A High Level Visual Notation for Understanding and Designing Collaborative, Adaptive Behaviour in Multiagent Systems

R.J.A. Buhr†, M. Elammari†, T. Gray‡, S. Mankovski‡

Carleton University† and Mitel Corporation‡

buhr@sce.carleton.ca,elammari@scs.carleton.ca,tom_gray@mitel.com,serge_mankovski@mitel.com

Abstract

We explain how to use a high level, visual notation called Use Case Maps (UCMs) to bring together the "what" and "how" of multiagent systems, for understanding and design. "What" refers to descriptions of what agents do, descriptions that are declarative from a detailed software design perspective (e.g., BDI models). "How" refers to descriptions of how the software does it, expressed with software design notations. Two important properties of agent systems that make them difficult to understand and design are multiagent collaborative behaviour and adaption by system self modification. BDI-style descriptions of what must be done to achieve these properties do not give a direct view of the properties, but leave them to emerge. Conventional software design descriptions swamp us with unnecessary and undesirable detail relative to both system properties and BDI-style models. UCMs were invented to raise the level of abstraction of software design in precisely the way required to overcome these problems. UCMs are particularly suitable for multiagent systems because they bring together the "what" and "how" of collaborative behaviour and system self modification in a coherent way, in a single high level visual notation. A companion HICSS'98 paper titled "Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example" illustrates the approach.

1. Introduction

This paper and a companion paper [7] come out of a research project on high level design and prototyping of agent systems [5]. This project is applying use case maps (UCMs) [8][9][10][11][12][13] as part of a new approach to agent systems. This paper makes a new contribution to both UCMs and agents by developing a concept of dynamically pluggable behaviour patterns that is missing from previous UCM publications and by showing how this concept can be applied to get a *system* view of collaborating agents in terms that relate directly to well known BDI (Beliefs-Desires-Intentions) [18] agent models. The contribution is to a new way of describing multiagents systems to get a better handle on understanding and designing them *as systems*, not to new ways of implementing agents. This paper concentrates on principles illustrated with very simple examples, leaving more realistic examples to [7]. A report [8] describes a multimedia communications example that was included, in part, in the initial version of this paper but had to be removed from this final version for lack of space. Another report [6] explains more than we have space for here about the application of UCMs to large, complex, self modifying systems. The concept of dynamically pluggable behaviour patterns presented here is covered in [6] along with much more about UCMs; however, [6] does not cover the relationship between UCMs and agents presented here.

A subset of the elements of the UCM notation is summarized in Appendix A, although not all elements in Appendix A are used in this paper (see [7] for use of other elements and [6] for the rest of the notation).

1.1. Motivation for Developing a Relationship Between UCMs and Agents

Software agents are being seen as a possible way of building distributed systems that are adaptable to changing user needs, environmental disturbances, new services and new technology. The prospect is attractive but has the down side that systems composed of multiple adapting, evolving agents are likely to be complex and therefore difficult to understand and design *as systems*.

One property that makes systems of agents complex is multiagent collaborative behaviour. A second is adaption through system self modification. A third is the interwining of these two properties on the same time scale while the system is running.

Adapation is required to accommodate environmental contingencies and changing user needs. Adaption is accomplished by system self modification in two ways: agents appear, disappear and move around the system; and behaviour patterns in and among agents change over time to achieve the same objectives under different system conditions. Self modification occurs on the same time scale as ordinary multiagent collaborative behaviour, and so becomes intertwined with it. The result is particularly hard to understand and design, because the structure of the system—the set of components present in the system (agents, resources, and so forth) and their relationships to each other—changes as we look at it, while the collaborative behaviour we are trying to understand is taking place (it can be helpful to think of this as analogous to "morphing" in computer

animation).

It is helpful to distinguish *system self modification*, as identified above, from code self modification. Applied to code, the term “self modification” means that code may sometimes treat parts of itself as data that can be modified and passed around. For as long as software has existed, self modifying code has been well known to be difficult to understand. Extending the concept of self modification to systems means that systems may treat their components as data (components such as agents, processes, and objects); and not only their components, but their behaviour patterns as well.

The distinction is useful because code can be locally self modifying without it being visible as a change in structure or behaviour at the system level. It is only when self modification is visible at the system level—and is intertwined with ordinary collaborative behaviour at this level—that the complexity issues in which we are interested arise. Fixed systems are hard enough to understand, because collaborative behaviour emerges from the details of the components and we must piece the details together to understand the whole. However, we can at least draw diagrams that show fixed system structures and trace the collaborative behaviour through these structures. Having everything changing at once complicates matters.

In the well known BDI model for agent systems [18] adaption is achieved through modification of beliefs, desires, and intentions over time. The beliefs, desires, and intentions express *what* must be done by the software of the system, as a set of requirements, without saying *how* the software will do it. As software designers, we must be concerned with *how* the software will do it. Unfortunately, popular low level software design models swamp us with unnecessary and undesirable detail relative to BDI models (detail at the level of objects, processes, methods, messages, state machines, and variables). Self modification at the system level and multiagent collaborative behaviour at the system level become so swamped in a sea of detail that the big picture is lost. With these techniques, the big picture must be designed and understood through the details.

We offer Use Case Maps (UCMs) [9][10][11][12][13] as a solution to this problem (Figure 1). UCMs provide a high level visual description that combines “what” and “how” in a high level way. We shall show that UCMs give visual expression to important elements of agent BDI models (“what”). We shall show that UCMs also provide a coherent view of multiagent system structure and behaviour (“how”) that expresses both collaborative behaviour and self modification in a high level manner, above the level of software details. We shall indicate that the gap between UCMs and more conventional software design models can be bridged systematically and give a brief indication of how (see [11] for more).

