Use Case Maps for Attributing Behaviour to System Architecture

R.J.A. Buhr

SCE-96-2: February 3, 1996

Contribution to the Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii

Department of Systems and Computer Engineering
Carleton University
Colonel By Drive, Ottawa, Canada K1S 5B6
buhr@sce.carleton.ca
http://www.sce.carleton.ca/faculty/buhr
(613) 520-5718
(613) 520-5727 (fax)

Use Case Maps for Attributing Behaviour to System Architecture

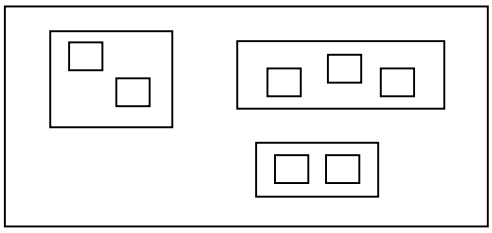
R.J.A. Buhr
Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada
buhr@sce.carleton.ca, http://www.sce.carleton.ca/faculty/buhr
Keywords: architecture, behaviour, design, use cases, use case maps, scenarios, systems

Abstract

The ability to attribute behaviour to architecture is important for high-level understanding, designing, evolving, and reengineering all kinds of systems (from object-oriented programs to parallel and distributed computer systems). Scenarios are a good way of doing it, but popular scenario techniques, such as message sequence charts, that use intercomponent "wiring" as their starting point do not scale up well. Use case maps provide a new, scenario-based way of attributing behaviour to architecture that solves the scaleup problem. The notation enables compact, composite maps to be drawn to represent behaviour patterns of whole systems in terms of causal paths, without reference to "wiring". Through an example, the paper aims to convince software and system engineers that the approach has depth and adds value, despite (and because of) its simplicity and deferment of detail.

1.0 Complexity Factors in Systems

Systems are often characterized at a high level of abstraction by block diagrams such as the following (called here *component context diagrams*), in which the



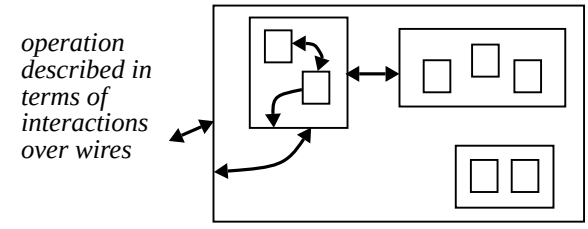
boxes represent collaborating components, every one of which is potentially—through recursive decomposition—itself a system.

To attribute behaviour to systems described with component context diagrams, connective tissue must be added. One form of connective tissue is intercomponent *wires*, meaning conceptual or actual connections that support interactions—such as calls or messages—between components through interfaces (a component context diagram with wires is called a *wiring*). Wirings are assumed to identify explicitly the names and parameters of all possible interactions, without reference to the internal logic of components. Behaviour may be attributed to wirings by describing interaction scenarios with interaction diagrams [7] (which we shall henceforth refer to by the generic term *message sequence charts*). We argue below that the combination of wirings and message sequence charts will overwhelm us with detail for systems of any size or complexity. We need to stand back from such detail to see the big picture clearly, in other words to see the system in architectural terms.

Furthermore, wirings that are concrete enough to be useful for understanding behaviour in the above terms are too detailed to be considered as architecture diagrams unless systems are one-off, small, and simple. For families of large or complex systems, architecture should have more to do with rules or guidelines for *creating* concrete wirings over the family than with the specifics of individual wirings. Diagrams showing only the existence of relationships such as *uses*, *communicates with*, or *has contract with* are useful for architecture, but they are not wirings in the terms of this paper, and are therefore not starting points for attributing behaviour to architecture.

Now let us examine the complexity factors associated with wirings and message sequence charts that will cause us to be overwhelmed with detail. They fall under three headings: *operation*, *assembly*, and *manufacturing*.

