2002-11

Approach for Highly Dependable Software-Intensive Systems

Luqi; Liang, X.; Luqi; Liang, X.

Naval Postgraduate School

Luqi and X. Liang, "Approach for Highly Dependable Software-Intensive Systems", Technical Report, NPS-SW-02-012, November 2002.
https://hdl.handle.net/10945/65205

NPS-SW-02-012

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Perspective-based Approach for Highly Dependable Software-Intensive Systems

Technical Report

By

Luqi, XianZhong Liang

November 2002

Prepared for: NSF

PROJECT SUMMARY

The objectives of the proposal are creating new methods and tools for effective constructing software-intensive systems based on multiple perspectives that are used to reflect differing stakeholder's concerns. The models and methods will incorporate rapid prototyping, explicit architecting and consistent engineering techniques into a synthesis approach for highly dependable software-intensive systems (HDSIS).

To improve the dependability of software-intensive systems as well as their affordable flexibility, and increase the effectiveness of requirements validation techniques for HDSIS, this proposed research addresses two aspects to dependability: getting the requirements right, and getting the realization of the system to meet the requirements. Integration of the proposed research with requirements validation techniques such as rapid prototyping, and integration of differing stakeholder perspectives and concerns into the models and methods to be used addresses this dominant dependability issue.

To put high confidence on a sound objective basis, via a systematic method for expressing dependability objectives via measurable localized constraints associated with the subsystems of the architecture, the proposed research addresses whether the system meets the requirements by enabling the degree of confidence to be put on a quantitative statistical basis that is rooted in measurements, and to factor assurance processes into independent tasks associated with each subsystem to the extent possible, thus simplifying analysis and reducing computational complexity.

To reduce the amount of re-certification effort required after each requirement change that stays within the envelope of some invariants, this proposed research is trying to maintain the assurance of dependability as the system evolves. Successful systems are in a constant state of change. This is particularly challenging for high confidence systems. We address this problem by investigating principles that enable parts of the assurance to be based on properties of the architecture that are invariant with respect to system requirements changes.

There are three main barriers to the synthesis approach for HDSIS: perspective model, constraint localization and software tool support. Modeling a system with multiple perspectives needs to specify, from the viewpoint of different stakeholders, which is crucial in the successful development and evolution of DHSIS. It is impractical to characterize all of stakeholder' s concerns with a single model, while multiple models to characterize specific concerns relative to specific stakeholders may not be compatible. Constraints localization represents how to translate dependability, its translation to quantitative constraints and applying them in the design, construction, deployment and evolution of the system, which is the problem we have to face. Intellectual models and methods should be designed for manipulable automation by CASE tools. To be truly usable and useful, the real challenge is to develop the sophisticated software tools that are needed for automation support.

The proposed research will explore the following:

1) Conceptual Framework for constructing HDSIS
2) Rapid modeling system via multiple perspectives.
3) Explicit architecting system via compositional patterns.
4) Derivational Evolving system via generalized framework.
5) Quantitative formulating dependability as localized constraints.
6) Automatic synthesizing approach by CASE tool support

The key areas of research are in Software Engineering, Rapid Prototyping, Software Architecture and Component-based Development. To be highly dependable, the development of software-intensive systems will also involve accompanied research areas: design inspection. Design inspection provides dominating methods for detecting errors in software systems. Numeric questions must be answered in each of these areas in order to evaluate this approach. However, if these questions can be answered sufficiently, then the overall impact on software development could be huge. Improving the dependability of software-intensive systems as well as their affordable flexibility is crucial. Creating a set of compositional patterns that support explicit architecting throughout the whole procedure of the synthesis approach. By translating macro dependability into quantitative constraints, while localizing them on compositional patterns, the dependability of software-intensive systems as well as their affordable flexibility should are dramatically improved.

This research could have broad impact on the area of Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) Architectures to support acquisition of system that will meet the needs of military coalition. The use of architectures is to address increased uncertainty about requirements, rapid changes in technology, changes in organizational structures, and a widening of spectrum of mission and operations. Today, military organizations must respond to a variety of situations by quickly assembling and organizing coalitions from different components. Compositional patterns are used to promote heterogeneous interactions among components, govern system composition and localized constraints on the architecture. The perspectives of an architecture are smoothly transformational via compositional patterns, they work with well-grained components, heterogeneous interaction and hierarchical composition and flexible configuration. In the synthesis approach for HDSIS, we have tools for rapid prototyping, explicit architecting, and consistent engineering the software-intensive systems.

PROJECT DESCRIPTION

We propose research at Software Engineering Automation Center (SEAC), Naval Postgraduate School to create a new method and tools for effective constructing software-intensive systems based on based on multiple perspectives that are used to reflect differing stakeholder's concerns. The models and methods will incorporate rapid prototyping, explicit architecting and consistent engineering techniques into a synthesis approach for highly dependable software-intensive systems (HDSIS).

