A Survey of Approaches For Describing and Formalizing Use Cases

Russell R. Hurlbut
Expertech, Ltd.
P.O. Box 4151 Wheaton, IL 60189
Document: XPT-TR-97-03
rhurlbut@charlie.iit.edu

ABSTRACT. *A use case is an object-oriented modeling construct that is used to define the behavior of a system. Interactions between the user and the system are described through a prototypical course of actions along with a possible set of alternative courses of action. Primarily, use cases have been associated with requirements gathering and domain analysis. However, with the release of the Unified Modeling Language (UML) specification version 1.1, the scope of use cases has broadened to include modeling constructs at all levels. Due to this expanded scope, the representation of use cases has taken on increasing importance. Accordingly, this paper presents a survey of approaches to describing and formalizing use cases. Key aspects are presented from various approaches that have appeared in trade journals, conference proceedings, white papers, and product literature. The major elements of each approach are classified from two perspectives. The first perspective is the format of how the use case is represented. The primary representation format categories are textual, graphical, and dynamic. The second perspective is use case focus. The primary focus categories are static, dynamic, policy, and process.*

KEY WORDS: *Use Case Formalisms, Scenario, Speech Acts, Workflow, Business Process Modeling, Object Oriented Analysis and Design, Unified Modeling Language*

1 Introduction

The Unified Modeling Language (UML) specification version 1.1 defines a use case as “The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system” [UML97]. Ivar Jacobson is regarded as the inventor of use cases and has authored a great many books and articles that address the subject [JCJO92] [JEJ94] [JBJE95]. Most of these ideas have been integrated into the UML specification. His work has clearly had the greatest influence on defining use cases and establishing the boundaries of what constitutes a use case formalism. One of his papers discussed formalizing use case modeling and suggested two directions for further development of use cases [Jaco95]. The first direction addresses process and human related concerns. The other addresses model and language related concerns. Jacobson stated that use cases should be formalized as far as possible as long as the three rules are maintained:

1. Internal communication among occurrences inside the system cannot be modeled.
2. Conflicts among use-case instances cannot be modeled.
3. Concurrency cannot be modeled.

He provided descriptions for several use case formalisms. The *basic* use case consists of structured English description which included alternative and exceptional behavior. Specific scenarios may also be developed for use during development to explain different perspectives of use. Static descriptions for each actor class and each use case class may also be provided. The second formalism is a class association. These include the use case «uses» and «extends» stereotypes and actor inheritance hierarchies. The third type is an *interaction diagram*. This diagram shows the different paths that the conversation will follow including iteration, repetition, branching, and parallelism. Pre-and post conditions, such as that the user is logged on to the system may also be specified. The fourth type is a *contract*, which specify an object's interface in detail. Actors may provide a contract that involves multiple use cases. Conversely, a use case may provide a contract that multiple actors use. The fifth formalism type is a state-transition diagram in which a stimulus from an actor will cause the use case to leave its current state and perform a transaction. Each transaction is associated with pre- and post-conditions

In spite of the inclusion of all of these formalisms into the UML specification, detailed mappings of use cases to other modeling constructs that implement these use cases, and the elaboration of use cases for requirements gathering and domain analysis is ambiguous. Several authors have proposed solutions to provide more formalism to use cases. This paper is not intended to be an exhaustive survey of all such approaches. Nevertheless, it provides does a comprehensive analysis of a wide variety of techniques directly related to use case along with many other research efforts that have an indirect bearing on use case formalisms.

This paper is organized as follows. Section two contains thirty-one different aspects of use case modeling. The approaches are presented in an order generally based on having relatively more significance or being more comprehensive. Section three contains twenty-six related works that have an indirect bearing on use case formalisms. These works are organized loosely by related topics. Section four provides a discussion of how the approaches relate to each other. It also provides a comparison of the approaches with respect to their focus and representation formats.

2 Use Case Modeling Approaches

This section surveys a wide variety of well known as well as lesser-known use case modeling approaches. Some of these methodologies have merited treatment of a full textbook. Others approaches are merely presented on a few pages as position papers for conferences. Nonetheless, each offers a perspective on use case modeling that provides insight into the issues and problems that must be addressed in requirements engineering.

2.1 Use Case Components

Jacobson, Griss, and Jonsson provide a thorough methodology addressing architectural, process, and organizational aspects of software reuse [JGJ97]. The basis for the methodology stems from Object-Oriented Software Engineering (OOSE) and reflects the semantics contained in UML specification. External stakeholders, end users and customers cooperatively determine what the system will do and how they will use it. This is done through *use scenarios*, first captured informally, then expressed more formally in a *use case model*. A GUI prototype may also be developed. An analysis model deals with the high level static structures of the system. This is the first step toward the system architecture consisting of structures, interfaces, and dependencies. The robustness analysis allocates parts of the use cases to responsibilities undertaken by separate parts of the system, stereotyped as «boundary», «control», and «entity» objects. The collaboration of these object types resembles the Model-View-Controller pattern. The design model defines how the use cases will be performed by the system in terms of interfaces, data structures, and methods. The test model describes how the system should be inspected and tested. The use case model is the starting point for the test model. Each “execution” of a use case described as a scenario will correspond to one test case.

The authors paid considerable attention to component systems and variability mechanisms. Components include use case, analysis, design, and implementation model elements. A «facade», which is a stereotyped package, enables a component system to export only a subset of its types, classes, and other work-products. Variability in component systems occurs at variation points and utilizes one of three mechanisms: inheritance, configuration, and parameterization. Inheritance is used to specialize or extend behavior through the «uses» and «extends» stereotypes. Configuration slots are filled by choosing alternative component implementations. Parameterization can take the form of a bound variable, a template instantiation, or an evaluated expression. Simple bound variables utilize frame technology [Bass96], permitting either values or macros to be placed at the appropriate variation points. Frames are appropriate when there are several small variation points for each variable feature, such that a single parameter can be used to fill all of them. Template instantiation allows type adaptation. Expression can take the form of scripting languages or wizards. A generator is required to transform wither the conversational based wizards or the problem-oriented scripting language into the evaluated resulting construct.

Components may consist of only use cases or actors. The use case model should be derived from existing use case components. This makes it easier to reuse object components. This assumes that requirements closely match the available use case components. This may sometimes require either changing application requirements in small ways or negotiating more substantial changes. An actor component is also designed and packaged to be reusable in several systems. Use case and actor components may be combined to specify variation in how users or system errors are dealt with. However, not all use cases should be reusable components, such as those that are too small or too oriented to technology to be relevant for reuse during requirements capture, only being useful as application use cases are designed and implemented.

Although the use case model is described in a number of documents, the most important one is use case description that defines the responsibilities. Each component system should also have its own glossary of terms. Use cases provide usage-oriented view of component system documentation that enables a developer to learn more quickly how to design from the component systems by seeing how they were intended to be used. Interaction diagrams are then used to restate the use cases in terms of stereotyped analysis types. Built in use case variability then guides the selection of object implementation strategies. For example, the authors suggest that varying constraints or business rules can be implemented through use of a designated «control» object or use of the Strategy design pattern. Responsibilities specific to a use case are placed in a distinct «control» type. More general responsibilities common to several use cases are placed in «entity» types. The «control» type is akin to the Mediator pattern. Extensions are expressed through the Decorator pattern. The Composite and Observer patterns are also appropriate for «consists-of» and «subscribes-to» associations. They contend that most sound design patterns can be identified as collaborations that implement an abstract use case. This implies that abstract use cases can be documented the same way as design patterns in pattern catalogs as suggested by Gamma, et.al. [GHJV94].

Some abstract use cases are domain specific, however many are also are generic, solution specific use cases. If an abstract use case is too small, it is likely only suitable for documentation of its underlying collaboration, rather than for requirements capture. It is not necessary to use the same variability mechanism to express variability in two different models, but two different models of the same system express the same variability. Some variation points are only relevant to a certain model (e.g. implementation issues).

In a layered architecture, each superordinate use case can be allocated to the application and component systems by splitting them across these systems. The superordinate use case is partitioned into *use case segments* corresponding to design subsystems in the superordinate subsystem. Each design subsystem offers a number of use cases that correspond to the use case segments allocated to the subsystem. In practice, the use case model for the superordinate system should only define transactions that involve more than one application or component system and are relevant to the only those subordinate design subsystems that need specialization or consideration of some sort. A sequence diagram can show packages representing each subsystem that participates in a superordinate use case. The modeler can highlight portions of the sequence diagram (i.e. the double line areas along the package lifelines) where input from the superordinate design model is provided.

2.2 Structuring Use Cases With Goals

Cockburn identifies four dimensions to use case descriptions: *purpose, content, plurality,* and *structure* [Cock97]. Each of these dimensions has an enumerated domain value. *Purpose* can be either for *stories* or *requirements*. *Content* can be either *contradicting, consistent prose,* or *formal content*. *Plurality* is either *1* or *multiple. Structure* can be *unstructured, semi-formal,* or *formal structure*. His approach is defined as *requirements, consistent prose, multiple scenario, and semi formal structure*. This is the same as that of the UML, which is derived from Jacobson's OOSE methodology [JCJO92].

Three levels of use cases are also identified through enumerated domain values: system scope (strategic, system), goal specificity (summary goal, user goal, sub-function), and interaction detail (dialog interaction, semantic interaction). Of the 12 combinations, only 5 are useful: 1) strategic scope/summary goal, 2) system scope/summary goal, 3) strategic scope/user goal, 4) system scope/user goal, and 5) system scope/sub-functions. Only semantic interaction is considered relevant for use case modeling with respect to the third level. Dialog interaction is equated to user interface requirements, which are likely to change too often.

Cockburn first focuses on the main conceptual constructs in use case modeling. He provides this definitions:

> A use case is a collection of possible scenarios between the system under discussion and external actors, characterized by showing how the primary's actor's goal might be delivered or might fail.

An *action* connects one actor's *goal* with another's *responsibility*. *Primary actors* are actors with the goal. *Secondary actors* are actors assisting the system. A developer writes responsibility in the function name and in the comments states the goal informally. If the secondary actor does not deliver on its responsibility, then a *back up action* occurs. An interaction can be simple or compound, but it does not contain branching or alternatives. It only describes the past in the form of an actual execution sequence of the system or a definite future in terms of a conceptual scenario. A use case collects all of the scenarios that might occur during one interaction of the primary actor. The scenarios and use cases go until goal success or abandonment.

The author proposes to control scenario explosion through three techniques: subordinate use cases, extensions, and variations. Variations utilize semi-formal structuring within and across use cases to track higher levels variations that are known to occur. These variations are particularly concerned with ones that require different data formats. Steps in a scenario are considered intermediate goals of the actor. Failure of a step is either handled by another scenario or and extension scenario. Subordinate use cases prescribe a recursive structure of use cases and scenarios. He uses a

“Striped trousers” analogy describing a dichotomy of success and failure of goals as the legs with each stripe representing a main course, alternative course, or recovery scenario that result in either the success or failure. Actors usually don’t know the condition of the scenario at the start of their interaction with the system. This is determined as conditions get stricter down the scenario chain.

Another analogy that Cockburn uses is the shape of sailing ship. This is intended to focus the attention of the user at water line, which represents the user goals, or use cases themselves. They use cases are abstracted into orthogonal membership into summary groups that relate to the tapering of the sails the higher one goes above the water line. The web of sub-functions is below the surface. These again taper to a relatively few widely shared functions.

Cockburn states that that empirical evidence shows that readers prefer to have actions numbered and starting on new lines. This helps keep the narrative clear, thus improving traceability from requirements to design to test. It also allows specific line references needed in the extensions section of his prescribed template. His template is relatively straightforward with mostly associations. These sections are: 1) characteristic information – goal in context, scope, level, preconditions, success end condition, failed end condition, primary actor, trigger; 2) main success scenario, 3) extensions with references to the steps in the main success scenario, 4) variations that will cause eventual forks in the scenario, 5) related information such as priority, performance target, frequency, superordinate use case, subordinate use cases, channel to primary actor, secondary actors, or channels to secondary actors, 6) open issues, and 7) schedule.

He refers to a *used* use case as a single line item in a scenario. An extension is another scenario that refers to some point inside the scenario. This conflicts syntactically with the UML specification, but essentially provides compatible results. His general format for writing use case statements as:

<time or sequence factor>… <actor>…<action>…<constraints>.

Example: *At the end of the month, she sends a credit memo to all customers whose credit is larger that a certain value.*

Cockburn cites a few problems that remain. First, there is continued confusion about the levels of use cases. His introduction of terms identifying the three levels helped, but it is still multi-level and there is little that can be done about that. Second, data variations still requires ad hoc carrying forward to the appropriate level of refinement. Third, partial delivery of system features and functionality often cross multiple use cases, making tracking more difficult.

2.3 Adaptive Use Cases

Hurlbut presents an adaptive use case extension to the UML specification that integrates conformity with The Workflow Management Coalition’s Workflow Reference Model (WRM) issue 1.1 and thew GUIDE Business Rules Model [Hurl97] [Hurl97a]. Use cases, as defined by the UML, are system-oriented. The focus is a single entity that represents the system along with other entities outside the system that are modeled as actors. Incorporating speech act semantics shifts the focus to an actor-oriented view. Only a single actor is considered and the system scope can vary to include those entities modeled as actors in the system-oriented view. In additional to supporting an actor/end-user and system/customer oriented view, additional use case refinements allow the focus to shift to a component/architect, framework/designer, or class/implementer oriented view.

Adaptive frames provide a behavior-oriented perspective through its focus on scenarios rather than the generalized collective behavior represented in use cases. Whereas the integration of speech acts focused on the system boundary and secondary actors, integration of adaptive frames focuses on package boundaries and variant scenarios.

Another distinction that is made along the same lines deals with UML model elements and their instances. The *Class* model element has instances such as Person, Address, and Company. Unlike the UML *Class* meta-model element where the types of attributes for the classes will likely be quite different, the *UseCase* model element attributes will often be identical for a given project team or domain model. This is why the WRM specifications identify a workflow definition as the root entity. The WRM specifies the state or a process instance as being either initiated, running, active, suspended, completed, or terminated. This could easily be translated into an attribute for the all use cases as a UseCaseStateKind enumeration.

In order to comply with the well-formedness rules of the UML, a scenario cannot be a use case. Otherwise its association with the owning use case would be a violation of the constraint that use cases can not have associations to use cases specifying the same entity. Instead, scenarios can be defined as *Class* model elements. They can then be declared a type of attribute of the use case. Providing structured use case representations through attached scenarios offers several advantages:

- *Scenarios do not have to be named.* Scenarios are instances of the class 'scenario' rather than having a new model element for each scenario as is done for use cases. Use case action sequences and individual use case actions are also instances rather than named model elements.
- *Scenarios allow management of finer specification detail.* Scenarios keep the use case as a specification model rather than implying realization though associations to actual actions. This association can come through a dependency relationship as part of a transformation process. Thus, scenarios and its use case actions primarily serve as text management constructs.
- *Scenarios facilitate use case evolution.* Scenarios enable better management of changes of a use case by easily allowing additional variations to be added or removed. This can even occur dynamically as the system is running if dealing with self-modifying systems.
- *Scenarios provide a mechanism for model element to be compatible with multiple standards.* They allow definition of use case formats as UML stereotypes that conform to WRM specifications.
- *Scenarios provide a mechanism for integrating business rules.* Since scenarios are UML C*lass* model elements, they support the notion of templates. Therefore, parameters can be integrated into the structures. This provides the support for business rule binding. Moreover, support is also provided for the different data types that result from use case variations that accommodate actor role subtypes.

Scenarios can transcend varying levels of abstraction. In order to maintain consistency with the integration of speech acts, it is appropriate that alternative sequences of actions not perceived from the outside should be suppressed. If all scenarios adhere to this guideline, then each becomes the description of a use case variant. Adaptive frames place one additional constraint upon a use case by requiring that the basic course of actions, referred to as the archetype scenario, does not contain any conditional branching. Once an archetype scenario has been declared, all alternative courses of action, exceptional behavior, and error handling can be completely defined in terms of scenarios built by describing their differences from the archetype scenario. Adaptive frames run contrary to strict inheritance, where structural and behavioral features can be replaced or added, but not deleted. Instead of trying to coerce a contrived sequence of actions upon each alternative scenario, only the actions explicitly declared as common actions between scenarios are reused. This reuse may also occur across use cases, since action sequences of one or more actions can be contained in their own frame for reuse. Once they have been defined, these frames may then be grouped into packages. Using fine grained frames consisting of actions and action sequences provides much greater opportunities for use case reuse than the large grained generalization stereotypes defined in UML.