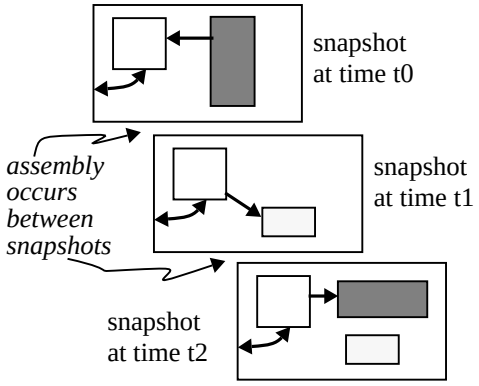
Operation. Wirings provide a means of understanding the *operation* of a system in terms of interactions over wires between components. Wirings



must include wires not only between peer components at

the same level of recursive decomposition but also between components at different levels of recursive decomposition, through all levels of decomposition (*symbolized* by the arrows in the figure, remembering that there would be many more arrows in actual wirings). Such wirings give us a view of system *operation* in terms of compositions of many details, tending to overwhelm the big picture with details. The effect is compounded by two other factors.

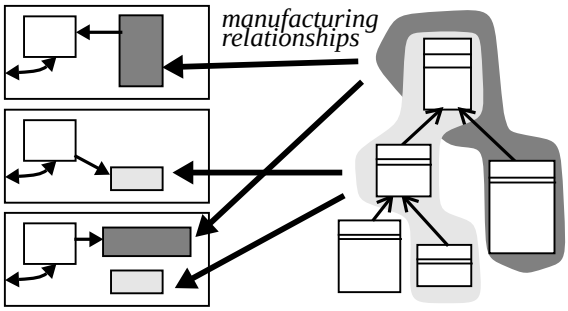
Assembly. Systems may change form while they are running, in other words, they may be structurally dynamic (why we associate the term *assembly* with this will be explained shortly). Structural dynamics is a routine property of systems and software, not an unusual one. When systems are viewed in terms of wiring diagrams as above, structural dynamics means that both the components and the wiring may change over time (symbolized in the following diagram by a sequence of snapshots, each of which show different components and different wiring at different times). We say that the



wiring in each snapshot shows connections that enable *operation*, that sequences of snapshots imply *assembly* (meaning that new components and wiring must be assembled in between snapshots), and that operation + assembly = *behaviour*.

Even without any specific description of how assembly is to be accomplished in between the snapshots, viewing assembly in terms of changing wirings obviously adds complexity to an already complex picture. This complexity seems counter-intuitive, considering the lightweight way in which assembly happens in code (pointers are assigned for new software components and passed around to other components).

Manufacturing. The third complexity factor is that behaviour (in the above sense of operation+assembly) becomes intertwined with *manufacturing* (in the sense of manufacturing objects from inheritance relationships in class hierarchies). This is symbolized in the following figure by showing a class hierarchy on the right from which the operational components in the snapshots on the left are manufactured (the shaded shapes in the class diagram symbolize the idea that the inheritance hierarchy is, in effect, collapsed into concrete classes, instances of which become components in the snapshots). In object-



oriented development, classes are where component behaviour and intercomponent wiring are defined, but diagrams showing class relationships do not themselves give the system picture on the left, which must be inferred from details in the class descriptions. Thus the system picture is a second-class abstraction in relation to classes (second-class, because it depends on details of classes, namely their interfaces and methods). Many diagrams and snapshots may be required to see the whole picture.

In Summary. Attributing behaviour to systems through wiring drags the level of abstraction down below a level appropriate for architecture. It requires visually and mentally combining elements abstracted from a messy, difficult-to-separate soup of details. The approach may be tolerable for small systems, but does not scale up well.

2.0 Essence of Use Case Maps

Use case maps [2] raise the level of abstraction by simplifying all three of these complexity factors:

Operation. Wires are replaced as connective tissue by cause-effect paths. Behaviour is not represented in terms of interactions between components via wires, but in terms of cause-effect sequences between responsibilities of components. Wires to achieve the interactions required to implement the cause-effect sequences are deferred as details. In other words, interfaces of components, and interactions between interfaces, such as calls, messages, or IPC, are deferred as details. Responsibilities can be coarser grained than such quantities as calls or messages. The net effect is to greatly reduce the level of commitment to detail.

