Systems Engineering Foundations of Software Systems Integration

Peter Denno and Allison Barnard Feeney

National Institute of Standards and Technology,
Gaithersburg, MD 20899, USA
peter.denno@nist.gov, abf@nist.gov

Abstract. This paper considers systems engineering processes for software systems integration. Systems engineering processes, as intended here, concern how engineering capability should be factored into problem-solving agencies for application to software systems integration tasks, and how the results produced by these agencies should be communicated and integrated into a system solution. The environment in which systems integration takes place is assumed to be model-driven; problem-solving agencies, working from various viewpoints, employ differing notations and analytical skills. In the course of identifying the systems engineering process, the paper presents a conceptual model of systems engineering, and reviews a classification of impediments to software systems integration.

1 Introduction

Software systems integration entails systems engineering, whether one consciously practices it or not. Systems integration starts with the recognition of new requirements and is not complete until one validates the resulting system against those requirements. A premise of this paper is that choice of systems engineering process is a matter of significant concern, particularly if one hopes to automate portions of the process in a model-based environment. [1]

A *model-based environment* (MBE) is an environment for systems integration that emphasizes the role of models in automating an integration process. The environment envisaged in this paper would provide an incrementally refinable account of a business process and the system that implements it. The account is derived from the union of all available views of the system. Views are provided in various notations (or "viewpoint technologies"). The MBE serves three purposes: (1) it fosters coherence among views by recognizing refinements and interrelations among disparate models; (2) it enables communication between problem solvers (human or automated) working in differing viewpoints toward

[1] Commercial equipment and materials are identified in order to describe certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

the resolution of an integration task; and, (3) it provides links from views that state or elaborate requirements to those that posit design commitments.

The concerns just mentioned, heterogeneous views, viewpoint communication, requirements and their allocation[2] to problem solvers, beg the question of what sort of systems engineering process the MBE should implement.[3] The answer to this question is the central point of this paper. The issue (1) above, of providing inter-model coherence is discussed in [8].

Section 2 of the paper describes a conceptual model of systems engineering. *Section 3* summarizes previous work which presents a classification of impediments to the integration of software-intensive systems. [3] *Section 4*, concerns the central point of the paper. It considers choices of systems engineering processes for an MBE for software systems integration. *Section 5* outlines our MBE architecture. *Section 6* discusses related work and future plans for the MBE described in the paper. *Section 7* provides conclusions. The final section, *Appendix A*, is a glossary of terms from the systems engineering conceptual model.

2 A conceptual model of systems engineering

Systems engineering is any methodical approach to the synthesis of an entire system that (1) defines views of that system that help elaborate requirements, and (2) manages the relationship of requirements to performance measures, constraints, components, and discipline-specific system views. *Figure 1* provides a UML class diagram of a conceptual model of systems engineering. *Appendix A* provides definitions of concepts presented in the UML.

The key terms in this definition of systems engineering are "requirements" and "views." Systems engineering is foremost about stating the problem, whereas other engineering disciplines are about solving it. A *view* is a representation of the whole system from the perspective of a related set of concerns [14]. An example of a view that "help[s] elaborate requirements" is a functional model of the system, that is, a view that identifies the resources, roles, and processes involved in fulfilling the purpose of the system. A *viewpoint* is a method founded on a body of knowledge of some engineering or analytical discipline and used in constructing a view (derived from [14]). A view is an application of a viewpoint. An architecture description language (ADL), such as Wright [1], provides a viewpoint that serves this role in systems engineering.

[2] The term *requirements allocation* refers to the task of charging problem solvers with the task of meeting requirements. *Technical problem solvers* meet requirements by making design commitments. *Conceptual problem solvers* refine original requirements and assert derived requirements.

[3] We distinguish a *systems engineering process* from a *design methodology* in that a design methodology only prescribes a path through the space of possible refinements (or design commitments). A systems engineering process prescribes the decomposition into agencies (viewpoints, problem solvers) that make the commitments, and a means to orchestrate these agencies.