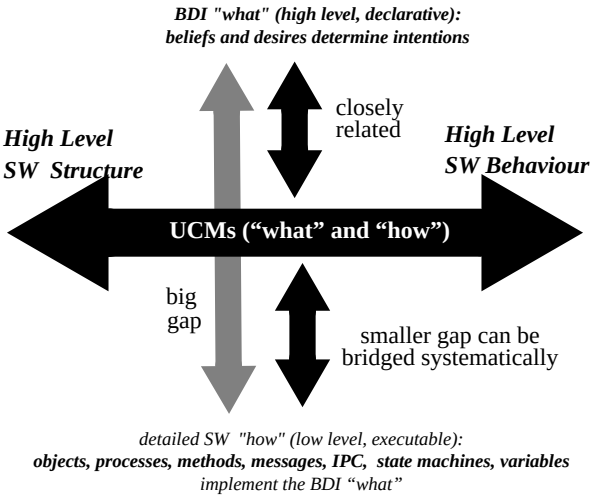


Figure 1 UCMs and the BDI model of agents.

2. UCM Basics

Jacobson [16] popularized the concept of *use cases* for object-oriented software engineering. A use case is a prose description of one or more related *scenarios* that explains how a system works from a user perspective. The term *user* is not restricted to human users interacting with system externals—in general, the term includes the possibility of scenarios involving many systems or subsystems that are “users” of each other. Use cases may, in general, have many related scenarios. Every use case has at least one. We often use the term scenario instead of use case, understanding that we mean one of the scenarios associated with a use case (if there is only one, the terms mean the same thing). In general, we use the term scenario to mean a description of either required behaviour for a system not yet constructed, or of observed activity for an actual running system.

UCMs (Figure 2) are maps showing the paths of use case scenarios through some underlying system. This is an *unbound* UCM because it shows no system components (we leave them out to introduce the path concepts first) and it is a particularly simple example

because is traces only one scenario: getting money ($)

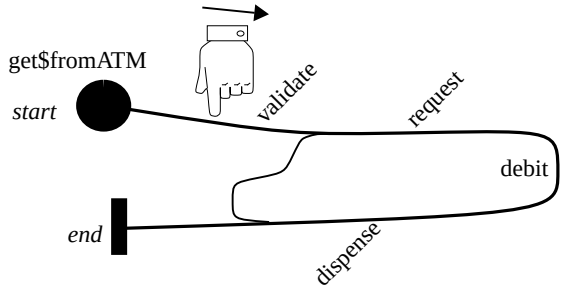


Figure 2 An unbound UCM.

from an automated teller machine (ATM). The pointing finger with the associated arrow suggests how to interpret the UCM (by tracing a path through it with your finger) but is not part of the UCM. Think of it as a token that must be mentally moved along the path to interpret the UCM. If the pointing finger was replaced with a highlighter pen, the result would be analogous to a highlighted journey on a road map.

In Figure 2, the main path from start to end shows a validate-request-debit-dispense sequence. The fork after dispense and the join before request indicate that a validated scenario may make more than one request before ending, in other words that the request-debit-dispense sequence may be repeated within the same scenario. The labels *validate*, *request*, *debit*, and *dispense* identify *responsibilities* of components in the underlying system (in this case the ATM system). An unbound UCM does not show the components, but their existence is implied. The path chains the responsibilities together in *causal* order. We say *causal* because there is nothing in the UCM to suggest *how* the sequence is to be achieved in a detailed way. From a detailed software design perspective, a UCM indicates *what* must happen, without indicating *how* (this is why UCMs are natural for expressing BDI models visually).

An important property of UCMs is their ability to condense information about many scenarios into a single understandable diagram. A hint of how this is done is given by the forked path after *dispense* in Figure 2. The fork means that this UCM represents more than one possible scenario. In this case, the different scenarios are all variations of get$fromATM, but, in general, quite different scenarios with different start and end points might follow shared subpaths for a while.

Obviously, we do not need a diagram just to *specify* responsibility sequences such as validate-request-debit-dispense (a textual list would do). The usefulness of these diagrams comes when we want to project such sequences onto the underlying components of the system, for example, agents and resources. Basically, UCMs are diagrams that project use case paths onto components of systems in a high level way. The result is called a *bound* map (Figure 3, which shows the same get$fromATM scenario as earlier). The component substrate onto which the paths are projected gives the shape to the paths.

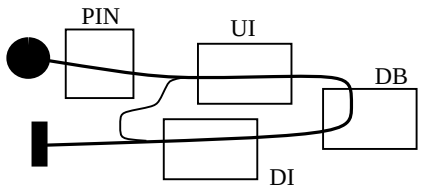


Figure 3 A bound UCM.

The UCM notation deliberately has no graphical symbols for responsibilities because such symbols would result in bound maps being visually cluttered. Responsibilities are simply points along paths or on components. Responsibility labels are then easily either shown, or omitted (as in Figure 3) to get an uncluttered overview that is helpful to human viewers. In this particular diagram, an observer can easily hold in the mind that the PIN component has the *validate* responsibility, the UI component the *request* responsibility, the DB component the *debit* responsibility, and the DI component the *dispense* responsibility.

The basic assumption of UCMs is that the projection of use case scenarios onto a component substrate can be represented in a simple way with paths. This is a very common characteristic of the types of systems with which we are concerned.

The path notation (Appendix A) may be used to create compound maps reflecting multiple scenarios that may share path segments, have internal concurrency, be concurrent relative to each other, and have different forms of coupling, both designed and inadvertent (e.g., due to races, conflicts, and resource contention). See [7] for examples of conflicts.

