A Requirements Analysis Framework for Open Systems Requirements Engineering

Behzad Bastani
Computer Laboratory
University of Cambridge
Cambridge, UK
Behzad.Bastani@cl.cam.ac.uk

ABSTRACT
Requirements Engineering [26, 27] is an old discipline which has been conveniently subject to being overlooked by system designers. Most requirements analysis writings either advise general guidelines which are short of any concrete operational aspects, or they are not simply at the scale of complex open systems construction. This paper presents a new analytical framework and a method which considers an end-to-end view of a system, and is specifically designed to support the requirements analysis and design of open systems. The paper briefly discusses the conceptual shortcomings in this area, presents an analytical perspective for requirements engineering, and proposes a new framework called "Abstraction-oriented Frames". This framework is a hybrid model consisting of three major parts offering a specific operational approach based on a consistent end-to-end analytical view of the system.

CATEGORIES AND SUBJECT DESCRIPTORS
D.2.10 [Software Engineering]: Design - *Methodologies, representation*

GENERAL TERMS
Modeling, Design, Method, Framework

KEYWORDS
Requirements Analysis, Requirements Engineering, Operational Requirements Management, Requirements and Architecture modeling, Specification, Design, Design Patterns, Methodologies, Abstractions.

1. INTRODUCTION
A requirements analysis [28] is normally the first step in developing a new system. However, it is interesting to note that such activity is only a first step theoretically, while in practice, this is a mostly omitted step in spite of the fact that many analysts consider such omission as part of the reasons for collapse of a good number of large software projects. A number of studies show the considerable influence of early requirement analysis on the reduction of unnecessary costs, confusion and complexity in the later phases of software development [1, 17]. However, intangibility of software, among other reasons, causes the requirement analysis to be much more complex and less comforting than many other engineering areas. More importantly, "little effort has been devoted to date to techniques for deriving architectural descriptions in concert with the requirements specifications"[6], and such realization requires an end-to-end analytical perspective accompanied with a practical framework that makes it possible for the relationships, properties and connections to become visible. One other problem is that a substantial amount of requirements literature appears like general ethical advice void of much concrete guidelines. Such advice is more focused on business or managerial procedures. Developing an analytical perspective and an integrated framework to address such concerns while having the needs of open systems at mind, is the motivation and subject of this paper.

Requirements elicitation belongs to the so-called problem domain. It is necessary to show how the requirements relate to the problem domain and conversely, what roles the problem domain play in characterization of the requirements. The conceptual focus of this paper is on providing a context to be in support of open evolvable systems and towards this goal proposes a new framework called "Abstraction-oriented Frames". The modeling framework is designed to accommodate the specifications of such systems. A fair question is, what are the evolvable system features to be supported at the blueprints level? In response we suggest the following capabilities at the blueprint level:

- Flexibility in changing requirements at any phase, including after construction.
- Flexibility in changing architecture and design at any phase.
- Flexibility in adding, removing, modifying entities and processes at any phase.
- Capability of keeping track of any change and reflecting it properly from initiation point to implementation – initiation point is always the relative problem domain.

To meet these capabilities, the requirements framework presented in this paper consists of dynamics that facilitate subsystem modifications at any level of abstraction or any part of the system.

This model is developed as a facilitating framework for the Open Evolvable Systems research project, which is carried on under the umbrella Pebbles Project, at the Computer Laboratory of the University of Cambridge. The project is funded by the Cambridge-MIT Institute (CMI).

Section 2 analyzes some core concepts of requirements engineering and our definition. Section 3 presents the ingredients of the new modeling framework and its associated methods. Section 4 offers a brief conclusion and the future work.

2. A PERSPECTIVE ON ANALYTICAL REQUIREMENTS ENGINEERING

It might not be surprising to find a strong tendency to skip [3] a rigorous requirements analysis at the outset of a system construction, as some wonder what is a requirements analysis and how should it be handled efficiently. Requirements Analysis deals with the problem domain or problem context. However, the requirements have no physical realization in the problem context, and cannot be considered as a subdomain of the problem domain. Therefore a central question can be, how do you come up with requirements for a system, and how do you express them.

To start, I suggest a definition: "For every problem definition P (whether it is a type or an instance), there is a "conceptual set" C of "discrete states" x, each might be a *desired state* as a selected resolution to the problem, any of which would be called a requirement." C = {x | x = desired state}; x = requirement.

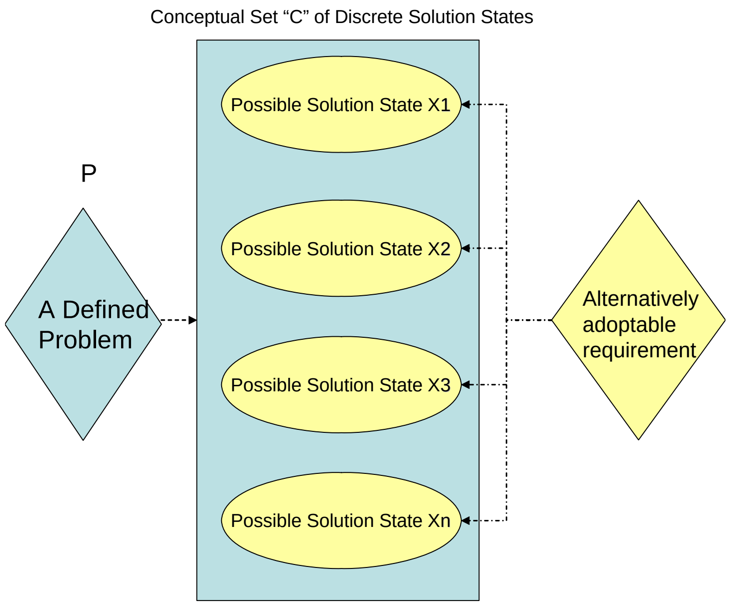

Conceptual Set "C" of Discrete Solution States
P
A Defined Problem
Possible Solution State X1
Possible Solution State X2
Possible Solution State X3
Possible Solution State Xn
Alternatively adoptable requirement

Figure 1: Reflection of a "Requirement" definition

We can take an alternative view on the issue to clarify the above definition. We discussed two fundamental concepts above: a "problem" and a "resolution" to that problem. Mapping these two concepts to a software system, we can conceive the "problem" as the action taken by the user on a system. Although it might not be exactly the problem per se, the action taken on a software system by a user is the direct result of a real world problem for which accordingly the user seeks a resolution. If the user's problem is "missing a phone number", the action of submitting a query for the phone number is a direct result of that real world problem in the problem domain. The requirement then is conceptually equivalent to *what* the user expects the system to do for her, regardless of *how* the system accomplishes it. From a system point of view, a requirement is a condition or capability to which the system must conform [(IEEE Standard 610.12-1990), 9]. Therefore, in the context of preparation of a requirement model, both the problem and *what* the system will do to resolve that problem should be reflected in an integrated manner. The user can be a human user, a machine or a program. To support this view, I refer to an excellent analysis M. Jackson presents [2]: "Requirements are all about – and only about – the environment phenomena…The customer for the system is interested in the environment, not in the machine…If you take this view of programs, specifications, and requirements, you can see that every specification is, in a formal sense, both a requirement and a program". Therefore the burden squares on our shoulders to provide means to separate the ingredients of this operationally integrated trinity into a conceptually layered end-to-end model with such a flexible architecture that can easily accommodate any future modifications.

The requirements model is the normally missing link between the world of the user and the developer. The user point of view and expectations should get reflected in the requirements document and become the foundation for the architectural and design planning and activities. The requirements document is usually understood as a written statement. However, such a document can be more formally organized [21]. Based on our definition above, a requirements statement should clearly define the solution states, the attributes, traceability to other possibly existing software artifacts, and interrelations with other requirements. One question which is always there to manage at each phase is, what should be the level of detail in a requirements statement.

One way of expressing and organizing functional requirements is utilizing use cases or use case patterns. Normally requirements are organized from a user's perspective. Expressing requirements in the form of use cases enjoys tools support as well as better preparation for integration with subsequent design and development phases. However, functional requirements are not everything involved and we clearly need a much wider perspective to cover all system concerns.

In this paper I propose a framework called "Abstraction-oriented Frames" which in addition to its theoretical and research foundations, relies partly on my industrial and practical experiences in this field working on super large software development projects. This framework is a hybrid model that modifies and extends three other major approaches in requirement analysis, and builds requirements models according to its new framework. The three approaches used as premises consist of the idea of problem frames [12] as a baseline framework, the well-known use case approach as its instrumental driver [18], and abstraction-based [13, 14] classification as its analytical method. The framework has a level zero startup phase which is based on XML schema modeling [19].

3. A MODELING FRAMEWORK AND A METHOD

Abstraction-oriented Frames as an analytical framework is an extension and amalgamation of three major conceptual approaches and is explained in the following sections under, Integrated Triple Sequence Model, Use Case Instrumentality, and Abstraction-oriented elicitation and Classification. Figure 2 presents a big picture view of the whole integrated model. The presentation of this big picture at this point serves as a conceptual organizer for the following subsections, showing the relative positioning of the model ingredients.

3.1 INTEGRATED TRIPLE SEQUENCE MODEL

The philosophy and rationality behind the *problem frames* is stated eloquently by Michael Jackson [12] and seems quite natural to be used as a startup point in system development. “A problem frame defines the shape of a problem by capturing the characteristics and interconnections of the parts of the world it is concerned with, and the concerns and difficulties that are likely to arise. So problem frames help you to focus on the problem, instead of drifting into inventing solutions” [12]. Having this as a criterion, still one can find high level solution ideas in the actual problem frames presented, mixing the description of the problem with statements about requirements combined with general solution specifications or assumptions about solutions. Furthermore, while suffering from other limitations [16, 20], Problem Frames as an approach “does not provide the designer with a means” for “decomposing an end-to-end system requirement into machine specification” [4], in spite of having this decomposition as a conceptual emphasis in the original theory.