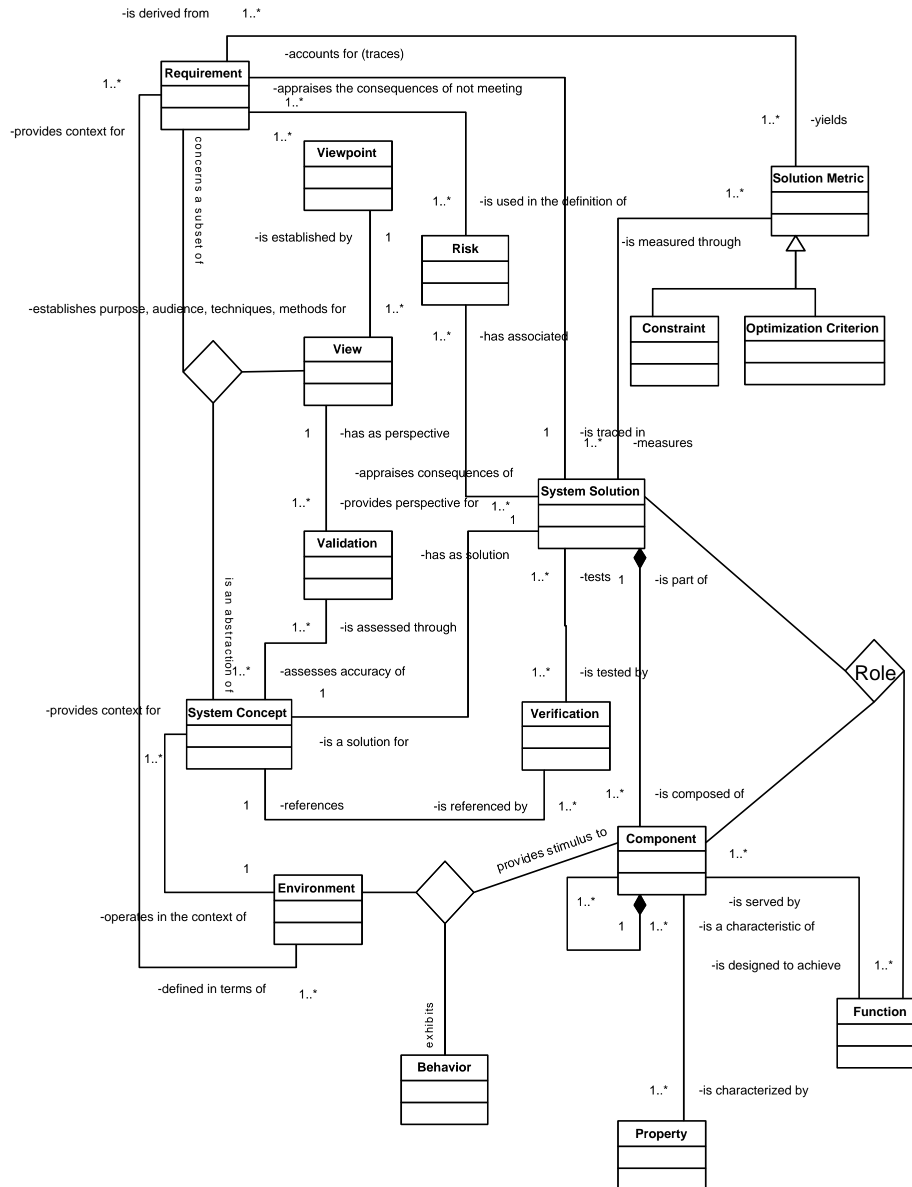

Fig. 1. Conceptual model of systems engineering

Because an ADL describes a functional model of the system,[4] it posits, to some level of specificity, a componentization of the system and an implied correspondence to requirements. That is, every component has its purpose and that purpose can be traced to the requirements.

A view represents a collection of requirements. The utility of a view depends largely on whether or not there exists a corresponding body of knowledge and technology to draw from; that is, whether its viewpoint exists. The body of knowledge underlying the viewpoint (the technology and expertise in its use) may serve to refine requirements, assess how well a design meets the requirements, or it may posit a design or design commitments that would satisfy those requirements [17].

2.1 Requirements, system solution, and system concept

Requirements should make reference to the environment in which the system will operate. Requirements are about the environment [30]. The environment includes those components of the original system that remain unchanged (in structure and usage) in the new system. The environment also includes components whose usage is mandated. A *system solution* is an assembly of interacting components forming a whole and serving requirements. A system solution is an instance corresponding to its system concept. The *system concept* is a conceptual entity; it is a characterization, based on the requirements, that holds for anything that satisfies the requirements.

In engineering design, and its formalization and automation, it is important that statements of requirement do not presuppose mechanisms intended to achieve those requirements. An MBE should distinguish viewpoints and problem solvers that make design commitments from those that refine statements of requirements and produce derived requirements without making design commitments. The reasons for this stipulation concerns the invariance of original requirements and the relative tractability of a design process that relies on this fact. The following example illustrate this point.