Software-intensive systems are those complex systems where software contributes essential influences to the design, construction, deployment and evolution of the system as a whole. Generally, they are characterized as distributed, heterogeneous, network-based and time-critical systems [44, 45].

C.1 Objectives and Significance

The main objective of the proposed work is to enable a perspective-based transformation process for HDSIS based on explicit system architecting. Because of increased uncertainty about requirements, flexible configuration in organizational structures, and rapid application development, HDSIS inevitably involve different kinds of stakeholders such as customer, architect and implementer. A collection of products is needed to document different stakeholder's concerns in developing the HDSIS. We propose to develop a synthesis approach embodied in computational prototyping, compositional architecting and transformational derivation. Explicitly architecting a system aims to attain the benefits of reduced costs and increased quality, because an architecture of the HDSIS can effectively bridge the gap between evolving software requirements and system implementations.

We also seek to develop methods for explicit architecting systems, which is recognized as a critical element in the successful development and evolution of software-intensive systems, for rigorously formulating dependability into quantitative constraints that can be associated with the compositional patterns, and for automatic generation of a well-designed architectural framework (generic templates) to realize dependable construction, convenient composition, rapid deployment and flexible evolution of software-intensive systems.

Proposed new methodologies and automated tools will incorporate rapid prototyping, explicit architecting and consistent engineering techniques into highly automated, easily applied CASE tools. There are five research thrusts in the proposed research:

- *Conceptual Framework for constructing HDSIS*
- *Rapid modeling system via multiple perspectives.*
- *Explicit architecting system via compositional patterns.*
- *Derivational Evolving system via generalized framework.*
- *Quantitative formulating dependability as localized constraints.*
- *Automatic synthesizing approach by CASE tool support*

C.1.1 Significance and Benefits

The results of this research should improve the dependability of software-intensive systems as well as their affordable flexibility.

There are two aspects to dependability: getting the requirements right, and getting the realization of the system to meet the requirements. Software engineering research has shown that the majority of software faults are due to requirements and specification errors, and that this proportion is particularly high for novel systems. The integration of the proposed research with requirements validation techniques such as prototyping and the integration of differing stakeholder perspectives and concerns into the models and methods to be used addresses this dominant dependability issue. The proposed approaches and models should increase the effectiveness of requirements validation techniques for HDSIS.

High confidence is a perception regarding dependability, at the current time a vague and informal concept. This research seeks to put high confidence on a sound objective basis, via a systematic method for expressing dependability objectives via measurable localized constraints associated with the subsystems of the architecture. This aspect of the research addresses whether the system meets the requirements, and should enable the degree of confidence to be put on a quantitative statistical basis that is rooted in measurements, and to factor assurance processes into independent tasks associated with each subsystem to the extent possible, thus simplifying analysis and reducing computational complexity.

Successful systems are in a constant state of change. This is particularly challenging for high confidence systems, because the assurance of dependability must be maintained as the system evolves. We address this problem by investigating principles that enable parts of the assurance to be based on properties of the architecture that are invariant with respect to system requirements changes. The proposed approach should reduce the amount of re-certification effort required after each requirement change that stays within the envelope of these invariants.

C.1.2 Technical Barriers

In order to develop the synthesis approach for HDSIS, following technical problems should be thoroughly studied and then solved

- *Perspective model problem.* Modeling a system with multiple perspectives needs to specify, from the viewpoint of different stakeholders, what stakeholder's concerns are and how they are formulated as key aspects in the successful development and evolution of software-intensive systems. Since stakeholder's concerns are not only often diversified but also sometimes contradictive. It is impractical to characterize all of stakeholder' s concerns with a single model, while multiple models to characterize specific concerns relative to specific stakeholders may not be compatible. It is crucial to find key factors that enable a transformable process between perspectives. The process should be able to "zoom in" the commonness and "zoom out" or identify the difference among stakeholders' concerns.
- *Constraint localization problem.* Constraints localization represents translation of dependability to quantitative constraints, and applying these constraints in design inspection. The dependability of a system is global qualitative requirements that are abstracted as availability, reliability, safety, security, integrity, maintainability, and so on. They need translating into semantic quantitative constraints during system development. What constraints are needed to quantitatively attain the benefits of largely reduced costs and highly increased dependability and how they can be applied (localized) in the design, construction, deployment and evolution of the system is the key problem that has to be faced.
- *Software tool support problem.* Intellectual models should be designed for manipulable automation by CASE tools. The associated engineering tasks, such as rigorously reasoning about desired properties, mechanically translating formulas from each other and formally applying constraints, cannot be done very well manually because of cost and excessive human error rates. Formal models should be represented as semantic formulas so that they fit with reasoning and manipulation by CASE tools. To be truly usable and useful, the real challenge is to develop the sophisticated software tools that are needed for automation support.