In a compound UCM, individual paths may cross many components and components may have many paths crossing them (Figure 4), giving a path-centric view that provides a start on developing component centric views.

As path-centric descriptions, UCMs push down details to give us a bird's eye view of a system as a whole (notice that we do not need interfaces, messages, or internal logic of components to describe the path). A segment of a path that crosses a component only tells us that the component must perform the responsibilities along the segment within the component boundaries, but not how the component will discover or be told that it must do this. A segment of a path that joins responsibilities on two different components tells us only that the components must collaborate (directly or indirectly) to achieve the causal sequence, but not how they will interact to accomplish this. In general many possibilities for interaction exist, such as exchanging a series of messages, starting with either component, or interacting through some underlying layer not shown.

If a component in a diagram such as Figure 4 is a black box view of a subsystem that is meant to contain internal components, then the same ideas apply recursively to it. The component-centric view of it becomes a path-centric view at the next lower level of decomposition. The fact that such decompositions may be developed without committing to component interfaces makes relatively lightweight work of exploring design alternatives down through several levels of decomposition.

If a component in a diagram such as Figure 4 is primitive—meaning it has no further levels of component decomposition underneath it—the path centric view of it gives a starting point for developing its internal logic to implement the path segments crossing it. This requires recourse to design models at lower levels of abstraction than UCMs (see Section 4 and Section 5 for more on this).

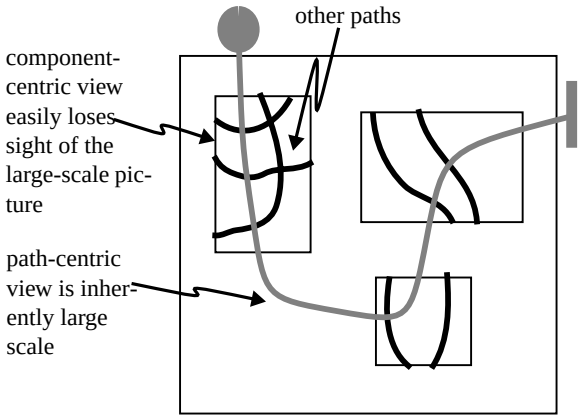


Figure 4 Path-centric and component-centric views

Of course, each path in a UCM only reflects an *example* of system behaviour and a set of examples cannot, in general, no matter how carefully chosen, provide a complete specification for a system. However, examples give insight. To get as close as possible to complete component-centric specifications, one tries to develop UCMs in which the path-centric view of a set of components covers all the important path segments in the component-centric view of each component. Although the result may not be a complete specification of the whole system, it provides a good starting point for developing the details of each component, with the added advantage of being tied directly into a system-level description. This is good design information, as well as being helpful for system understanding.

We have found that UCMs can exploit human visual pattern recognition ability to aid system understanding, by arranging components and paths into recognizable visual forms that can be understood and reasoned about by those familiar with a system, even without textual labels.

In addition to the visual aspect, every UCM has separate documentation that includes information such as descriptions of elements in the map, information about the scenarios, such as preconditions and postconditions, and information about relationships to other maps.

This overview has only scratched the surface of UCMs. See [6] for more.

3. UCMs and BDI Models of Agents

Visual techniques are well known to be needed for human understanding. However, in both the agent community and other communities, such as the object-oriented community, there is a big gap between models at the BDI/UCM level of abstraction and popular software design notations. Referring back to terminology used in Figure 1, software design notations express “how” in a detailed way, leaving “what” to emerge from the details. For example, Kendall [17] models agent systems using object-oriented-style diagrams that require commitment to agent-centric details (e.g., pairwise interactions via messages, pairwise inheritance relationships). There are many tools in the object-oriented community that support modeling of systems at this level of detail (for example, [4][21]). In the agent community, the Clearlake tool [15] specifically supports design of agent systems at a similar level of detail. Workflow models (e.g., as discussed and applied by Kendall [17]), while less detailed than the design models just mentioned, are more detailed than UCMs because intercomponent workflows must be defined. None of these models provides an easy means of describing agent system self modification.

When we depart from the world of low level software design models for agent systems, what remains is abstract textual description of “what” without any visual representation of “how” in software. Examples are COOL [2] and Shoham’s AOP [23], which represent agents formally with logic.

Thus we have two styles of description in common use in the agent community, high level, non-visual “what” descriptions and low level, visual “how” descriptions, with nothing in between. UCMs can be used to bridge the gap (see Figure 5, which continues the ATM example).

During forward engineering, the progression is from BDI models to UCMs to detailed software design models. During reengineering, UCMs may be used as high level frameworks for changing software details, or the UCM descriptions may themselves be changed to accommodate new requirements (we mean requirements that change BDI models, not adaption within the framework of an existing BDI model—the latter is expressible directly with UCMs by means about to be explained).