The relationship between the archetype scenario and the variant scenario is established through an «adapts» UML stereotype. An «adapts» stereotyped relationship between a variant scenario as the source and an archetype scenario as the target results in a set of adaptive frames. Each variant can also be adapted. Without the «adapts» stereotype or similar construct as part of the UML meta-model, the modeler must choose between no reuse or contrived reuse through «extends» and «uses», resulting in model elements that may be more appropriately defined as "abstract use case fragments." In the UML Extension for Business Modeling, a use case is prohibited from being partitioned over several use case packages. The integration of adaptive frames conforms to this restriction by directly associating a use case with its archetype scenario and then incorporating all variant scenarios as an adaptation of this base line use case. Thus all parts of a concrete use case is contained in a single package, regardless of the fact that it may adapt through a web of frames contained in other packages.

The action statements rely on a semi-formal structured grammar that adopts the GUIDE Business Rules Model text ordering conventions as well as variations of existing restricted form grammars. Each action statement is bound to a *UseCaseAction* model element. Thus, although the use case steps remain in a semi-formal English syntax, each statement maintains a formal linkage through the adaptation mechanism. Since *UseCaseAction* is a stereotyped derivation of the UML ActionSequence, a direct mapping to UML *Action* model elements and associations to *Operation* and *Method* model elements is provided. In this manner realizations via state machine models and frameworks (via pattern collaboration models) is possible.

The approach of integrating business rules with use cases introduces a policy-oriented view to domain models. Workflow oriented rules are primarily used for process flow control. By introducing business rule bindings to use cases, the static object and data structures of an application can also be controlled. However, unlike class model which directly integrate rules as constraints and invariants, these business rules are bound to process definitions and are only indirectly related to classes through the realization of use cases. Business rule binding allow process oriented speech acts to highlight the different contexts of system boundaries that isolate structural differences. They also enable structurally oriented adaptive frames to be used to isolate behavioral differences.

Business rules utilize templates to create sets of scenarios that comprise a use case. These business rules are not directly integrated into the scenario; rather they are bound at build time. In this manner, several different parameterized data structure formats can be accommodated in a single use case. For example, in a use case called openNewAccount for a banking application, an individual, a corporation, or a trustee may play the role of Actor. The information required by each of these will vary to some degree. However, the realization of the use case will require coupling across the collaborations of cooperating objects in order to consistently deal with the variety of data structures accompanying the different players of the actor role. Parameterization in the form of the business rule allows a single parameter to declare the mutual constraints that define this coupling.

This parameterized business rule binding approach has many similarities with dynamic reflection. First, one of the key features is maintaining knowledge about dependencies among elements so that when changes are made to various objects, these changes can be automatically propagated throughout the entire system. Second, functionality is abstracted and embedded into the system. Third, an application is build as a series of layers, with the topmost layer being a use-friendly scripting language that represents domain-specific concepts and application specific rules. The primary difference between parameterized business rule binding and dynamic reflection through meta-object protocols relates to the emphasis of analysis in the former and design in the latter. Dynamic reflection tracks dependencies of classes and subclasses. Parameterized rules tracks dependencies of use cases, which may be realized by many different object systems and therefore is more generic.

Business rules assert the necessary data structures, while use cases declare the necessary behavioral semantics. Thus, use cases drive the selection of components, classes, or manual processing through domain asset fit assessment and project resource constraints.

2.4 Enhanced Scenarios for Modeling Global Behavior

Beringer argues that the flat scenario model with a matrix relationship between data model entities and scenarios forces an assumption that the externally visible behavior of the system can be subdivided into more or less independent scenario types [Beri97]. Although this matrix approach is utilized by most use case modeling approaches, it is limited when complex systems are modeled. Some of these problems are: 1) relationships and similarities between different scenario types cannot be expressed, 2) dependencies between scenario types are not modeled, 3) only one abstraction level can be represented, and 4) the use case analysis model may produce a low quality object design model with a strong bias towards data modeling.

The author's enhanced scenario modeling technique, called SEAM, is proposed as a semi-formal extension to the Fusion methodology. Although related to workflow management and business process engineering, these domains are excluded from the research. SEAM includes composition, aggregation, specialization and extension hierarchies of services. Interacting object are represented at varying levels of abstraction, including atomic objects, subsystems or entire systems. Scenario types, which identify the possible interaction sequences for a specific service, can also be modeled at different levels of abstraction. These scenarios can be represented from both an external and an internal point of view. A user view is defined as a special form of external view in which a single actor perspective is taken. The internal view contains both the interactions with the environment and the interactions among internal objects. Since modeling efforts may result in overlaps of internal interaction diagrams and global state transition diagrams, several automatable transformations are necessary. The author cites two examples: life-cycle expressions → state transition diagrams and scenario trees → regular grammar → state machines.

Beringer describes *global behavior* as focusing on the services provided by the whole system of interacting objects. This contrasts local behavior, which is considered the concern of class interfaces and contracts. Reasons for such global scenarios are: 1) the external view is partitioned into meaningful pieces; 2) representations such as test, tables, or sequence diagrams facilitate communication, including role-playing and walk-throughs; 3) correct and complete requirements model can be produced; 4) test scripts can be produced; 5) incremental development can follow; and 5) leads to detection and specification of objects in the design model.

The author also distinguishes between conceptual and technical interactions. A conceptual interaction is concerned with the problem domain and analysis models. A technical interaction is concerned with construct in the software system that realizes the problem domain and design models. During the refinement process the transition from purely conceptual to technical events and from signals to request occurs. A further distinction is made between scenario types and scenario instances and event types and event instances. If all parameters are assigned a value, then the construct is an instance, otherwise it is a type. Thus, most diagrams are scenario types since some values typically remain as parameters (such as the telephone number is a telephone call).

Algorithmic representation of a scenario type specifies all possible orders of events through iteration, alternation, and concurrency annotations. Declarative descriptions only identify state changes and interactions, not their order. The scope of a scenario type has two dimensions, its length and the number of possible instances, that are defined in a tuple of criteria referred to as a *classification scheme*. The length dimension contains five gradations ranging from a single input event to the entire lifecycle of the system. In between these two extremes, the scenario is allowed to be partitioned at the boundary of reaching an essential system state. Larger aggregations may be defined as a collection of interactions as long as additional input events do not trigger scenarios of their own. Finally, this last criteria may be relax to include any sequence short of the entire life-cycle of the system that may be perceived as part of the same overall task. Similarly, the number of possible instances may be partitioned between two extremes. At one extreme, the sequence of event types remains the same, and only the values of the system's state and the event parameters may vary. At the other extreme, any sharing of common parts or handling of similar tasks may be combined. In between these two extremes, the primary partitioning of scenario types is based upon the initial event type remaining the same. Additional partitioning may be based on the content of the input events or on the system's state at initiation. Regardless of the scope of a scenario type, it should specify the following elements: 1) its *name*, which is often the name of the input triggering event (there may be more than one); 2) the *reaction* of the system in terms of output events, calculation of output parameters, and state changes; 3) the *conditions* which determine the reactions of the system and order of the interactions; and 4) the *interaction types* and *intermediate interactions* with agents, if any.

SEAM defines its parameters of interaction through a provisional external data model represented in an *entity relationship model*, a *data dictionary* with *business rules* and a *concept map*. Services are then defined through the following properties: *information flow*, *interactions* (both input and output as well as the possible scenarios), *state changes*, and *preconditions*. Pseudo-code annotations to sequence diagrams specify control flow. Detailing and abstracting a scenario is considered a zooming in and out of a service in a manner that can not be automated. It is more than hiding of interactions; the conceptual interactions and pseudo-code annotation of the more abstract views must be created explicitly by a developer. Special bar notation along the left side of the bracket code of the sequence diagram denotes which parts of the diagram are available in a more detailed view.

Inheritance, aggregation, and composition hierarchies for services have slightly different semantics from their object counterparts. Inheritance hierarchies are used to represent specialization and extension of services. Aggregation hierarchies group services together that follow each other. Aggregation may be complete or partial. A complete aggregate service is an aggregation of several (possibly elementary) services. A partial aggregate service reuses one or more other service, but contains elementary services. Unlike the external oriented view of the first two, composition hierarchies reflect an internal view that corresponds to the services of the objects of the decomposed subsystems. The relationships between services of different models do not appear in the final documentation of the scenario model, but are considered important for the modeling process. Services that have the same responsibilities, information flow, and server object may be modeled equally in different models. However, their specifications may differ by the *names* used, the position for a *selection criteria*, the *algorithm* used, the *internal composition* of objects, or the composition of *element services* in an aggregation. A selection criteria position can be either a parameter type or a value from the type that is used in the interaction name. For example, *open(accountType)* in the former case and *openSavingsAccount* or *openCheckingType* in the latter case.

The SEAM meta-model is based on the paradigm of systems of interacting objects. A distinction is made between the *services* and their *scenario types*, between *technical* and *conceptual* interactions, between *requests* and *notifications*, between *high-level* and *elementary* services and between *system services* and *services of atomic objects*. It is composed of two diagrams: a service inheritance hierarchy and an entity-relationship diagram between *object, service, scenario type, interaction, state change, action,* and *event.*

Object life-cycles can be represented through regular expressions that show the order of services offered by an object, rather than the order of input and output events of a system. The following operators are used: Sequence (A.B), interleaving or concurrent services (A||B), optional service ([A]), alternation (A|B), repetition of zero or more (A*), repetition of one or more (A+), interleaving or concurrent repetition (A ||*, A||+), and precedence ([], *, +, ||*, ||+, ., |, ||). Regular expressions show all possible orders of services in a non-deterministic manner. Conditions, which specify the selection of which parts of a service to perform, should be specified through pseudo-code. Preconditions may be used in conjunctions with regular expressions and pseudo-code. The latter two would be used to show the system's life-cycle as a whole. The preconditions would be used to indirectly specify the life-cycles of the internal objects, thus avoiding having separate life-cycle models for each internal object.

2.5 Agent Based Use Cases

Kendall, Malkoun, and Jiang discuss the application of object oriented analysis to agent-based systems [KMJ97]. Structured messaging and negotiation protocols represent a significant feature of agent based systems. They argue that event traces are too complex in large systems, instead use cases allow compounding, abstracting, and specializing. For example, the *extends* relationship in use cases can be conditional, and occurs when one use case is inserted into another if certain circumstances exist. This corresponds to context or invocation conditions that determine when an agent's plan is to be intended. They expand the traditional notional of uses cases to include depictions of how humans or organizations enter and interact with each other as well as systems.

Agents can be classified as either weak or strong. A weak agent is described as autonomous, social, reactive, and proactive. A strong agent adds in mentalistic notions, such as beliefs, desires, intentions, rationality, veracity, adaptability, and learning. Knowledge query and manipulation languages, such as KQML, allow agents to negotiate and engage in conversations or protocols that can be represented as speech acts. Objects are used for beliefs, sensors, and *effectors* that are combined with agents. A correspondence between IDEF functions (i.e. input, control, output, and mechanism/resource), agents, and use cases is mapped.

Agents are used when proactive behavior is mandated. They augment or replace human decision making, i.e. replacing or assisting actors. The agent has belief, goals, intentions, and sets of use cases with associated plans. The agent gets input through sensor objects and acts through *effector* objects. It also sends and receives control or coordination messages.

Use case representations do not indicate which use cases give rise to control or decision making behavior, however some workflows do. Instead, use case representations have workflow equivalences, such as when use case diagrams and IDEF diagrams are completed to the same level of detail. The IDEF functions appear directly as use cases. Specifically, a use case groups a behaviorally related sequence of transactions, with the IDEF functions adding in control flow.

The IDEF models describe a hierarchy of functions that are decisions, activities, and actions. However, only those functions with control output are important to agent oriented systems. In other word, those corresponding to a use case scenario and agent's goals and plans. Control input determines which use case extension or plan to follow. When more than one agent is involved, coordination follows a protocol. With abstraction, use cases and scripts lead to identification, specialization, and abstraction of agent protocols as well.

2.6 Use Case Maps

Buhr has developed use case maps as a visual notation for comprehending and developing the architecture for emergent behavior in large, complex, self modifying systems [Buhr97]. He describes this notation as existing at a higher level of abstraction than UML. His basic premise is that behavior emerges during execution, and is determined by details of the components. The property of self-modification allows for a system-wide behavior to be different at different times (such as stress loads). This allows the population of components, and relationships among the components may change over time. Both the components and the behavioral patterns may be modified in progress.

Use case maps provide a birds-eye view of whole system behavioral patterns. The maps are useful for reverse engineering, or understanding complex systems. The maps also facilitate forward engineering, or the designing of new systems. UCMs are a two-dimensional map of cause-effect chains from points of stimuli through the system to points where responses are observed. Continuous paths traverse an underlying geography of the operational component boxes, which make no distinction between hardware and software. These component boxes can be fixed or slots that are occupied dynamically by pluggable path stubs. Buhr makes a distinction between *components* of the system view and *classes* of the construction view. Classes are specifications that may be used to construct operational components, not the operational components themselves.

Use case maps consist of three primary constructs. Responsibilities are represented as dots, with the responsibility described by active natural language phrases keyed to the dot’s label. Causality is represented as a path that connects the dots, with start and end points that have associated pre- and post-conditions. Components are represented as simple boxes with associated responsibilities, but without commitment to interfaces, protocols, messages, state machines, etc. Components eventually become objects, processes, and modules.

Development of UCM can occur in one of three ways, distinguished by which construct is added last. The causal path is meaningless by itself, so it can never occur first. An unbound UCM represent the purest form of behavioral patterns in that they do not have a component substrate. This is essentially a “connect the dots” approach. A modeler may alternatively choose to begin with the component substrate. Trial path traversals through the components may reveal responsibilities and explore alternative component bindings. Responsibilities may also be applied to the components directly, with path completion being done last.

Once completed, the inter-component path segments determine the sequences of interactions with data becoming explicit. Sets of cross-component paths (with responsibilities) determine component logic and state. Edge crossings determine component interfaces with data becoming parameters. System-environment edge crossing determine system interfaces. Multiple paths through the UCM denotes the sharing of some responsibilities. These multiple paths may create an appearance of forks and joins, but they are really superimposed paths (e.g. a scenario along a black path cannot take a gray fork).

UCM path notation provides support for representing internal concurrency and couplings. Coupling can occur through responsibilities, synchronous coupling, or asynchronous coupling. Responsibilities are unsynchronized and represent the weakest type. The results may vary depending on the order that the responsibilities are performed, e.g. a save before/after a retrieve.

Synchronous coupling is the strongest type. A many-to-one join forces synchronization of multiple incoming paths to proceed along a single outgoing path. Asynchronous coupling represents an intermediate level of coupling. This is accomplished through *waiting places* that appear along one path that touch another path that clears the waiting condition. This may involve the concepts of *Real time* and *Timers*. There may be timeouts, resulting in aborts and exceptions. An exception coupling is represented as a *watchdog path*, one or more *failed paths,* and a *handler path*. The *watchdog path* discovers an error condition affecting other paths. The *failed paths* are those paths that can affect the error condition. The *handler path* handles the error. Failure points may cause scenarios to end abnormally, typically an OR-fork leading to the end point.

Use case maps can be refined through decomposition. Since UCMs only represents a set of example scenarios, they cannot provide a complete specification of a system. Decomposition of a component at one level, the component-centric view of it becomes a path-centric view at the decomposed level. This allows for finer grained specification of the system. Use case maps may also be partitioned, referred to by Buhr as factoring. Factoring is accomplished by cutting paths into fragments at judicious points, adding start/end point symbols, and then separating into separate UCMs.

When decomposing UCMs, responsibilities are allocated to progressively finer grained sub-components. Path segment crossing leads to stubs for sub-paths and sub-components that remain to be developed. However by itself, stubbing is a path concept, not a component concept. A higher level layer has hidden nuances, such as hidden synchronization. An indirect relationship may be established that is not traceable directly from the higher UCM, except for the knowledge that there is a layer underneath. The binding of actual paths in the higher layer to implicit stubs in the lower layer results in a detouring to the lower layer where operational meaning is revealed.

