TOWARD DOMAIN-SPECIFIC DESIGN ENVIRONMENTS

Some Representation Ideas from the Telecommunications Domain

Sol Greenspan and Mark Feblowitz

GTE Laboratories Incorporated
Computer and Intelligent Systems Lab
40 Sylvan Road
Waltham, Massachusetts 02254
617-466-2962
greenspan@gte.com

Introduction

ACME[1] is an experimental environment for investigating new approaches to modeling and analysis of system requirements and designs. ACME is built on and extends object-oriented conceptual modeling techniques and knowledge representation and reasoning (KRR) tools [Greenspan, et. al. 1991]. The most immediate intended use for ACME is to help represent, understand, and communicate system designs during the early stages of system planning and requirements engineering.

While our research is ostensibly aimed at software systems in general, we are particularly motivated to make an impact in the telecommunications domain, especially in the area referred to as Intelligent Networks [IEEE Comm., Dec. 1988], [IEEE Comm., Feb. 1992]. Intelligent Network (IN) systems contain the software to provide services to users of a telecommunications network (e.g., call processing services, information services, etc.) as well as the software that provides the internal infrastructure for providing the services (e.g., resource management, billing, etc.). The software includes not only systems developed by the network proprietors but also by a growing group of independent service software providers.

The kind of software design problem we are interested in is at a high level. It involves, among other things, deciding where, in a distributed heterogeneous system, to locate program logic, data, and other resources; conceptually speaking, how to assign responsibilities and capabilities for carrying out the services [Greenspan 1991]. The situation is often an evolving one: given an existing situation, new requirements arise, such as the need for a new service or a new capability, and the design problem is how to (re)design the system to respond to the change.

We are quite sure that IN systems analysts and designers use a great deal of domain knowledge to make decisions about how to design an IN system to meet new requirements, and that their familiarity with the domain is a dominant factor affecting the ultimate success of the system design. The question is what that knowledge is and how it can be represented in a domain-specific environment. In this paper, we will briefly survey a few of these representation ideas and how they contribute to the goal of domain-specific software design. To the extent that these ideas are cogent applications of general software engineering principles, their essence should apply to other domains as well.

Design from domain-specific building blocks

In the telecommunications domain, there are several mandates for having a stable set of building blocks from which service software can be composed and rapidly implemented. One impetus for building blocks is the need for the industry to agree on a basic set of services and capabilities that can be assumed as universal so that services can interwork over company and national boundaries. Another impetus is that the US federal government seeks to promote fair competition by making sure that a common set of building blocks is available to all potential service developers/providers (not only to the telecommunications network proprietors).

Although the forces that motivate the use of building blocks may be largely nontechnical or quasi-technical, the emphasis on a building block approach turns out to be a valuable idea from a design point of view. It narrows the search space for solution software because all solutions must be composed from officially sanctioned building blocks. Moreover, the resultant software can be more correct, reliable, and so on, since building blocks are subject to intense scrutiny and analysis. The building block approach may appear to bring with it a loss of design

---

1 ACME is an acronym for A Conceptual Modeling Environment.

freedoms, since software is not allowed to decompose into arbitrary software components, but the premise here is that the gain in manageability of the design process is worthwhile compensation for this

The essential ideas of a building block approach are as follows. First, building blocks need to be (a) adequate to compose the desired set of services, and (b) implemented effectively in components of the systems that provide the services. Secondly, building blocks need to be reliably, efficiently, safely (etc.) implemented in the embedded system base. Thirdly, the introduction of new building blocks into the system fabric needs to be a controlled process. Suppose an organization desires to offer a new class of services that requires building blocks not already available in the system; the newly required building blocks need to be carefully identified, implemented, and tested, and importantly and nontrivially, their interactions with the old building blocks need to be taken into account.

It is important not to confuse the building block approach with the general notion of reusable components. The main idea of reuse, in its most general sense, is an asset management idea, namely that prior investment in software artifacts (code, specifications, or whatever) can be capitalized on by reusing the artifacts. If an enterprise restricts its software development to a specific domain, then the existence of domain-specific reusable components may enable one to achieve a higher degree of reuse. It has been pointed out that domain analysis is a way to achieve this (e.g., see [Arango & Prieto-Diaz, 1991]). However, this is still not the idea of building blocks. Reusable components refer to a library of assets that happen to be available to designers, while building blocks refer to the set of software components that have been designed into the system infrastructure of the operational system.