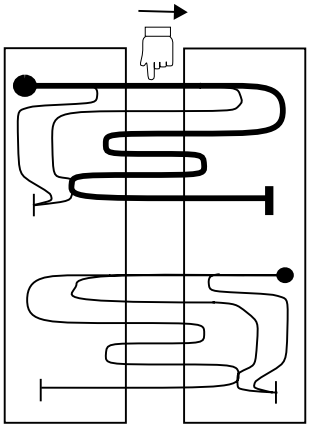
Assembly. The modeling of assembly as a sequence of snapshots of changing components and wirings is eliminated. The wiring part disappears

because wires have disappeared. The changing components part is still required, but is modeled in a simple way through the use of *slots*. Slots are like positions in human organizations that exist independently of their occupants or occupancy. They are also like slots that exist in equipment racks of computer systems for circuit boards that may be plugged in at any time while the system is running. At the level of use case maps, the only requirement for pluggability of a component into a slot is that the component is able to perform the responsibilities defined for the slot. Setting up the wiring to accommodate the component (where wiring is used in the very broad sense of physical and software connections) is deferred as a detail. Slots are not specifically illustrated in this paper (see [1][2] for examples), but are an important feature of use case maps.

Manufacturing. Dependence of behaviour descriptions on manufacturing details is eliminated by the fact that use case maps are first-class models in their own right that may be developed independently of class descriptions.

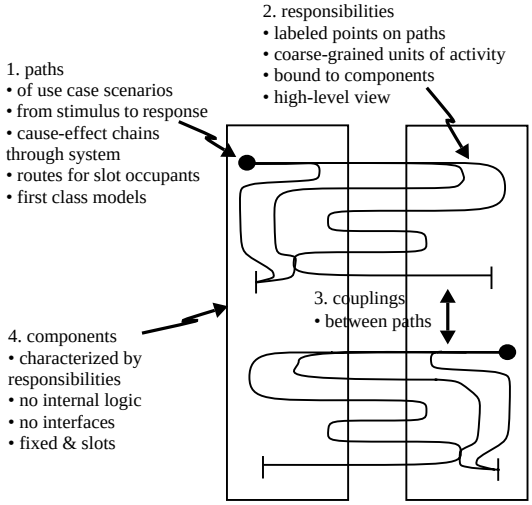
The Result. By eliminating dependence on wiring, use case maps raise the level of abstraction to an appropriate one for architecture.

The essence of use case maps is extremely simple. The diagram below is a use case map consisting of

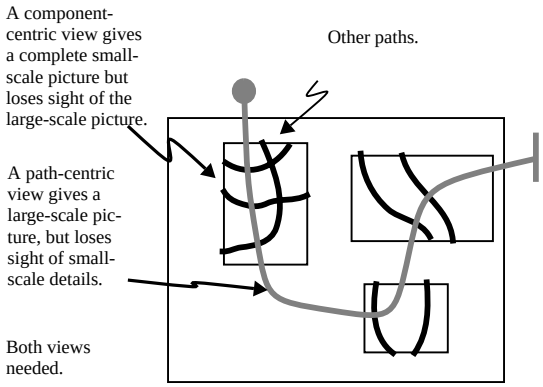


many paths superimposed on a component context diagram containing two components. The meaning of this particular map in terms of the specific system it describes is explained in Section 3.0. The only thing to understand at this point is that it is basically like a road map in which any end-to-end route (such as the one highlighted here with thick lines) may be traced with your finger to explain cause-effect sequences that follow from stimuli occurring at the start of the route (the filled circle). A use case map is basically a record of routes traced by one or more such finger-pointing sequences (called “scenarios”). Concurrency is represented, not by specific map notations, but by allowing multiple scenarios to be traced at the same time, so use case maps are equally applicable to sequential or concurrent systems.

Use case maps have four main elements: *paths* that trace scenarios through the components of a system from points where stimuli occur to points where responses are felt; *responsibilities* that link paths to components; *couplings* between paths that express couplings between scenarios, such as one scenario pausing to wait for another; and *components* that perform responsibilities.

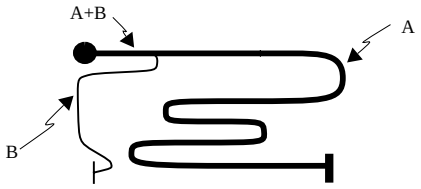


Paths. Use case maps provide a path-centric view of systems that is, in a sense, a dual of the more familiar component-centric view.