Stubs are another important concept in use case maps. Components and slots are used for representing variability in static structures. Components can be created and moved into a slot, moved from a component pool to a slot, or moved from one slot to another slot. Stubs and their plug-ins are use to specify variability in behavior, as represented by the causal paths. Plug-ins are described in a separate diagram and bound to stubs as parameter associations. A plug-in from a design pattern library may be more general than needed, so the unused paths need to be marked as such. When they fill the path-stubs, the start and end points of the stub disappear. Thus, this plug-in behavior only contains components; it is not actually a component when filled. Choices are made according to system conditions as to which path is taken. This allows for dynamically pluggable elements to accommodate self-modifying systems.

According to Buhr, UCMs demonstrate these desirable properties: 1) they aid human reasoning at a high level of abstraction rather than entering details into a computer tool, 2) their meaning is not dependent on details of components or code, 3) they combine system behavior and structure into a single coherent diagram, 4) they express system self modification in a compact lightweight fashion, 5) the diagrams are easily grasped as visual patterns for s system as a whole, 6) they provide a macroscopic view for forward engineering, reverse engineering, maintenance, evolution, and reengineering, 7) they encourage design experimentation, 8) they scales up, 9) they provide a high level supplement for any detailed model/method, 10) they suggest a new concept of architecting behavior, and 11) they can be saved for documentation and maintained without unreasonable effort. Buhr also cites criticism that Petri-nets and LOTOS knowledge representation can do the same thing. UCMs also share some affinity to the class responsibility card technique (CRC) – both are centered around causal sequences of responsibilities.

2.7 Change Cases

Ecklund Jr., Delcambre, and Freiling focus on use cases that identify future requirements [EDF96]. Change cases are intended to capture anticipated future changes to a design in order to increase its robustness to handle change. Essentially, change cases are use cases that have links to affected use cases and to change requirements for traceability purposes. Their premise is that by getting the interfaces right, maintenance and evolution of the system is easier. Bi-directional traceability links to each artifact are a prerequisite to make this work. They consider four types of change: 1) market demands, such as a large customer wanting things done their way; 2) business requirement change, such as new policies or operational processes; 3) legislative and regulatory change; and 4) imaginative users.

They also identify three key characteristics of change: 1) focus, in terms of system responsibilities; 2) scope, in terms of artifacts; and 3) degree of definition, as to what extent the particulars of the potential changes are understood. These characteristics are used to measure the impact of change. Their first metric is *impact of change at a level*. This is the percentage of use cases affected by the change at the same level (i.e. requirements, analysis, design, implementation, documentation, and test). The second metric is *constructs impacted at each level through traceability links*. This is the relative number of affected constructs to total constructs at that level. The third metric is *downstream impact*. This is a robustness measure that compares the impact at one use case level with another level. Their final analysis combines all changes to compare two designs. Analysis can also be made to compare known cost of completed models with the *impact of change at a level* to determine cost of change with and without immediate redesign of additional robustness.

Budgetary constraints define change cutoffs that tend to limit upstream changes. Traceability of change cases from design to analysis help prevent design decisions that reverse analysis decisions intended to provide robustness. This typically requires a judgment call by the design team to identify most important changes in order to keep within budget. Artifacts useful in this process are a change analysis budget and a change-extended analysis model.

2.8 Task Scripts

The OPEN Modeling Language is a competing meta-modeling to the UML. Both UML and OML represent the merger of three main contributing methodologies – for UML, these are OMT, Booch, and Objectory; for OML, these are SOMA, MOSES, and Firesmith [HSFG97][Hend97][Grah94]. A key principle behind OPEN is the notion of *tasks* and *techniques*. A task may be accomplished by one or more appropriate techniques. For example, in order to find objects, the choice may be between, say, using use cases, using noun analysis, identifying concepts and their responsibilities, using CRC cards, etc.

The use case metamodel is an aggregate of *External* and *Use Case*; which itself participates in a short type hierarchy consisting of *Essential Use Case*, *Use Case*, and *Scenario*. In addition to use case diagrams, OML defines two types of scenario class diagrams – task script diagrams and mechanism diagrams. The relationship between a use case and a scenario is described in several ways –as a specialization of a use case, an instance of a use case, and as a component of a use case. A use case links with objects via a *participation* association. The *behavior model* then consists of the use cases and the objects as actors in the model. There is also a link through to events. The OML meta-model also defines a specific category of relationships for scenarios. Scenario relationships describe connections within a use case/task script model. These three relationships are precedes, invokes and uses.

Capturing user requirements involves the use of *task scripts*, which are primarily used to discover business objects and then retained for testing purposes. Task scripts are supported by a task-action grammar that consists of a: Subject – Verb – Direct.Object – Preposition – Indirect.Object (SVDPI). Although lying in a different domain than the application object, these task scripts are also objects and can be organized into composition, classification and usage structures. *Component scripts* are derived through hierarchical task analysis. *Side scripts* deal with exceptions that require the flow of control to be redirected for special case handling. Task scripts deal with *business processes* and not business functions. The authors make this distinction in order to avoid the danger of capturing the requirements in the form of narrow functional specializations that are often present in the existing organization.

The meta-model also has explicit support for rule assertions that an object's operations must not violate. They take the form of pre- and post-conditions and invariants. Graham defines four types of rules: 1) control rules, relating object behavior to control handling of defaults, multiple inheritance, exceptions and general associations with other objects; 2) business rules, which typically relate two or more attributes; 3) exception handling rules; and 4) triggers, which relate attributes to operations [Grah95].

2.9 Use Case Classes

Jansson describes an approach that reifies a use case as a class [Jans95]. A use case is reified through a use case instance that constitutes the performance of the sequence of actions specified in a scenario of the use case. The general approach begins with making the use case model into a class category. Each use case is an operation of the class *System*. *AllActors* become another class. Each actor is an instance of *AllActors*. A class is then created for each use case operation on *System*, each scenario is added as an operation of this class. An actor object starts a scenario by sending a use case message to the System, which in turn instantiates a use case object. The second message sent by the actor is the scenario name. It is sent to the instantiated use case object. Additional objects, that are an actual part of the system, can then be instantiated by the scenario. The actor interacts with the new class objects normally. The actor can also send additional use cases messages (to *System*) or scenario messages (to *UseCase* objects). The rationale for instantiating use cases as classes is primarily for traceability links for testing source code. The use case classes serve as system and function test drivers.

2.10 Use Case Formats

Harwood defines three formats for use cases that correspond to macro-process phases [Harw97]. The *conceptualization* phase establishes core requirements. It utilizes a free text form use case for requirements capture and specification. Its template headings include: reference, title, actors, general description (2-3 lines of text), specific description (sequential description of basic path, triggering events, outcome in numbered paragraph form), and alternate paths (why, what outcomes). The *analysis* phase develops a model that describes behavior. It utilizes a structured textual format for discovery of the problem domain. It adds system preconditions to the template. It also replaces a specific description with an *event-response list*. This list describes the chronological context of the event, the desired system responses, and provides a reference to the relevant portion of the requirements document. This list is repeated for each alternative path. The *design* phase creates the architecture and is captured by an event trace diagram.

2.11 Directed Use Case Graphs

Kosters, Pagel, and Winter present an approach for mapping use cases onto static classes and methods [KPW97]. The authors expand the definition of a scenario step to embody actions that may occur outside the system. This is necessary to capture the complete workflow. A scenario step may be an interaction with the system, a responsibility to be carried out by some human (the primary or secondary actor), or a referenced use case.

Use case explosion is controlled through directed use case graphs where nodes relate to the scenario steps. A scenario is similar to a control flow in a use case graph. Conditions and outcomes of each scenario step correspond to certain object models, referred to as *object constellations*. Each interaction constitutes a unique thread of execution that relates to a sequence of method calls in the object model. The collection of classes correspond to the textual descriptions, referred to as *pre-scope* (conditions) and post-scope (outcomes) in an analogy to pre- and post-conditions. Each scenario step is refined by an *episode*. An episode is a tree of methods, each of which is specified in the class model. The analyst decides on the appropriate decomposition, all the way down to attribute access if necessary. Activating the same root method with different pre-scopes and parameter settings (goals) may result in totally different episodes. Each interaction must be refined and each method referenced by an episode. Unused methods are suspicious.

2.12 Scenario Trees

Hsia, Samuel, Gao, and Kung describe user oriented *scenario trees* that represents all scenarios for a particular user [HSGK94]. A scenario tree consists of *state* nodes and *event* directed arcs. This approach is best suited for a single thread of control and well defined state transition sequences that have few alternative courses of action. Since all scenarios must be modeled independent of each other, concurrency and interacting user scenario dynamics are not supported. Regular expressions are used to formally state the user scenario that results in a deterministic finite state machine with a single state that defines both its initial and terminal state. Automated support for well formed expression checking and prototype generation of the conceptual state machine enable the modeler to validate the *scenario schemas*, which correspond to each unique path through the scenario tree.

2.13 Message Sequence Charts

Andersson and Bergstrand present a method to formalize use cases that introduces an unambiguous syntax through message sequence charts [AB95]. This approach utilizes three levels for the use case model. The system level describes the functional view of the system and corresponds directly to the OOSE use case model with an extension to include associations to reuse structure blocks, which identify an event sequence that is reused in several use cases. This construct essentially replaces the *uses* inheritance association. The structure level describes the use case behavior without going into detail by utilizing a hierarchical view that allows information hiding. Informal text is used to define the actors, start conditions, and constant declarations. Structure blocks are again used in this level to describe the behavior from the actors point of view. It can be composed of another structure diagram. Flow lines connect the structure blocks sequentially and operators that enable the expression of several event flows within a single structure level are provided. The basic level describe the detailed interaction between the system and actors. These are essentially sequence diagrams the have structure blocks overlaid over the lifelines and message arrows.

Five operators that utilize the standard Message Sequence Chart notation are used: sequence, alternative, repetition, exception, and interrupt. Alternative operators are allowed to share initial sequences of events, thus allowing a delayed choice to be made when it is clear which alternative has been chosen. Repetition can be specified by unbounded intervals. Exceptional and interrupt sequences of events terminates the use case, the difference being that the former occurs at defined moments relative to other events, the latter at any given time.

Three different levels of reuse are defined. *Identical reuse* involves the same event sequence and actors. This type of reuse is accomplished by merely referencing the structure block by name. *Reuse with redefined actors* is denoted through formal instance parameters that establish context, e.g. UseCaseName(INST newActorName:oldName). Several actor redefinitions can appear in the same diagram, provided no individual actor appears more than once. *Reuse with specialization* share the same event sequences, but are allowed to vary at one or more occasions. Source type event sequences are marked as virtual and may be redefined to provide reuse, e.g. SpecializedEventSequence INHERITS VirualEventSequence.

2.14 Elicitation of State Machines by Use Cases

Mitchell and Lecoeuche base their research on the premise that although the mapping between use cases and classes is clear and uncontroversial, they do not provide a mechanism for constructing a state machine that supports varying levels of abstraction [ML97]. A use case corresponds to several states. A state can be represented by several use cases, but they would not occur in the *same* use case model because use case commonality is abstracted into a supplier use case.

They create the state machines through a process of transformations. First, the use case model is developed. The use case model is followed by development of an analysis model using the OOSE stereotyped objects (i. e. interface, entity, and control object). The analysis model is followed by the creation of statecharts with transitions. This involves discovering candidate states that are present in the object’s dynamic view and determining the transitions between them. There are two invariants for this mapping. First, every state the system may hold must map to a use case. Second, every use case (supplier) that has dependent use-cases (clients) must correspond to a super-state.

They describe the use of CRC cards as a tactical alternative to use case models. Use cases are described as high level in terms of external agents and internal processes. CRC cards are described as the storyboarding of individual classes. They use Harel’s state transition matrices (STMs) which can be used as state machines in their own right. The STMs are useful in cross checking and in modeled using spreadsheet or database applications for CASE tool interrogation. They have developed their own graphical notation to extend class diagrams with state modeling. Each class encapsulates at least one FSM (state package). States and sub-states are modeled through inheritance.

2.15 Formalized Use Cases Based on State Charts

Glintz describes a formal notation for validating and simulating a behavioral model representing the external view of a system [Glin95]. Use cases, referred to as scenarios, must be structured such that they are all disjoint. Any overlapping use cases must be either merged into a single use case or partitioned into several disjoint ones. Such structuring allows for each use case to be modeled by a closed state chart, i.e. a single initial state and a single terminal state. Composition of use cases is performed though sequence, alternative, iteration, or concurrency declarations. The formalized model allows for consistency and completeness checks, which include provisions for stub to be placed where a component scenario is not yet modeled. Deadlock, reachability, naming and mutual exclusion tests are performed.

2.16 Synthesized Use Case Modeling

Regnell, Kimbler, and Wesslen proposed a *synthesis phase* extension to the OOSE use case modeling approach [RKW95]. In their approach, separate use cases are integrated into a *synthesized usage model*. The synthesis phase consists of three activities; formalization, integration, and verification. The formalization of each use case is accomplished through message sequence charts in which abstract interface objects and atomic operations are identified. The integration of use cases identifies user and system actions in order to create and integrate abstract usage scenarios, which are represented through diagrams consisting of system actions, user actions, messages, and external system states. A usage view is synthesized from the abstract usage scenarios for each actor. The aggregation of these usage views comprise the synthesized usage model which also contains descriptions of actors, use case specifications, use and system actions, abstract interface objects, and data dictionary with problem domain objects. Verification is first performed on each use case specification through manual review with respect to completeness and consistency. The synthesized usage model is then checked to ensure that every use case specification is completely covered. This second step could be automated by checking that every abstract usage scenario is a possible path in the corresponding usage view. Finally usage testing through automatic generation of test cases are derived from the usage views. Functional and statistical properties of system usage can be used to enable certification of the system's reliability.

A number of differences with OOSE are cited. First, the semantics of actors and use cases is changed to enforce a multiple-role, single-actor view. Since a system may allow a user to play multiple roles at the same time, single-role actors can lead to use cases which address too narrow aspects of system usage, and thus disabling analysis of how different system operations interact. A *single-actor-view* approach simplifies the use case concept. Any secondary actors are communicated with through the system, meaning that no direct actor-to-actor communication is modeled. The use case description also contains a list of conditions defining a context in which the specific flow of events of the use case can occur, including *invocation, termination,* and *flow* conditions. Flow conditions state the assumptions about the user and system behavior at some point during the use case, but is not necessarily true at its invocation. These different conditions are used in the synthesis phase for finding relations between use cases. Use case descriptions are structured and terminology is unified through the data dictionary.

2.17 Use Case Dialog Maps

Wiegers focuses on the aspect of use case development that involves interaction with the customer [Wieg97]. Use case workshops are conducted to walk through use cases in order to identify the actual functional requirement, exception and decision situations. Complex use cases used *dialog maps*. The maps represent a possible user interface at a high level of abstraction. They are similar to a state transition diagram with each dialog element, such as a screen, window, or prompt shown as a state. Test cases are developed to verify both the specifications and the dialog maps. Business rules are separately documented. Multiple iterations allowed the specifications to grow.

His approach utilizes the concepts of threads to design and implement use cases incrementally. New modules and classes are developed as needed to support any new use cases. Thus, a thread is a grouping of such modules or classes that implement a specific set of related requirements. Use cases provide the mapping of requirements to threads.

2.18 Service Usage Models

Kimbler and Sobirk describe an approach for use case driven analysis of feature interactions that utilizes service usage models as its central element [KS94]. The goal is to detect and resolve undesirable interactions on the early stages of the service life-cycle. This method assumes that a requirements specification, consisting of a list of services described as basic functional components, or features. Activation, parameterization, and invocation procedures are also described. A use case model is then developed and transformed into a *service usage model* (SUM) which describes the dynamic behavior of the system services from the user's perspective. Each service in the SUM is modeled in a separate *service usage graph* (SUG), which is a state oriented diagram demonstrating dynamic relations in the form of transitions among user states and features of the services. Automating the creation of service usage graphs is considered the most difficult step.

The transformation from the use case model to the SUM is accomplished through a series of steps. First, the informal use case descriptions are converted into event sequences resulting in an auxiliary *analyzed use case model*. External system states in which the use can make alternative decisions are then identified. These *usage states* are combined into another auxiliary model, referred to as the *usage model*. Service usage graphs are then derived from the usage model by merging the action sequences from the analyzed use cases into the usage model, thus describing what happens when the system goes from one state to another. A *common state* is a state that occurs in more than one SUG and reflects the fact that several services can be combined into a single use case.