We suspect that a building block approach is already being used in other software domains and is worth making an explicit principle for building domain-specific environments. Further insights can be gained by drawing parallels between software development and other forms of manufacturing, where a set of building blocks (or "parts") are used to assemble products. Software is different in the respect that an infinite variety of "parts" can be created, which is both an opportunity and a management problem.

Domain-specific layering based on design decisions

In the telecommunications domain, standards groups are discussing a four-level IN conceptual model [Duran & Visser 1992] that organizes Intelligent Network systems in a useful way that might apply to other domains. While the model itself is not complete in any sense and is continually evolving, there are some ideas worth noting. We will not give a literal description of the four-plane IN conceptual model but rather give a rough summary and extract some of the key ideas, using vocabulary convenient for the purposes of this paper.

The layers/planes are roughly the following, from top to bottom:

1) Services -- The software applications for the end user.
2) Service Building Blocks -- As discussed above, software components that are used to compose services and which are provided by the underlying service-providing system/network.
3) Logical System Entities -- A set of standard system components (called "functional entities" by the standards groups), each of which offers methods that implement the building blocks.
4) Physical System Entities -- A set of available system components that can be developed or procured and installed in the embedded base. They are, conceptually, packages of logical system entities. Vendors build these.

These planes are usefully chosen so that the relationships between adjacent levels involve key types of design decisions. We already discussed the relationship between Services (level 1) and Service Building Blocks (level 2).

The main rationale for level 3, Logical System Entities, is that the industry needs to have a way of identifying, specifying, and integrating systems in a vendor-independent manner. Besides this motivation, level 3 also seems to be the focal point for several design concerns. Level 3 identifies the perceived infrastructure of logical, service-providing systems. Elaboration of this plane would describe the perceived standard subsystems that comprise the domain, such as making phone calls, billing, reporting error messages, and so on. This level will be quite rich with domain-specific content, representing, in effect, a model of the service-providing enterprise (discussed further below).

The relationship between Service Building Blocks and Logical System Entities (i.e., between levels 2 and 3) concern how capabilities are distributed among logical system components. The relationship between Services (level 1) and Logical System Entities (level 3) concern design decisions about what system entities are responsible for playing various roles in services.

The design decisions relating Logical and Physical System Entities (levels 3 and 4) mostly concern designing a physical system to meet nonfunctional requirements. Logical systems will have associated nonfunctional requirements (e.g., concerning performance, reliability, security, etc.) that must be met by the physical system entities. The original sources of nonfunctional requirements might actually be traceable to any of the levels. In any particular domain, one must identify and specify the nonfunctional properties that are most critical to success in that domain. Arguably, the design knowledge about how designers design physical systems to successfully meet the nonfunctional properties seems one of the most difficult subjects to formalize and automate.

This discussion of the four-plane model is intended to point out some of the more generic (high-level) design issues that might transfer across domains. The industry has

developed other, more detailed, layered models (such as the seven-layer OSI architecture), which is more intensely domain-specific to communications and less likely to transfer.

Enterprise domain knowledge

The domain-specific design environment for systems in our domain should be able to take advantage of the fact that all of these systems ultimately are part of an enterprise that provides services (either to end-users or to internal agents responsible for tasks necessary for providing the service). Given what is known about the nature of these systems, there are a lot of assumptions and constraints that can be built into the design environment.

For example, in the domain of telecommunications services, there are customers who subscribe to services. Services are tasks performed by service provider agents for customers, usually involving sensing and changing the state of objects in the customer environment and performing communication acts across a network of objects. It is further known that when a customer signs-up for (or subscribes to) a service, some service-related objects may need to be installed (e.g., a telephone at the customer premises, a wire to the customer's residence, customer information in a system database -- this is called "provisioning"). Another part of the domain is that services are performed in exchange for payment, which requires data on the use of services by customers. These and other aspects of the enterprise can be and should be part of a domain specific environment for designing systems in that domain.

One advantage to be gained by ACME from the presence of enterprise domain knowledge is that model acquisition can be supported by intelligent assistance, as in [Reubenstein 1990]. For example, since the assistant knows that provisioning is done when a new customer signs up for a service, the system can know something about what information needs to be specified (and can partially fill it in).