Suppose that one has defined the original requirements of a new business process. A problem solver with an enterprise view may identify the components that are involved in fulfilling these requirements by identifying the relevant information, who possesses it, and who requires it. This problem solver may posit two derived requirements: (1) a derived requirement stating that the an information flow must occur between the identified components; (2) a derived requirement stating the triggering and temporal ordering of events that lead to the exchange of the information. These new assertions draw on information in the environment (of both the existing and to-be system). Other than to identify roles, the information does not concern the means by which the business process is achieved.

[4] In models that embody commitments to mechanism, *function* is defined as the activity by which something fulfills it purpose. However, in models that do not make commitments to mechanisms (*i.e.* conceptual models), this notion of function cannot be represented. *Utility* – what purpose something serves – is the the corresponding term in conceptual models.

As long as the original requirement remains valid, this information need never be retracted. The assertions made by these problem solvers are part of the system concept.

On the other hand, any of a number of problems might impede the implementation of this information flow (see *section 3*). If the problem is that both components behave as servers, one must take initiative to update the state of the component requiring the information. Potential solutions include: (1) component A, who has the information, triggers a process to communicate with component B, who needs the information; (2) a delegate of component B polls component A for changes in the information, or; (3) both components A and B subscribe to an event service tracking the original production of the information. The choices made here are design commitments, since they concern mechanism, and their quality is contingent on design commitments made elsewhere to address other original requirements. If, for example, the totality of requirements suggests that a new event-based channel of communication is warranted, choice (3) may become increasingly attractive. That is, it may become necessary to retract assertions that are design commitments. The assertions made by these problem solvers are part of a system solution, not the system concept.

2.2 Behavior, function, and role

Systems[5] are made of components, which themselves may be viewed as systems (subsystems to the system). Systems may be described efficiently in terms of their function, but in *ad hoc integration*,[6] when a subsystem is integrated into a system, what is most relevant is not what its function was as it operated in isolation, but rather whether its behavior serves a function. This is so because it is ultimately the behavior of the subsystem that must be controlled and exploited to service the needs of the system. The function of the component as it was in isolation may not be relevant. For example, a text editor may be used to write reports. The editor may serve this function at various points in the processes of an organization. It may be the case that the editor has a regular expression search capability that can be used to to recognize exceptional conditions in data collected from the factory floor. The regular expression search behavior can be harnessed for this purpose. Its report writing function is not relevant here.

The behavior of a component may serve multiple functions, and each instance of its application in a system may serve a different function. The function of the instance in the context of the system is called the *role* of the component. The notion of role cannot be distinguished from the notion of function, except for the fact that a role is defined in the context of the usage of an occurrence of the component in a particular system context.

[5] Unless it is necessary to distinguish the concept of the system from its solution, we will henceforth use the term *system* to refer to a system solution. This is consistent with common usage.

[6] *ad hoc integration* is integration where a communication flow is required between components that were not conceived with the intention of providing or receiving that flow.

3 Impediments to integration

Integration is about enabling components to act jointly toward a goal. In the scope of the model-based environment, the problem solvers that are orchestrated in the systems engineering process work toward enabling joint action. Joint action is impeded by various obstacles. [3] identifies five broad categories of impediments to the integration of software-intensive systems and classifies problems within these. That work is summarized below.

Technical impediments concern problems in communication and process flow arising from the technology and logistics employed at the interfaces of components. These include control conflicts (*e.g.*, every component is designed to be a client, or every component is designed to be a server) and differences in the syntax of messages.

Semantic impediments concern how well information communicated among the components of the system serves the joint action that fulfills the purpose of the system. *Semantics* refers either to a theory of behavior or a theory of reference [25]. Terms have a sense and a reference. Regarding reference, communicating agents may differ with respect to the objects to which a term refers. These differences may be (but are not always) detrimental to joint action. Regarding sense, the behavior that a message elicits from the recipient may be in conflict with what is expected and intended by the speaker.

Information may be conveyed directly to known recipients or published. Information conveyed to known recipients intends to elicit particular behaviors from the recipients and utterances are designed for that specific purpose. Published information is information that should be true in the context of the system. The behavior that published information is intended to elicit is known only by the systems engineer, not the components that provide the information.