Once the service usage model is constructed, scenarios may be analyzed to detect interaction-prone feature pairs, i.e. those features that access the same service or modify the same data. The detection algorithm has three phases. First all pair-wise feature combinations in a service-free context are analyzed. Next, all potential service combinations are derived from the SUM resulting from common states. Any intra-service feature interactions are also detected in this phase by analyzing all possible sequences of features and states for each SUG. Finally the information from the first two phases is combine to detect cases when two interaction-prone features are invoked in different services.

2.19 Scenario Contexts

Zorman argues that scripts, storyboards, and object interaction diagrams are more structured formats for scenarios than text based use cases [Zorm95]. Three relationships are explored: within scenarios, between scenarios, side scenarios. Relationships within scenarios involve objects, measures/types, spatial elements, temporal elements, and behavioral elements. Relationships between scenarios involve different viewpoints, scenario evolution, elaboration, disjunction, and composition. Side relationships serve to describe a building block's details, such as value ranges for a unit of measure.

Scenario representation should conform to these properties: 1) no domain knowledge in the representation, 2) understandable to people with different backgrounds, 3) support for both depiction and description, 4) support for a range of structures, 5) serve as building blocks with domain knowledge that form the basis for automated tools, and 6) provide expressive behavioral constructs including temporal and causal behavior.

2.20 Scenario Strategies

Coad, North, and Mayfield identify fifteen strategies for working out dynamics of object models with scenarios. They define a scenario as a sequence of object interactions that deliver a specific feature of the system [CNM95]. These strategies are intended to also discover addition responsibilities and scenarios and for testing. The first five scenarios deal with the process of developing and using scenarios. The *select key scenarios* strategy emphasizes demonstration of the satisfaction of key system features through key object interactions. Sub-scenarios may aid in comprehension. The selection of scenarios that stretch the model for completeness are recommended. The *where to begin a scenario* strategy suggests that human interactions as the likely place to begin, but that problem-domain and system-interaction services should also be considered. The *act it out* strategy adopts a role-playing of objects, essentially adopting a CRC approach. The *two-pass scenario development* strategy supports making an explicit decision regarding how much detail should be included in a specific scenario. The first pass only uses existing objects, if possible. The second pass adds objects, responsibilities and interactions as needed. The *describe scenarios with a scenario view* involves a recognizer, end-to-end performance constraints, and a time-ordered sequence of sender and receivers services. The remaining strategies concern object interaction strategies for implementing the scenarios, such as passing an object as a parameter, abstracting interfaces, and active objects.

2.21 Use Case Modeling Concepts for Large Business System Development

Armour, Boyd, and Sood describe a use case modeling approach that prescribes steps that result in several artifacts [ABS95]. One artifact is a *context diagram*, which includes identification of external parties. Another is a *functional area model* that identifies functional areas along with the use cases that involve these areas. Two other artifacts are a *use case summary matrix* and a *use case dependency diagram*. The former serves as a glossary of use cases as well as defining other terms. The latter provides a mapping of use cases to workflow models as well as linkages among individual use cases.

Four levels of use cases are specified. *High level use cases* describe the interactions by actors. *Expanded use cases* describe the main course of action only. *Detailed use cases* describe any exceptions or alternative courses of action. *Abstract use cases* described common functionality. Annotations of nonfunctional assumptions, such as performance and security are placed in one section.

2.22 Distributing New Requirements On Existing Architecture and Use Case Episodes

Regnell, *et. al.* describe a methodology for modeling how new requirements are distributed on a hierarchy of existing system components [RD97]. Their approach combines use case models with a hierarchical component model through activities of *functionality specification*, *component specification*, and *functionality distribution*, which are carried out recursively on different abstraction levels. The *functionality specification* activity creates a use case model for a component from a black-box point of view, based on textual requirements and its architectural environment, which are expressed in terms of supporting components and secondary actors. The *component specification* activity creates a component model for a component by describing its contained components and its internal and external communication structure in terms of interfaces. The *functionality distribution* activity creates white-box use cases using message sequence charts specifying how requirements are distributed on contained components.

One particular problem encountered was the fragmentation of use cases, in which many use cases only comprised a small part of the functionality required by each actor. The "uses" and "extends" relations made it difficult to apply and resulted in increased fragmentation. Another project placed greater emphasis on use case definitions reflecting the actor goals and undivided use cases functionality. The significant reduction in the number of use cases was offset by redundancy between use cases. The introduction of the episode concept was proposed as a solution to this problem. This was introduced in a supporting paper as part of a use case meta model [RAB96]. Their meta-model describes a *use case* as a composition of *episodes* that are in turn composed of *events*. An event is a *stimulus*, *response*, or *action* and has associated *parameters*. A use case has an associated *context* that defines *preconditions* and *post-conditions*, and is used to describe a *service*, in order to satisfy a *goal* originated by a *user,* which is an *actor* that participates in the *use case*.

2.23 Use Case Reuse

Knowles argues treating requirements model objects as versioned components is relatively expensive. Fined grained versioning may also be unnecessary, but could be of value for managing modifications. He question what the process should be for developing a use case for a component or for standardizing roles within industry domains [Know95].

2.24 Use-Cases, Interaction Diagrams, Hypermedia and Visualization

Alvarez, Dombiak, and Prieto consider object interaction diagrams a good starting point for a formalization of the information contained in use cases [ADP95]. They argue that an evolution of use cases must trace in both directions – forward from the beginning to observe the evolution; backward to understand reasons for the evolution. They also describe animation visualization of objects involved in the use case. Hyperlinks are used to explore detailed use cases.

2.25 Hypertext Use Cases

Zeien proposes that use cases should be constructed as a dynamic web of documents allowing a reader the ability to follow the business process, including exploration of alternative paths, through hypertext links [Zeie95]. Comments, issues, and questions can also be associated through hypertext links. He recommends definition of a set of standards several areas. Standards for overall web architecture should establish how multiple use cases will be linked and how readers can browse the information. Standards for common project information, such as issue logs, comment logs, phone lists, project glossaries, and checklists should also be defined at this level. A standard use case template should be defined identifying what should be recorded. Establishing the set of related documents that make up a single use case should also be defined, i.e. where the information is placed and how the links are established. The author also suggests developing a predefined toolbar to facilitate document jumping. Process standards should include adding uses cases, updating information, providing notifications of updates, reviewing information, and adding comments. Finally a common approach to the use of sound video and graphics should be developed.

2.26 Requirements Scripts

Bertreaud and Bezivin describe requirement scripts that are used in the construction of a requirements model and supported by graphical browser [BB95]. The conceptual entities used in this approach are a hierarchical structuring of a document, sections, and scripts. A document is an abstract entity specifying a set of files of a precise subject, It is composed of sections which capture interviews with domain experts in any form, such as text, graphics, or sound. Scripts are associated with sections and are the focus of this approach, defining the key requirement aspects that must be captured in the section.

A requirement script has four types of attributes. Textual attributes include information such as objective, pre- and post-conditions, measures, performance, reliability, security, date, writer, and status. Attributes referencing other scripts include inheritance, sequence, and compositional information. Actor and object attributes identify the actor that triggers the script and objects are used by the script, respectively. The attributes are displayed both textually and graphically using the OOSE graphical notation for use case and object stereotypes.

2.27 Animating Use Cases

Thomas cites three problems with use cases [Thom95]. First, each use case consists of fixed written steps that trace one path through the usually infinite set of possible paths. Secondly, use cases are static, meaning that it is not easy to amend a use case once its behavior has been written down. Finally, use cases are the product of what is essentially paper and pencil technology.

The author advocates her TOA system that includes both text and graphics. Object diagrams are animated to enable the modeler to determine if the system behaves as expected. Facilities exist in TAO to simulate methods by input values from the keyboard as the animation progresses. As errors are found in the model, changes may be made to the object diagram and the method stubs. The system was limited by testing which is still ad-hoc. There is a need to generate scenarios automatically. Second, there is no way to ensure essential cases are tested and regression tested when model changes.

2.28 Requirements Definition and Verification Through Use Cases

Hansen and Miller define primary use cases as supporting major functionality and secondary use cases as alternate, supporting, reused use cases [HM95]. A stratification grouping is aimed toward framework development. Use case points which combine analysis objects, transactions and technical complexity is the first project estimate metric. This is followed by a prototype complexity analysis that adds feature interactions, reuse, and framework assistance.

2.29 Formal Verification of Use Cases

Mendoza-Grado describes use case validation as formalizing behavior as a set of execution sequences [Mend95]. Object interaction diagrams are one form of a use case formalization. One example of validations is verifying that no message is accepted more than once by a recipient. Another example is that every message sent is received at least once by the intended recipient. There can be terminating sequences and cyclic sequences. However, there should also be consistency to avoid deadlocks, livelocks (i.e. infinite loop), and improper terminations. He considers the use of temporal logic to express things that should never happen as very complex to analyze. He provides a three step process: 1) binding uses cases to objects, 2) elaborating message sequence charts, and 3) defining the data exchange among objects as fully as possible.

2.30 Validating Component Use Cases

Davies, May, Wardell, and Wooding describe techniques for developing reusable business components [DWMW96]. Component partitioning is validated through CRC techniques and role playing of use cases. Applying multiple use cases to the same components helped to probe the correctness of each component's behavior in different contexts. A correlation between design quality perception and the number of use cases it satisfied was found.

2.31 Iterative Use Case Prototyping

Collins cites use case refinements as user requested changes [Coll95]. Some are user interface presentation changes; others affect the way the system works. Some symptoms of the latter are requests to change the order of performing commands, or combining two options that currently do not work together.

3 Approaches Related to Use Case Modeling

This section contains treatments on a wide range of related topic to formalisms for the representation of use cases. An attempt was made to provide some organization of the topics such that they would lead in to one another. Accordingly, this section begins with several papers that relate to workflow automation and business processes. We next look at modeling of events and roles which lead into transactions and business rule modeling. A slight shift of gears moves the topic to processes for developing, evolving and transforming requirements. We then get more abstract again by looking at the representation of information and knowledge. We wrap up with more concrete aspects related to collaborations and contracts.

3.1 Action Workflow and Speech Acts

Medina-Mora, Winograd, Flores, and Flores describes the Winograd/Flores speech act model which has been integrated into an action workflow management system [MWFF92]. The workflow is defined as an interaction between a customer and a performer and comprises four phases. The proposal phase is where the customer requests (or the performer offers) completion of some action. The agreement phase captures a mutual understanding on the conditions of satisfaction, with an implicit shared background of standard practices. The performance phase concludes when the performer declares to the customer that the action is completed. The satisfaction phase is where the customer declares that the competed action is satisfactory. Each phase can be decomposed into separate action workflow phase loops.

The Winograd/Flores model presents speech acts with five illocutionary points: assertives, directives, commissives, expressives, and declarations. Assertives commit the speaker to something being the case, such as ‘it is raining’. Directives get the hearer to do something. Commissives commit the speaker to future action. Expressives express a psychological state of affairs, such as apologizing. Declarations establish a correspondence between a proposition and reality, such as pronouncing a couple married.

3.2 Workflow Management System for BPR environments

Beedle describes a workflow manager in the form of several process patterns: CaseWorker, ApplicatioFollowsProcess, ReifyTheProcess, and SharedBusinessObjects [Beed97]. The process begins with a case worker is assigned work through a work list, which can be either pushed or pulled. Case workers are provided the ability to route work. Business scenarios have polymorphic interfaces according to different clients and situations, which are designed and managed within the software. Enabling applications cooperate to form a corporate memory to remember the steps of the process.

3.3 Abstract Business Processes

Gale and Eldred describe abstract business processes (APBs) based on the Porter Value Chain. The thrust is that an enterprise is a sequence of activities taking source materials and producing an output for the next step [GE97]. *Value* is determined by the consumer of the output in terms of what they will exchange in order to derive its benefits. The processes are recursive, covering the functional areas of management & control, marketing, inbound logistics, operations, sales & services, outbound logistics. Six generic use cases are defined: customer interface (for opening, negotiation), customer product transfer, customer assessment, customer payment, and customer interface (for service). The management and control APBs communicate through their superior or subordinate APBs through five use cases: direction intervention, performance feedback (either superior or subordinate), and information gathering.

3.4 Organization Knowledge

Fingar, Clarke, and Stikeleather contend that as corporations become more extended and externally integrated, community memory and organization knowledge become the property of the information systems, not of the people within the organization [FCS97]. Cognitive science plays a central role in new-era information systems. An information architecture provides a business semantic framework for the information components of the business: *subjects, events, roles, associations*, and *business rules*.

They also describe several architectural layers. The *events* layer essentially replaces the traditional transaction by allowing users to define events of the business that may influence previously defined scenarios. For example, events such as a competitor changing its price or a new employee being hired, tend to generate many transactions. The *agent* layer allows the organization of event driven workflows with the *view* layer supplying renderings for the human interface. These two layers are intelligent, recognizing both the context and content of the information being exchanges with the external world. Simulators in the form of business object models will be used to design critical business processes. Techniques such as requirements gathering use cases are likely to give way to more powerful approaches such as those used in knowledge engineering. Future methods will focus on using, not creating objects. This represents a shift from procedural development to component assembly. Defect free components are essential

3.5 Key Event Dictionary

Winant and Frankel present that their key event dictionary which is intended to enable clear delineation of functional requirements, the classes involved, the required response sequencing, and the state dependent behavior [WF97]. The following elements are identified: 1) the *key event*, which is a concise phrase naming the event trigger, 2) the *detection mechanism*, indicating whether it is external, temporal, or a class data item that is monitored, 3) *event input data* comprised of a list of data items expected to arrive with an external event detection, 4) *responses* in terms of required functional activities, 5) *class data* indicating which class attributes are affected by the execution of a single response, and 6) the access type, which is either read or write. There can be many responses and related class data and access type to any key event.

Use cases are described as meaningful sequences of key events. They are considered complementary to the key event dictionary. Use cases overlay their actions onto the object states until all key event dictionary details have been allocated and the object state machines execute as a well oiled machine in response to external and internal stimuli. Classes with behavior will need to track the current state of each existing instance with a new status attribute. Use cases employed to drive the process are easily prioritized by dependence, mission criticality, performance thresholds, customer interest, prototyping capabilities, and phased releases. Key event dictionary details are allocated according to the priority level.

3.6 Behavioral Lifecycle Modeling

Chafi presents an event triggered behavioral lifecycle model that is supported by a set of stereotyped pattern based behavioral service methods [Chaf96]. The generic model is composed of five phases: event, recognition, communication, decision, and transaction (ERCDT). The *event* phase contains the behavior triggering event. The *recognition* phase provides the services to recognize that relevant events have occurred. The *communication* phase concerns notification of all interested objects in the event. The *decision* phase established the course of action. The *transaction* phase organizes how the services cooperate to provide required actions in response to an event.

Complex event behavior that spans multiple objects is supported through an event *scanner* that monitors for its occurrences. The *decider* tracks multiple event occurrences as part of its decision making process through use of a *coupler* service method. If more than one transaction is required, each one is assigned to a *reactor* method for transaction management. The reactor guarantees that all rules in the transaction are applied. An *executer* service method is called upon to carry out each necessary rule execution, which in turn calls upon *checkers* that verify conditions and *invokers* that make sure the actions have been executed.

3.7 Transaction Based Analysis

Rawsthorne argues that functional requirements can be captured through object interactions discovered as a result of transaction based analysis [Raws95]. This approach is based on both use cases and responsibility driven design. A transaction is an ordered set of collaborations between objects. Once a transaction is identified, it will lead to sub-responsibilities, which in turn lead to new objects. The object that owns the responsibility and orchestrates the transaction is called the transaction manager. Even without object interaction, a transaction should be established, since an object may later be partitioned. Patterns can be captured by recognizing similar ‘shapes’ in different transaction diagrams.

3.8 Transaction, Event and Rule Models

Silva discusses an active object-oriented database modeling technique that consists of several multi-level modeling views [Silv95]. The object model provides representation of rules in addition to attributes and operations for objects and classes. The event model provides declarative and active behavior through events. The rule model provides declarative and active behavior through rule sets, coupling modes and interactions. The behavior model provides modeling of database transactions and object methods at a conceptual level so that the context of rule evaluation is understood. The rule model provides a visual notation as well as semantic for overriding rules and coupling modes of rule execution. Each rule is decomposed into component pieces with accompanying graphical notation, such as constraints and attributes.