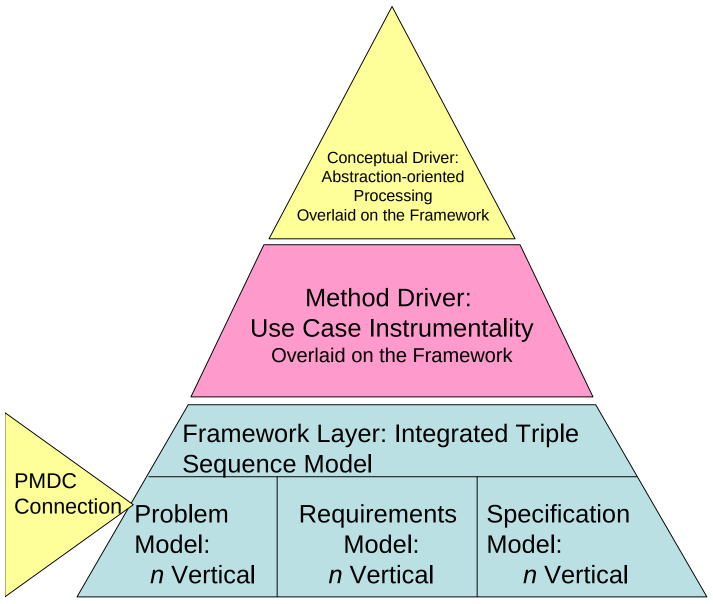


Figure 2: Big picture view of the Abstraction-oriented Frames

One question is, where are the requirements coming from conceptually? Formal requirements literature say the requirements come from many diverse sources, including system owners, users, and business experts. Our conceptual definition is, *requirements reflect human choices in response to a real problem in the world outside the computers.* Therefore we should recognize that one focus should be on the problem in its natural context (whatever part of the world it might be) to analyze it regardless of any presumed solution. The other focus should be on structuring the demands of those dealing with the problem the way they choose to deal. This part is a humanistic choice and could be anything. In other words, whatever the choice should be, one cannot claim there is anything wrong with it mathematically or formally. As such, it imports factors of fluidity and uncertainty into the system development effort. The fluidity is not always the result of uncertainty, but could be the result of more material factors like limited budget that might be increased in the future and create a demand for inclusion of more requirements in the picture – some of which may have been even envisioned in the past. To formulate this analytical point of view into a more structured and applicable framework of software development, I propose *Integrated Triple Sequence Model* which would be applicable to any given level of abstraction (higher or lower) for a certain problem, starting from the idea of *problem frames*, but extending it into an integrated collection of *Problem Model*, *Requirement Model* and *Specification Model*, while limiting the scope of the sequence to the initial problem in the Problem Model. Each of these is normally a partial model – for the whole system – at a certain level of abstraction. That is why at each level they can be disintegrated into more specific models and in turn start a new triple sequence, as shown in the Figure 3.

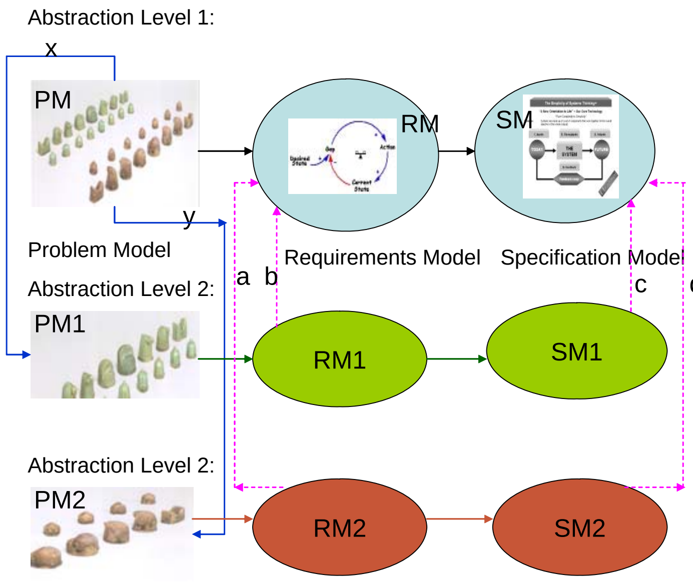


Figure 3: Integrated Triple Sequence Model

The Triple Sequence makes the foundation framework layer for the model. The framework layer in fact has two horizontal and vertical coordinates. The horizontal axis or abscissa represents the progression form the identification of a problem to the design of a system specification at each single level of abstraction, and includes three distinct phases as shown in the model above. The vertical axis or ordinate represents the progression from the higher level architectural conceptualizations and choices towards the lower level details involved, and for each system there are *n* number of them depending on the complexity and size of the system. One very important feature of such organization and modeling is that it delicately fits into the iterative process model for implementation and testing purposes. Iterative process is a fundamental concept for Unified Process (UP) and its derivatives like Rational Unified Process (RUP) [22, 23]. In our model, every Triple Sequence can be a partial iteration. The scope of the iteration is set by the initial problem at each abstraction level which is, from a practical point of view, a subsystem. Yet it is a full iteration given its scope, as the solution can be substantiated by its full specification expressed by its Specification Model, and then tested. This is only if the level of abstraction is at the design level, as opposed to architectural level, so that the level contains enough details for substantiation. Therefore, one added advantage of this framework over the conventional iterative process is its further refinement of the iteration unit. Conventionally, an *iteration* is understood as either a mini-version or a mini-project within the entire entity of the software to be developed. One considerable contribution of the "Abstraction-oriented Frames" is that it reduces the granularity of the iteration from the mini-version or mini-project level to a fine granularity of a *single (net) problem* level. That means at the lowest level in the *n vertical* chain (Figure 2) of every Problem Model, there is one single problem that initiates one single *iteration* that leads to an implementation and testing of that one problem. Reducing the granularity of the iteration level will drastically reduce the *risk*.

Before explaining the responsibility of each of these models, as it follows, it should be emphasized that such explanation –in this section– only defines the conceptual foundation of the models, yet the methods of using them will be explained in subsequent sections of 3.2 and 3.3.

A "Problem Model" is responsible to show a specific problem in its natural context. The problem might be at any level in terms of its generality. In other words, a problem might be expressed at a very high level in its general context (like the very starting point of thinking on a problem), and then processed for getting broken down to lower levels or subproblems. The most important point for a Problem Model is, not to include any indication of a solution, a feature we name it *solution-neutrality*. Even the jargon used to describe the problem should be clear from usage of terms that indicate a prejudice with respect to some solutions. To preserve control over expression of the problem, one naturally needs to use some constraint concepts over the dimensions of t he problem. Yet the concept of constraint also finds a different shade of meaning in this context. We might show constraints on a Problem Model, but we should be careful these should not be requirements or design constraints, but should only reflect constraints imposed by the context or the natural environment of the problem on the elements.

In Figure 3 a Problem Model is shown at abstraction level 1 which is a high level expression of the problem at its natural context. At this specific high level of abstraction, the Problem Model is followed by a high level Requirements Model and Specification Model, explained below. However, a part or a module of the problem is chosen to be reflected as a new problem in itself, transferred through link x to abstraction level 2, which is a lower abstraction level compared to abstraction level 1 –dealing with more details. The links can be modeled using UML <<trace>> relationship. This process can continue recursively for submodules of PM1, transferring each of them to subsequent abstraction levels 3 and 4 (not shown in the figure). At the same time, another module of PM gets transferred through link y to abstraction level 2 creating the new PM2, and follows the same path as explained for the first module PM1.

A Requirements Model reflects on the Problem Model but extends it by adding the *desired* general solutions that a user or system stakeholder might anticipate. I would like to distinguish between two conceptual levels of a Requirement Model. The first conceptual level reflects user's desires, choices and anticipations in response to the nature and dimensions of the problem. I would call this level, *Choice Requirements*. As it is named, this level only reflects user's arbitrary choices and there is almost nothing right or wrong in terms of logical or mathematical correctness for such choices. A user can decide to choose any response to a problem, even though the response might look irrational to others. A system can be built to deliver such responses, and as long as the system delivers expected result, it is a correct system relative to its requirements.

The second conceptual level is a category of the requirements that impose themselves as mandatory as a result of the *Choice Requirements*, and I would call them *Consequential Requirements*. The criteria can be stated as, these requirements are logical, mathematical or technological requirements that follow *Choice Requirements* whether the system stakeholder likes them or not.

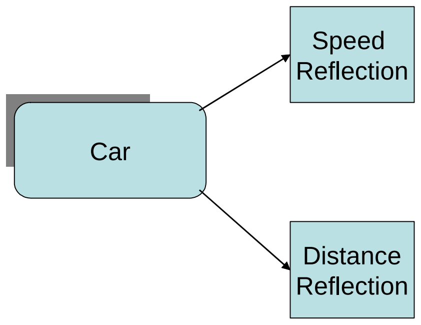


Figure 4: High-level Problem Model

Let's expand on an example. Suppose we are building a custom designed car. We know that one problem scenario about the car is that the speed of the car and the distance traveled must be reflected to the user. So far we just have a problem at hand and this should get recorded in the Problem Model without any preconception about a specific solution. Figure 4 shows the Problem Model. Although this is an extremely simple problem, the point is demonstrating that a problem Model does not have any presupposition about either requirements or the solutions, but just shows the ingredients of the problem in its natural context and the possible relationship between the entities. Here the Problem Model only shows a fact of the problem domain that the car reports the speed and distance to two separate reflectors without any indication about the type of reflectors, and this is demonstrated in two one way arrows, meaning the problem domain has no indication of anything returning from the reflectors to the car. We know in the real world there are other constraints, like the main odometer of a car cannot be reset to zero. But the fact is that this is not naturally indicated by the problem domain, but it is a universal constraint that has been artificially defined as a requirement, hence should not appear in a Problem Model.

Based on *Integrated Triple Sequence Model*, the first step in the resolution of this problem is extending the Problem Model by adding *Choice Requirements*. Let's consider three sets of alternative choices in response to the problem. The first set is reflection of the speed and the distance traveled by a mechanical indicator (interface) in the front panel (Figure 5). A choice of "mechanical" generates certain design requirements and application of certain technologies in the solution architecture.

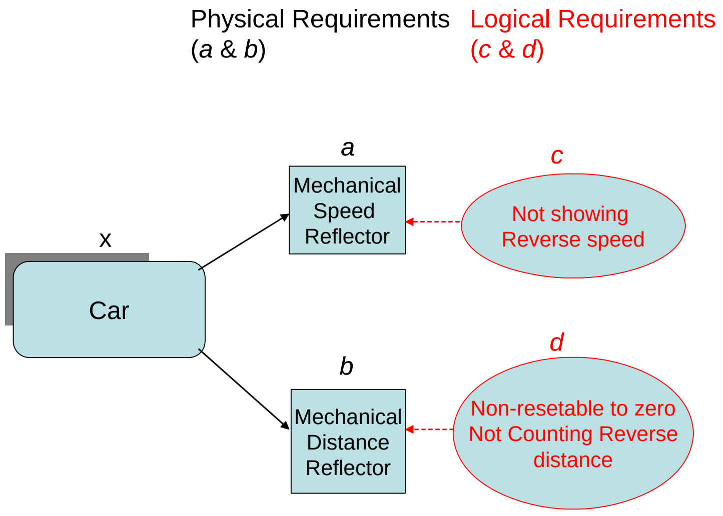

Physical Requirements
(a & b)
Logical Requirements
(c & d)
x
Car
a
Mechanical Speed Reflector
b
Mechanical Distance Reflector
c
Not showing Reverse speed
d
Non-resetable to zero Not Counting Reverse distance

Figure 5: High-level Requirement Model for Requirement Choices

The second set is reflection of the speed and the distance traveled by a digital indicator through digital reporting (Figure 6). The third set is reflection of the speed and the distance traveled by a voice announcement based on digital calculation at the point of the generation of the signal. Categorized as *Choice Requirements* at this level, each of these three sets of choices then generates specific set of *Consequential Requirements* that are mostly different from each other although they have a small common subset as well. The different requirements are in the areas of interfaces and the technologies employed, as requirements *a* and *b* in Figure 5 are different from requirements *a* and *b* in Figure 6. However there is a common requirements subset that defines the abstract logic for the general operation of a speedometer and odometer regardless of their physical type. Hence we distinguish between *physical requirements* and *logical requirements* that might appear as part of either *Choice Requirements* or *Consequential Requirements*. Logical requirements can be common over different physical requirements. Therefore requirements *c* and *d* in Figure 5 are exactly as their counterparts in Figure 6. However, there is an extra logical requirement *e* in Figure 6 that demands speed and distance values be calculated digitally at the point of origination before transmitting to the display environment.