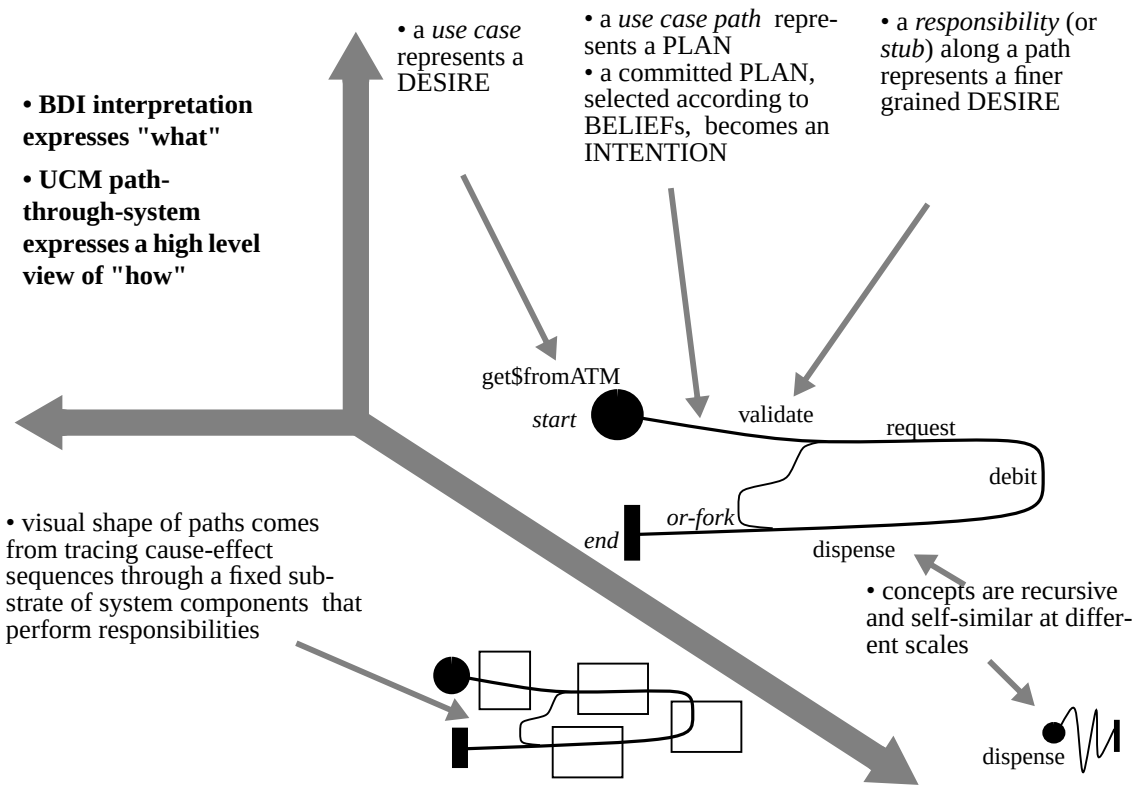


Figure 5 From UCMs to BDI models.

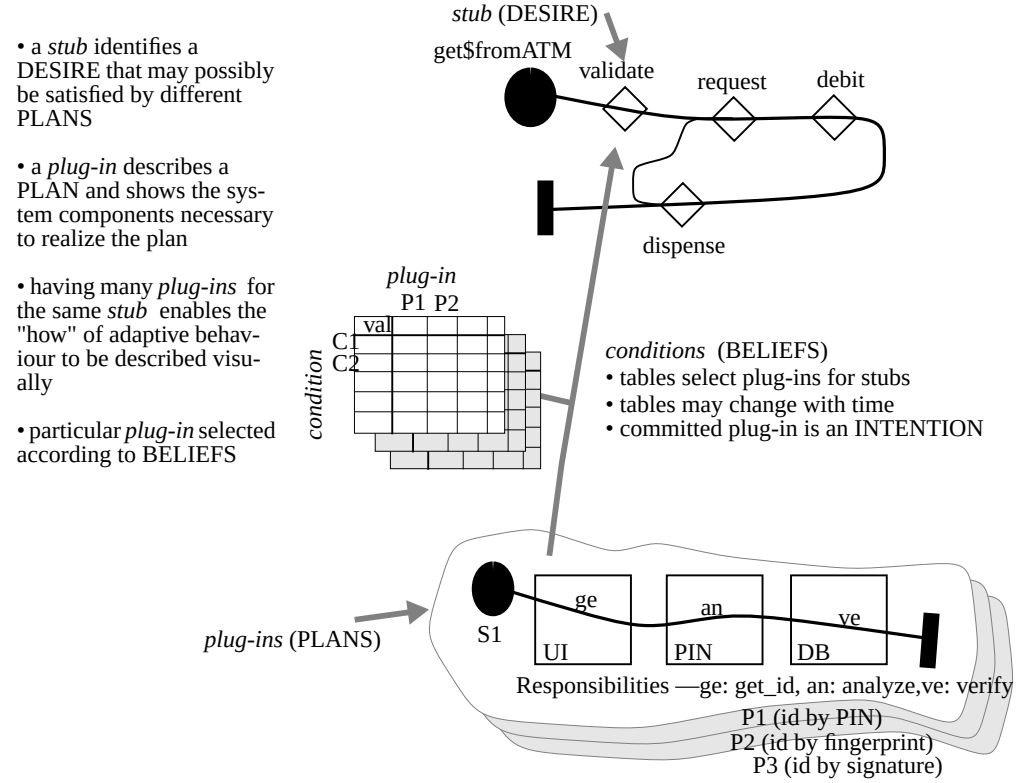


Figure 6 Concept of dynamically pluggable behaviour patterns.