Designing systems in terms of the enterprise domain knowledge is much easier than working at a general systems level. General-purpose CASE environments, which offer generic concepts such as objects, properties, entities, processes, and so on, leave too large a semantic gap between the subject matter and the representation scheme. (On the other hand, we are in favor of building on general-purpose modeling concepts; see [Greenspan, et. al. 1991].) In [Greenspan 1991], we actually propose the use of an intermediate level of domain-specificity, called Service-Oriented Systems (SOSs), that takes advantage of some of the knowledge of service-providing systems in our domain but still remains relatively generic.

Process domain knowledge

The above argument for using domain knowledge can be extended to process knowledge, namely the process of designing and developing the system. This is sometimes called process knowledge or methodology modeling. Process domain knowledge deals with how the system/software artifacts are created and how they evolve. Given that we know that the artifacts we are designing belong to a specific domain (e.g., systems that provide IN services), we can specialize our view of the process that creates these artifacts. We are not creating just programs, or subsystems -- we are creating services, service-providing systems, and so on. Each of these concepts refers to a type of artifact that needs to be designed, maintained, and evolved. This process domain knowledge needs to be represented in the environment, too.

A service-providing system in our domain is built (or evolves) by specific actions such as Create Service, Install Capability, and so on. These process operations can be considered as services themselves, where the user is the software designer/developer/maintainer rather than the usual service customer. The ability to rapidly create new services and alter the enterprise systems to provide the services is critical and therefore comprise important (meta-)services in themselves.

Thus, we think that work on general models of software process should be specialized to specific domains.

Summary

By exploring some of the manifestations of domain-specificity in our domain of IN systems, we have found some representation concepts that could have parallels in other domains.

Note how some general SE principles were instantiated but restricted to impose constraints that help gain control over the domain.

> Domain-specific building blocks are like reusable components that result from domain analysis of the services and service-providing systems in the domain. However, they play a stronger role in constraining the design.
>
> Domain-specific levels based on design decisions are similar to levels of abstraction in software engineering but there is a fixed number of levels. We do not do an analysis/design to find out how many levels a system will have.
>
> We study systems in the domain and then design a fixed set of appropriate levels.
>
> Enterprise domain knowledge is similar to knowledge represented in generic environments. However, this knowledge is built into the environment (at the meta-level) to become part of a domain-specific framework.
>
> Process domain knowledge specializes the vocabulary and tools of the software process, so that domain experts have a more direct understanding of the process.

We close by mentioning a couple of issues that could be discussed at the workshop:

How can we build on general-purpose/vanilla methods and tools? Some fairly well-understood vanilla notions of behavior, function, structure, etc. are converging in AI. Similarly, some forms of object models, dataflow diagrams, state-transition, etc. from the CASE world are becoming fairly standard. We need to understand now to systematically construct domain-specific structures on top of (or next to) these.

Are there some common subdomains whose subject matter knowledge and design knowledge would be useful across several different domains? Is anybody working on representation and reasoning frameworks for important domains and packaging them to be shared across domains?

What is a "good" high-level design? For example, suppose high-level design includes assigning responsibilities to agents and assigning ownership of resources to agents. Then a "good" design might be one in which all agents who have responsibility for an action own the resources needed to carry out the action. However, this might be too restrictive; a suitable design might be one in which an agent responsible for an action either owns the needed resources or has access to an agent who does. This needs to be developed, and a framework for expressing designs is needed. (If there is already some work on this, we would like to become aware of it.)

References

Greenspan, S., M. Feblowitz, C. Shekaran, and J. Tremlett, 1991. Addressing Requirements Issues Within a Conceptual Modeling Environment. In Proceedings of the International Workshop on Software Specification and Design.

IEEE Communications Magazine, December, 1988. Issue Featuring Building the Intelligent Network 26(12).

IEEE Communications Magazine, February 1992. Issue Featuring Intelligent Networks 31(2).

Greenspan, S.1991. Analysis and Design of Composite Service Systems. In Proceedings of AAAI Symposium on Composite System Design.

Arango, G.; Prieto-Diaz, R., 1991. Introduction and Overview: Domain Analysis Concepts and Research Directions. In Prieto-Diaz, R., and Arango, G. (eds), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

Duran, J.; Visser, J., 1992. International Standards for Intelligent Networks. in [IEEE Comm., Feb. 1992].

Reubenstein, H., 1990. Automated Acquisition of Evolving Informal Descriptions. Ph. D. diss., M.I.T. Technical Report 1205.