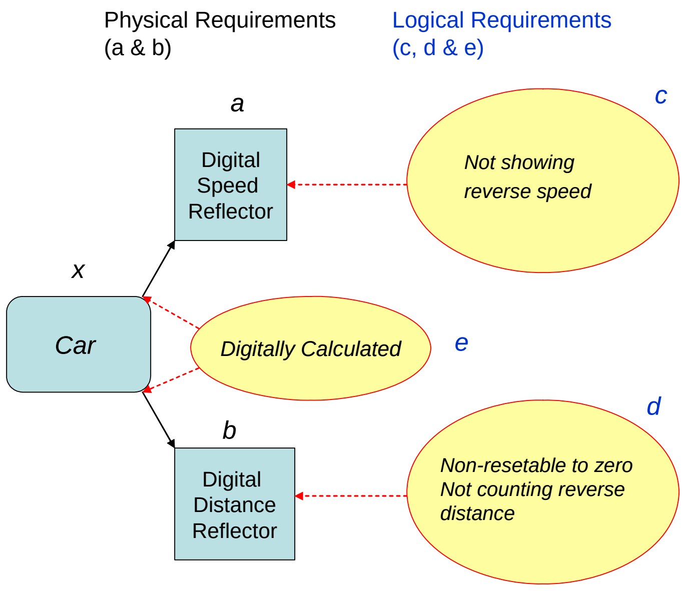


Figure 6: High-level Requirement Model for alternative Requirement Choices

There are two critical concepts that should be discussed with regards to the integrated feature of the *Integrated Triple Sequence Model*: traceability and mapping. Traceability means every lower level requirement should be traceable to a higher level Requirement Model. However we know that at lower levels, new requirements will be generated and these new items might not be directly traceable to higher level models simply because they are silent with regards to more detailed issues. In this case every new requirement should be mapped to certain higher level models that act as a logical container of these new requirements. In other words, the lower level requirements cannot contradict the higher level requirements and also cannot take a scope that clearly crosses that of the upper limit requirements. This is why *Integrated Triple Sequence Model* considers it *mandatory* that every lower level requirement should be either traced or mapped to higher level models to guarantee the integrity of the system. Figure 3 shows the tracing/mapping links of *a* and *b* from RM1 and RM2 to RM.

A Specification Model is a solution response to a Requirements Model, although it drastically changes the presentation of the issues involved. There is a causal relationship between every part of a well defined Requirements Model and a well designed Specification Model. Although a Specification Model will show an explosion of new concepts compared to a corresponding Requirements Model, still every piece of a Specification Model *must* be logically traceable to some concept of its corresponding Requirements Model. Therefore the Requirements Model serves as the upper level logical constraint and boundary setter for its Specification Model. At its highest level, a Specification Model shows the grand architectural choices in response to its corresponding Requirements Model. In Figure 3, SM reflects the grand architectural choices in response to RM. However at this high level, since RM lacks many requirements details, so is SM abstracting many layers of solution. But the critical point is, architectural choices expressed in SM set the grand conceptual boundaries of the system.

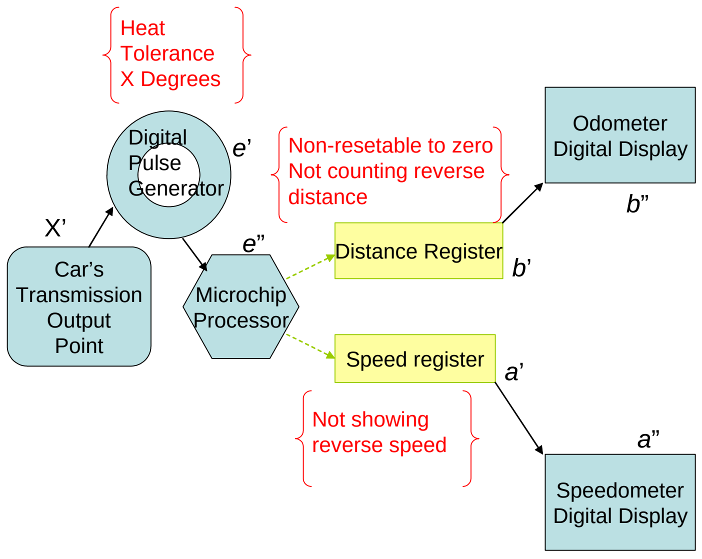


Figure 7: High-level Specification Model corresponding to figure 4

Figure 7 shows a transformation of the Figure 6 Requirements Model. The general entity x as a car transforms the specific point of interface relevant to the subject matter of the Requirements Model. In this specific case the design team has decided that the monitoring point for the car's movement would be the transmission output point (as opposed to rear wheel). Therefore the general entity *x* (a token of the problem domain) in the Figure 6 gets replaced by the specific entity *x'* in Figure 7. The requirement *e* (Figure 6) gets replaced by the special purpose design element *e'* which is a digital pulse generator that attaches to the transmission and transforms the mechanical output motion to a digital pulse readable by a microchip. The arrows indicate the direction of the input flow or the direction of the control. Both entities *e'* and *e"* (Figure 7) trace back to requirement *e* (Figure 6). It should be noted that the two registers *a'* and *b'* (Figure 7) are in fact internal modules of *e"*, and therefore shown by a different representation while broken arrows indicate internal flows. The requirement *a* (Figure 6) gets satisfied by the entities *a'* and *a"* (Figure 7). The logical requirement *c* referring the entity *a* (Figure 6) transforms to a constraint on Speed Register *a'* (Figure 7) shown in brackets following a UML notation. Likewise *b* (Figure 6) translates to *b'* and *b"* (Figure 7) and the logical requirement *d* (Figure 6) transforms to the constraints over *b'* (Figure 7) in brackets. Providing a comprehensive explanation of the Specification Model is out of the scope of this discussion. However this example was presented to provide a conceptual contrast for a Specification Model as opposed to a Requirements Model as well as showing the concept of traceability in spite of the drastic model transformation. Although in these figures the concept of *horizontal traceability* between a Specification Model and Requirements Model was presented, it should be noted that a *vertical traceability* is also involved. Figure 3 shows the vertical traceability links of *c* and *d* from SM1 and SM2 to SM. Vertical traceability provides that the lower level Specification Models are always bound to conceptual upper limits set by the higher level specification Models. That means the lower level SMs cannot contradict the higher level architectural choices, while the latter act as the conceptual upper-bound constraints for the former.

3.2 USE CASE INSTRUMENTALITY

Use cases are known as helpful instruments for reflecting and elaborating the behavior and functionality of a system or application [24, 25]. Use cases are only concerned with what a system does, not how it is implemented. This is exactly why they can be very efficient in reflecting the requirements of a system and for this reason we choose them as an integral part for the *Abstraction-oriented Frames*. In this way use cases will be employed as abstraction instruments for expressing requirements in the *Integrated Triple Sequence Model.* The standard expressiveness of the use cases also helps people with different background and roles, such as architects, developers, project managers and system stakeholders to be able to communicate in a more transparent manner. Use cases are also instrumental in validation of the architecture and verification of the system over its development course [15]. To see the role of use cases in the *Integrated Triple Sequence Model*, we need to focus specifically on each model:

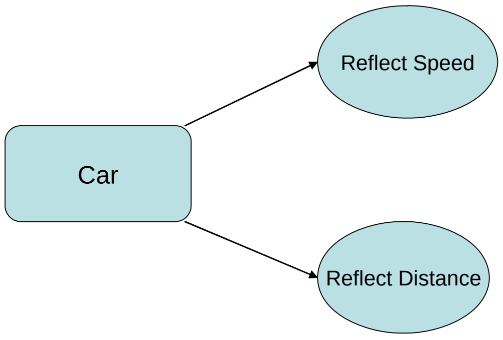


Figure 8: Application of use cases to a Problem Model

3.2.1 PROBLEM MODEL: We discussed the responsibility of a Problem Model as showing a specific problem in its natural context. Looking at a problem in its context at the first glance tends to generate a static picture of a problem. However, a static only picture is most possibly a misleading picture of the reality. Employing use cases in the Problem Model provides a better conceptual environment for viewing the elements of the problem in their dynamic nature. Yet not every element of the Problem Model has to be converted to a UML equivalent [10]. The most important point in a Problem Model is showing the problem in its natural context without any indication of a possible solution. To develop a Problem Model, speaking practically and from a tools point of view, it is best to use a *free form diagram* that most of the current tools support, as opposed to a UML diagram. Figure 8 shows application of use cases to the model in the Figure 4 above. One might imagine reflecting the "car" as an "actor", based on the idea that one definition of the *actor* is an *external system*, and one could conceptualize the car as an external system that those use case are associated with. Yet this is too much conceptualization and contrary to the point that a Problem Model should show the problem in a clear manner to all the roles involved, including the user, in an intuitive way. Still transforming the static concept of "Speed Reflection" (in Figure 4) to "Reflect Speed" use case would prepare the ground for the dynamic aspect of the problem.

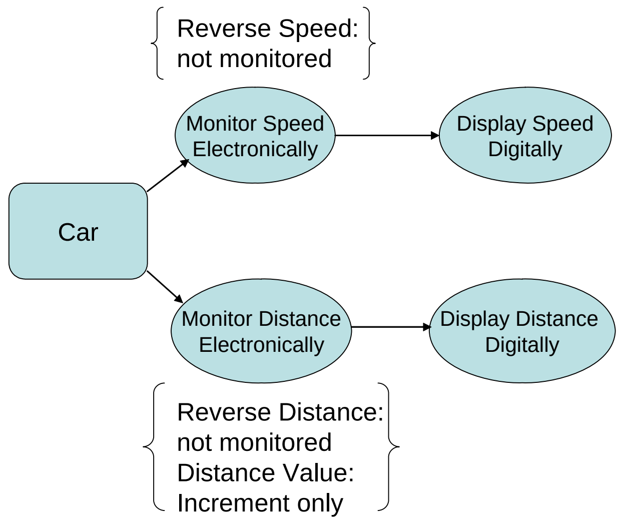

Reverse Speed: not monitored
Monitor Speed Electronically
Display Speed Digitally
Car
Monitor Distance Electronically
Display Distance Digitally
Reverse Distance: not monitored
Distance Value: Increment only

Figure 9: Integration of Use Cases into Requirement Model Applied to Figure 6

3.2.2 REQUIREMENT MODEL: Use cases can be utilized to reflect the functional requirements [8] and as such we integrate them into our defined Requirement Model. We distinguished between two conceptual levels in our model, *Choice Requirements* and *Consequential Requirements.* Both of these categories can consist of functional and nonfunctional requirements. It is to be emphasized that only the functional requirements can be reflected by use cases. The nonfunctional requirements are normally represented in the form of constraints over different system elements. There is a specific advantage in integrating use case instrumentality into our Requirement Model. According to the best practices in modeling use cases, a well-structured use case represents a unit of system functionality that is neither too comprehensive nor too specific. Although expression of *Choice Requirements* might naturally start from some most general levels (as seen in Figure 6), the application of use cases along with their best practice recommendations would guide the modularization of the requirements to more optimal levels at the initial phases of the system architecture.