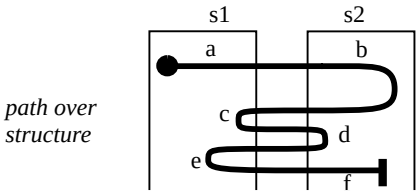


There is more to paths than just the visual representation. Documentation of maps includes at least the following: naming and prose descriptions of scenarios associated with paths; preconditions and postconditions; labelling of path segments to indicate the scenarios associated with them (e.g., A+B for a segment shared by scenarios A and B, to provide information that would be shown in diagrams by shading or colouring); and cross references between large scale paths and factored smaller scale subpaths for parts of systems (Section 3.0

gives an example of factoring).

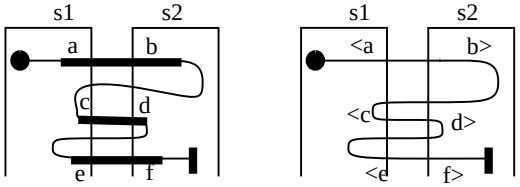


Sequences of responsibilities are shown in use case maps as labelled points on paths over structure, not as activity segments of timelines in separate time sequence diagrams. Thus they are not just a visual notation for responsibility sequences in use cases [5], they are a direct link between use cases and architecture that combines both in one diagram. Many paths can be



combined in a composite map to convey—in a compact way that is specifically related to architecture—information about behaviour patterns extending over multiple scenarios. Thus maps exploit human pattern recognition capabilities to increase understanding.

At the level of system architecture, we need to stand far enough back from fine-grained detail to see the big picture. This means that responsibilities may be coarse grained quantities, compared to calls or messages, and that sequences of responsibilities that span components may be coarser grained than sequences of calls or messages between components. Therefore, pairs of responsibilities that span pairs of components, such as ab, cd, or ef in the foregoing diagram, may actually represent situations in which multiple, detailed, back-and-forth *interactions* occur between the components to complete each pair of responsibilities. In other words, we may have to think of such reponsibilities as *shared*. The diagram below shows two ways of indicating



shared responsibilities

shared responsibilities: as the ends of thickened responsibility segments stretching between components, or with a special syntax for the responsibility labels (a third way is by map documentation). (Section 3.0 gives an example.)

Components include fixed components (indicated by solid outlines) and *slots* (indicated by dashed outlines—not illustrated). Slots enable assembly to be expressed with fixed use case maps in as lightweight a fashion as it is implemented in code, without leaning on programming concepts. A simple notation indicates points along paths where components are created, moved, or destroyed. At points of movement, the movement is into or out of slots or pools (a slot is a place where a single component may be active, a pool is a place where many components may be held in readiness to move into slots). Paths between such points indicate sources and destinations, without indicating the transport mechanism.

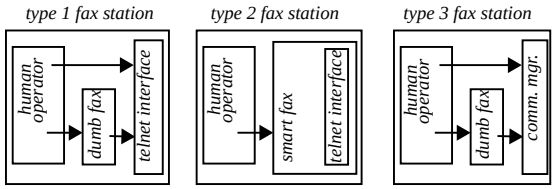
In fact, in software, components do not actually move, they only change visibility. Saying a component *moves* from one place to another is just another way of saying it goes from being visible in one place to being visible in another, where "visible" means it can be interacted with. From a software perspective, the movement model is a convenient fiction that contributes to a lightweight representation of assembly in use case maps.

Interpath coupling may be implied by scenario patterns or indicated explicitly by coupling notations. In either case, the existence of coupling is visible in the maps, it is only a question of whether or not there are explicit coupling notations joining paths. Examples of coupling implied by scenario patterns are as follows: the postconditions of one path become the preconditions of another; paths share some physical resource that implicitly couples them (e.g., a communication channel); and a responsibility along one path, by definition, affects a responsibility on another path through the state of some shared component. (Section 3.0 gives and example of implicit interpath coupling through shared resources and states.) An example of a type of coupling for which there is an explicit notation is interpath rendezvous. Explicit notations to indicate this and other types of coupling (not illustrated by this paper) are given in [2]. In all cases, the purpose is only to indicate at the path level that coupling *occurs*, without committing to the means.