3.9 Collective Behavior in Business Rules and Software Transactions

Kilov, Harvey, and Tyson contend that behavioral specification, the precise specification of system semantic, is essential to understanding business rules [KHT95]. Object interaction diagrams emphasize what messages are exchanged, not about effects. Interactive machines cannot be expressed well in first order logic, however object interactions are best modeled this way. They conclude two viewpoints, the analyst/end user view and the designer/developer view, must be linked explicitly through an invariant. Furthermore, graphical notations may not be easier to understand than textual ones and that exactness may sometimes be traded for clarity.

3.10 Business Rules Model Based on Events, Conditions, and Actions

Herbst provides a meta-model for business rules that intended to derive a rules-based systems analysis methodology that incorporates a repository system for rules [Herb95]. The structure of a business rule contains an event, condition, then-action, and else-action. Static constraints and entire processes can be defined in this manner. For processes, the action component of a rule provides the link to raise events to trigger other business rules. Abstraction and specialization of business rules is based on the data-flow diagram leveling technique – all inputs and outputs remain the same regardless of the level. In terms of business rules, this means that the triggering event and the raised events must be identical. A specific subset of business rules may be assigned to a more detailed level corresponding to sub-processes. The specialized business rules always raise the same events but they may trigger different business rules, depending on the subset of business rules which define the current sub-process.

The Herbst business rule meta-model consists of six sub-models. The *Business Rule* sub-model is composed on *Events*, *Conditions*, and *Actions*. The *Modeling Construct* sub-model embraces the entity-relationship model, which is represented through a *Data Schema* and is sub-typed as *Entity*, *Relationship*, and *Attribute* elements. Any one of the business rule components may directly reference any modeling construct. The remaining sub-models place the business rules in context. *Origin* indicates whether the rule originate inside or outside the organization. External rules are either *natural facts* or (legal) *norms*. Internal rules are designated as *primary* if originating from a source document, otherwise they are derived and referred to as *secondary. Organizational Unit* is concerned with *intra-unit* and *inter-unit* rules. Inter-unit rules exist where an event, condition check, or performance of a action may occur in different organizational units. Business rules only define certain aspects of the *Process* sub-model element, such as the starting and ending rules. Other attributes such as name, owner, involved actors and organizational units, or relationship to other processes is not defined by business rules. Critical points in the process may depend on events that from the environment. These result in broken links in the process that prevent the chaining of rules to define the complete process. *Software Component* identifies where each of the three component of a business rule is realized – i.e. either as a software component, such as a function or database trigger, or through the system environment.

3.11 Business Rules Model Based on Assertions and Derivations

Hay and Healy define a business rule as a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business [HH95]. The GUIDE project developed a model that includes three business rule sub-types: *structural assertions* (terms and facts), *action assertions* (integrity constraints, condition, and authorizations), and *derivations* (mathematical calculations and inferences). Text orderings in facts can be rearranged to suit the perspective of the first role referenced. Action assertions may also be classified in two other ways. They can be typed as *enabler, timer*, or *executive*. An *enabler* permits an attribute's value or activates a rule if true. A *timer* is a specialized enabler that bases its truth-value on whether or not a time threshold is satisfied. An *executive* causes execution an action if true. The other classification for assertions is whether the rule is controlling in nature or merely influencing in nature, i.e. must vs. should). Several other types of actions assertions derived from the Ross method [Ross97] are also included.

They place their research in context to the Zachman reference model to define the scope of their work. The reference is from the designer's stakeholder perspective and data and motivation aspects. Aspects that are explicitly excluded include softer rules involving human judgement, work flow processes, rule maintenance, and modeling situations where rules are ineffective.

3.12 Hierarchical Policy Model

Nassiff describes a Hierarchical Policy Model (HPM) consisting of five components: 1) a policy proponent. 2) a policy approving authority, 3) the importance of a policy as it relates to the hierarchical structure of an organization, 4) the relationship of the policy statements to the data model and its constructs, and 5) the relationship of the policy statement to other models in the architecture [Nass91].

Policies are created through a three phase process. First, the *collection* phase gathers source documents such as regulations and memos. External policy, including laws, treaties, and customs are also assembled. Next, facts are deduced from the policies. Finally, tests for incompleteness, redundancy, are inconsistencies are performed. The *validation* phase is assures that all policies are consistent, concise, and complete. Each policy must also be approved and supported. The final phase is *classification*, which generates a five-tuple in the form of: P(group, number, authority, proponent, hierarchy). The *process group* associates with a use case (cluster) or functional process. The *policy number* associates the policy with an organization hierarchy element in order to establish the proponent and authority. The *policy authority* is a person who can override or approve a specific policy. The *policy proponent* is the person with the responsibility for creation and maintenance of the policy. The *policy hierarchy* rates the policy in accordance with the following binding strength: 1) contract, constituting an agreement enforceable by law, 2) directive, which is an order or authoritative instruction with specific authority, 3) business rule, which represents a principle or accepted rule governing the procedures followed by the organization, 4) plan, indicating a scheme or method of action for obtaining a specific goal, or 5) opinion, signifying a belief of judgement that rests on grounds insufficient to produce complete certainty.

Policies can be place into tree structures based on different policy values. For example, the hierarchical value can reveal levels of management control. Authority value reveals a chain of command. Proponent level reveals functional characteristics of the organization.

3.13 Agent Based Language for Building And Eliciting Requirements

Dubious, Du Obis and Petit describe ALBERT, an agent based language for building and eliciting requirements [DDB93]. They provide graphical declarations and complement them with textual constraints. Traditional requirements specification through formal language are combined with object-oriented conceptual modeling. Systems are viewed as a composite of software components, manual procedures, real-world entities (e.g. stock price), specific hardware devices (e.g. thermometer). This approach deals with commitments, similar to contract, but they provide a dual focus on agents and interactions. The constraints that compose their approach are state behavior, effects of actions, responsibilities, perceptions, publicity, and commitments. A formal proof of their language is provided.

The authors discuss the elaboration of the requirements document as a process involving a sequence of transformations of the current state of the requirements document. First, the problem specifications are expressed monolithically, i.e. as a black box. Then, new subsystems are identified and responsibilities are attached to each of them so that the refinement preserves the original prescribed behavior. The process continues recursively until arriving at the “terminal” sub-systems. A “terminal” sub-system is one that designers have agreed to implement based on its attached properties.

3.14 Incremental Evolution of Specifications

Johnson, Benner, and Harris describe Aries – acquisition of requirements and incremental evolution of specifications [JB93]. It is a system that partitions requirements and relates them to certain perspectives of domain models. Some requirements, such as federal regulations, may be necessary across all systems. Others address specific aspects, such as aircraft flight, which has different, but overlapping requirement for traffic control than for flight navigation. The system provides a library of evolutionary transformations that constitutes reusable knowledge about the *process* of requirements analysis. When a change to a component is desired, the system retrieves all transformations that can perform it, ranked according to generality. Steel’s Constraint language is used to enforce nonfunctional requirements

The system is biased towards simulation over theorem proving because it is tractable for large specifications and more tolerant of incompleteness. It can provide feedback about behavior even with incomplete specifications. It also allows dealing with appropriate levels of abstraction by focusing on aspects of relevant behavior and suppressing the irrelevant details. Specifically, validation is accomplished through scenarios and anti-scenarios – behavior that the system must (not) exhibit. An influence analysis determines which parts of the specification potentially interact with the actions described in the validation questions. Those components that do not interact can be replaced with simplified approximations, otherwise executable scenarios are needed.

3.15 Commitment-based software development

Mark, Tyler, McGuire, and Schlossberg describe their development approach as a process of negotiation in which implementation decisions are changed to meet new specifications [MTMS92]. Architecting is used to meet requirements rather than coding. The design comprises a large number of interacting constraints with commitments representing pre-selected constraint types such as input/output, data access, and control. The intent is not to build executable specifications, rather it is to develop robust descriptors for retrieval of modules based on behavioral requirements. There is no guarantee that module descriptions can be realized in working code, or that the code correctly implements the module. Primary attention is paid to the interaction of design decisions in an evolving design through context specific guidance. Substituting modules is deliberately interactive. The developers are expected to play an active role.

The general process is to initiate design modifications by finding the set of more specific module descriptions that are consistent with the subject description and then compute the new commitments for each alternative to be included in the design. The process continues recursively, until each commitment can be met by further module description modifications.

3.16 Baseline Requirements Document

Burns and Halliburton discuss detailed customer involvement with OO design and application generators [BH89]. An ongoing requirements review and approval process includes a signed baseline requirements document. Changes to this baseline are performed as a result of simplification, consistency, completeness, and clarification requests. Customer involvement in the operation walk-through focused on specific details of system usage. Software technology assistance through an application generator generator (i.e. a generator to create domain specific application generators) is also discussed.

3.17 Deriving an Object-Oriented Design from Functional Specifications

Alagar and Periyasamy describe a methodology that emphasizes the role of formal specification and application domain modeling in deriving a design [AP92]. Formal specifications derive from informal requirements. Automated transformations then take the formal specifications through to design. The main goal of the formal specification phase is to describe abstractly what the system is supposed to achieve. It provides a formal model of objects in both the problem domain and the system domain so that we can formally reason about the behavior of objects.

They appear to argue for a formal domain model that bridges requirements and design. What is still not provided is the technique for formalizing the specification from informal requirements. They have focused on the formal specification to design transformation step they use VDM as the formal specification technique in the provided example. Their research provides motivation for increasing the level of formalism in specification models in order to allow design models to be automatically generated.

3.18 Information Mapping

Horn presents a conceptual framework for the analysis, linkage, and display of knowledge hypertext based systems [Horn89]. The dimensions of hypertext systems cover the scope of information sharing, the mode of operation (i.e. using, authoring, editing, or administration), and the scope of the application. System design issues need to address the sizes of nodes, the kinds of links and filters, and how to anticipate the users needs. User issues include inadequate and missing reading cues, accommodating reading behavior for both novices and experienced users, meta-cognitive skills and general leaning skills. Meta-cognition is concerned with understanding and controlling one's own cognitive processes by organizing, monitoring, and modifying them as a function of learning outcomes. For example, meta-cognitive skills include the ability to set realistic learning objectives, determine prerequisites, and when to seek expert advice or utilize resources outside of the system. General Learning skills, on the other hand address such things as how to put something into long term memory, taking notes, practicing, or the ability to paraphrase, summarize, and abstract information.

Horn describes three domains for structured hypertext. A relatively stable discourse domain is best suited for documentation and training, such as technical manuals and policy manuals, or an on-line product knowledge data base. The disputed discourse domain can be used for argumentation analysis. The experimental discourse domain is appropriate for scientific reports and articles including background and bibliographic information leading up to the research, conducting the experiment, and reporting the experiment.

Information mapping is based on four principles for constructing blocks. First, all information must be in small, manageable units. Second, each chunk should contain information that relates to one main point based on that information's purpose or function for the reader. Third, similar words, labels, format, organizations, and sequences should be used for similar subject matter. Fourth, every chunk and group of chunks should be labeled according to consistent and specific criteria.

Managing completeness by key block and topic matrix is accomplished through a series of analysis processes including scope, information gathering, organization, sequencing, and presentation. Examples of the most frequently used block types are: checklist, definition, diagram, example, flow chart, outline, parts-function table, and rule. The blocks are integrated through hyper-trails, which are a set of various types of links that organize and sequence information. Prerequisite links can map what users must do in order to understand more advanced topics or accomplish more advanced skills. Classification links can find more specific or general information. Other types of links include chronological, decision, and example links.

3.19 Problem Frames

Jackson focuses on a conceptual framework centered around problems rather than solutions [Jack94]. A problem can be characterized by its principal parts (hypothesis, conclusion) and a solution task (to show how the conclusion follows from the hypothesis). The principal parts of a problem to construct are the unknown, data, and condition. The solution task is to construct the unknown so that it satisfies the condition with respect to the data.

He provides three examples of problem frames: the JSD problem frame, where the solution task is to construct a *system* that models, or simulates, the *real world* and satisfies the *function;* the workpiece frame where the solution task is to construct the *machine* to perform the *operations* on the *workpieces* in response to the *operation requests;* and the environment-effect frame where the solution task is to construct the *machine* so that it senses and controls the *environment* through the *connection,* and ensures satisfaction of the *requirement.*

The problem frames are far from interchangeable. The chosen frame must fit the problem. A good software development method prescribes a very specific problem frame and exploits its properties to the fullest. A simple problem is a problem for which we have a close-fitting frame and an effective method that exploits it. For example, decomposition has traditionally stood as if they were self-sufficient. “Having divided to conquer, we must re-unite to rule”. There is difficulty when the same domain object appears as two different principal parts in two different problem frames. To develop one implementation that conforms to both applies “the Shanley Principle” should be used. A classic illustration of this principle is a comparison of the V-2 rocket vs. the Saturn rocket. The Saturn uses the fuel pressure to functionally replace structural rods of the V-2, thus solving two problems with one domain entity.

3.20 Analysis Patterns

Fowler has developed a set of analysis patterns that address common business modeling problems, such as organization structure and observation/measurements [Fowl96]. One of his modeling principles is to explicitly divide a model into operational and knowledge levels. The operational level records the day to day events of the domain. The knowledge level records the general rules that govern the structure.

One set of analysis patterns is concerned with planning, protocols, and related actions. These patterns have a close affinity with the structure of use cases. Since an action is often performed according to a protocol, the *protocol* pattern addresses the problem of performing repetitious standard procedures. A protocol (i.e. how we go about doing something) can be a name, description, textbook page, video, etc. This pattern is a simple behavioral meta-model that provides for decomposition into sub-protocols. The *plan* pattern is more flexible and permits more accurate tracking of individual protocol steps. The source of the pattern stems from the medical treatment problem domain, so accordingly combining plans and protocols injects an ad-hoc nature into the sequence of actions the make up the protocol. This characteristic is important to such professional when considering the circumstances of an individual patient.

When the two patterns are combined, they share the same sequences of actions. The protocol pattern exists at the knowledge level while the plan pattern exists at the operational level. Directed acyclic graphs can be used to represent both plans and protocols. A plan is a directed acyclic graph of proposed actions with the arc on the graph corresponding to the dependency relationships on the action references. A protocol’s directed acyclic graph, which exists at the knowledge level, is slightly different. Instead of being composed of subsidiary protocols, the graph is formed from protocol references. This is necessary since one protocol can appear in more than one step of another parent protocol. At knowledge level, protocols are a bag, equivalent to a group of actions. However proposed actions, which appear at the operational level, are considered a set since the same action cannot be performed twice but two actions can have the same protocol.

Actions are involved in several related patterns. One distinguishes between proposed and implemented actions. Others deal with action states of completed, abandoned or suspended. An interdependency or linking of actions is also discussed, such as when one action can effectively handle actions in two plans.

Fowler also discusses related issues to the analysis patterns, including implementing variability and rule firings. Five approaches for introducing variability in models behind an interface are described. The *Singleton* class is generally the poorest choice because subtypes are rather artificial and it leads to may classes. The *Strategy* pattern involves sub-typing a method rather than the whole class. It is better that the Singleton class because the method class performs the calculation and the method choice is eliminated. An *Internal case statement* adds a case clause and private operation, which is not much more work that method classes of strategy pattern. *Parameterized methods* can hold a number the can use the same method if simple, but is limited to a few parameters. The *Interpreter* parses a string and is good for simple formulas. Fowler prefers a combination of the strategy pattern that is parameterized.

Several strategies are offered for separating the rule firing from the rule itself. They have been generalized here from the accounting domain analysis patterns that he described. *Eager firing* can occur at transaction creation, individual entry creation, or through an *observer* that registers with a trigger. Observers are useful, but complex and difficult to debug. On the other hand, transactions no longer need to activate the rules themselves. *Entity based firing* assumes that the entity will be told to process itself. It must keep track of what has not been processed and for maintaining the correct processing order. This approach is good for daily batches. *Rule based firing* is similar to entity based firing, except the responsibility for deciding what has not been processed now lies in the rule instead of the entity. The rule is invoked by an external agent. *Backward chaining firing* is an entity based variant that causes dependent entities to process themselves. This is accomplished by maintaining knowledge of which entities are triggers for a rule that has itself as output.

A related issue to rule firings is the method of selection over a collection for applying rules. Three strategies are described. First, the whole collection can be returned and the entity then performs the selection. This involves a considerable amount of copying and moving. Second, a method can be created. However, if there are many different methods, the interface can become ver large. Finally, a filter can be used such that an object encapsulates the query. One additional concept discussed was the issue of rule instance overriding, which relates the whether a rule should be copied or shared.