Functional impediments concern conflicts arising from a mismatch between actual behavior and the behavior that is expected for a particular role. An agent may perform activities that are beyond those called for in its designated role. The effect of these extraneous activities may be (but are not always) detrimental to joint action.

Qualitative impediments concern how well a component performs the role to which it is tasked. Qualitative impediments include accuracy, security, trust, credibility, and timeliness of results.

Logistical impediments concern the impact of the designed system on the system in which it is embedded. Problems here concern system validation, that is, how well the deployed system, in fact, satisfies requirements. Important requirements may have gone unstated and unfulfilled. Requirements may have changed while the system was being designed.

4 Systems Engineering Processes

A *systems engineering process* prescribes a decomposition of engineering capability into problem-solving agencies, and a means to orchestrate these problem

solvers. A design methodology prescribes a path through the space of potential refinements (of both requirements and specifications). Differing design methodologies can be employed within the individual problem solvers as the means by which they function. The choice of systems engineering process determines how requirements are classified and allocated to problem solvers. In part because different system engineering processes suit different problems, systems engineering has not defined a best practice systems engineering process. Such a thing might not exist [11].

The engineering of complex systems typically follows a top-down and then bottom-up development process, depicted as a *V*, where the left half of the *V* represents the requirements definition and decomposition effort, and the right half of the *V* represents integration and verification [12]. (See *figure 2 (a)*). This basic flow holds generally, including *ad hoc* integration and business process reengineering situations. In more complex systems development, the conceptual architecture itself may be subject to modification during development. This process has been described as a *modified-V* pattern [28]. (See *figure 2 (b)*.) This flow somewhat resembles one in software systems engineering in which one evolutionary path of development (and the "stove-piping" it entailed) is terminated and a reconceptualization is made to simplify broad areas of the design.

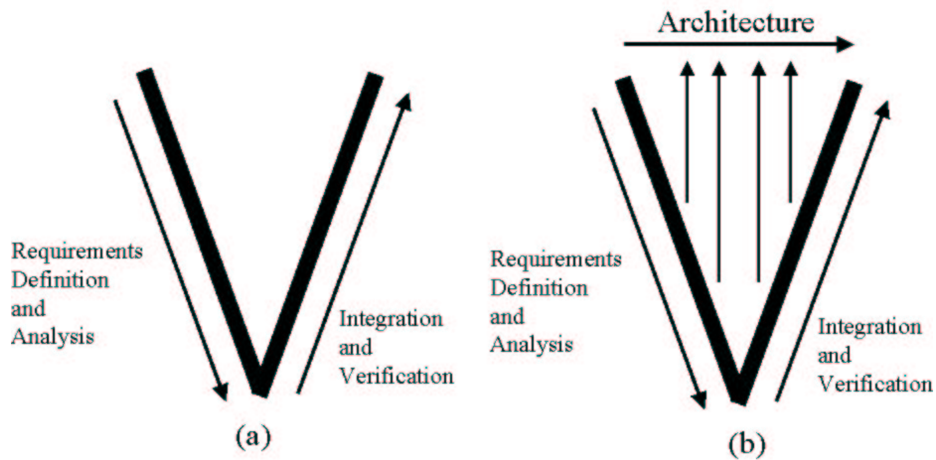

Fig. 2. Two patterns of systems engineering process

We are motivated in this discussion to identify the systems engineering process and problem-solving architecture that best address the nature and requirements of software systems integration in a MBE. These requirements include (1) the facilitation of heterogeneous viewpoints and notations that express the many relevant views of the system; (2) the communication of results among these

viewpoints [30]; (3) modularity that accommodates new viewpoints; (4) simplicity; (5) end-to-end (requirement definition to validation) completeness, and; (6) leverage of our knowledge of the problems of systems integration, including classification of integration impediments.

Of these requirements, (1), and (2), appear to be most difficult to satisfy. Among the many issues here, knowing what information need be communicated among viewpoints and what form that communication should take are problematic. Agencies do not act in total isolation, however the complexity of engineering the entire system requires that details that are inconsequential to the decisions of other agencies remain isolated in the agency from which these details originate [18]. Deciding what information needs to be exposed requires systemic domain knowledge. An analogy from the systems engineering of physically-intensive systems [22] illustrates the problem: Suppose a spacecraft project has a weight budget and a choice of integrated circuit (IC) technology must be made. It may appear that the choice has little to do with weight (all chip weights are approximately equal). However, one IC technology may entail greater power consumption and thus the need for additional solar power collectors and batteries. An analysis of similar problems, termed *nearly decomposable problems*, is provided by Simon [26].