As demonstrated in Figure 9, the restructuring of the *Requirement Model* in Figure 6 with use cases displays a much better readability, modularization of the functionality, and association of the constraints with their exact underlying modules. Although we have not displayed a "trace" link between different models here, as the models are presented as single slides, "trace" links are applied between the elements of different models when actual UML support tools are used. *Trace* is an UML dependency notation that indicates a historical or process relationship between two elements – in different models – that represent the same concept, without defining rules for deriving one from the other, represented in the form of <<trace>> on the link[15]. We also discussed the mandatory concept of *mapping* between models as well as abstraction levels before. This concept will be implemented by the UML notation <<refine>> that represents a refinement process between two elements at two different semantic levels. If we think about tracing or mapping elements from one model type to another, or from one abstraction level to the other, we realize that there might be in fact a long chain of relationship from the Problem Model to the last implementation detail in the Specification Model, starting from a high abstraction level and ending in a very low abstraction level. Such a chain of relationship should be explicitly and independently documented– in addition to visually showing on the models – in a relational database, as the models form. This process is specifically important for two purposes: 1. a top-down system evaluation on whether and how every single high-level requirement is satisfied, with in fact a reference to the original problem, thanks to the integrated nature of the model; 2. a bottom-up verification on the necessity, authenticity, and traceability of the implemented system specifications. This integrated end-to-end traceability creates a self-controlling process that prevents proliferation of what I call *orphan-specifications* –specifications that are oddly inserted at development levels without being supported by system requirements. Another byproduct of this tight integration is a much higher degree of control over system modification process. Any change in the problem domain –the system environment– or requirement domain which reflects evolution of user demands, can be clearly followed for modification to the last implementation chain. These two byproducts are extremely important for large systems, where no imagination can comprehend what is related to what end-to-end.

Finally, since the focus of this paper is on the presentation of a requirement model, it does not seem it would be a great idea to embark on the details of role of the use cases in the Specification Model as the third element of the *Integrated Triple Sequence Model.* Starting such a discussion, given its dimensions, can very well divert the focus of this paper, and is considered beyond the scope.

3.3 ABSTRACTION-ORIENTED ELICITATION AND CLASSIFICATION

So far we presented a specific framework, *Integrated Triple Sequence Model*, and we extended it by making it partially *use* *case driven*. In this section we present another driver for this modeling framework. However, contrary to the previous driver which was in fact a method driver, this new driver would be a conceptual driver referencing system elements. The necessity of this driver comes from the fact that user requirements or *Choice Requirements* in our model, are initially expressed in imprecise literal language, with no software-oriented conceptualization. Therefore as system designers, we need to have a pattern or a method for reconceptualizing user input to make them appropriately digestible for system design activities. At a fine grained level, this reconceptualization has to target the very basic problem domain elements.

A textual requirements document prepared through a mixture of methods including interviewing the client and prospective system users, is essentially a set of discrete views on collections of desired functionality. Although from a requirements document's perspective, these views might have been defined as separate domains or subdomains, the fact is there are many shared entities, relationships and cross-references that interleave such views. One cannot understand a large system requirements document easily unless many details are abstracted out and the information is modeled properly. In the initial phase of this attempt, the primary system elements should be identified. The practical utility of the AbstFinder [13] application might be under serious questioning, and I consider its analysis out of this discussion, however I consider the analytical model underlying it as useful and choose to modify and extend it as part of our presented model here.

The identification of the primary system elements does not need to be necessarily in the context of the requirements. In other words, identification of the primary conceptual elements of the system at this phase would be performed regardless of their web of relationships. The identification only focuses on the prominence of the concept in the formation of the system. This way we apply the principle of separation of the concerns as a conceptual approach, and try not to miss any primary conceptual system element in the requirements document. *Primary* here is defined as having non-trivial role in the system.

To perform this in a process-oriented way, I propose a Pre-Problem-Model Discovery and Categorization Method (PMDC). This method is a conceptual modification and extension to both KAOS [14] and AbstRM method [13]. KAOS uses a framework of meta-concepts consisting of agents, actions, entities, goals and constraints. Our choice of this framework is based on such reasoning that a requirements document should essentially clarify four categories of conceptual system elements: *who*, *what*, *how*, and *when*.

The "*who*" category includes action initiators and in KAOS meta-concepts framework matches "agents". The "*what*" category includes entities that are subject to actions as well as logical components, and match the "entities" of the KAOS framework. The "*how*" category is the operational link between *who* and *what*, and matches "actions". Furthermore, there is a non-functional attribute of the "actions" which is normally expressed in terms of the *quality of service* or *general delivery of a service*, and therefore "*how*" matches both "constraints" and "goals" in the framework. It is important to note that "*how*" in the requirements context does not have implementation connotation but it is a general conceptual category that basically covers the dynamic aspects of a system – regardless of their ordering – that happen to be the linkage between *who* and *what* categories. The ordering aspects of the dynamics of a system, that can be viewed as a flow-oriented concept, corresponds to the "when" concept, which does not seem to be addressed by the KAOS categories. Therefore we extend the KAOS categories by a new category of "*relations*" explained later. It also should be noted that the "*goals*" category has the potential of becoming controversial at times, but the fact is in the context of requirement analysis, it might be a very close concept to the "*constraints*" category. Figure 10 presents a mapping and some examples.

General Conceptual Categories	KAOS Meta-concepts	Examples	Notes
who	agents	customer	
what	entities	account	
how	actions	buy	
how	constraints	confidentiality	
how	goals	fast execution	
when		flow/execution order	Our extension

Figure 10: Mapping general conceptual categories to KAOS meta-concepts

In understanding the Pre-Problem-Model Discovery and Categorization Method (PMDC) we need to distinguish between two domains, *requirements document* and *Problem Model*. The requirement document is the document that is prepared as a result of many interviews and exchanges of a designated requirement team with the system stakeholders, reflecting the stakeholders' views and understanding of both the problem domain and requirements, and is normally written in plain English *without* a lot of technical fantasies. The temptation is to jump from the requirement document straight to developing a requirement model. However, according to our *Abstraction-oriented Frames* method, in any iteration, we need to start by developing the first of the *Integrated Triple Sequence Model* which is the "Problem Model". Therefore the prepared *requirements document* plus technical guidance by some independent domain expertise need to initially generate a Problem Model, and then the Problem Model transform into the Requirements Model (Figure 11). The PMDC focuses exactly on this preparatory phase.

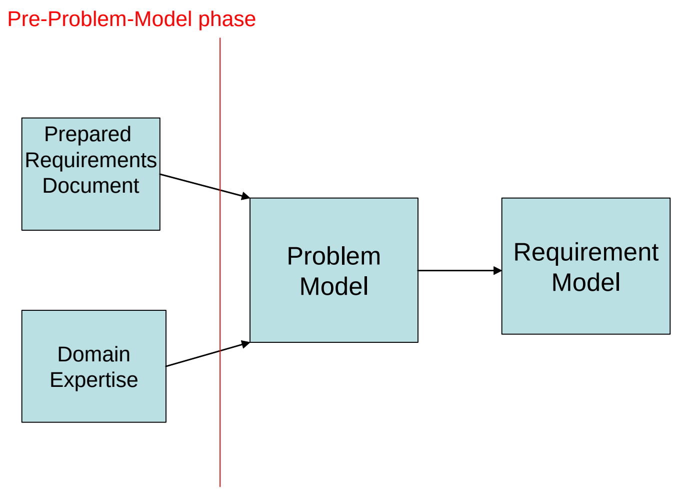


Figure 11: Pre-Problem-Model Discovery and Categorization Method activity scope

As an instance, one might consider a startup Company X that intends to offer stock trading services to the public. In fact we are shifting the example from the above car modeling case to get a more complex model with elements better known to the general audience. In this example the Company X hires software developing Company S to develop its platform software. The designated requirements team of the Company S prepares a plain English requirements document after long discussions with entrepreneurs of the Company X, which reflects the range of services the company intends to offer, the quality attributes of those services, as well as the understanding they have about the environment they intend to operate in. However, there are tons of regulations, practical bottlenecks and possible legal issues that are not simply part of the general knowledge of the entrepreneurs of a stock trading company, but need to be practically integrated into the software platform. Here the requirements team of Company S needs extensive expert advice to understand and model the general problem domain, while modeling the specific problem domain of the Company X. The *domain experts* perform independent of the plans and needs of the Company X. Furthermore, the Problem Model is independent of any possible future solutions, and only reflects the microcosm of the Company X working environment with the boundaries it has selected for activity.

Focusing on this preparatory phase, Pre-Problem-Model Discovery and Categorization Method (PMDC) is responsible to organize the environment by abstracting, classifying and processing problem domain elements. This is a type of conceptualization which should be considered a bit different from design-oriented concepts like use cases, actors, classes, attributes and operations. Among other reasons, it is partly because of the type of the people involved – like domain experts – who are not software engineers, and partly because of the universe of thinking, which is the problem domain as opposed to a software project environment. At this point it seems best to explain PMDC in the context of this practical example. Figure 12 shows an incomplete Pre-Problem-Model – for simplification purposes – for the Company X platform. We choose to design a Pre-Problem-Model in an XML schema or XSD format. The reason for this architectural choice is that we are technically trying to convert a plain English document to a structured model as we parse through it. We also need to have an environment that we can easily expand and modify different parts of the model, while having the modifications automatically reflected through all parts of the document. Yet we do not want to get engaged with approaches like expert UML modeling or use of OCL, because we work in a phase that unstructured human interactions (that of system stakeholders) as well as necessity of readability for non-software-engineers is a fact of life and we mean to be free of all software-oriented preconceptions to be able to reflect the problem domain. It should be emphasized that PMDC phase by no means tries to capture any solution domain model or even strict requirements modeling. Therefore an XML schema model can be a very appropriate choice. Furthermore, an XML schema can easily translate to a visual model that greatly facilitates the architectural development and conceptualization. This is obviously in addition to the fact that custom designed logic and programs can further machine process such a transformed requirements document. In fact this line of thinking can be considered as a positive potential of this architectural choice and be designated as “future work” in this area.