3.0 A Fax System Example

This example, which is a slightly modified and extended version of an example from Chapter 2 of [2], is intended to give a sense of how to work with use case maps, not to cover all aspects of the notation or its interpretation. See [1] and [2] for other examples and the complete notation. The example is a pair of fax stations (each of

which is of one of the following types) interacting

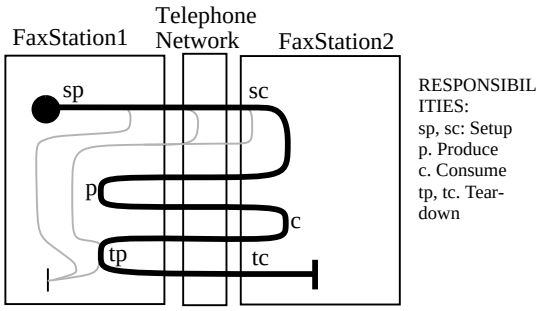


through a telephone network. A type 1 fax station is a human operator and a dumb fax machine interacting through an ordinary telephone handset. A type 2 fax station has a smart, software-controlled fax machine that itself provides the interface to the telephone network. A type 3 fax station is typified by a home office with an ISDN connection in which a software-controlled communications manager controls interactions over the circuits provided by the telephone company.

Use case maps will be used to develop required high-level behaviour patterns without making a commitment to any one of these configurations, but with the idea in mind of binding the behaviour to one or the other later.

This is a sufficiently complex example to illustrate the power of use case maps, because the big behaviour picture can easily be swamped by details at the level of telephony signals and fax machine commands that use case maps defer. The fact that we will be able to describe behaviour at this level without appealing to details of interactions among the human operator, the fax machine, and the telephone network will illustrate the ability of use case maps to represent behaviour as a first-class abstraction, independent of such details. The fact that the example is not (at the level of the diagrams we shall develop), a software example, will help to illustrate that use case maps are technology-independent.

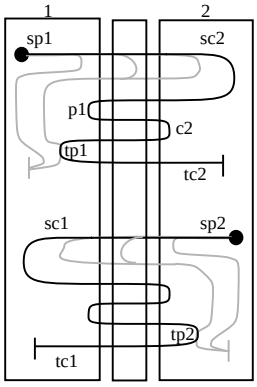
The following block diagram shows a use case map describing a negotiated-producer-consumer pattern to set up one station as a producer and the other as a consumer.



Selected scenarios in this map identify different cases of success or failure of the negotiation (the thick black path is a success path and the thin grey paths are failure paths). The responsibilities *sp*, sc, *tp*, and *tc* mean setup and teardown as a producer or consumer. The responsibilities in this map are very large-grained quantities compared to operator commands, telephony signals, and data transfers (indeed, the completion of the pairs *sp-sc*, *p-c*, or *tp-tc* may involve many back and forth interactions that are below the map level of abstraction, such that the performance of the individual responsibilities of each of these pairs may overlap in time—in other words, the pairs may be shared between the two stations).

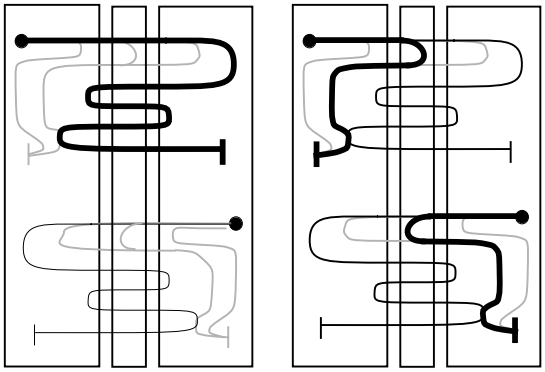
This map is best approached first from the perspective of the success path. This is a path in which station 1 is set up as a producer and station 2 as a consumer (after completion of the sequence *sp-sc*), a produce-consume sequence is completed (*p-c*) and finally the setup is undone at both ends (*tp-tc*), freeing each station to be a producer or consumer in a new relationship. There is an implied mirror-image map that enables station 2 to do the same thing. However, both stations cannot be producers and consumers at the same time, which is the reason for the setup and teardown responsibilities and the existence of the failure paths. Both stations may start down mirror image paths, intending to set themselves up as producers (the occurrence of such race conditions is common in concurrent systems, and understanding them at a high level is an important application of use case maps), but both cannot succeed. The paths in the mirror image maps must interact to prevent this.