We consider three general processes:

Discipline-centric process: A discipline-centric systems engineering process is one in which engineering capability is organized by engineering discipline (*e.g.*, network analysis, concurrency analysis). That is, engineering capability is organized into problem-solving agencies by classification of their area of expertise. Examples of this approach include the time-tested means by which many physically-intensive systems such as automobiles, aircraft, and spacecraft, are developed. The process is most effective when applied to the development of a class of similar products, where knowledge of the flow of information among agencies (that is, a design procedure) has been well established.

In this process, problem solving follows an established route.

Component class-centric process: A component class-centric systems engineering process is one in which engineering capability is organized by expertise in the development of a class of subsystems (*e.g.*, database or factory floor data collection). This class is related to the discipline-centric process in that disciplines and subsystems tend to correspond. The process is most effective when applied to projects that require the development of new subsystems. The procedure is at a disadvantage in situations where the specification of the interfaces between components are subject to modification.

In this process, problem solving follows a route determined by functional decomposition.

Problem-centric process: A problem-centric systems engineering process is a variation of a discipline-centric process in which engineering capability is organized to address classes of well-known problems. However, here no design procedure has been established. Instead, derived conceptual requirements identify instances of the well-known integration problems. The approach may become

excessively complex if these problems are revealed not by derived conceptual requirements, but only through detailed design commitments.

In this process, problem solving follows a route that emerges only with refinement of requirements and assertion of commitments.

Of the approaches considered, the problem-centric process appears to be most appropriate to the design of an MBE for software systems integration. A discipline-centric process is not responsive to the *ad hoc* emergence of integration tasks, as might be required to resolve a control conflict (see section 3), for example. A component class-centric process may avoid this problem but may be inappropriate where the concerns to be resolved are predominantly impediments to integration not design syntheses. In the use envisaged, enterprise views of the baseline system provide a conceptual model upon which derived conceptual requirements can be formulated. Derived requirements that concern needed information flows, temporal ordering, and timeliness constraints between identified components provide a basis to review the subject components with respect to the impediments to integration (*Section 3*). The demarcation between requirements (invariantly asserted) and design commitments (subject to possible retraction) is the point at which the information flow, its temporal constraints, and a neutral representation of the information to be exchanged[7] are identified and provided to problem solvers corresponding to particular concerns. Design commitments made by problem solvers may introduce new integration concerns.

5 A Model-based environment for software systems integration

This section describes an MBE for software systems integration based on the problem-centric systems engineering process. The approach suggested in the previous section involves multi-disciplinary knowledge, heuristic techniques, and a flexible problem-solving ability.

Success in the use of the approach requires, foremost, a comprehensive collection of information, including enterprise models, information models, and technical models. The scope of information required must be sufficient to reveal obstacles to the implementation of the new business process. Some of the information necessary may exist as artifacts of earlier design efforts. Information such as a mapping of information structures to an ontology of concepts in the business process, imposes an additional cost on systems integration. However, in the system envisaged, the ontology and references between models should provide enduring value; it can evolve with the enterprise system it represents. As described in [8], the information collectively embodies an enterprise model of the subject business processes. The model is an *emergent enterprise model* in the sense that it comes into being through the accretion and interrelation of the

[7] This representation may be, for example, an ontology of concepts relevant to the business process being implemented. Problem solvers may need access to a view that maps these concepts to information structures they expose in interfaces.

various models generated in the course of the development and evolution of the enterprise's infrastructure.

Two other obstacles, these concerning the 'human element' must be addressed: (1) original statements of requirement are typically informally specified, potentially inconsistent or wrong, and; (2) modeling notation (especially graphical ones) may be interpreted in differing and potentially conflicting ways.

With regards to (1), implementation of the approach imposes the additional work of translating the original requirements into a formal notation consistent with concepts in the ontology of the business process. The consequences of (2) are that inferences made from such models are apt to be invalid. The system should recognize certain modes of use (*e.g.*, a UML class diagram used to represent conceptual entities versus a UML class diagram used to implement classes) and make inferences accordingly.

Figure 3 depicts the basic concept of the envisaged system. The design uses a blackboard architecture [5] to orchestrate viewpoint-specific problem solvers. The blackboard tracks the developing system concept and system solutions. Technical problem solvers provide results that are contingent upon assumptions about prevailing designs. The systems engineering executor is responsible for partitioning design commitments by the assumption set on which they are based. An assumption-based truth maintenance system (ATMS) [6], [7] may serve this purpose. Conceptual problem solvers provide the systems engineering executor with derived requirements, which are not assumption-based.

