Submission to First International Workshop. from Software Requirements to Architectures. (STRAW'01).,* 2001.

[29] Book, M. and Gruhn, V. "Modeling Web-based dialog flows for automatic dialog control". *Automated Software Engineering, 2004. Proceedings. 19th International Conference on,* pp. 100-109, 2004.

[30] Book, M. and Gruhn, V. "A dialog flow notation for web-based applications". *Seventh IASTED International Conference on Software Engineering and Applications*; Marina del Ray, CA; USA; 3-5 Nov. 2003. pp. 100-105. 2003

# Appendix A

| Aspect | Darwin | Rapide | Wright | Koala | Weaves | AADL | Acme | ADML | xADL | UML | DFN |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Visualizations** | Textual, graphical | Textual, graphical | Textual | Textual, graphical | Graphical | Textual, graphical | Textual | Textual | Textual, graphical | Textual, graphical | Graphical |
| **Components** | Components | Interfaces | Components | Components | Components | Component type, category, implementation | Components | Components | Components | Classes, objects, components | Masks |
| **Connectors** | Instantiation and binding | Messages /events | Connectors | Instantiation and binding | Queues | Connections | Connectors | Connectors | Connectors | Associations | Actions |
| **Interfaces** | Require/provide interfaces | Same as components | Ports and roles | Require/provide interfaces | Write/read ports | Ports, buses | Ports, roles | Ports, roles | Interfaces | Interfaces | Events |
| **Configurations** | Hierarchical composition | Structures | Configurations | Hierarchical, with variation points | Tool fragments | Networks, processes, and more | Systems | Systems | Links, general groups | Diagrams | Modules |
| **Non-functional aspects** | No support | No support | No support | No support | Implied by architectural style | User-defined properties | User-defined properties | User-defined properties | User-defined extensions | Minimal, through annotations | No support |
| **Static/dynamic aspects** | Static and dynamic | Static and dynamic | Static and dynamic | Static | Static | Static and dynamic | Static and dynamic | Static and dynamic | Static and dynamic | Static and dynamic | Dynamic |
| **Dynamic models** | No support | No support | No support | Variation combinations | Implied by static model | No support | Programmatic manipulation | Programmatic manipulation | Programmatic manipulation | Tool-dependent | Presentation channels |
| **Domain** | Multiple | Event/message based systems | Safety-critical systems | Consumer electronics | Multiple, object streams | Critical real-time | Multiple | Multiple | Multiple | Multiple | Web-based applications |
| **Tool support** | Darwin Framework | Rapide toolset | Wright toolset | By Philips and Open Source | Weaves toolset | Multiple and growing | AcmeStudio | XML editors | ArchEdit, Archipelago | Multiple and growing | Dialog Control Framework |