Architectural Structures and Views

Alexander Ran
Nokia Research Center,
3 Burlington Woods Drive #260,
Burlington, MA 012803, USA
alexander.ran@research.nokia.com

1. ABSTRACT

This is an outline of a conceptual framework for architecting complex software. The framework identifies multiple independent structures of software that support different kinds of requirements, making possible to partition requirements so that each group can be supported by a different architectural structure. Architectural views are separated from architectural structures making it easier to define different processes and allocate different development stages for design and description of architectural structures and views.

1.1 Keywords

Requirements, software architecture, architectural views, structures of software

2. INTRODUCTION

Many new projects developing software intensive products begin from design of system software architecture. Often one of the first steps in this process is building a common for the project members understanding what is software architecture. Being present in project meetings addressing this question often I have to present my opinion.

In such situations I always remember the classical definition by Perry and Wolf of architecture being elements, form, and rationale [1]. Though this definition masterfully captures the essence of architecture, I don't usually dare to pronounce it as an answer to a product development team. This is because their real question is "what is an appropriate conceptual framework for design and description of software architecture for this product?".

This paper is an outline of a conceptual framework for architecting complex software. Here I present my understanding regarding what are architectural structures of software, why they are different from each other, and how they can be used to support desired system properties – the rationale for the elements and the form. I will describe how different architectural structures support different kinds of requirements, suggesting that there are ways to partition requirements so that each group can be supported by a different architectural structure. I will discuss partitions of requirements based on architectural structures that play major role at different stages of software lifecycle like write time, configuration time, (re)-start time, or run time. Some of the ideas may be only applicable to embedded software, as this is the domain I am having in mind when writing this paper.

3. ARCHITECTURAL STRUCTURES

Architecture of software directly affects system-wide properties like availability, reliability, security, etc. Well-structured software also supports requirements for change, reusability, interoperability with other systems, etc. If all different requirements were supported by the same architectural structure it would be impossible to satisfy them independently. And indeed this is often the case. For example requirements concerning performance and reliability interact since software execution structure affects both kinds of requirements.

Software exists in multiple component domains as a set of modules, a set of set threads, a set of processes, a set of files, etc. In each component domain a system can form a different structure. Often system requirements may be grouped so that requirements in different groups may be addressed by different and at least partly independent software structures established by partitions of software in different component domains. Such partitions exist simultaneously and often are independent of each other.

A few examples:

Run-time requirements are addressed by partitioning software into execution threads of varying priority (or utility), specifying thread scheduling policies, regulating use of shared resources, etc.

Portability requirements are addressed by defining software layers and establishing conformance of layers and their interfaces to existing standards.

Reuse requirements are addressed by partitioning software into modules - substitutable, unit-testable components having well-defined boundaries, predictable interaction with the environment, and minimal, well-specified dependencies on other modules.

Effective work division is addressed by partitioning software into subsystems that limit the domain expertise necessary for their development and further partitioning the subsystems into separately testable components that implement a meaningful in the product domain function, with minimal interaction with other functions that can be modeled and controlled.

An architectural structure is created by configuration of a partition. By configuration I mean an instantiation of components (parts) and their relationships.

As it happens these and other useful structures of software in different component domains may be very different from each other.

Structure / Stage	**Requirements Domain**	**Component Domain**
Execution structure is essential at run-time	Performance, availability, reliability	Execution threads, communication channels, schedulers, shared resources
Loading structure is essential at start-up / shut-down time	Independent re-start / upgrade, protection	Processes/executables, data stores
Module structure is essential at "write" time or construction time	Change management (for evolution, porting, diversification), incremental / concurrent development, reuse,	Modules, provided and required interfaces
Increment structure is essential at integration time	Incremental system development integration and testing	Increments
Subsystem structure is essential at work division time	Work division, concurrent development, outsourcing,	Subsystems

Table 1 Architectural Structures of Software

For example the partition into modules has little or no relationship to partition into execution threads. Partition into modules is done to enable incremental construction, testing, evolution, and reuse of specific functionality. Layers of a protocol stack are an example of partitioning data communication functionality into modules. Partition into execution threads is done to simplify system design while addressing performance and possibly reliability and availability requirements. The same module structure such as a protocol stack for example may get assigned to or split over an arbitrary structure of interacting execution threads. Thus there needs to be no relation between the two structures.

As another example consider the partition of software into processes. This partition is used to address requirements for independent loading and protection. Though processes bound sometimes execution threads, often execution threads span multiple processes. Such could be the case when parts of a protocol stack need to be independently (re)loadable and / or upgradable. One way to address these requirements is by partitioning the stack into different processes. At the same time the passage of a packet through a protocol stack happens in a single execution thread. Also the very existence of remote procedure call mechanism is due to the fact that a single execution thread may get partitioned into multiple processes.