Figure 12 reflects the five KAOS categories modeled in XSD, but does not show our extension category, “*relations*”, just to indicate here that the latter belongs to a second iteration of processing. The first layer of the model is intentionally organized under an <xsd:sequence> element set descriptor. Although from a classification point it wouldn’t have made any difference if we used <xsd:all>, the connotation is that there is a method-oriented priority on the higher order classifiers. Unified Process (UP) recommends initiating the model development activity with identification of “actors” and roles [8]. This is why we list “agents” as the first in sequence to keep this method recommendation visually and practically emphasized. We define “agents” as a set of actors or behavior initiators. We record such definitions directly in the model using <xsd:annotation> and <xsd:documentation>, and the model would be self explaining. The subset of “agents” and other second layer elements are organized under <xsd:all>.

“Agents” conceptually connect to “entities” by means of “actions”. So our abstraction identification eyes, when parsing the requirement document, is focused on the abstract triple sets of “who” agent reaches “what” entity by means of “how” action. Sequencing these three elements indicates their conceptual priority. The dynamics are then qualified and constrained by the other two elements “goals” and “constraints”. The elements of these five categories are mapped during the first major iteration of parsing of the requirements document as shown in Figure 12.

The second major iteration moves to refining the model by adding the “relations” category which is a reflection of dynamics of the system. The role of this element in the model is one of a moderator and adjustor. This element does not collect any new abstraction per se directly from the requirements document. However, by focusing on the dynamics of the model, redefines the previously mapped elements. Since most of the root elements out of requirements document are modeled as global elements, any modification of them under "relations" automatically reflects to all other parts of the model which means reorganization of the model given the dynamics of the system. We define "relations" as a flow-oriented concept or element, in terms of its modeling. Keep in mind we are in the problem domain and we need to focus on how flows naturally occur in the environment. Since the model-level flows are fundamentally initiated from the "agents", the "relations" is modeled, at least at its higher layers, in an agent-flow oriented approach. Hence we have elements like "customerFlow" and "tradingCompanyFlow" (Figure 13). Using the "ref" feature, customer added here (<xsd:element ref="customer"/>) is exactly the same element already listed under "agents". As Figure 12 shows, customer and account are two independent global elements extracted from the requirements document and each placed in its own classifier. Although we know they have some relationship, the model at this level does not show any relationship. Figure 14 shows the model extended with the "relations" element, with the subelements defined in terms of agent flows, like "customerFlow". When customer is added to this element initially, "customer" does not have any subelements. By the token of the requirements document we know that customer has an account to work with and can reach customerService and technicalSupport. So we add these elements to the customer right at this place in the model.

As soon as we add such dynamic based concepts to this part of the model, the "agents" section of the model gets automatically

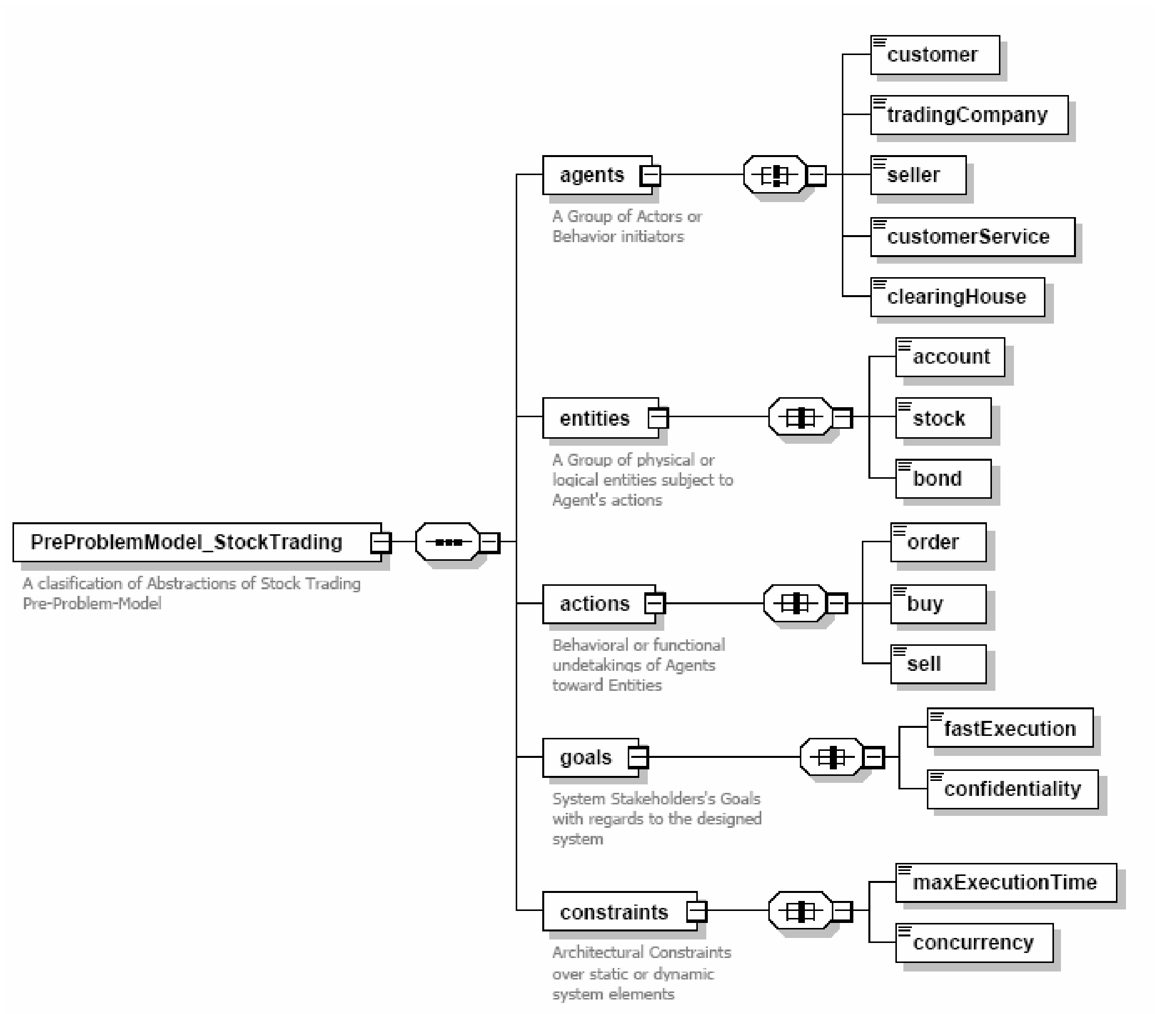

PreProblemModel_StockTrading
A clasification of Abstractions of Stock Trading Pre-Problem-Model
agents
A Group of Actors or Behavior initiators
customer
tradingCompany
seller
customerService
clearingHouse
entities
A Group of physical or logical entities subject to Agent's actions
account
stock
bond
actions
Behavioral or functional undetakings of Agents toward Entities
order
buy
sell
goals
System Stakeholders's Goals with regards to the designed system
fastExecution
confidentiality
constraints
Architectural Constraints over static or dynamic system elements
maxExecutionTime
concurrency