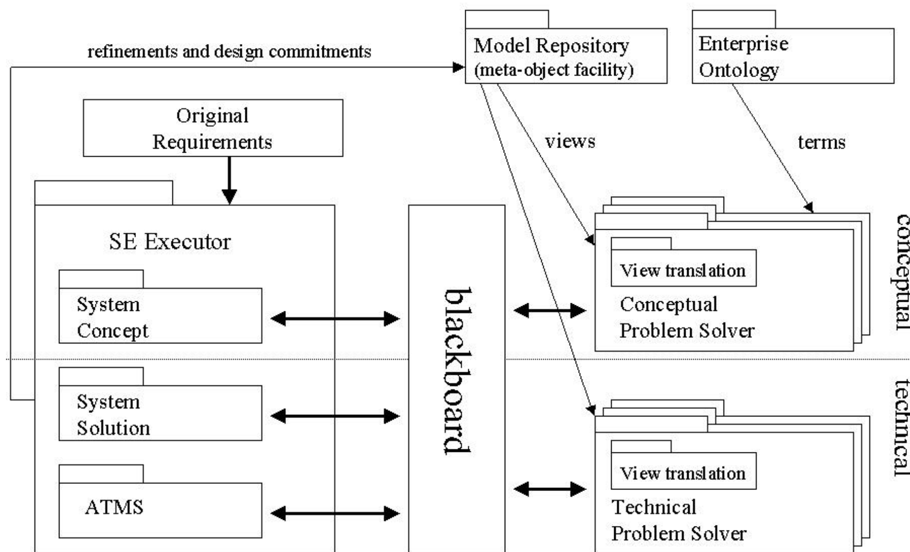

Fig. 3. System design

The nature of the language on the blackboard is a matter of great importance in the design of such a system. The language is the means by which problem solvers communicate with each other. It must be capable of expressing requirements and specifications at varying levels of detail. Further, it must be capable of providing content to the various viewpoints of problem solvers. In designing a language with these capabilities there is a risk of disproportionate reliance on the representation scheme of some given problem solver. To reduce this risk, the systems engineering executor component must operate only on information that is meta-level to the content that motivates the problem solvers. This requires that the language distinguish (1) problem domain content from (2) constructs that concern the structure and modality of the content, and provides directives to and from the executor. The nature of the designed systems engineering process is affected primarily by choice of representation scheme for message structure, modality, and executor directives, whereas the nature of the designed domain problem solving capability is affected primarily by the choice of representation scheme for the content language.

The purpose of the content language on the blackboard is to present problems for solution by the problem solvers. Problem solvers are responsible for translation to and from the blackboard content language to whatever form they find useful. The language concerns requirements, specifications, and the refinement of both. Earlier work in this area has addressed issues of accommodating various notations [16], [31], and refinement of requirements [30], [27]. More recently, interest in the *family of languages* concept of UML [4] has directed attention to providing a semantics and formal foundation to relate a collection of modeling notations.

The blackboard content language of the subject model-based environment, may be patterned after the predicate logic-based representation described by Zave and Jackson [31]. It should accommodate a notion of refinement such as found in the Z specification language [27].

Problem solvers draw on *prescriptive views* (views corresponding to the problem-solving tasks for which they are specialized) as well as *concept links* that resolve terms referenced in requirements to information structures relevant to the views on which they work. Prescriptive views and concept links (views themselves) are retained in a model repository based on the OMG meta-object facility [20]. Terms referenced in requirements correspond to concepts defined in the enterprise ontology. The enterprise ontology could be implemented in a description logic such as Powerloom [24].

Prescriptive views are typically founded on existing viewpoint technology. Although a complete analysis of the relationship between prescriptive views and the integration impediments on which they have bearing is beyond the scope of this paper, the following observations can be made:

- Prescriptive views serving technical impediments include ADLs [1] and behavioral modeling notations such as statecharts [13].
- Prescriptive views serving semantic impediments include database schema, ontologies, and tools that identify conceptual differences in these.

- Prescriptive view serving functional impediments include the functional specification of components, activity diagrams such as IDEF0 [10], statecharts [13], functional flow block diagrams [23], and enterprise models [9].