UCMs can be used to express BDI-style adaption visually (Figure 6). Suppose changing beliefs require different plans to be selected to satisfy desires. We indicate this visually by showing the desires as stubs (diamond shapes) rather than just labelled points. Stubs (BDI desires) are points where details are defined by submaps, called plug-ins (BDI plans). Different plugs-ins, possibly with different paths and components, may be selected (selection amounts to committing to BDI intentions) for different system conditions (BDI beliefs). (UCM stubs may have multiple entry and exit paths, not just single ones as shown here, to allow multiple paths to be coupled through stubs. Plug-ins for such stubs are then fully general UCMs with multiple start and end points.) The conditions that select a plug-in may change over time, causing different plug-ins to be selected at different times. The preconditions and postconditions of the paths, and the conditions that determine selection of plug-ins, are related to the belief sets of agents. In general, conditions could be such that more than one plug-in could be a candidate for the same stub, in which case there may be a need to resolve conflicts between plug-ins. We are working on ways of using UCMs to identify possible conflicts of this kind, using telephony feature interaction as an example (preliminary work is reported in [7]).

Note that component boxes identify components that operate where shown, but stub boxes represent only path decomposition; any components are left to the decomposition.

The concept of dynamic plug-ins is a new one relative to earlier UCM work. Stubs were originally viewed in [11] as a static decomposition technique for paths analogous to the way we statically decompose components by drawing black-box and glass-box diagrams of them. Stubs were viewed as analogous to black-box components and plug-ins as analogous to their internals shown in separate glass box diagrams. UCMs have from the start had the concept of dynamic component pluggability ("slots" identify places where this may occur). However, bringing UCMs together with agents made us realize that dynamically pluggable stubs are just as useful as dynamically pluggable slots (slots are effectively component stubs). This concept is useful for agent systems because it captures in system terms the concept of dynamically selectable plans.

Figure 7 shows a mechanism, expressed with a UCM, for implementing the concept of Figure 6. This is a standard pattern that would normally be assumed for all cases. The new notations here are borrowed from the standard UCM notations for dynamically pluggable *components* (dashed outlines indicating dynamic pluggability, pools for storing elements to be plugged in, and small arrows to indicate movement of the elements). The possibility of having to abort an active plug-in is indicated (the notation also offers the option of specifying an exception path to clean up after the abort).

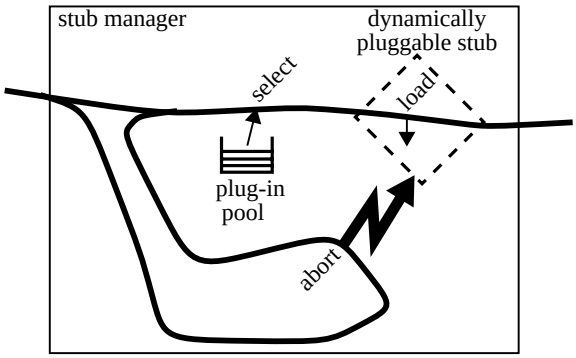


Figure 7 A mechanism for dynamically pluggable stubs

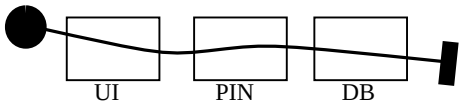
4. From UCMs to Design Models at Lower Levels of Abstraction

For multiagent systems, paths of UCMs imply collaborative behaviour of multiple agents. In other words, the agents are intended to collaborate to realize the path sequences. However, the high level nature of UCMs defers the question of exactly how this is to be accomplished in detail. Figure 8 shows how the P1 plug-in for the validate stub of Figure 6 might be translated into a detailed design. The path of the plug-in through the component substrate may be *implemented* in different ways that differ only in details, as illustrated at the bottom of Figure 8 (in the style of object-interaction diagrams).

Each of the bottom configurations in Figure 8 commits to a different set of component interfaces and a different set of interaction possibilities among the components. Some of them even add a new control component. Yet the causal path they implement remains the same. For static plug-ins, implementing the chosen configuration would simply be a matter of programming the interactions directly. Dynamic plug-ins require implementing a mechanism such as that of Figure 7. How to implement such a mechanism is a subject of current research (see [7] for an example in which the mechanism must resolve conflicts between plug-ins).

Notice in Figure 8 that the path is no longer explicit in the detailed design diagrams, but is implicit in the internal details of the components. To see the paths in the details we must consult the details to reconstruct the path in our minds. This is the kind of commitment to details required by conventional software design models that swamps us in unnecessary and undesirable detail for software design at the level of agent systems

and BDI models.



**plug-ins select parts of substrate,
causal nature of path sequences defers
control/interaction details**

**contrast with object-interaction-style diagrams
(a few of many possibilities)**

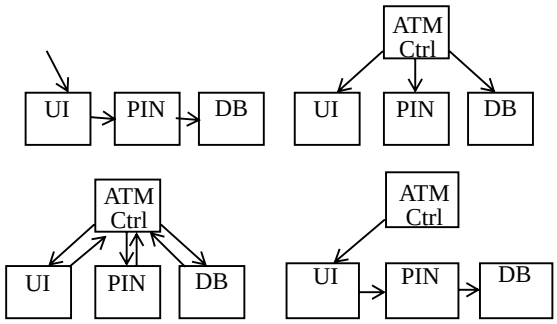


Figure 8 UCMs defer details.

5. Discussion

5.1. Why a New Notation?

We have said that agent systems are complex because they combine, at the system level, multiagent collaborative behaviour with self modification. Systems that combine these properties are particularly hard to understand because their structure changes as we look at it, while the collaborative behaviour that we are trying to understand is taking place.

However, why do we need a new notation? We have tried to explain why in the previous sections and will now try to summarize the rationale.

Popular software/system design notations written up in textbooks and standardization documents and possibly supported by CASE tools (for example, [3][4][14][16][20][21]) are unable to give a *bird's eye view* of end-to-end causal sequences running through large, complex, self modifying systems. With notations such as these, a bird's eye view must be composed in the mind's eye by mentally combining details. Experts are able to do this to form personal, idiosyncratic mental models that, although helpful to guide their thinking, cannot be easily written down or communicated to others; non-experts are left out in the cold. What distinguishes UCMs from other notations is the way they explicitly join the above elements together in a compact visual form that can easily be held in the mind's eye and communicated to expert and non-expert alike.