One effective way to identify independent (or partly independent) requirements and component domains is by identifying structure of software that play major role at alternative stages of software life cycle. A typical (though somewhat simplified) set of stages when different partitions of software play major role include write-time, build-time, configuration-time, start-time, and run-time.

Thus write-time related requirements like feature addition and evolution, porting, and diversification are primarily addressed by appropriate module structures that play major role at write-time. Similarly, start-time related requirements (like order, presence, independent operation and failure modes) are primarily addressed by appropriate process (executable) structures – the startup / shutdown unit or component. And, of course, run-time related requirements like performance or availability are addressed by the structure of execution threads – the primary run-time software component.

Table 1 lists some of the most common partitions, their requirement domains, component domains, and software lifecycle stages concerned.

In many software development projects there is significant pressure to structure the system identically in different component domains. This invariably leads to problems in development and occasionally in final products (see [2] for some real life examples). Therefor it is very important to recognize the existence of multiple component domains, independent partitions of software, and their relations to different requirement domains.

It is interesting to notice that architectural structures can be (and often are) defined without ambiguity. For example the module partition specifies names and interfaces of modules, and module configuration specifies module instantiation and binding.

Subsystems are essentially groupings of modules and are best described by specifying the modules that they contain. Subsystems are commonly "vertical" sections. Such subsystems usually aggregate modules that implement related functions

Loading partition can be specified by names of the programs, shared (dynamically linked) libraries, data stores, and parameter data. The configuration of loading partition is specification of loading and unloading order often indirectly defined by process dependencies.

Though architectural structures must be described unambiguously in most cases such descriptions do not necessary require a special architecture description language unless a specific kind of analysis or generation can be performed. In the later case the main question is whether the analysis or generation capabilities would justify the overhead of additional language.

If we were able to establish all architectural partitions necessary to address product and development requirements we would have not needed architectural views. However this is not the case for most non-trivial systems.

It is usually too hard to design or even to understand all architectural partitions without some graduate approach through simplified conceptual models of software. Such simplified conceptual models that are needed for design of architectural partitions and their configurations constitute architectural views of software.

4. ARCHITECTURAL VIEWS

For most software systems it is possible to identify three classes of important concepts: application domain concepts, implementation domain concepts, and architectural concepts. Application domain concepts result from application domain analysis and jointly form domain model. Implementation domain concepts result from implementation domain analysis and jointly define infrastructure, virtual machine or platform.

Architectural concepts are not found from analysis of requirements or implementation platform. Key concepts for architecture of software need to be invented to simplify the task of bridging the product requirements and implementation platform.

Thus one of the primary tasks of software architects is to establish and communicate to the rest of the team all the important concepts necessary for effective software design and implementation. A proven way to approach this goal is by creating partial models that relate different architectural concepts and their role in addressing architectural problems and concerns. These models reflect various aspects of software construction and execution and provide partial views on architecture of the software. Together these views make conceptual architecture of software.

Architectural views are created before the system is designed to any significant degree of detail and usually exist more as a vague intuition than a precise structure. To communicate these intuitions to the development team, to define architectural partitions, and to develop detailed designs, architects must rely more on evocative concepts than formal descriptions. This is the primary reason why verbal interaction is considered so important for successful communication of conceptual architecture.

The term "architectural views" is commonly used to mean a broader category of architectural descriptions following the well-known work of Philippe Kruchten on the "4+1" views model of software architecture [3]. In the "4+1" model architectural descriptions are grouped into logical,