Figure 12: Pre-Problem-Model for the Company X platform

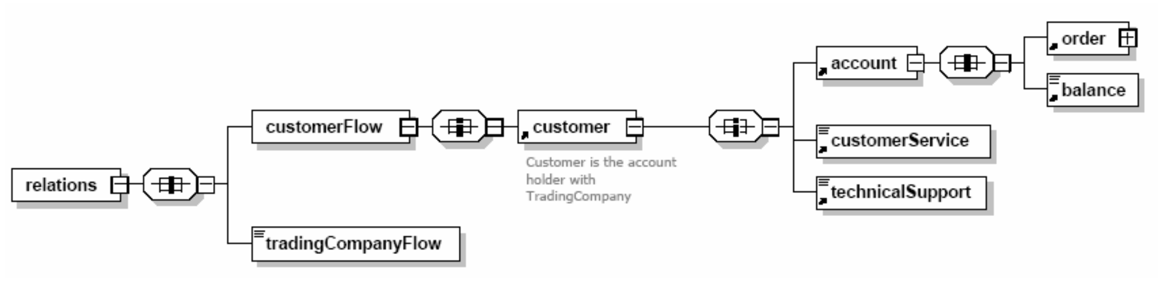


Figure 13: Relations model element

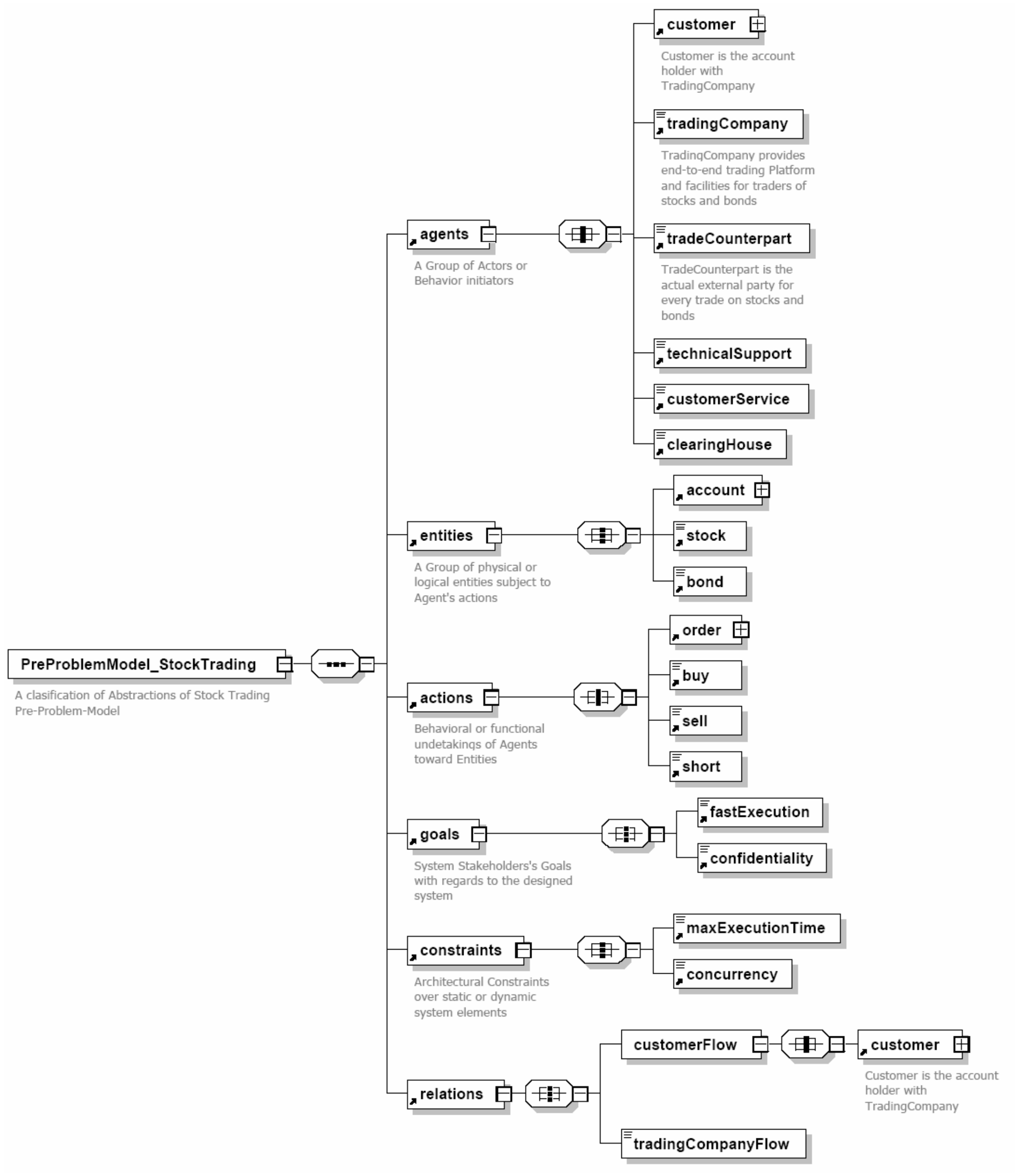

PreProblemModel_StockTrading
A clasification of Abstractions of Stock Trading Pre-Problem-Model
agents
A Group of Actors or Behavior initiators
customer
Customer is the account holder with TradingCompany
tradingCompany
TradinqCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
tradeCounterpart
TradeCounterpart is the actual external party for every trade on stocks and bonds
technicalSupport
customerService
clearingHouse
entities
A Group of physical or logical entities subject to Agent's actions
account
stock
bond
actions
Behavioral or functional undetakings of Agents toward Entities
order
buy
sell
short
goals
System Stakeholders's Goals with regards to the designed system
fastExecution
confidentiality
constraints
Architectural Constraints over static or dynamic system elements
maxExecutionTime
concurrency
relations
customerFlow
customer
Customer is the account holder with TradingCompany
tradingCompanyFlow

Figure 14: Extended Pre-Problem-Model for the Company X platform

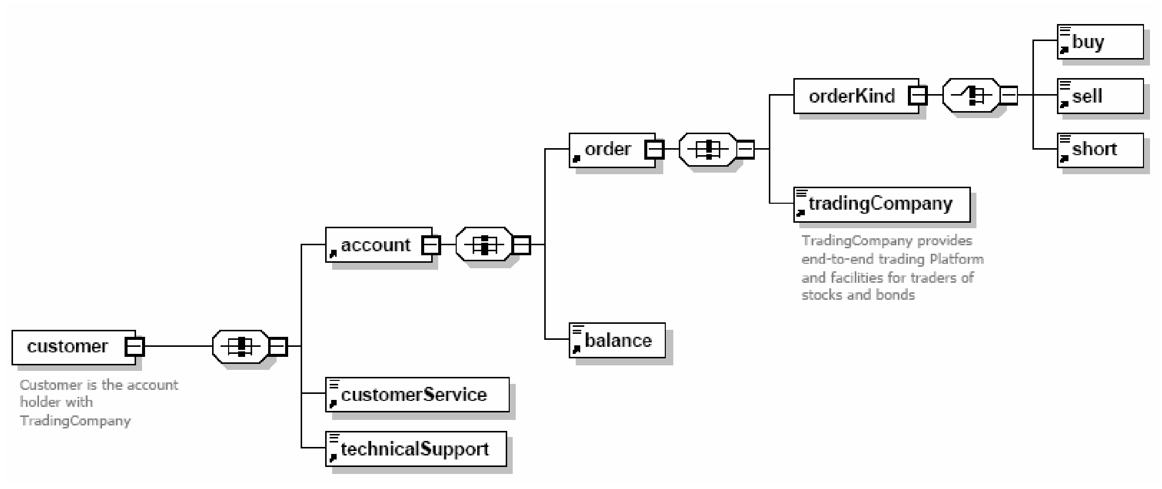

customer
Customer is the account holder with TradingCompany
account
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
balance
customerService
technicalSupport

Figure 15: Customer element with its dynamically added subelements

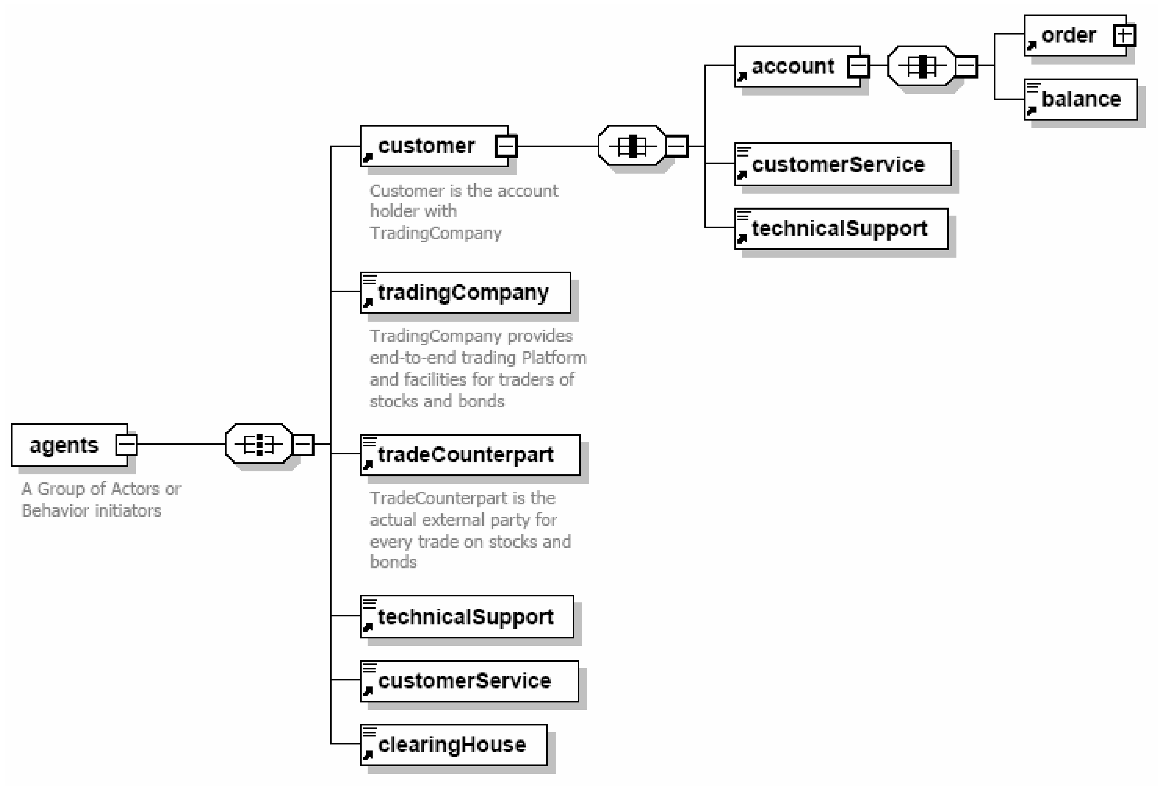

agents
A Group of Actors or Behavior initiators
customer
Customer is the account holder with TradingCompany
account
order
balance
customerService
technicalSupport
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
tradeCounterpart
TradeCounterpart is the actual external party for every trade on stocks and bonds
technicalSupport
customerService
clearingHouse

Figure 16: Agents element automatically updated

updated, showing customer to have the added subelements (Figure 16). Likewise account and order belonging to two separate classifiers, do not show any relationship at the first iteration of *abstraction identification*. Yet the dynamic view describes that a customer places an order in her account. Therefore under the relations, we add order as a subelement of account and this automatically updates the “entities” section of the model (Figure 17). It should be noted that in the recent figures we have been showing the graphics just at the level of single elements as the model has been getting much wider than the width of a printed page.

On the other hand, the old model in Figure 12 shows three elements under “actions” classifier, order, buy and sell. They were listed there in the first iteration which was *abstraction identification*, picked as a result of parsing the requirement document and assigned to the relevant classifier. Now in a flow-oriented approach we recognize that buy and sell are only two types or kinds of an order. So we generate a new element of orderKind (Figure 15), which is a new local element by itself, and put buy, sell and short as its subelements. Although done under relations, this leads to “actions” and “accounts” classifiers to get automatically updated showing the new relationships (Figures 18 and 19). At the same time all the cumulative changes get reflected under the customer element of the model which is a member of “agents” classifier (Figure 15). As it can be seen clearly, this method and modeling technique has the capacity and advantage of developing models in an exponential way while preserving the consistency of the model as it gets larger. On the other, hand since the human focus for the model development and relationship design is at the level of one element at a time, the model management and readability improves drastically. After all, any subsequent changes and modifications to the model elements get *consistently* and *automatically* reflected throughout the entire model. Appendix 1 shows the code for the generated model which has passed the check for well-formedness and is also a validated code. The completion of such an XSD model at the Pre-Problem-Model phase constitutes most of the Problem Modeling. Depending on the case and domain, this can even be considered by an architect as the Problem Model if s/he does not intend to develop a model using other methods. In other words, only if the architect intends to develop a Problem Model using another language such as UML, or trying to present different perspectives, s/he needs to convert the model, and in that case the semantics are basically there laid out in the present model. It should also be noted that developing such XSD models can contribute to libraries of extensible domain-oriented models for different types of problems, in this case a stock trading platform.

3.4 IMPLEMENTATION

Contrary to some methods or frameworks that are offered through research papers and are dependent on a specific supplemental proprietary implementation environment, written by the author(s) and not normally accessible for testing, “Abstraction-oriented Frames” is a general purpose framework that can be implemented on general purpose and publicly available platforms.

Although there are quite a few industrial-grade UML implementation environments that could be utilized, for our purposes, we have implemented this framework through a combination of tools. For the main framework we used a combination of the Rational RequisitePro and IBM Rational Software Architect V 6.0.1.1. We extended the requirement types of the RequisitePro by our four suggested concepts, *Choice Requirement*, *Consequential Requirement*, *Physical Requirement* and *Logical Requirement*. RequisitePro was then plugged into IBM Software Architect. This way a preliminary model developed in RequisitePro can be directly used in the Architect environment. The Integrated Triple Sequence Model was implemented on the IBM Rational Software Architect. Each of the three models is implemented as a logical package. For extensive or complex systems, each model can be initially implemented as physical package that subsequently holds further logical packages. Each *n+1* vertical layer (Figures 2 & 3) under each of three model types appears as a child package for the higher abstraction layer. This way the system has a self-guiding mechanism routing through lower levels of abstraction while strictly confining and packaging the elements at the same level. To better keep track of the corresponding layers across the Triple Sequence, an *n+1* numbering notation is used as part of the naming convention for the packages that clearly indicates where in the vertical ladder each package is situated. The Problem Model is implemented as a use case model, although it could also be a blank model. The Requirement Model is implemented as a use case model. The Specification Model is implemented as an analysis model for the higher level modeling and design phases. Each model occupies one modeling file (.emx) and can hold any number of packages. PMDC can be implemented in the same environment as an XSD model, or can be implemented in external platforms and then plugged into this environment or imported.