It is true that many notations are available for describing systems and sequences and there is nothing fundamentally new in the simple fact of being able to describe such things with UCMs. A comparable statement in the programming language field would be that there is nothing fundamentally new in high level languages because assembly languages are capable of expressing anything that high level languages express. Such statements miss the point. What is new in both cases is expressiveness from a human perspective.

UCMs provide a significant reduction, relative to other techniques, in the number of diagrams required to get a bird's eye view of a system. Unfortunately, paper-sized examples are never sufficiently large scale and complex to *prove* this. There is never sufficient space and, even if there was, very few readers would be interested in seeing the details. Therefore some imagination must be exercised when studying examples such as those in [7][8][10][13]. Imagine scaling up the examples such that inch-thick stacks of conventional diagrams would be required, because this is representative of large, complex systems in practice. Then it is not difficult to imagine how UCMs can be useful as a way of condensing the essence of a system into many fewer diagrams.

Some discussion of a few examples of other notations for describing systems and sequences may be helpful:

- So-called "high level message sequence charts" under development by the Z120 community [19] are an attempt to give a bird's eye view of sequences by providing road maps through more detailed sequence charts. However, they are still sequence charts, separate from diagrams describing component organizations, forcing the viewer to piece together mentally two different types of diagrams, each with their own box notations, to get a UCM-like picture. In a formal sense, UCMs and the result of this piecing together provide similar information. However, UCMs condense it into a more compact visual form.
- Data flow diagrams (DFDs) show data precedence relationships in networks of data processing units that feed data to each other. One way of using DFDs is to give an overview of data relationships in conventional systems. In such cases, UCMs and DFDs can provide useful complementary views (e.g., UCMs can show causal sequences running through data processing units). A use of DFDs that would not be helped much by UCMs is specifying data flow machines (hardware or software) that will perform data processing with as much parallelism as possible (e.g., some signal processing systems).
- Petri nets provide a visual formalism for executable specifications of concurrent systems. They focus on specifying details that will allow behaviour to emerge. The emergent behaviour can be traced to reveal UCM-style paths, but the paths are not first class elements in the specifications. In

relation to UCMs, Petri nets are best viewed as one of many ways of developing executable models from UCMs.

- Workflow models and process models may have relationships to UCMs that remain to be explored. However, we remain confident that UCMs are a useful contribution, independent of what may have been developed in these fields. The UCM approach was developed from the ground up out of frustration with existing techniques in the software design field, the approach is novel in that field, the field is sufficiently complex in its own right to challenge any approach, and UCMs have, so far, met the challenges.

5.2. Process Issues

UCMs provide a notation and a way of thinking that can accommodate many different development processes in combination with other techniques.

The reader can see that UCMs are at a higher level of abstraction than popular software design models. Here is a summary of the ways in which this is so:

- Paths show causal sequences, above the level of intercomponent interactions.
- Responsibilities are named points, without any visual indication of parameters or results, and without any commitment to how they are implemented, e.g., as functions, methods, or sets of them.
- Components are represented in a very simple way, only by named boxes with associated responsibilities. There is no commitment to interfaces, protocols, methods, messages, state machines, or anything else that defines how components are implemented, how other components interact with them, or how quantities are transferred between components.
- Details that are part of UCM interpretation (e.g., nature of responsibilities, preconditions and postconditions of paths) are expressed informally in prose.
- Many details are missing, such as the explicit representation of data as a separate element from components (basically, if you want to indicate data storage, you show a component that stores it, and if you want to indicate data flow or the data state of a component, you imply it through prose descriptions of the notational elements).
- Implementation details are entirely missing, except for indicating the *existence* of components.

UCMs *supplement* other descriptions at lower levels of abstraction and provide traceability to such descriptions (and back from them). Figure 9 gives some insight into how. In this diagram, the steps indicated by the thick grey arrows start with basic information from the UCM to which details are added by human designers. From a tool perspective, the process would be one of machine assistance, not machine translation. Here, design patterns (in the sense the term is used in the object-oriented community) could enter the picture to provide standard ways of filling in details.

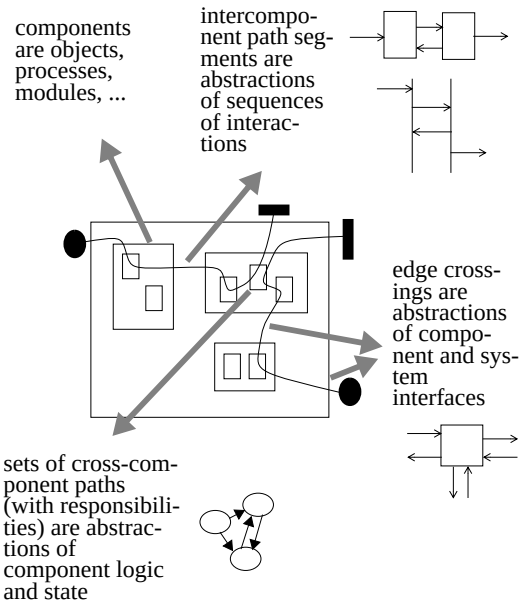


Figure 9 From UCMs to detailed design models.