The following composite map includes both the original use case map and its mirror image. By symmetry, the top or bottom half of the composite map is sufficient to define the pattern, but it is not sufficient to understand the pattern when confronted with it for the first time.



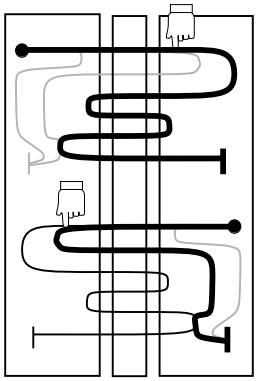
The thick paths in the following copies of this

composite map highlight the most obvious success and



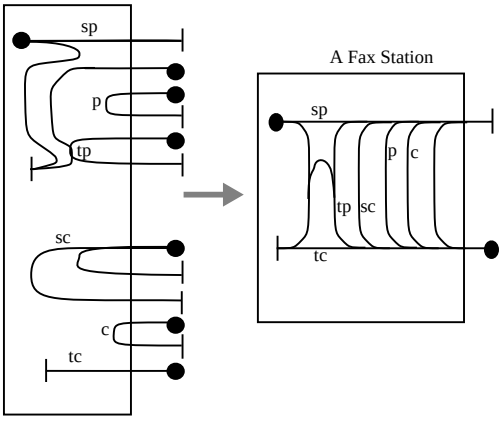
failure scenarios. On the left, success occurs for the side that sets out to be a producer, because the other side is not doing anything. On the right, both sides set out to be producers but both fail, through collision in the telephone network (at a more detailed level, both would get busy signals).

Maps like the composite one above can be used to ask what-if questions about more subtle forms of behaviour. For example, is the composite scenario in the following figure possible (considering that it implies both paths get through the telephone network to the points indicated by the pointing fingers before finalizing the producer-consumer decision)?



It is certainly possible with type 1 fax stations. One operator could accidentally pick up the telephone just as it is about to ring with a call from the other operator, with the result that the operators would find themselves talking to each other. People are able to resolve such collisions by discussion. This is an example of paths being coupled through a shared resource (a telephone connection) and the states of some components (the fact that each operator starts out with the assumption that its state is “producer”). The same kind of thing could happen with a type 3 fax station, with the communications managers resolving the conflicts.

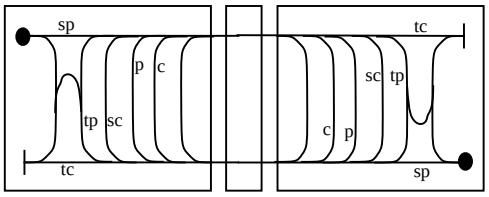
To illustrate that use case maps are easily manipulated to accommodate component decompositions (and, in the reverse direction, compositions), here is how a path-centric view spanning more than one component can be converted into a component-centric view around a selected component by factoring. The steps are: 1) draw a map in which all



paths traversing a component of interest are shown; 2) isolate that component from the rest of the map by cutting the paths traversing it outside its boundaries and terminating the ends; 3) (optionally) tidy up the result by superimposing path segments with common start points, end points, and responsibilities.

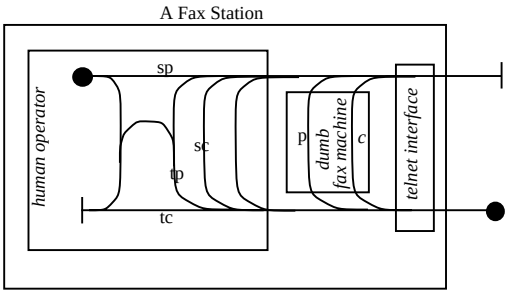
The result is a local map for the component that can be used as a starting point for designing and implementing it as a separate system. The big picture is visually lost in the tidied-up local map (although the assumption is that documentation would provide cross referencing so that it is not actually lost), but the benefit is that the local map is itself a big picture relative to the internals of the fax station.