3.21 Archetypes, Deltas, and Frames

Basset addresses the problem of inaccurate modeling of domains, stating that inflexible inheritance and software entropy are barriers to software reuse [Bass96]. He prescribes software models that bridge the gap between a vague problem domain and the preciseness of applications – combining two or more models leads to inconsistencies.

Simple and reversible changes are needed to reduce entropy. The process of *binding* provides variability at the expense of performance and possibly also at the expense of usability. We can change methods and data structures at *construction time*, but not at *run-time*. This essentially separates the operations of reuse from use. *Rogue objects* cause problems with class hierarchies. Multiple inheritance may help, but generally creates a plethora of *polymorphs* that obscures the variability in a sea of redundancy.

His solution is to utilize construction-time, as opposed to run-time, changing of data structures and methods through archetypes, deltas, and frames. Archetypes are decomposed to appropriate levels of granularity, thus making deltas smaller (i.e. 0-15%), resulting in reduced redundancies and clearer interactions. The archetypes are implemented through generic components called frames, which are lists of properties that can be overridden at construction time by other frames. This allows the modeler to keep wanted properties, but prune those not needed. The result is a change from passive inheritance to active adaptation.

3.22 Collaboration Frameworks

D'Sousa and Wills define a collaboration refinement as a relationship between two collaborations, where one refines the other. This claim is supported by a mapping of the refined model and actions to the abstract model and actions [DW97]. Collaboration refinements, which are an underpinning of their Catalysis methodology, usually entail two aspects. The first aspect deals with what sequences of finer actions induce each abstract action. These can be specified through state charts, object interaction diagram details, pre- and post-conditions specifications or finer operations. The second aspect deals with more detailed type models of each participant. Many interesting object-oriented implementation frameworks consist of a set of compatible types being "plugged" in to define the specialization. This framework makes explicit the *parameterized* dependencies between a *family* of types rather than focusing on a single type. This results in simultaneous specialization in mutually compatible ways.

3.23 Collaboration Based Design

VanHilst and Notkin suggest that use cases and responsibilities represent collaboration based design methodologies [VN96]. Their approach utilizes role components, which are implemented as source code entities through class templates designed in their stylized way. Different combinations of roles can be generated through templates without changing the role definitions themselves. Object interaction diagrams show operations and roles, but not how they are composed in a single class.

One critical aspect of their approach is the need to determine the order to compose the roles. Extensions and overriding are especially important to the order of template definition, since these hierarchies can be as deep as 20 classes. For example, if operations or attributes are defined more than once, resolution is needed to determine whether they are they shared, repeated, or overridden. Moreover, when one collaboration extends another, we need to identify the calls between roles with the same class. They refer to specialization by inserting ancestors as more common than extending the class hierarchy. Also, delegation is considered a distortion to the design of is-a and has-a relationships. Instead, in their approach change is encapsulated in roles that become integral to the class. Binding to a collaborator type is delayed to compile time. Thus, this static inheritance is more efficient because no indirection is involved.

3.24 Path Expression Groups

Adams and Corriveau describe a technique to describe the interactions of a group of cooperating objects through regular expressions [AC94]. An initiator is typically outside the group and is modeled at the beginning of the path expression using a '>>' symbol. These clients of the group are considered secondary participants in the interaction, as are other server object that are called upon by objects that belong to the group. These groups can participate in inheritance and composition hierarchies. Thus, group behavior can be derived from the path expression of component groups through composition of their respective path expressions. A distinction is made between typical, essential, and exception interactions. Key dependencies between object members of the group are captured through invariants.

3.25 Component Contracts

Holland discusses the design and representation of object-oriented components through the use of contracts [Holl92]. A *contract* is a package of object declarations and object behavior definitions enclosed by a scope of visibility. At implementation-time, a *conformance statement* is used to integrate a contract into an implementation. At run-time, an *instantiation statement* identifies a group of objects as the participants of a contract.

The syntactic description of a class implementation may be split up and distributed to two or more contracts. The different parts are subsequently 'glued' together with the conformance statement of the Contract language. The dynamic object behavior is changed in four ways. First, *multiple interfaces* are possible with each interface visible to a restricted set of objects. Second, *multiple implementations* per object for each message are possible. When a message is received by an object, the appropriate response is determined dynamically. Third, the implicit parameter list (e.g. self in Smalltalk or this in C++) is extended with a set of objects called the contract participants. Fourth, special objects called *Contract Lenses* are created at run time to control how application objects respond to messages and to store the contract participants.

The contract structure consists of these parts: 1) a name, 2) a refinement clause, 3) a participant list of either objects or anonymous set of objects, 4) obligation definitions specifying what each object contributes to the contract 5) include clauses, and 6) invariant definitions. A handy feature of the contract mechanism is the ability to determine if an individual class is eligible to participate as a participant in a contract. Furthermore through the use of conformance statements, differences in the methods names are bridged through the contract lens mechanism.

Contract refinement is accomplished by adding new participants and obligations, adding additional terms to the invariant, or specialization of existing obligations. Specialization of an obligation is very similar to the action of defining one class as a subclass of another. Type obligations specify that the participant must support certain variables and method interfaces. Behavioral obligations specify that the participant must perform an ordered sequence of actions (including message sends) and make certain conditions true in response to messages.

Contracts facilitate the replacement of objects by functionally equivalent objects when a change is needed. This contrasts with frameworks where the creation of object interactions was either hidden inside class constructors or implicit in certain sequences of method calls. This makes for better and easier understanding of the application architecture.

Contracts rely on a few basic principles between natural types and roles. An instance of a natural type cannot exist without being an instance of that type. For example, an instance of the natural type Person cannot stop being a person and still exist. A role instance can exist without playing that role. For example, an instance of Student can exist as an instance of Employee after graduation. The identity of the instance will not change, only the attributes used to describe the instance. Finally, a role must be a subclass of at least one natural type and no natural type is a subclass of a role.

3.26 Exception Handling

Stewart and St. Pierre relate their experiences with a manufacturing framework, citing that many of the recommendations made for the development of the CIM framework are generally applicable to other object-oriented framework developments [SSP95]. Of most interest was their discussion of exception handling. Exceptions are explored in the context of true exceptions rather than errors, i.e. a normal but alternative response. The author's remark that raising an exception is the method of choice in CORBA’s IDL for returning a significantly different signature. They cite four categories for exceptions: 1) no nonstandard exception signatures, 2) simple two-valued exception signatures, 3) more than two modes of return using an enumerated return type to distinguish results that have the same signature, and 4) complex, user defined exceptions.

They also offered the following recommendations: 1) adopt a single source electronic specification management approach, 2) increase supplier involvement in both specification and certification development, 3) develop usage scenarios to clarify the implementation and use of a framework, 4) build on existing specifications, and 4) expand the use of formal description techniques in specifications.

4 Comparison

Use case formalisms may be classified in many ways. The approaches surveyed in this paper present a myriad of classification dimensions. The comparisons made in this section primarily look at use case formalisms and related techniques from a broader perspective of defining, evolving, and managing domain architecture artifacts. Appendix A provides a concept map that embraces most, if not all, of the concepts presented in the approaches surveyed. There are two dimensions that have been chosen that provide sufficient common ground to allow a meaningful comparison to be made. The first one is the primary *focus* of the approach with respect to a use case model’s static, dynamic, and policy properties. The process of use case modeling is also considered in the dimension. Table 4-1 summarizes this *focus* comparison. The second dimension compares representation *formats*. These formats may be textual, graphical, or dynamic in nature. Table 4-2 provides a comparison of use case formalism formats. These tables merely denote the presence of some measure of meaningful coverage by each approach. The intent is to provide an indication of overlap of concepts. Relative merits are not indicated in the table. Therefore, the commentary in each section provides supporting information that may be of interest in making such a comparison. However, the reader should be able to utilize these tables to refer back to each section and further, to the referenced papers, in order to evaluate strengths and weaknesses for himself.

4.1 Use Case Formalism Focus

In order to provide a comparison of research approaches, we have partitioned the focus categories into four general areas of properties: static, dynamic, policy, and process. Although each approach will demonstrate characteristics of each of these four general areas, the basis for comparison is derived from the approach bias for the evolution of use case models either explicitly or implicitly expressed by the authors.

4.1.1 Static Properties

The *static* properties are concerned with how use case model elements relate to each other. *Inheritance* and *aggregation* are familiar object-oriented concepts. *Dependency* and *refinement* are two concepts that are well documented in the UML specification. It is assumed that the reader is sufficiently familiar with these concepts, so that none of these terms warrant further explanation here. The remaining two properties may need additional explanation. *Data* is concerned with the structure of information that is exchanged between actors and the system. *Context* is concerned with how use case model elements relate to each other.

As suggested in the introduction of this paper, for good or bad, use case modeling has been greatly influenced by the philosophy and advances of its originator. Jacobson's recent collaboration with Griss may very well be the single most important reference to use case modeling [JGJ97]. In particular, their enumeration of various mechanisms for introducing variability and discussion for the construction and reuse of use case components have provided significant advances in the maturity of use case modeling. With its close coupling to the emerging UML standard, this text also provides valuable insight as to how and why the UML Use Case semantics are presented as they appear in the current Version 1.1 specification. However, along with being a major shaper comes the burden of hanging on to conceptual constructs that have persisted in confusing use case modelers. In particular, this is a reference to the «uses» and «extends» stereotypes. Conceptually, they made much greater sense when use cases were highly informal narratives. However, with the strict semantics of UML semantics, many believe that these stereotypes have outlived their usefulness. For example, Cockburn establishes a dependency relationship for the «uses» stereotype by making it a single line item in a scenario [Cock97]. This could just as easily appropriately be called «invokes». Adaptive use cases, which are compatible with the UML meta-model, treat most application of the «uses» and «extends» stereotypes as additional scenarios rather than use case fragments that can not stand on their own [Hurl97].

Two other meta-models for use cases were presented – the OML use case model [HSFG97] and the Regnell use case model [RD97]. The former abstracts an *essential use case* model element, which are similar to the archetype scenarios of the adaptive use case UML extensions and the key scenarios from Coad's strategies and patterns [CNM95]. The latter defines and *episodes* and *actions*, which again resembles the adaptive use case meta-model's decomposition of use cases into action segments and delta scenarios. Change cases [EDF96] introduce a slightly different construct that deals with versioning of use cases.

4.1.2 Dynamic Properties

Most of the *dynamic* properties selected also correspond directly to UML semantics. *State, event,* and *action* are explicitly defined model elements that relate to state machines. *Contracts/patterns* and *roles* being derived concepts that relate to the UML collaboration model. The remaining two properties are not well addressed in the UML specification. *Transactions* are larger aggregations of interactions that may provide a basis for decomposition of a use case. *Agents* add mentalistic concepts such as beliefs and goals, which are more typically associated with workflow and business processes.

Several papers discussed business processes and workflow issues. In particular, the agent based use cases [KMJ97] demonstrated IDEF equivalences with use cases, allowing integration with control flow, agents, and plans. The UML extensions for business process modeling could have benefited from contributions in the workflow community rather than stretching the use case and analysis object types from Objectory. For example, the ActionWorkflow process models are the product of a long history of workflow automation research by Winograd and Flores [MWFF92]. Their speech act model provides a more robust representation of use case outcomes that the success/failure dichotomy of Cockburn's use case model. The state of the dialog will tend to modify goals and place definitions of success and failure within a context.

Several other examples of business process treatments include task scripts [HSFG97], the adaptive use case's linkage to the Workflow Reference Model [Hurl97], the addition of scenario steps that occur outside system boundaries in order to incorporate workflow [KPW97], abstract business processes [GE97], event driven workflows [FCS97], and the notion of persistent workflows [Beed97]. It could also be argued that the reification of use cases [Jans95] represents a form of workflow automation similar to Beedle's persistence of workflow objects.

Several approaches that are role based were included in this survey. Chafi proposed an alternative set of object role stereotypes that those presented in the Objectory and Business Process Extensions to the UML [Chaf96]. An approach dealing with role composition can be found in [VN96]. Holland's work on component contracts provides a formalized basis for interfaces that has been adopted in many other approaches [Holl92].

State machines played prominent parts in many approaches. Two of address the relationship between use cases and state machines from opposite directions. One elicits a state machines from use cases [ML97], while the other generates formalized use cases based on state charts [Glin95].

4.1.3 Policy Properties

Policy properties relate to business rules, which are subdivided into three types. First, behavioral *rules* deal with invariants, pre- and post-conditions. *Exception handling* makes a distinction between alternative courses of action and exceptional courses of action, including failures. This property is a focus of [SSP95]. *Composition* relates to functionality that is included rather than how it is enforced. Use case components [JGJ97], Business rule templates [Hurl97], adaptive frames [Bass97], and structure blocks [AB95] represent various mechanisms that guide declarative composition of use case model elements. Commitment-based software development relies on a more interactive architecting approach to meet requirements [MTMS92].

Although definition pre- and post-conditions is relatively common, most approaches only give passing references to business rules. The integration of business rule templates with adaptive use cases represents the most formalized approach [Hurl97]. The importance of declarative business rules warranted considerable treatment in this paper, obviously reflecting a bias of this author. Competing rule meta models can be found in [Herb95] and [HH95]. Silva and Ross present two classification schemes for business rules that decompose rules into atomic constituent parts [Silv95][Ross97]. Nassiff takes a slightly different approach to classification dealing with [Nass91] dealing with such aspects as authority and importance. Process Properties

The final general category, *process*, is concerned with the development process and participants. As stated at the beginning of this section, *evolution*, is the key basis for this comparison. Several aspects of use case evolution have been explored. Three noteworthy examples are maintaining a history of use case evolution [ADP95], metrics for measuring the impact of change [EDF96], and functionality distribution [RD97]. Another reference to metrics concerned use case points [HM95]. Process standards are addressed in [Zeie95]. Negotiation for scope and timing of functionality is closely related to evolutionary concerns. Commitment-based software development concerns the process of negotiation in which implementation decisions are changed to meet new specifications [MTMS92]. Use case components assume depend on requirements that closely available components, often resulting in either changing application requirements in small ways or negotiating more substantial changes [JGJ97]. Adaptive use cases rely on fit assessment models [Hurl97]. Change cases involve a process that considers budgetary constraints define change cutoffs that tend to limit upstream changes [EDF96].

Parameterization is closely related to evolution, since its intent is to provide future variability. Parameterization was the most frequent approach to introducing variability into use case formalisms. Pattern based variability [Fowl96][GHJV94], adaptive frame based parameterization [Bass97][JGJ97][Hurl97], and instance parameters [Beri97][AB95][RD97][DW97] are representative examples of this property.

Users addresses those approaches that elevate the importance of the user relative to the modeler. Scenario trees [HSGK94] address all scenarios for a particular user. Synthesized usage model take a multiple-role, single-actor view [RKW95]. Dialog maps represent a possible user interface [Wieg97]. Service usage models describe the dynamic behavior of the system services from the user’s perspective [KS94]. Baseline requirements documents define user requests as operations on these documents [BH89]. Collins’ iterative use case prototyping approach cites use case refinements as user requested changes [Coll95].

Tools simply refers to approached that describe automated tool support for their approach. Automated tool support to handle transformations was frequently discussed [KHT95][JGJ97][Hurl97][Beri97][KS94][DDB93][JB93][AP92]. Verification through automated tool support was another area that was well covered [HM95] [Mend95] [DWMW96].