C.2 Technical Approach

In order to enable effective constructing HDSIS, we propose three fundamental approaches: computational model for rapid prototyping, compositional patterns for explicitly architecting, and optimized object model for component object evolution, all of which are synthesized to the perspective-based approach. Based on compositional patterns, the synthesis approach calls for explicit treatment of software composition and architecting that involves (1) **three perspectives of architecture** characterized as computational activity, compositional architecture, and derivational transformation, (2) **two mappings** between three perspectives, embodied in explicitly architecting the system via compositional patterns and physically deriving the component from architectural role wrappers, (3) **a set of formal description** that is used to model well-grained components, heterogeneous / hierarchical composition, and flexible configuration, and (4) **the associated tool support** that makes the synthesis approach more applicable in design, analysis, evolution and executable system generation.

C.2.1 Multiple Perspectives for Software-Intensive Systems

Highly dependable software-intensive systems inevitably involve multiple stakeholders (e.g., customer, architect and implementer of the system) and their concerns (e.g., computational, compositional and derivational aspects). Associated with a set of concerns, a perspective will provide both conventional specification (for constructing and using the perspective of stakeholders) and patterns or templates from which to develop individual perspectives by establishing the techniques for its creation and analysis. Such a system should have an architecture and this can be concretely described as a collection of products to document the architecture.

Modeling a system via three perspectives starts with the formulation of customers's informal needs that, in the context of software-intensive systems, are usually a strategic operational concept. An operational concept is created to describe how undertaking collaborative missions will be carried out. A prototype model (in a prototyping language such as PSDL [1,2]) represents computational activity from which a compositional architecture can be derived. Three perspective models, embodied as computational activity, compositional architecture and derivational transformation, should provide a computer-aided foundation abstracted as significant attributes suitable for automated analyzing, reasoning and framework/code generating. Figure 1 illustrates a high level depiction of the synthesis approach for HDSIS development.

In the intersection of the prototype constructing and system architecting, there exists a centered formalism collecting vertical properties (prototyping) and horizontal properties (architecting) into a generated framework. They both are translated as quantitative constraints that can be used for steering design inspection (e.g., model checking [34-37]) and monitoring component derivation (e.g., run-time monitoring [42,43]). These formal verification techniques are embodied in the checking and analysis processes that ensures the software conforms to its specification and meets the needs of the customers who are paying for that software [37,39].

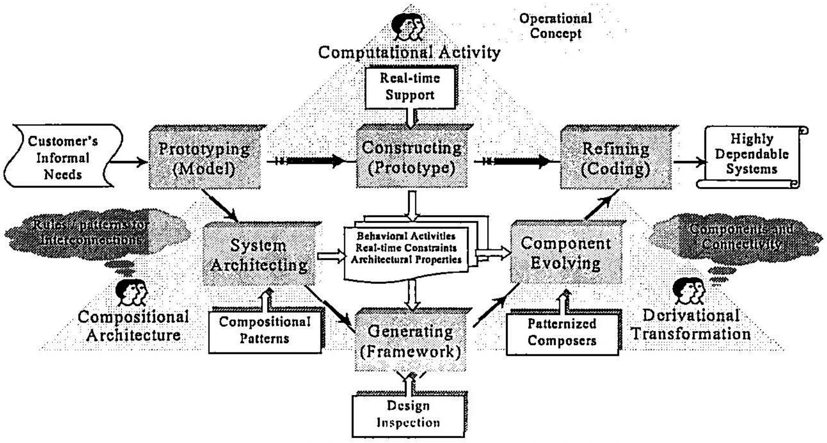


Fig. 1 Synthesis Approach for HDSIS

C.2.1.1 Conceptual Framework

In general, a system is a collection of components organized to accomplish a specific mission or set of missions. A system inhabits an environment, while the environment can influence that system by determining the settings and circumstances of developmental, operational, political, and other influences upon that system. A system has an architecture that provides the fundamental organization of the system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution. Fig.2 illustrates the conceptual framework that is used to support explicitly architecting software-intensive systems with multiple perspectives.