On first exposure to the UCM way of looking at things, people may be made uncomfortable by the absence of detail in UCMs and may want to fix the notation by adding “missing” elements, but Figure 9 shows that other models are intended to cover these elements. Adding them to UCMs would be self defeating. The notation—as is—seems about right for its intended purpose, namely presenting a human-understandable bird’s eye view of a system as a whole, including both its structure and its main behaviour patterns.

Have UCMs been used on a large project? In an informal way, yes, but nothing has been reported in the public domain.

When do UCM diagrams become too large to maintain intellectual control over them? The well known psychological fact that humans can hold roughly seven things in the mind at once suggests—rounding seven up to ten—that no more than ten scenario paths through ten components might be a practical limit on the comprehensibility of UCMs. No large scale complex system can be expressed this simply, although its essence might be expressible this way at the highest level of decomposition and layering. Decomposition and layering techniques described elsewhere [6] must be employed.

Would the descriptions be shared within an organization? Yes, we intend UCMs to provide a shared reference point for the development and understanding of more detailed models. We see some UCMs as providing design patterns of a high level kind [12].

Is there a potential for modeling the same behaviour redundantly and differently by various organizational components? Yes, UCMs supplement other

techniques and may be used in combination with them in any way that makes sense.

5.3. UCMs and Agents

We have attempted to convey how UCMs deal with agent issues at a high level of abstraction, as follows:

- Paths express multiagent collaborative behaviour, without committing to how it will be implemented. In other words, the paths indicate that agents along them must collaborate to realize the path sequences, without saying how the agents will do this, for example, by exchanging messages.
- Stubs and plug-ins are used to represent self modification in a way that mirrors BDI models. They give a visual representation of the way in which changing belief sets cause both the components involved in the collaborative behaviour and the collaborative behaviour itself to change. Thus, they join "what" (in the sense of BDI models) and "how" (in the sense of system design) in a high level way.
- The notations for system self modification are particularly compact and helpful compared to other techniques. They combine self modification with ordinary collaborative behaviour in a coherent way that keeps them clearly distinct while enabling them to be understand together. It enables us to have our cake (self modification) and eat it too (express collaborative behaviour in terms of fixed diagrams).
- The notation expresses "what" (e.g., dynamic components and plug-ins are stored and move as data) without indicating "how" (e.g., exactly what the software mechanism is, and whether the movement is a "pull" from one side or a "push" from the other). On the other hand, it also expresses a measure of "how" by showing how system components are involved in a high level way.

We used the BDI model for illustrative purposes because it is well known and forms the basis for a number of extensions. The industrial authors of this paper are directly involved in developing an agent framework that is intended to allow software interoperability across enterprise applications. For this purpose, the BDI model has been extended with some new concepts [1][24][25][26]. However, this does not change any of the principles explained so far. The details are outside the scope of this paper. However, here is the general idea. The framework provides for a dynamic matching of overall enterprise goals with enterprise resources through a negotiation process among agents. An *understanding* is a prelude and guide to *negotiation*. Through negotiation a *contract* is established in which the objects required to fulfil the understanding are supplied to the *motivated agent* by an agent with appropriate *authority.* This allows the decoupling of goals from resources and hence allows services to be defined independently of the enterprise's specific resources used to accomplish the goals. The result is expected to be a system which is evolvable for both new services and new technology. Current research is focusing on ways of moving systematically between system views with UCMs and agent internal models that implement the above ideas.

6. Conclusions

UCMs and agents are a good fit for a number of reasons. Agents are causal entities; UCMs describe causal sequences. Agent systems are self modifying and distributed; UCMs express these properties in a first class, unusually compact and easily understood manner. UCM concepts of stubs and plug-ins are easily mapped to agent BDI concepts. The discovery and modeling phases of agent *systems* are difficult with conventional software design techniques because of large conceptual gap between them and BDI concepts. UCMs bridge the conceptual gap for definition and instantiation of agent systems with extended BDI concepts. UCMs are particularly suitable for multiagent systems because they are capable of expressing the "what" and "how" of collaborative behaviour and system self modification in a coherent way in a single, high level, visual notation. UCMs are an original and unique approach that contributes to the development of agent-oriented programming as a discipline.

References

[1] S. Abu Hakima, I. Ferguson, N. Stonelake, E. Bijam, R. Deadman, *A Help Desk Application for Sharing Resources Across High Speed Networks Using a Multi-Agent Network Architecture*, Workshop of Distributed Information Networks, IJCAI 95, Montreal.

[2] M. Barbuceanu, M.S. Fox, *COOL: A Language for Describing Coordination in Multi-Agent Systems*, In Proceedings of the International Conference on Multi-Agent Systems, San Francisco, CA, 1995.

[3] G. Booch, Object-Oriented Design, Benjamin/Cummings, 1994.

[4] G. Booch, J. Rumbaugh, *Unified Method for Object-Oriented Development*, Documentation Set, Version 0.8, Rational Software Corporation, 1995.