Use Case Formalism Focus	Static						Dynamic							Policy			Process			
	Data	Inheritance	Aggregation	Dependency	Refinement	Context	State	Events	Transaction	Actions	Contract/Pattern	Roles	Agents	Rules	Exceptions	Composition	Evolution	Parameterization	Users	Tools
Use Case Components [JGJ97]		✘	✘	✘	✘		✘	✘		✘	✘	✘				✘	✘	✘		
Structuring Use Cases With Goals [Cock97]		✘	✘	✘	✘	✘						✘								
Adaptive Use Cases [Hurl97]	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘			✘	✘	✘	✘	✘	✘	✘	
Enhanced Scenarios [Beri97]		✘	✘	✘	✘	✘	✘			✘				✘				✘		
Agent Based Use Cases [KMJ97]		✘	✘				✘		✘	✘	✘		✘							
Use Case Maps [Buhr97]		✘	✘						✘		✘							✘		
Change Cases [EDF96]		✘			✘					`							✘			
Task Scripts [HSFG97]		✘	✘				✘	✘				✘		✘	✘					
Use Case Classes [Jans95]					✘															
Use Case Formats [Harw97]						✘		✘			✘				✘					
Directed Use Case Graphs [KPW97]					✘					✘										
Scenario Trees [HSGK94]							✘	✘											✘	
Message Sequence Charts [AB95]		✘	✘		✘			✘								✘		✘		
Elicitation of State Machines [ML97]							✘						✘							
Formalized Use Cases/State Charts [Glin95]							✘	✘												
Synthesized Use Case Modeling [RKW95]	✘	✘	✘				✘			✘									✘	
Use Case Dialog Maps [Wieg97]							✘							✘	✘				✘	
Service Usage Models [KS94]		✘					✘	✘		✘								✘	✘	
Scenario Contexts [Zorm95]	✘					✘										✘	✘			✘
Scenario Strategies [CNM95]					✘						✘	✘	✘							
Modeling Large Business System [ABS95]				✘		✘									✘					
Distributing Requirements/Episodes [RD97]				✘		✘		✘		✘								✘		
Use Case Reuse [Know95]					✘												✘			
Interactions/Hypermedia/Visuals[ADP95]																	✘			✘
Hypertext Use Cases [Zeie95]																			✘	✘
Requirements Scripts [BB95]		✘	✘																	✘
Animating Use Cases[Thom95]																			✘	✘
Requirements Definition/Verification [HM95]																✘				
Formal Verification of Use Cases [Mend95]	✘									✘										
Validate Component Use Cases [DWMW96]				✘																
Iterative Use Case Prototyping [Coll95]																✘			✘	

Use Case Formalism Focus	Static						Dynamic							Policy			Process			
	Data	Inheritance	Aggregation	Dependency	Refinement	Context	State	Events	Transaction	Actions	Contract/Pattern	Roles	Agents	Rules	Exceptions	Composition	Evolution	Parameterization	Users	Tools
Action Workflow/Speech Acts [MWFF92]								✘	✘	✘	✘				✘					✘
Workflow Management System [Beed97]											✘	✘	✘							
Abstract Business Processes [GE97]											✘									
Organization Knowledge [FCS97]								✘				✘	✘	✘						✘
Key Event Dictionary [WF97]						✘	✘	✘												
Behavioral Lifecycle Modeling [Chaf96]								✘	✘		✘	✘								
Transaction Based Analysis [Raws95]									✘		✘									
Transaction, Event and Rule Models [Silv95]		✘	✘			✘	✘	✘	✘	✘				✘	✘					
Collective Behavior/Business Rules[KHT95]											✘			✘					✘	
Business Rules Model (ECA) [Herb95]						✘		✘		✘				✘						
Business Rules Model (GUIDE) [HH95]														✘						
Hierarchical Policy Model [Nass91]		✘										✘		✘						
ALBERT [DDB93]						✘					✘		✘							
Incremental Evolution of Specs[JB93]		✘															✘			✘
Commitment-based software [MTMS92]		✘									✘			✘		✘				✘
Baseline Requirements Document [BH89]																			✘	✘
OO Design from Functional Specs [AP92]																				✘
Information Mapping [Horn89]	✘																		✘	✘
Problem Frames [Jack94]						✘														
Analysis Patterns [Fowl96]		✘	✘								✘	✘		✘						
Archetypes, Deltas, and Frames [Bass96]														✘		✘		✘		
Collaboration Frameworks [DW97]					✘		✘	✘		✘	✘	✘						✘		
Collaboration Based Design [VN96]	✘	✘										✘								
Path Expression Groups [AC94]		✘	✘																	
Component Contracts [Holl92]													✘	✘						
Exception Handling [SSP95]															✘				✘	✘
Design Patterns [GHJV94]		✘	✘								✘	✘						✘		
Business Rule Atomic Elements [Ross97]														✘						

Table 4-1: Use Case Formalisms Focus

4.2 Use Case Formalism Formats

Table 4-2 identifies several types of representations for use cases. Although each of the approaches can be fitted into one or more of these categories, the rich variety of representational schemes merits special attention. A basic argument to support the plethora of model elements and diagramming techniques can be found in Horn's information mapping techniques [Horn89]. First, no two domains are exactly alike. This notion is supported by Jackson's problem frames [Jack94]. Second, different personalities playing the various stakeholder roles can relate better to certain types of representation. Third, the maturity of a domain architecture influences the applicability or certain representations. Having a toolkit that consists of a wide selection of such techniques facilitates understandability, reuse, and accuracy. However with such an array of representational devices, it becomes imperative to have a single underlying meta-model that supports the transformation from one representation scheme to another.

4.2.1 Textual Formats

Five categories of textual format are defined in the table. Purely *textual* formats are *unstructured text* narratives or semi-structured *scripts*. Some approaches dealt with a restricted English format [Cock97][Grah95][Hurl97]. *Structured descriptions* refer to a form that is used as a template. Those papers that emphasized this aspect were [Harw97][Cock97] and [ABS95]. *Tabular* formats are similar to a database table, such as for state-event matrices. The use case summary matrix [ABS95] and key event dictionary [WF97] are two such examples. *Formal language expressions* are regular expressions in a well defined syntax. Three papers developed such regular expressions conforming to strict formal syntax. Beringer's regular expresses are used for representing object life-cycles [Beri97]. Path expression describe the interactions of a group of cooperating objects [AC94]. Formal expressions are also used to describe collaboration refinement [DW97]. The object constraint language that was incorporated into the UML specification would be another such example of a formal language expression.

4.2.2 Graphical Formats

Graphical formats include *structure*, *state*, *interaction*, and *implementation* diagrams similar to those defined in the UML notation guide. One additional type of diagram, the *rule* diagram is included for graphical representation of rules. Graphical representation of rules can be found in [Silv95] and [Ross97]. Several new modeling constructs were introduced. These included the use case and actor components [JGJ97], use case summary goal groups [Cock97], adaptive use cases [Hurl97], two types of model elements referred to as episodes [KPW97][RAB96]. All of these modeling constructs are intended to enhance reusability. Many different diagramming techniques representing new types of use case models were also described. These included directed use case graphs [KPW97], scenario trees [HSGK94], message sequence charts [AB95], service usage models [KS94], dialog maps [Wieg97], use case maps [Buhr97], context diagrams [ABS95], functional area models [ABS95], and use case dependency diagrams [ABS95].

4.2.3 Dynamic Formats

Dynamic formats imply automated tool support for the *assembly*, *visualization*, and *navigation* of use case model elements. Automated tool support for dynamic formats is immature, so these categories are mostly just propositions. Animation visualization is discussed in [ADP95] and [Thom95]. Hyperlinks navigation is discussed in [ADP95][Zeie95] and [Horn89]. A graphical browser of scripts and models is described in [BB95]. Use case driven dynamic assembly is covered in [Hurl97] and [Buhr97].

Two *other* types of dynamic formats that may not require automated support are *role playing*, such are with CRC cards, and *storyboards*. A storyboard differs from visualization in that there is no animation involved. Storyboard relate more to a slide show. Zorman make use of storyboards [Zorm95]. Wieger's use case dialog maps may also be considered a form of storyboard [Wieg97]. Two role playing examples are component partitioning validation through CRC cards [DWMW96] and the *act it out* strategy [CNM95].

Use Case Formalism Formats	Textual					Graphical					Dynamic			Other	
	Unstructured Text	Tabular	Structured Descriptions	Formal Language Expressions	Script	Structure	State	Rules	Interaction	Implementation	Assembly	Visualization	Navigation	Role Playing	StoryBoard
Use Case Components [JGJ97]			✘	✘	✘	✘	✘		✘	✘					
Structuring Use Cases With Goals [Cock97]			✘		✘	✘			✘						
Adaptive Use Cases [Hurl97]	✘	✘	✘	✘		✘	✘		✘	✘	✘	✘	✘	✘	
Enhanced Scenarios [Beri97]		✘	✘	✘		✘			✘					✘	
Agent Based Use Cases [KMJ97]			✘	✘	✘		✘		✘						
Use Case Maps [Buhr97]		✘				✘			✘		✘				
Change Cases [EDF96]				✘	✘				✘						
Task Scripts [HSFG97]			✘	✘	✘	✘	✘	✘	✘						
Use Case Classes [Jans95]						✘			✘						
Use Case Formats [Harw97]	✘	✘	✘		✘				✘						
Directed Use Case Graphs [KPW97]						✘			✘						
Scenario Trees [HSGK94]				✘			✘								
Message Sequence Charts [AB95]	✘					✘			✘						
Elicitation of State Machines [ML97]					✘		✘							✘	
Formalized Use Cases/State Charts [Glin95]				✘			✘								
Synthesized Use Case Modeling [RKW95]					✘	✘			✘						
Use Case Dialog Maps [Wieg97]			✘		✘										✘
Service Usage Models [KS94]	✘	✘					✘								
Scenario Contexts [Zorm95]	✘		✘		✘				✘						✘
Scenario Strategies [CNM95]									✘					✘	
Modeling Large Business System [ABS95]	✘	✘			✘				✘	✘					
Distributing Requirements/Episodes [RD97]	✘					✘			✘	✘					
Use Case Reuse [Know95]										✘					
Interactions/Hypermedia/Visuals[ADP95]					✘				✘			✘	✘		
Hypertext Use Cases [Zeie95]			✘										✘		
Requirements Scripts [BB95]					✘	✘			✘						
Animating Use Cases[Thom95]	✘				✘	✘			✘			✘			
Requirements Definition/Verification [HM95]	✘				✘							✘			
Formal Verification of Use Cases [Mend95]			✘		✘				✘						
Validate Component Use Cases [DWMW96]					✘									✘	
Iterative Use Case Prototyping [Coll95]	✘				✘										

Use Case Formalism Formats	Textual					Graphical					Dynamic			Other	
	Unstructured Text	Tabular	Structured Descriptions	Formal Language Expressions	Script	Structure	State	Rules	Interaction	Implementation	Assembly	Visualization	Navigation	Role Playing	StoryBoard
Action Workflow/Speech Acts [MWFF92]					✘				✘						
Workflow Management System [Beed97]						✘			✘						
Abstract Business Processes [GE97]					✘				✘						
Organization Knowledge [FCS97]	✘		✘		✘				✘		✘	✘			
Key Event Dictionary [WF97]		✘			✘		✘		✘						
Behavioral Lifecycle Modeling [Chaf96]						✘			✘						
Transaction Based Analysis [Raws95]			✘		✘	✘			✘						
Transaction, Event and Rule Models [Silv95]						✘	✘	✘	✘						
Collective Behavior/Business Rules[KHT95]			✘						✘						
Business Rules Model Based (ECA)[Herb95]						✘									
Business Rules Model (GUIDE) [HH95]			✘												
Hierarchical Policy Model [Nass91]	✘			✘											
ALBERT [DDB93]	✘		✘	✘	✘	✘			✘						
Incremental Evolution of Specs[JB93]				✘	✘	✘	✘		✘		✘	✘	✘		
Commitment-based software [MTMS92]				✘							✘				
Baseline Requirements Document [BH89]	✘										✘			✘	
OO Design from Functional Specs [AP92]	✘			✘											
Information Mapping [Horn89]	✘	✘	✘		✘								✘		✘
Problem Frames [Jack94]			✘												
Analysis Patterns [Fowl96]						✘			✘						
Archetypes, Deltas, and Frames [Bass96]				✘							✘				
Collaboration Frameworks [DW97]			✘	✘		✘	✘								
Collaboration Based Design [VN96]						✘			✘						
Path Expression Groups [AC94]				✘					✘						
Component Contracts [Holl92]									✘						
Exception Handling [SSP95]				✘	✘								✘		
Design Patterns [GHJV94]			✘			✘			✘						
Business Rule Atomic Elements [Ross97]			✘					✘							

Table 4-2: Use Case Formalisms Formats

Russ Hurlbut is a principal of Expertech, Ltd., an information systems consulting organization specializing in the development of business domain-specific architectures. He has taught graduate courses in Object Oriented Development and Technology Transfer at the Illinois Institute of Technology. This paper represents one aspect of his research towards his Ph.D. Thesis "Managing Business Domain Architectures through Use Case Formalisms."

5 References

[AB97] Michael Andersson and Johan Bergstrand, “Formalizing Use Cases with Message Sequence Charts”, Master thesis, Department of Communication Systems at Lund Institute of Technology, 1997 http://www.efd.lth.se/~d87man/EXJOBB/PS/ExBook.ps.Z.uu

[ABS95] Frank Armour & Lorry Boyd & Monica Sood, “Use case modeling concepts for large business system development”, OOPSLA workshop -- Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/toddHansen.ps

[AC94] G. Adams, J.P. Corriveau; Capturing Object Interactions; Proceedings of TOOLS Europe '94, Versailles, 1994

[ADP95] X. Alvarez & G. Dombiak & M. Prieto, “Use-cases, interaction diagrams, hypermedia and visualization”, OOPSLA workshop -- Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/Lifia.UseCase.Final.ps

[AP92] Alagar, V.S.; Periyasamy, K., "A methodology for deriving an object-oriented design from functional specifications", Software Engineering Journal, Vol: 7 Iss: 4 p: 247-63, July 1992

[Bass96] Paul G. Basset, “Framing Software Reuse: Lessons From the Real World”, Prentice Hall PTR, Upper Saddle River, New Jersey, ISBN 0-13-327859-X

[BB95] Regis Berteaud & Jean Bézivin, “Requirement modeling in the OSMOSIS workbench”, OOPSLA - First WORKSHOP on USE CASES Austin, Texas, 15 October 1995. http://www.unantes.univ-nantes.fr/usecase/Contributions/berteaud.ps

[Beed97] Michael A, Beedle "A 'light' distributed OO Workflow Management System for the creation of OO Enterprise System Architectures in BPR environments", OOPSLA-97 conference.", available at ftp://www.fti-consulting.com/pub/bpr-papers/oopsla.pdf

[Beri97] Dorothea Beringer, “Modelling Global Behaviour with Scenarios in Object-Oriented Analysis”, Ph.D. Thesis, Département d'Informatique, Ecole Polytechnique Federale De Lausanne, 1997 ftp://lglftp.epfl.ch/pub/Papers/beringer-thesis.ps.Z

[BH89] Burns, H.S.; Halliburton, R.A., "Tackling productivity and quality through customer involvement and software technology", GLOBECOM '89. IEEE Global Telecommunications Conference and Exhibition. Communications Technology for the 1990s and Beyond (Cat. No.89CH2682-3), p: 631-5 vol.1, 27-30 Nov. 1989, Dallas, TX, USA, IEEE

[Buhr97] Ray Buhr, “Use case maps (UCMs) Updated: A Simple Visual Notation for Understanding and Architecting the Emergent Behaviour of Large, Complex, Self Modifying Systems”, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/ucmUpdate.ps

[Chaf96] R. Chafi, “Generic Object Oriented Implementation Design”, Ph.D. Dissertation. Illinois Institute of Technology, 1996

[CNM95] Peter Coad, David North, Mark Mayfield, “Object Models – Strategies, Patterns, & Applications”, Yourdan Press, Englewood Cliffs, NJ, 1995, ISBN 0-13-108614-6

[Cock97] Alistair Cockburn, “Structuring Use cases with goals” – JOOP/ROAD 10(5) Sep ’97 and 10 (7) Nov '97, http://members.aol.com/acockburn/papers/usecases.htm and http://members.aol.com/acockburn/papers/uctempla.htm

[Coll95] Mark Collins, “Iterative Use Case Prototyping”, OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/mcollins.ps

[DDB93] Eric Dubois, Phillippe Du Bois and Michael Petit, “O-O Requirements Analysis: an Agent Perspective", ECOOP '93 – Object-Oriented Programming 7th European Conference, Kaiserland, Germany, July 26-30, 1993, Proceedings, pp. 458-81

[DW97] Desmond D’Souza and Alan Wills, “Component-Based Development Using Catalysis, Draft v 0.8”, ICON Computing, http://www.iconcomp.com

[DWMW96] Robin Davies, Pay May, Dave R. Wardell, and Trish Wooding, “Techniques for developing reusable business components”, JOOP 9(7), November-December 1996, pp. 40-43

[EDF96] Earl F.Ecklund, Jr, Lois M. L.Delcambre, Michael J.Freiling, “Change Cases: Use Cases that Identify Future Requirements”, OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1997.

[FCS97] Peter Fingar, Jim Clarke, and Jim Stikeleather, “The business of distributed object computing”, Object Magazine - 7(2) Apr 1997, pp. 28-33.