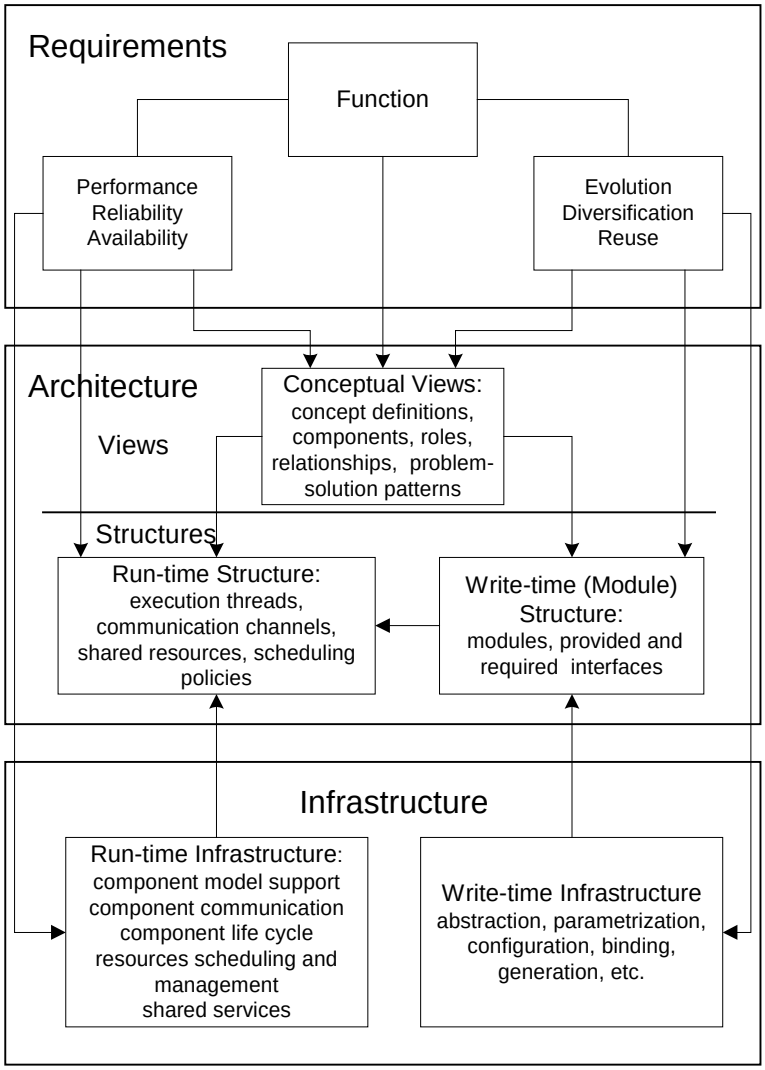

Figure 1 Partitioning Requirements Architecture and Infrastructure along run-time / write-time line

development, process, and physical view. While the logical view is a set of conceptual models, the other views of the "4+1" model correspond to concrete architectural structures of software.

There are several good reasons to clearly separate concrete software structures that exist at write-time, or start-time, or run-time, from abstract views necessary for early conceptualization, design, or understanding of complex software and its architectural partitions. While conceptual views of software architecture need to be built prior to more detailed design concrete architectural structures are best described along with detailed design and often after implementation is completed. Also the degree of detail and precision in describing conceptual models and architectural structures is different. Finally, it is significantly easier to communicate to software developers importance of concrete architectural structures than abstract conceptual models. Understanding the relationship between conceptual architectural views and concrete structures makes architecture more accessible to development team and thus increases its healthy life time,

Figure 1 shows an example of partitioning requirements architecture and infrastructure along run-time / write-time line. Additional partitions can be introduced as necessary for start-up time, configuration time, and other important stages of software lifecycle. The links on this diagram are not marked because they carry multiple meanings. The most general interpretation of the links is indication of dependency or flow of influence. This implies an order for definition, or in a spiral development order of progress in definition.

The diagram on Figure 1 also emphasizes separation of application architecture from infrastructure and shows that product (or rather product family) requirements influence design of infrastructure. Though the value of this partition is well understood it is often not seen as something to be designed as a part of product software architecture. This is quite acceptable for many types of computer software where advanced infrastructure is well established and is refined through use in numerous applications. Embedded software is quite different in this respect. It runs on top of application domain specific hardware machines that often consists of multiple devices integrated to provide functions necessary for a specific product or product family. Therefor embedded software architects need to specifically design the infrastructure appropriate for the product.

All the ideas we discussed in application to architecture are also applicable to infrastructure. This includes concepts definition, partial problem-oriented views, identification of independent structures, and partition of requirements in correspondence with available independent architectural structures.

Existence of write (or construction) time infrastructure may need some clarification. Construction time dimension of the implementation domain for software based solutions is made of techniques and support tools for software construction. This includes code generation tools, macro facilities, interpreters, compilers, configuration management tools and techniques, etc. Just as run-time infrastructure must be identified and managed to provide adequate support for execution architecture, construction time static infrastructure must be identified and often specifically designed for the particular product family. Write time infrastructure supports modularization of software. The key issue of modularization is localization of definitions for functionality that is subject to change. Though programming languages and other software construction tools are designed to solve the general problem, specific application domains may require and often can benefit from application specific write-time infrastructure that allows to localize definitions of functionality which otherwise would have to be replicated. Typical examples of advanced write-time infrastructure are meta-facilities, preprocessors, application specific languages, and some other code-generation technologies.

Just as run-time architecture rests on run-time infrastructure, module architecture rests on write-time infrastructure.

5. SUMMARY

In the early stages of software design one can only expect to outline partial models for structuring software that form abstract views of software architecture and communicate ideas for addressing different architectural concerns on an intuitive level. Later concrete architectural structures need to be designed and described precisely. Existence of multiple independent structures in different software component domains makes it possible to support different kinds of requirements at the same time. An effective conceptual framework for software architecture needs to specify a partition of requirements to independent architectural structures.

6. REFERENCES

[1] Perry, D.E. and Wolf, A.L. "Foundations for the Study of Software Architecture", Software Engineering Notes, ACM SIGSOFT, vol. 17, no. 4, October 1992

[2] Ran, A. and Kuusela, J., Selected Issues in Architecture of Software Intensive Products, in Proceedings of the Second International Software Architecture Workshop, ACM Press, 1996.

[3] Kruchten, Ph. "The "4+1" View Model", IEEE Software, 1995