Other suites like Borland Architect and Rational Rose could be employed as well. As mentioned, this is due to the fact that the “Abstraction-oriented Frames” is not a product or development environment but an end-to-end conceptual framework and method that could be carried out using any software suite or development environment that offers serious UML and XML schema support.

4. CONCLUSION AND FUTURE WORK

This paper presented a conceptual model for requirements analysis which is structured in the form of an operational framework combined with a method. The model is a hybrid model, thus effectively using some of the established procedures, but modifies and extends them into a new consistent model which is specifically designed to support the open systems construction. The proposed framework is called “Abstraction-oriented Frames” that consists of three distinct extended perspectives, conceptually glued together into a cohesive operational framework and method, built on the top of a level zero startup phase which is based on XML schema modeling. Although it is perfectly general purpose, this framework is developed to be used for the design of a model open evolvable system architecture under the umbrella Pebbles Project at the Computer Laboratory. A partial model is already designed using this framework, however the volume of the material prevents it from being demonstrated here as an example, and the completion of the work is in process.

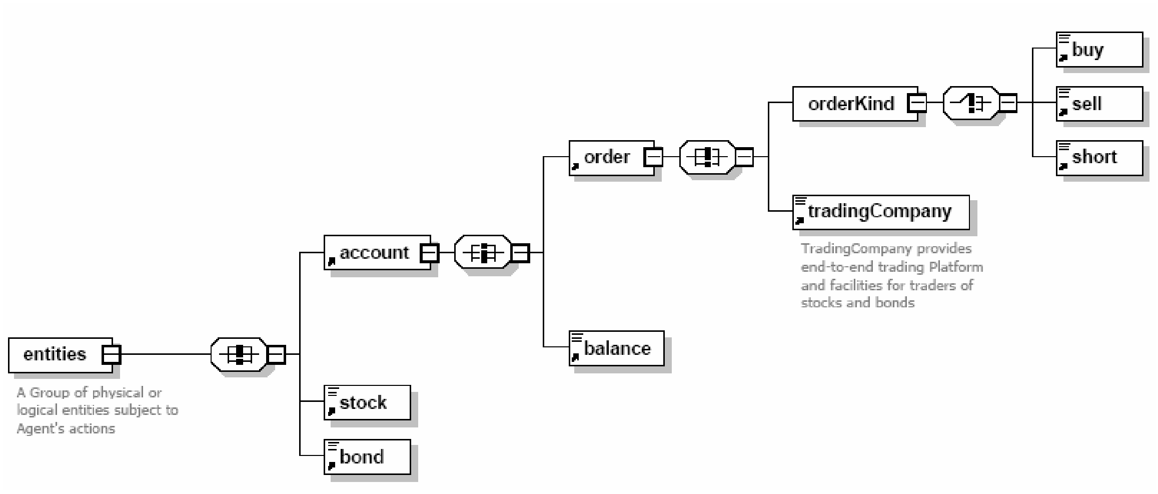

entities
A Group of physical or logical entities subject to Agent's actions
account
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
balance
stock
bond

Figure 17: Entities automatically updated

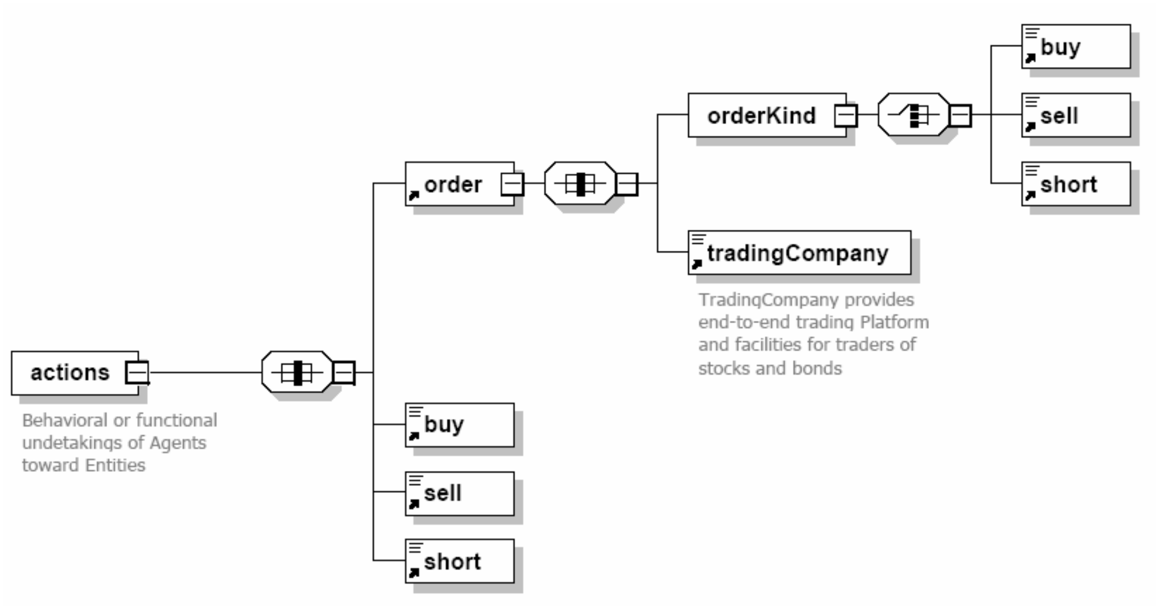

actions
Behavioral or functional undetakings of Agents toward Entities
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
buy
sell
short

Figure 18: Actions automatically updated

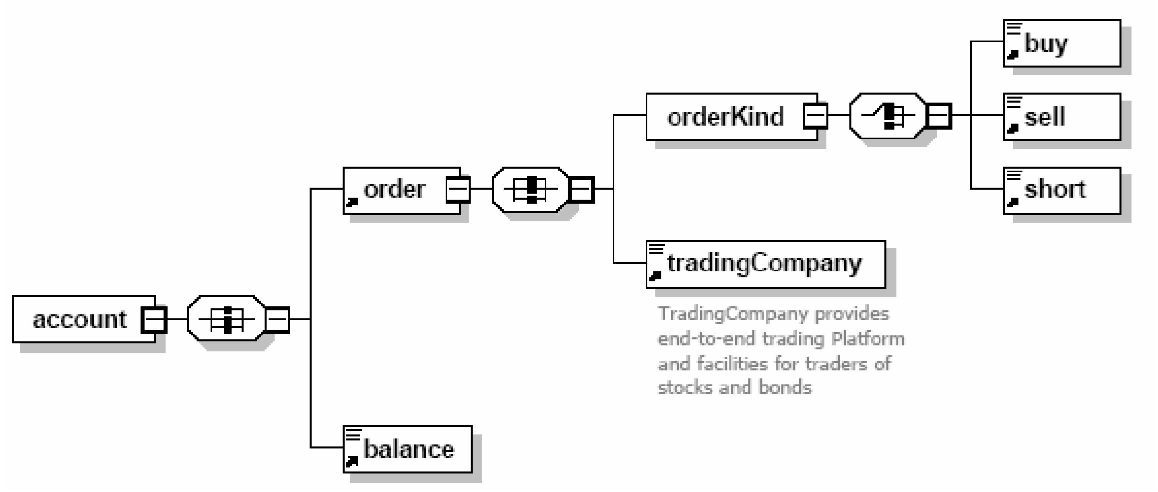

account
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
balance

Figure 19: Accounts automatically updated

REFERENCES:

[1] Sommerville, I., Ransom J. (2005): An empirical study of industrial requirements engineering process assessment and improvement. ACM TOSEM, Vol. 14, No. 1, pp 85-117.

[2] Jackson, M. (1995): Software Requirements & Specifications. New York, NY: The ACM Press.

[3] Escalona, M. J., Koch, N. (2006): Metamodeling the Requirements of Web Systems. In *WEBIST 2006- The Second International Conference for Web Information Systems and Technologies*, Setubal, Portugal, 11.04.2006.

[4] Berry, M.D., Lawrence, B. (March 1998): Requirements Engineering. IEEE Software 15, 2, pp 26–29.

[5] Seater, R., Jackson, D. (May 2006): Problem Frame Transformations: Deriving Specifications from Requirements. In *IWAAPF '06- Proceedings of the 2006 international workshop on Advances and applications of problem frames*.

[6] Castro, J., Kramer, J. (July 2001): From Software Requirements to Architectures. *Proceedings of the 23rd International Conference on Software Engineering.*

[7] Nuseibeh, B., Easterbrook, S. (June 2000): Requirements Engineering: A Roadmap. In *The Future of Software Engineering 2000*, ed. A. Finkelstein. ACM, Limerick, Ireland.

[8] Arlow, J., Neustadt, I. (2005): UML 2 and the Unified Process. Addison-Wesley.

[9] IBM-Rational (2003): Rational Unified Process software package integrated Help.

[10] Choppy, C., Reggio, G.A. (2004): A UML-Based Method for the Commanded Behaviour Frame. In *The Proceedings of IWAAPF04*, Eds. Karl Cox, Jon G. Hall and Lucia Rapanotti. IEE, pp. 27-34.

[11] Paulk, M.C., et al. (1993): Key Practices of the Capability Maturity Model. Technical Report, CMU/SEI-93-TR-25, Software Engineering Institute, Pittsburg, US.

[12] Jackson, M. (2001): Problem Frames. The ACM Press.

[13] Goldin, L., Finkelstein, A. (May 2006): Abstraction-Based Requirements Management. In *Proceedings of the 2006 International Workshop on Role of Abstraction in Software Engineering ROA '06.* ACM Press.

[14] Bertrand, P., et al. (April 98): GRAIL/KAOS: An Environment for Goal Driven Requirements Engineering. In *The proceedings of ICSE'98 - 20th International Conference on Software Engineering.* IEEE-ACM, Kyoto.

[15] Booch, G., et al. (1999): The Unified Modeling Language User Guide. New York, NY: Addison-Wesley.

[16] Lavazza, L., Del Bianco, V.A. (2004): UML-based Approach for Representing Problem Frames. In *The Proceedings of IWAAPF04*, Eds. Karl Cox, Jon G. Hall and Lucia Rapanotti. IEE, pp. 39-48.

[17] Jackson, M.C. (1997): Critical Systems Thinking and Information Systems Development. In *The Proceedings of Eighth Australasian Conference on Information Systems*, University of South Australia, Adelaide, South Australia, pp. 1-20.

[18] Jacobson, I., Christerson, M. (1995): Modeling with Use Cases: A Growing Consensus on Use Cases. Journal of Object-Oriented Programming (8), pp. 15-19.

[19] W3C, XML Schema Specifications. http://www.w3.org/XML/Schema

[20] Hall, J. G., et al. (2002): Relating Software Requirements and Architectures Using Problem Frames. In *Proceedings of 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pp. 137-144.