[Fowl96] Martin Fowler, Analysis Patterns : Reusable Object Models, Addison-Wesley, 1996, ISBN: 0201895420

[GE97] Thornton Gale and James Eldred, “The Abstract Business Process”, Object Magazine, 6(11) Jan 1997, pp. 21-37

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Pattterns: Elements of Reusable Software", Addison-Wesley ISBN 0-201-63361-2, 1994

[Glin95] M. Glinz, "An Integrated Formal Model of Scenarios Based on Statecharts", Proceedings of the 5th European Software Engineering Conference ESEC 1995, Sitges, Spain, 1995

[Grah94] Ian Graham, “Migrating to Object Technology”, Addison-Weslay, 1994, ISBN 0-201-59389-0

[Harw97] R. J. Harwood, “Use case formats: Requirements, analysis and design”, JOOP 9(8), January 1997, pp. 54-57.

[Hend97] B. Hendeerson-Sellers, “Choosing between UML and OPEN”, http://www.csse.swin.edu.au/cotar/OPEN/CHOOSING/choosing.html

[Herb95] H. Herbst, "A Meta-Model for Business Rules in Systems Analysis", Proceedings of the Seventh Conference on Advanced Information Systems Engineering, Berlin. Springer 1995, pp. 186 - 199. ftp://ftp.ie.iwi.unibe.ch/public/documents/papers/caise95.eps

[HH95] David Hay, Keri Anderson Healy, “GUIDE Business Rules Project - Final Report”, November 7, 1995 http://www.guide.org/ap/apbrules.pdf

[HM95] Todd Hansen and Granville Miller, “Requirements definition and verification through use case analysis and early prototyping”, OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/toddHansen.ps

[Holl92] Ian Holland, “The Design and Representation of Object-Oriented Components.” Ph.D. Thesis, College of Computer Science, Northeastern University, Boston, MA, 1992. 183 pages.

[Horn89] Robert E. Horn, "Mapping Hypertext: Analysis, Linkage, and Display of Knowledge for the Next Generation of On-Line Text and Graphics", Lexington Institute, Lexington, MA, 1989

[HSFG97] Brian Henderson-Sellers, Donald G. Firesmith, and Ian Graham, “OML Metamodel: Relationships and state modeling”, JOOP 10,1, March/April 1997

[HSGK94] Pei Hsia, Jayaranan Samuel, Jerry Gao, and David Kung,; Formal Approach to Scenario Analysis; IEEE Software, Vol. 11, No. 2., March 1994

[Hurl97] Russell R. Hurlbut, “Ph.D. Thesis Proposal: Managing Business Domain Architectures through Use Case Formalisms”, Department of Computer Science and Applied Mathematics, IIT, Feb 1997. http://www.iit.edu/~rhurlbut/hurl97a.pdf

[Hurl97a] Russ Hurlbut, “Managing Domain Architecture Evolution Through Adaptive Use Case and Business Rule Models”, Technical Report: XPT-TR-97-02, Expertech, Ltd., 1997 http://www.iit.edu/~rhurlbut/xpt-tr-97-02.html

[Jack94] Michael Jackson, “Problems, Methods and Specialisation”, in Special Issue of SE Journal on Software Engineering in the Year 2001.

[Jaco95] Ivar Jacobson, “Modeling with Use Cases: Formalizing use-case modeling”, JOOP (June 1995, Vol. 8 No. 3)

[Jans95] Par Jansson, “Use Case Analysis with Rational Rose”, Chapter 4, Rational Rose Application Notes, Software Release 3.0, 1995 Ration Software Corp, Santa Clara CA

[JB93] Johnson, W.L.; Benner, K.M.; Harris, D.R., "Developing formal specifications from informal requirements", IEEE Expert, Vol: 8 Iss: 4 p: 82-90, Aug. 1993

[JBJE95] I. Jacobson & S. Bylund & P. Jonsson, “Using Contracts and Use Cases to Build Plugabble Architectures”, JOOP, (May-June, 1995) http://www.unantes.univ-nantes.fr/usecase/JOOP.ps

[JCJO92] Ivar Jacobson, Magnus Christenson, Patrik Jonsson, Gunnar Overgaard, "Object-oriented software engineering : a use case driven approach",(New York) : ACM Press ; Wokingham, Eng. ; Reading, Mass. Addison-Wesley Pub.,1992

[JEJ94] Ivar Jacobaon, Maria Ericsson, Agneta Jacobson, “The Object Advantage – Business Process Reengineering with Object Technology”, ACM Press, 1994, ISBN 0-201-42289-1

[JGJ97] Ivar Jacobson, Martin Griss, Patrik Jonsson, “Software Reuse – Architecture, Process and Organization for Business Success”, ACM Press, New York, 1997 ISBN 0-201-92476-5.

[KHT95] Haim Kilov, Bill Harvey, Kevin Tyson, “Semantic integration in complex systems: collective behavior in business rules and software transactions”, OOPSLA '95 - Addendum to the Proceedings, OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1997.

[KMJ97] Elizabeth A. Kendall, Margaret T Malkoun, and C. Harvey Jiang. “The application of object oriented analysis to agent-based system”, JOOP 9(9), February 1997, pp. 56-63.

[Know95] Nicolas Knowles, “Reuse Cases?”, OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/nickKnowles.html

[KPW97] Georg Kosters, Bernd-Uwe Pagel, Mario Winter, "Coupling Use Cases and Class Models", Proc. of the BCS-FACS/EROS workshop on "Making Object

Oriented Methods More Rigorous", Imperial College, London, June 24th, 1997, pp. 27-30, ftp://ftp.fernuni-hagen.de/pub/fachb/inf/pri3/papers/winter/RoomAbstract.ps.gz

[KS94] Kristofer Kimbler, Daniel Søbirk, “Use Case Driven Analysis of Feature Interactions”, Feature Interactions in Telecommunications Systems, IOS Press, 1994, http://www.tts.lth.se/Personal/daniels/papers/fiw94.ps

[Mend95] Victor M. Mendoza-Grado, “Formal Verification of Use Cases”, OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/mendoza.ps

[ML97] Ian Mitchell and Hugues Lecoeuche, “On an improved approach to the elicitation of O-O state machines by use-case”, JOOP 9(9), February 1997, pp. 52-55.

[MTMS92] Mark, W.; Tyler, S.; McGuire, J.; Schlossberg, J., "Commitment-based software development", IEEE Transactions on Software Engineering, Vol: 18 Iss: 10 p: 870-85, Oct. 1992

[MWFF92] Raul Medina-Mora, Terry Winograd, Rodrigo Flores, Fernando Flores, "The Action Workflow Approach to Workflow Management Technology.", CSCW 92 Proceedings, November 1992.

[Nass91] E. F. Nassiff, “The Hierarchical Policy Model”, MS Thesis, Illinois Institute of Technology, 1991

[Raws95] Daniel A. Rawsthorne, “Transaction Based Analysis Capturing Functional Requirements Through Object Interactions”, OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/drawstho.ps

[RAB96] B. Regnell, M. Andersson, J. Bergstrand, “A Hierarchical Use Case Model with Graphical Representation”, Proceedings of Second International Symposium on Engineering Computer-Based Systems, Friedrichshafen, Germany, IEEE Computer, Society Press, ISBN 0-8186-7355-9, March 1996, pp. 270-277. http://www.tts.lth.se/Personal/bjornr/Papers/ECBS96.ps

[RD97] B. Regnell, A. Davidson. “From Requirements to Design with Use Cases - Experiences from Industrial”, REFSQ'97: 3rd Intl Workshop on Requirements Engineering - Foundation for Software Quality, pp. 205-222, Presses universitaires de Namur, ISBN 2-87037-239-6, Preceeding CAISE'97, Barcelona, June 16-20, 1997 http://www.tts.lth.se/Personal/bjornr/Papers/REFSQ97.ps

[RKW95] B. Regnell, K. Kimbler, A. Wesslen, “Improving the Use Case Driven Approach to Requirements Engineering”, Proceedings of Second International Symposium on Requirements Engineering, York, UK, IEEE Computer Society Press, ISBN 0-8186-7017-7, March 1995, pp. 40-47. http://www.tts.lth.se/Personal/bjornr/Papers/tts-94-24.ps

[Ross97] Ronald G. Ross, “The Business Rule Book – Classifying, Defining and Modeling Rules, Second Edition” Database Research Group, Boston, MA.

[Silv95] Mauricio J. V. Silva, “A/OODBMT - An Active Object-Oriented Database Modeling Technique” PhD Thesis, Illinios Institute of Tehcnology, Computer Science Department of Illinois Institute of Technology, 1995

[SSP95] S. L. Stewart and James A. St. Pierre, “Experiences with a Manufacturing Framework”, in Workshop Report: Business Object Design and Implementation. 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. Addendum to the Proceedings. OOPS Messenger 6:4: ACM/SIGPLAN October, 1995. http://www.tiac.net/users/jsuth/oopsla/stewapub.pdf

[Thom95] Lynda Thomas, “Animating Use Cases”, OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.unantes.univ-nantes.fr/usecase/Contributions/lThomas.ps

[UML97] Unified Modeling Language, Version 1.1, http://www.rational.com/uml

[VN96] Michael VanHilst, David Notkin, “Using role components to implement collaboration based design”, OOPSLA: The First Eleven Years Conference Proceedings 1986-1996. CD-ROM. ACM Press, New York, 1997.

[WF86] Terry Winograd and Fernando Flores, "Understanding Computers and Cognition: A new Foundation for Design", Ablex Publishing Corporation, Norwood, New Jersey, 1986.

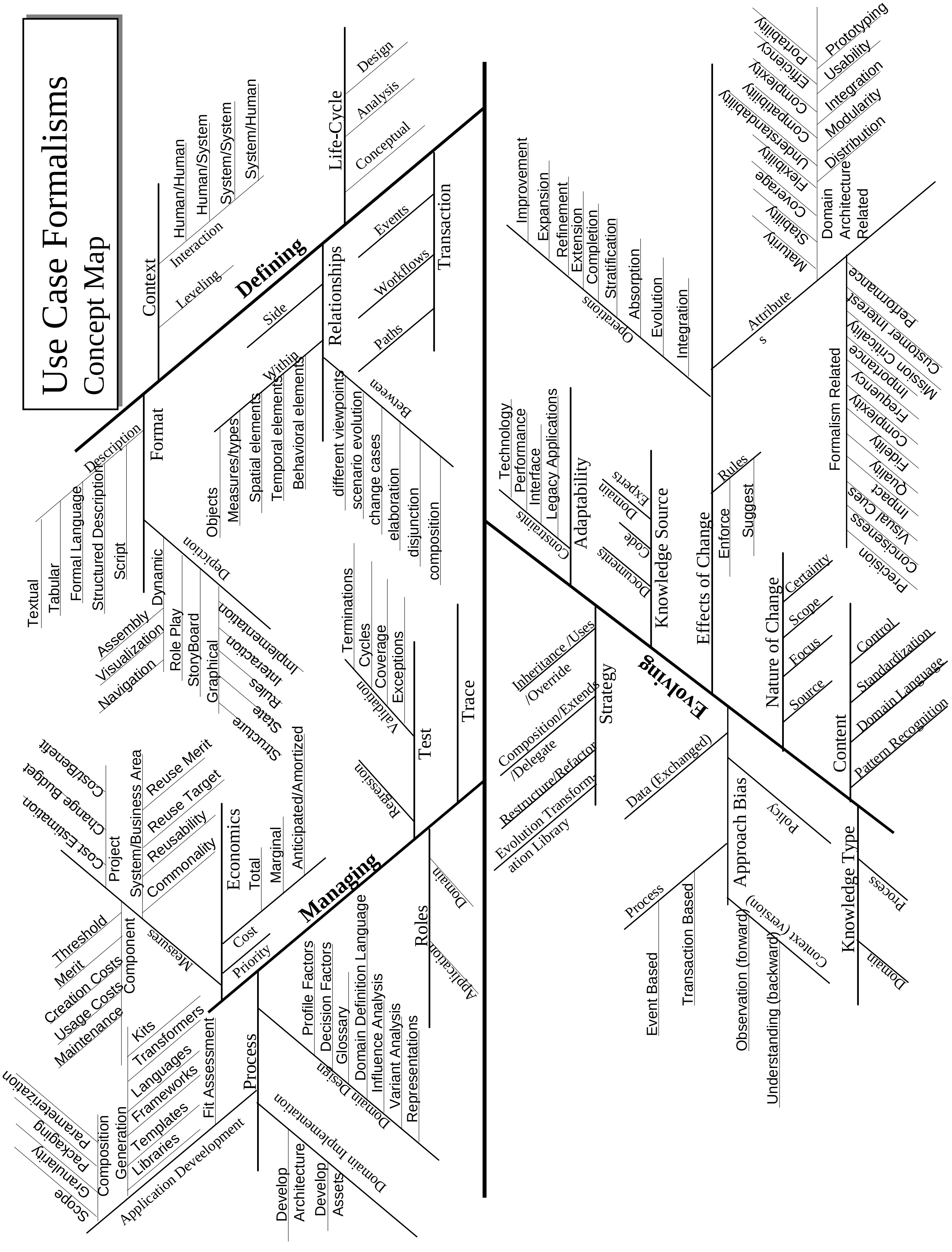
[WF97] Becky Winant and Mike Frankel, “Solving object state model mysteries using a key event dictionary”, JOOP 10(1), March-April, 1997, pp 52-58.

[Wieg97] Karl Wiegers, “Use Cases: Listening to the Customer’s Voice”, Software Development, March 1997, Vol 5., No.3.

[Zeie95] Gary J Zeien, “Weaving a web of use cases”, Object Magazine Nov/Dec 1995

[Zorm95] Lorna A. Zorman, “The context and composition of scenarios”, OOPSLA workshop – Requirements Engineering: Use Cases and More, Sunday October 15, 1995, AUSTIN, Texas, http://www.isi.edu/soar/lorna/oopsla95.ps

Appendix A. Use Case Formats


Use Case Formalisms
Concept Map
Defining
Context
Interaction
Human/Human
Human/System
System/System
System/Human
Leveling
Life-Cycle
Design
Analysis
Conceptual
Side
Relationships
Transaction
Events
Workflows
Paths
Within
Between
Format
Description
Textual
Tabular
Formal Language
Structured Description
Script
Dynamic
Assembly
Visualization
Navigation
Role Play
StoryBoard
Graphical
Depiction
Implementation
Interaction
Rules
State
Structure
Objects
Measures/types
Spatial elements
Temporal elements
Behavioral elements
different viewpoints
scenario evolution
change cases
elaboration
disjunction
composition
Test
Validation
Terminations
Cycles
Coverage
Exceptions
Regression
Trace
Managing
Economics
Total
Marginal
Anticipated/Amortized
Cost
Priority
Measures
Project
Cost Estimation
Change Budget
Cost/Benefit
System/Business Area
Reuse Merit
Reuse Target
Reusability
Commonality
Component
Threshold
Merit
Creation Costs
Usage Costs
Maintenance
Roles
Domain
Application
Process
Fit Assessment
Kits
Transformers
Languages
Frameworks
Templates
Libraries
Generation
Composition
Parameterization
Packaging
Granularity
Scope
Application Deveelopment
Develop Architecture
Develop Assets
Domain Implementation
Domain Design
Profile Factors
Decision Factors
Glossary
Domain Definition Language
Influence Analysis
Variant Analysis
Representations
Evolving
Operations
Improvement
Expansion
Refinement
Extension
Completion
Stratification
Absorption
Evolution
Integration
Adaptability
Constraints
Technology
Performance
Interface
Legacy Applications
Knowledge Source
Domain Experts
Code
Documents
Effects of Change
Rules
Enforce
Suggest
Attributes
Domain Architecture Related
Maturity
Stability
Coverage
Flexibility
Understandability
Compatibility
Complexity
Efficiency
Portability
Prototyping
Usability
Integration
Modularity
Distribution
Formalism Related
Performance
Customer Interest
Mission Criticality
Importance
Frequency
Complexity
Fidelity
Quality
Impact
Visual Cues
Conciseness
Precision
Nature of Change
Certainty
Scope
Focus
Source
Content
Control
Standardization
Domain Language
Pattern Recognition
Strategy
Inheritance /Uses /Override
Composition/Extends /Delegate
Restructure/Refactor
Evolution Transformation Library
Data (Exchanged)
Approach Bias
Policy
Process
Event Based
Transaction Based
Observation (forward)
Understanding (backward)
Context (version)
Knowledge Type
Process
Domain