The factored maps can be reconnected as shown below to resurrect the big picture, but the result,



although more compact than the big-picture map from which they were derived, loses some visual insight.

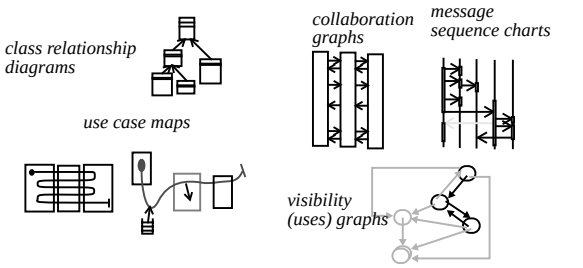
The factored use case map can now be bound to



the chosen internal configuration of the fax station (shown here is a binding for type 1 fax stations). Choosing a type 2 or type 3 fax station would require assigning the setup and teardown responsibilities to internal software components of the black boxes. There is nothing new at the use case map level in doing this.

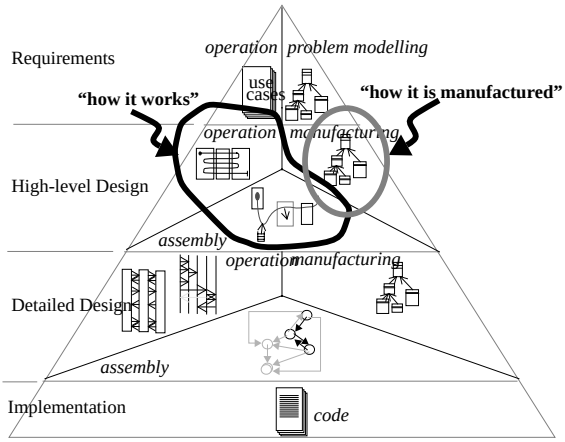
4.0 A Suite of Design Models

Using the following stylized symbols for different



design models, the triangle in the following diagram provides a background for explaining how use case maps are intended to fill a gap in our suite of design models. The gap is indicated by the outlined area labeled “how it works”. In this gap, design magic is normally required. Use case maps do not replace other design models but complement them by filling a gap that normally requires design magic.

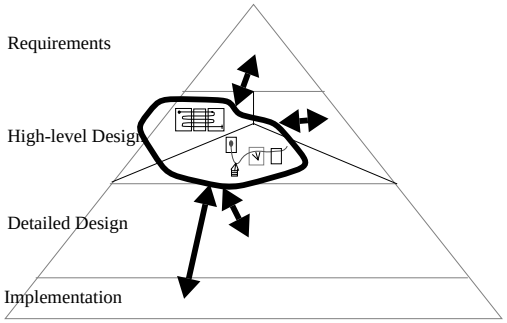
The triangle identifies a hierarchy of design models ranging between most distant from implementation at the apex to implementation at the base. The categories of *operation* and *assembly* accommodate models of different aspects of system behaviour identified in Section 1.0; the category of *manufacturing* accomodates models of how components are to be constructed by class inheritance.



5.0 Directions for Development

Recent case studies of practical systems have confirmed that use case maps are useful for a wide variety of systems, but that their use is hampered by the lack of tools to create and maintain them (which is one reason why the studies are not yet published). Examples studied include a distributed data base system for intelligent networks, a transaction processing system in a distributed multimedia database environment, a distributed end-user-training access system that obtains materials from the internet, a distributed application performance management system, concurrency patterns in distributed object-oriented systems, virtual path testing in ATM networks, a PBX case study that builds a bridge from use case maps to ROOM [4], and reengineering a continuous media file server.

Tools are visualized for preparing, editing, and maintaining use case maps as first-class design models (the outlined area in the diagram below) and for providing links to other models at the same and different levels (the arrows in the diagram below).



Other models may be supported by other tools, in which case intertool links are required. An example of a useful intermodel, intertool link is the association—with paths—of time estimates to execute methods (of classes) that will be used to perform responsibilities along the paths. In object-oriented implementation practice, this kind of information is often not discovered until detailed design and implementation, necessitating the use of expensive cut-and-try procedures. We also visualize maps being formalized to make them analyzable and executable. This would enable, for example, validation of detailed design models against use case maps, or perhaps generation of partial detailed-design models by behaviour-preserving transformations of use case maps.