[21] Ferreira, M.J., Loucopoulos, P. (2001): Organisation of Analysis Patterns for effective Re-use. In *Proceedings of the International Conference on Enterprise Information Systems. ICEIS 2001*. Setubal, Portugal.

[22] Jacobson I., Booch G. & Rumbaugh J. (2000): The Unified Software Development Process. Addison-Wesley-Longman.

[23] Bergström, S., Råberg, L. (2003): Adopting the Rational Unified Process: Success with the RUP. Addison-Wesley Professional.

[24] Alexander, I., Maiden N. (Eds), (2004): Scenarios, Stories and Use Cases: Through the Systems Development Life-Cycle. John Wiley & Sons.

[25] Mavin, A., Maiden, N. (2003): Determining Socio-Technical Systems Requirements: Experiences with Generating and Walking Through Scenarios. In *Proceedings of 11th International Conference on Requirements Engineering.* IEEE Computer Society Press, pp 213-222.

[26] IEEE (1998). IEEE Recommended Practice for Software Requirements Specifications. IEEE STD-830.

[27] Kovitz, B. (1998): Practical Software Requirements: A Manual of Content and Style. Manning Publications, Greenwich, CT.

[28] Robertson, S., Robertson, J. (2006): Mastering the Requirements Process. Addison-Wesley Professional.

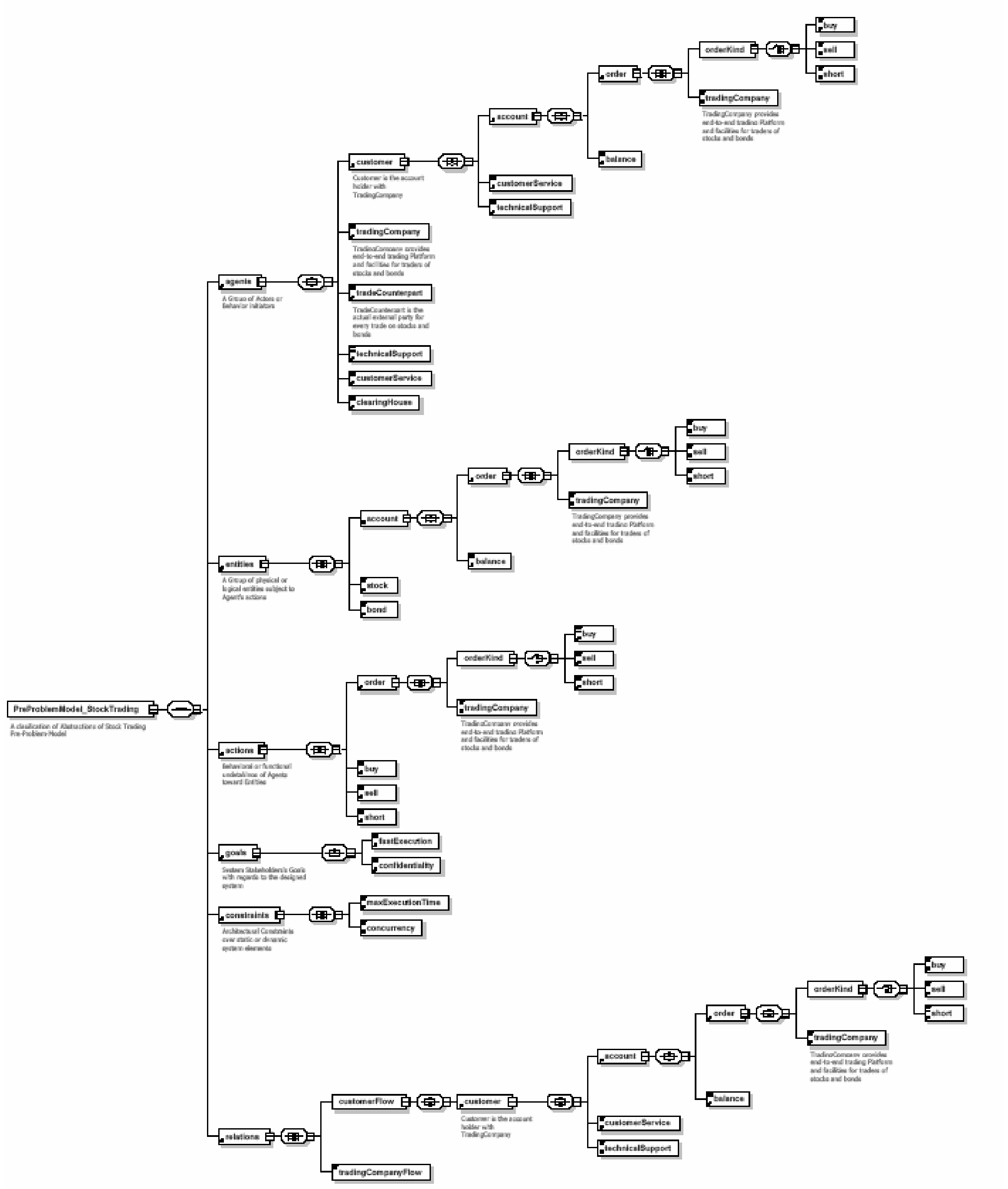
APPENDIX 1

Validated XML code for the PreProblemModel_StockTrading.xsd in Figure 14:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- PreProblemModel_StockTrading) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
	<xs:element name="PreProblemModel_StockTrading">
		<xs:annotation>
			<xs:documentation>A classification of Abstractions of Stock Trading Pre-Problem-
Model</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="agents"/>
				<xs:element ref="entities"/>
				<xs:element ref="actions"/>
				<xs:element ref="goals"/>
				<xs:element ref="constraints"/>
				<xs:element ref="relations"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="customer">
		<xs:annotation>
			<xs:documentation>Customer is the account holder with TradingCompany
</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:all>
				<xs:element ref="account"/>
				<xs:element ref="customerService"/>
				<xs:element ref="technicalSupport"/>
			</xs:all>
		</xs:complexType>
	</xs:element>
	<xs:element name="agents">
		<xs:annotation>
			<xs:documentation>A Group of Actors or Behavior initiators</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:all>
				<xs:element ref="customer"/>
				<xs:element ref="tradingCompany"/>
				<xs:element ref="tradeCounterpart"/>
				<xs:element ref="technicalSupport"/>
				<xs:element ref="customerService"/>
				<xs:element ref="clearingHouse"/>
			</xs:all>
		</xs:complexType>
	</xs:element>
	<xs:element name="tradingCompany">
		<xs:annotation>
			<xs:documentation>TradingCompany provides end-to-end trading Platform and
facilities for traders of stocks and bonds</xs:documentation>
		</xs:annotation>
	</xs:element>
	<xs:element name="tradeCounterpart">
		<xs:annotation>
			<xs:documentation>TradeCounterpart is the actual external party for every trade
on stocks and bonds </xs:documentation>
		</xs:annotation>
	</xs:element>
	<xs:element name="customerService"/>
	<xs:element name="clearingHouse"/>
	<xs:element name="account">
		<xs:complexType>
			<xs:all>
				<xs:element ref="order"/>
				<xs:element ref="balance"/>
			</xs:all>
		</xs:complexType>
	</xs:element>
	<xs:element name="stock"/>
	<xs:element name="bond"/>
	<xs:element name="entities">
		<xs:annotation>
			<xs:documentation>A Group of physical or logical entities subject to Agent's
actions</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:all>
				<xs:element ref="account"/>
				<xs:element ref="stock"/>
				<xs:element ref="bond"/>
			</xs:all>
		</xs:complexType>
	</xs:element>
	<xs:element name="actions">
		<xs:annotation>
			<xs:documentation>Behavioral or functional undertakings of Agents toward
Entities</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:all>
				<xs:element ref="order"/>
				<xs:element ref="buy"/>
				<xs:element ref="sell"/>
				<xs:element ref="short"/>
			</xs:all>
		</xs:complexType>
	</xs:element>
	<xs:element name="order">
		<xs:complexType>
			<xs:all>
				<xs:element name="orderKind">
					<xs:complexType>
						<xs:choice>
							<xs:element ref="buy"/>
							<xs:element ref="sell"/>
							<xs:element ref="short"/>
						</xs:choice>
					</xs:complexType>
				</xs:element>
				<xs:element ref="tradingCompany"/>
			</xs:all>
		</xs:complexType>
	</xs:element>
	<xs:element name="buy"/>
	<xs:element name="sell"/>
	<xs:element name="goals">
		<xs:annotation>
			<xs:documentation>System Stakeholders's Goals with regards to the designed
system</xs:documentation>
		</xs:annotation>
		<xs:complexType>
			<xs:all>
				<xs:element ref="fastExecution"/>
				<xs:element ref="confidentiality"/>
			</xs:all>
		</xs:complexType>
	</xs:element>
```

```
      <xs:element name="fastExecution"/>
      <xs:element name="confidentiality"/>
      <xs:element name="constraints">
            <xs:annotation>
                  <xs:documentation>Architectural Constraints over static or dynamic system
elements</xs:documentation>
            </xs:annotation>
            <xs:complexType>
                  <xs:all>
                        <xs:element ref="maxExecutionTime"/>
                        <xs:element ref="concurrency"/>
                  </xs:all>
            </xs:complexType>
      </xs:element>
      <xs:element name="maxExecutionTime"/>
      <xs:element name="concurrency"/>
      <xs:element name="relations">
            <xs:complexType>
                  <xs:all>
                        <xs:element name="customerFlow">
                              <xs:complexType>
                                    <xs:all>
                                          <xs:element ref="customer"/>
                                    </xs:all>
                              </xs:complexType>
                        </xs:element>
                        <xs:element name="tradingCompanyFlow"/>
                  </xs:all>
            </xs:complexType>
      </xs:element>
      <xs:element name="technicalSupport"/>
      <xs:element name="short"/>
      <xs:element name="balance" type="xs:double"/>
</xs:schema>
```

VIEW OF THE COMPLETE CODED PMDC MODEL:


PreProblemModel_StockTrading
A classification of Abstractions of Stock Trading Pre-Problem-Model
agents
A Group of Actors or Behavior initiators
customer
Customer is the account holder with TradingCompany
account
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
balance
customerService
technicalSupport
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
tradeCounterpart
TradeCounterpart is the actual external party for every trade on stocks and bonds
technicalSupport
customerService
clearingHouse
entities
A Group of physical or logical entities subject to Agent's actions
account
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
balance
stock
bond
actions
Behavioral or functional undertakings of Agents toward Entities
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
buy
sell
short
goals
System Stakeholder's Goals with regards to the designed system
fastExecution
confidentiality
constraints
Architectural Constraints over static or dynamic system elements
maxExecutionTime
concurrency
relations
customerFlow
customer
Customer is the account holder with TradingCompany
account
order
orderKind
buy
sell
short
tradingCompany
TradingCompany provides end-to-end trading Platform and facilities for traders of stocks and bonds
balance
customerService
technicalSupport
tradingCompanyFlow