Finally, the following observations can be made about this design:

- The design does not preclude the possibility that some problem solvers could be human.
- The design does not entail the definition of a formal notation for prescriptive views that lack one (*e.g.*, statecharts). How a problem is resolved through use of the view is a choice left to the problem-solver implementation.
- The systems engineering executor (serving the role of initiator of the blackboard) does not require global knowledge concerning the expertise of problem-solvers.
- The systems engineering executor is not responsible for transformations to problem solver notations.
- Subject system design choices can be traced to requirements.

6 Related work

The Model Driven Architecture (MDA) of the Object Management Group (OMG) describes an approach that may allow a model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mapping to specific platforms. The model, called a *platform independent model* (PIM), provides a specification of the structure and function of a system that abstracts away technical detail. [19].

The MDA is primarily concerned with providing freedom in the selection of middleware technology, and hence addresses a significantly narrower class of problems than the approach suggested by this paper. The MDA approach assumes agreement among communicating components on four of the five integration impediments described in *Section 3* (functional, semantic, qualitative, and logistical), and resolves only a subset of technical impediments.

The MDA PIM provides the viewpoints provided by the Unified Modeling Language. These concern structural and control commitments that can be directly transformed into interface specifications. The PIM does not address most semantic concerns, such as mismatch of scope, granularity of abstraction, and temporal basis [29]. A PIM may provide a UML activity diagram that, like *Z*, does not indicate the agents performing the activities. This can be an advantage when the the matter of who should act as server is a design issue. However, as this paper suggests, the choice can be contingent on a wide spectrum of related concerns.

UML Version 2 (currently under development) intends to provide a common underlying semantics for the viewpoints of UML.[21] This would further the goals of MDA, and provide to this project valuable input towards developing the content language of problem solvers.

ARIES [16], and the work of Zave and Jackson [31], is related to this work in providing environments for representing requirements and refinement amid multiple viewpoints. Those works sought a common underlying semantics for composition of specification across viewpoints. Like UML Version 2, those works may serve to provide a content language for the problem solvers of our work. They differ from what we propose in that we use the blackboard executor to focus attention on the impediments to integration, and use a model repository for accruing inter-model dependencies.

7 Conclusion

In this paper we reviewed ideas from systems engineering so as to identify the full scope of challenges affecting the ability to automate tasks of software systems integration. We identified a systems engineering process based on the resolution of integration impediments as the most efficient towards meeting these challenges. The solution outlined implements the systems engineering process as a blackboard executor mediating problem solvers working from multiple viewpoints.

The resolution of certain integration problems involve design choices, and hence are likely to be addressed only through heuristic, knowledge-based problem solvers. Some semantic impediment, resist solution through automated means. In these situations, it might be possible to create and rely on semantic links between information structures exposed at interfaces and a domain ontology. But this approach is unproven and could prove costly.

Our implementation of the ideas presented in this paper have been limited to the development of a basic meta-object facility, components of a blackboard and investigation of problem-solver technology such as the description logic systems [24]. Work towards the system envisaged is continuing.

A Glossary of the terms from the conceptual model

Behavior - how something acts in response to stimulus

Constraint - an expression derived from a requirement that partitions system solutions into those that meet the requirement and those that do not

Environment - the context in which a system operates

Function - mode of action or activity by which a thing fulfills its purpose

Optimization Criterion - a mathematical expression derived from a requirement that provides an ordering and metric on system solutions indicating how well each solution meets the requirement

Requirement - an optative statement intending to characterize and identify a system solution

Risk - a probabilistic expression that appraises the consequences of not meeting particular requirements

Role - the characteristic function or expected function of a thing in the context of a system solution

Solution Metric - expressions derived from requirements that are used to measure system solutions

System Concept - the concept of an assembly of interacting components forming a whole and serving requirements

System Solution - an assembly of interacting components designed to meet requirements

Trace - an account of the relationship between a requirement and a design decision

Validation - the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model [2]

Verification - the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model [2]

View - a representation of a whole system from the perspective of a related set of requirements (derived from the definition of the term in [14])

Viewpoint - methods founded on the body of knowledge of some engineering or analytical discipline and used in constructing a view. *N.B.*, viewpoints establish the purpose and audience of a view.