There are a number of directions for future work, in addition to the development of tools. Systemize methods and documentation of use case maps (like [3]). Systemize connections to use cases and requirements analysis (for example, in relationship to [5]). Develop visualization techniques (e.g., fisheye, 3D) and apply them to explaining complex industrial examples. Develop repositories of design patterns with use case maps (e.g., modeled on [3]). Develop cookbooks for integrating use case maps with other methods. Employ

use case maps to describe how components are intended to operate in the context of systems based on standard components (e.g., ACE [6]).

6.0 Properties of Use Case Maps

Use case maps have the following properties:

- They provide a visual composition of use cases and architectural block diagrams in compact form (they are more than just a visual notation for use cases).
- They are first-class design models that may be used as equal partners with other models such class relationship diagrams (used in object-oriented design).
- They attribute behaviour to architecture at a level above wiring.
- They can be used as a starting point for designing wiring rules for architectures, and individual wirings.
- They scale up better than representations that depend on details at the level of wiring.
- They are technology-independent, making them applicable in a uniform fashion across a wide range of system types, from sequential object-oriented programs to parallel and distributed computer systems.
- They give a system-wide view in a compact way that exploits human pattern-recognition abilities.
- The absence of wires enables them to represent total behaviour (operation + assembly) in a lightweight fashion.
- They can be concrete work products for design traceabililty and reuse of high-level design patterns.
- They can be used to characterize architectures in terms of such properties of path patterns as regularity, knotting, and clustering that may relate to robustness of architectures.
- They may be used to develop peformance measures of architectures in time-along-path terms.
- They have semantic depth that enables them to be used to ask what-if questions about large-grained behaviour patterns and architectures to realize them.
- There is no free lunch: the high level view loses sight of details.
- The lunch may be worth the price because they replace high-level design magic with a rational and traceable progression from requirements to the design of details with popular methods/tools.
- They are most useful for systems with predominantly point-to-point stimulus-response patterns (in other words, systems in which the effects of stimuli do not spread throughout the system like ripples on the surface of a pond).

7.0 Conclusions

The ability to attribute behaviour to architecture is important for high-level understanding, designing, evolving, and reengineering all sorts of systems (from object-oriented programs to parallel and distributed computer systems). Use case maps provide a new, scenario-based way of attributing behaviour to architecture that scales up well as a result of the simple trick of representing scenarios in terms of cause-effect sequences of responsibilities along paths, instead of interaction sequences traversing wires between components. The notation enables compact, composite maps to be drawn to represent behaviour patterns of whole systems in path terms. The paper described highlights of the approach, its relationship to other approaches, work in progress, and issues and directions for future work. Through an example, the paper aimed to convince software and system engineers that the approach has depth and adds value, despite (and because of) its simplicity and deferment of detail.

Acknowledgments

NSERC and TRIOSOFT are providing current financial assistance for aspects of this work. BNR provided financial support for some of the initial development of the ideas. Colleagues, industrial collaborators, and many students, too numerous to list, have helped greatly by performing interesting case studies of significant applications, doing work to extend the ideas, and asking difficult questions.

References

[1] R.J.A. Buhr, R.S. Casselman, T.W. Pearce, *Design Patterns with Use Case Maps:A Case Study in Reengineering an Object-Oriented Framework*, SCE 95-17, http://ftp.sce.carleton.ca/UseCaseMaps/dpwucm.ps.

[2] R.J.A. Buhr, R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[4] B. Selic, G. Gullickson and P.T. Ward, *Real-time Object-Oriented Modeling*,Wiley, 1994.

[5] B. Regnell, M. Andersson, J. Bergstrand, *A Hierarchical Use Case Model with Graphical Representation*, Proc. ECBS96, IEEE Second International Symposium and Workshop on Engineering of Computer Based Systems, March 1996.

[6] D.C. Schmidt, *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*, 11th and 12th Sun User Group conferences, San Jose, CA, Dec 7-9, 1993 and San Fransisco, CA, June 14-17, 1993.

[7] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.