[5] R.J.A. Buhr, *High Level Design and Prototyping of Agent Systems*, Research project description, www.sce.carleton.ca/rads/agents

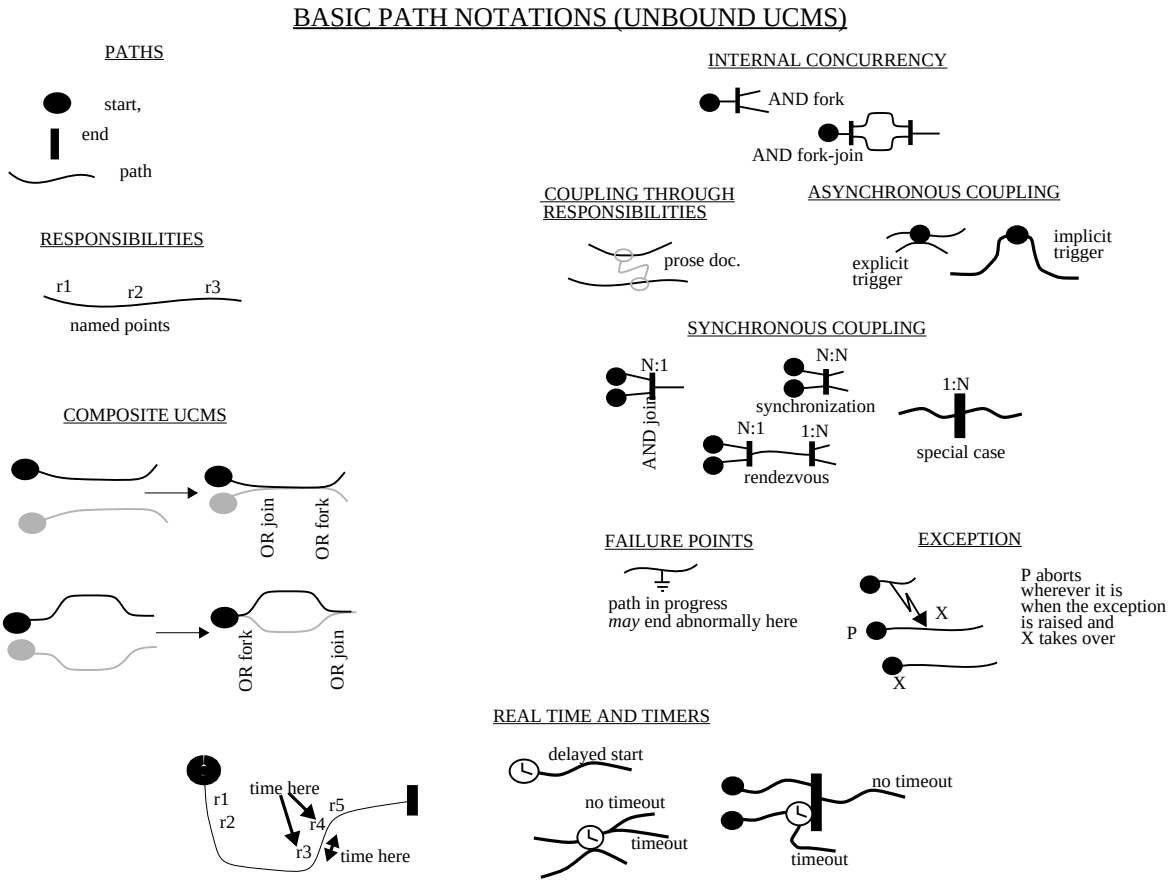
[6] R.J.A. Buhr, Use Case Maps Updated: *A Simple Notation for Architecting the Emergent Behaviour of Large, Complex, Self Modifiying Systems*, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmUpdate.ps

[7] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, *Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example*, to appear in Proc. HICSS'98, Hawaii, January 98.

[8] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, D. Pinard, *Understanding and Defining the Behaviour of Systems of Agents, with Use Case Maps*, Report (presented as a Poster Ses-

sion at PAAM'97), http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam97.ps
[9] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, SCE-96-2: September 23, 1997, Contribution to the Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii, http://ftp.sce.carleton.ca/UseCaseMaps/attributing.ps.
[10] 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.
[11] R.J.A. Buhr, R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.
[12] R.J.A. Buhr, *Design Patterns at Different Scales*, presented at PLoP96, Allerton Park Illinois, Sep 96. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps.
[13] R.J.A. Buhr, A. Hubbard, *Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE-Framework Application*, Hawaii International Conference on System Sciences, Jan 7-10, 1997, Wailea, Hawaii, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss-final-public.ps
[14] CCITT Recommendation Z120: Message Sequence Charts (MSC), undated document.
[15] Guideware Corporation, *The Implementation of Business Processes with Mobile Agents*, Technical Paper, Guideware Corporation, Mountain View, California, 1995.
[16] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.
[17] E. Kendall, *A Methodology for Developing Agent Based Systems for Enterprise Integration*, IFIP Working Conference of TC5 Special Interest Group on Architectures for Enterprise Integration, Queensland, Australia, November 1995.
[18] A. Rao, M. Georgeff, *BDI Agents from Theory to Practice*, Technical Note 56, AAII, April 1995.
[19] 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.
[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modelling and Design*, Prentice Hall 1991.
[21] B. Selic, G. Gullickson and P.T. Ward, *Real-time Object-Oriented Modeling*, Wiley, 1994.
[22] Shaw and Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.
[23] Y. Shoham, *Agent-Oriented Programming*, Artificial Intelligence, 60(1), pp 51-92, 1993.
[24] M. Weiss, T. Gray, A. Diaz, *A Middleware for Developing Distributed Multimedia Applications*, Proceedings of the ISCA International Conference on Parallel and Distributed Computing, Orlando 1995.
[25] M. Weiss, T. Gray, A. Diaz, *An Agent Based Distributed Multimedia Service Environment*, International Conference on Tools with AI, Herndon VA, 1995.
[26] M. Weiss, T. Gray, A. Diaz, *A Service Environment for Distributed Multimedia Applications*, Workshop of Distributed Information Networks, IJCAI 95, Montreal.

Appendix A: Selected Subset of the UCM Notation



DYNAMICALLY PLUGGABLE COMPONENTS
(ADAPTED IN BODY OF PAPER FOR DYNAMICALLY PLUGGABLE BEHAVIOUR PATTERNS)

(D.C. = dynamic component)