References

1. Allen, R., Garlan, D., "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology.* July, 1997.
2. American Institute of Aeronautics and Astronautics, *AIAA Guide for the Verification and Validation of Computational Fluid Dynamics Simulations (G-077-1998)*, AIAA Standards Series, 1998.
3. Barkmeyer, E. J., (Editor). *Concepts for Automating Systems Integration*, NIST Interagency Report, National Institute of Standards and Technology, Gaithersburg, Maryland, To be published.
4. Cook, S., "The UML Family: Profiles, Prefaces and Packages" *UML 2000 — The Unified Modeling Language: Advancing the Standard*, Proceedings of the Third International Conference, York, UK, Springer Lecture Notes in Computer Science, Vol 1939, October 2000.
5. Craig, I., D., *Formal Specification of Advanced AI Architectures*, Ellis Horwood Limited, Chichester, West Sussex, 1991.
6. de Kleer, J., "An Assumption-Based TMS," *Artificial Intelligence* 28(2): 127-162, 1986.
7. de Kleer, J., "A Perspective on Assumption-Based Truth Maintenance," *Artificial Intelligence* 59(1-2): 63-67, 1993.
8. Denno, P., Flater, D., Gruninger, M., "Modeling Technology for a Model-Intensive Enterprise," Proceedings of SSGRR-2001, *Infrastructure for e-Business, e-Education, e-Science, and e-Medicine*, Scuola Superiore G. Reiss Romoli, L'Aquila, Italy, July, 2001.
9. Esprit Consortium AMICE (editors), *CIMOSA: Open System Architecture for CIM*, 2nd revised and extended edition, Springer-Verlag, Berlin, 1993.

10. Federal Information Processing Standards, *Integration definition for function modeling (IDEF0)*, National Institute of Standards and Technology, Gaithersburg, Maryland, 1993.
11. Gabb, A., "Requirements Categorization," *Requirements Working Group of the International Council on Systems Engineering (INCOSE)*, 2002.
12. Grady, J., O., *System Integration*, CRC Press, Boca Raton, Florida, 1994.
13. Harel, D., "Statecharts: a visual formalism for complex systems". *Science of Computer Programming* 8, 3, June 1987.
14. IEEE 1471, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems.* Institute of Electrical and Electronics Engineers, Inc., September, 2000.
15. ISO/IEC IS 10746, ITU-T X.900, *Open Distributed Processing – Reference Model*, International Organization for Standards, 1996.
16. Johnson, W. L., Feather, M. S., and Harris D. R., "Representation and Presentation of Requirements Knowledge," *IEEE Transactions on Software Engineering*, Vol 18, No. 10. October, 1992.
17. Mark, W., et al., "Commitment-Based Software Development," *IEEE Transactions on Software Engineering*, Vol 18, No. 10. October, 1992.
18. Minsky, M., *The Society of Mind*, Simon & Schuster, New York, New York, 1985.
19. Object Management Group, *Model Driven Architecture (MDA)*, http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, July 1, 2001.
20. Object Management Group, *Meta Object Facility (MOF) Specification*, Version 1.3: ftp://ftp.omg.org/pub/docs/ formal/00-04-03 March, 2000.
21. Object Management Group, *Request For Proposal: UML 2.0 Infrastructure*, ftp://ftp.omg.org/pub/docs/ad/2000-09-01, September, 2000.
22. Oliver, D. O., *Personal communications.*
23. Oliver, D. O., Kelliher, T. P., and Keegan, G. J., *Engineering Complex Systems With Models and Objects*, McGraw Hill Text, 1997.
24. Preece, A., et al. "Better Knowledge Management through Knowledge Engineering," *IEEE Intelligent Systems* 16:1, Jan-Feb, 2001.
25. Quine, W. V., *From a Logical Point of View*, second edition, Harvard University Press, Cambridge Massachusetts, 1980.
26. Simon, H. A., *The Sciences of the Artificial*, Third Edition, The MIT Press, Cambridge, Massachusetts, 1996.
27. Spivey, J. M., *The Z Notation: A Reference Manual*, Prentice-Hall, London, 1989.
28. Thomas, L. D., "System Engineering the International Space Station," NASA Langley Research Center, *International Space Station Video Conference '97*, 1997.
29. Wiederhold, G., "Mediators in the Architecture of Future Information Systems," *IEEE Computer Magazine*, March, 1992.
30. Zave, P., Jackson, M.: "Four Dark Corners in Requirements Engineering," *ACM Transactions on Software Engineering and Methodology*, Vol 6, No. 1, January 1997.
31. Zave, P., Jackson, M.: "Conjunction as Composition," *ACM Transactions on Software Engineering and Methodology*, Vol 2, No. 4, October, 1993.