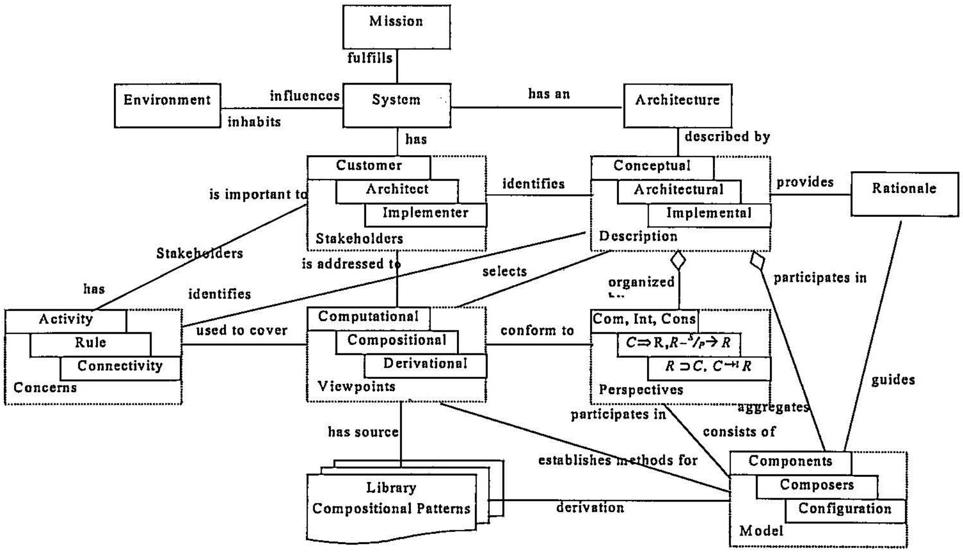


Fig.2 Conceptual framework supporting explicitly architecting with multiple perspectives

In this research, a system has three kinds of stakeholders who have specific concerns relative to the system. Involving system stakeholders, life cycle and uses of architectural description, the system architecture will provide multiple documents to reflect multiple stakeholder's concerns. From viewpoint of customer, *computational perspective* conforms to requirement acquisition From viewpoint of architect, *compositional perspective* conforms to explicit architecting that covers architect's concerns, so that the computational activity can be accomplished. From the viewpoint of implementer, *derivational transformation* conforms to system implementation that covers implementer's concerns under the support of a specific architecture.

The architectural description selects three viewpoints for use. This choice depends on the concerns of three stakeholders, all of which need to be addressed by the architectural description. Similarly, Joint Technical Architecture framework (e.g., JTA [45]) has selected three such viewpoints, but IEEE-std-1471-2000 [44] does not prescribe any particular viewpoints. A viewpoint may be defined with the architectural description, but it may also be defined externally and only used in the architectural description. Such externally defined viewpoints are termed library of compositional patterns.

C.2.1.2 Architectural Perspectives

Computational Perspective. Computational activity provides the perspective for customer whose main concern is **computation** and **interconnection** with qualitative dependable requirements. This perspective outlines the activities and information flows that will accomplish the operational concept (e.g., a computer-aided prototyping system such as CAPS [1]), that is, what activities are needed and how their interactions are associated with workflows, networking and plans necessary to support customer' s operations [45]. Computational perspective addresses system requirements (computation and interconnection) and the associated macro dependability (translated into constraints) by capturing three kinds of formal arguments: well-grained *components* from which the system is built, *interconnections* enforcing interactions among components, and *constraints* on both components and interconnections. This perspective can be represented as follows

$$P_{\text{computation}} = [COM_c, INT, Cons\ (COM_c, INT)]$$

Where COM_c is the set of conceptual components, INT is the set of interconnections among components, $Cons\ (COM_c, INT)$ is the set of constraints localized on components and their interconnections, respectively.

Conceptual components should take responsibility for carrying out the activities to accomplish mission-collaborated operational concept. By the way, granularity is one of the importance features that will be considered as a kind of constraints localized on the components, which are crucial factors for constructing complex systems. Well-grained components are helpful not only to increase productivity but also to improve the understandability, reliability and maintainability.

Compositional Perspective. Compositional architecture provides the perspective for architect whose main concern is explicit treatment of **system composition** and **architecting** with constraints localized on the patterns. This perspective details what kinds of rules (patterns) are used to govern the interactions among components and how quantitative constraints are associated with the patterns (e.g., JTA framework, software architecture and architecture description languages [10,12,18,22,44,45]), so that the operational concept can be accomplished. Compositional perspective addresses what kinds of interactions are applied among components and how to translate macro dependability into quantitative constraints and how to partition them and how to associated them with the patterns by generalizing three kinds of formal arguments: interactive *roles* components are assigned to play in the architecture, architectural *styles* for specifying interactions between specific roles, communication *protocols* for transporting information during interaction. This perspective can be represented as follows

$$P_{\text{composition}} = [COM_c \Rightarrow R, R_o \xrightarrow{S/P} R_i, Cons\ (R, S, P)]$$

Where $\boldsymbol{COM_c \Rightarrow R}$ is the set of roles abstracted from conceptual components, $R_o \xrightarrow{S/P} R_i$ is the set of compositional patterns: R_o interacts with R_i via architectural styles $\boldsymbol{S}$ while complying with communication protocols $\boldsymbol{P}$, $\boldsymbol{Cons(R, S, P)}$ is the set of constraints localized on roles, styles, and protocols, respectively.

The interactive roles play an important part in compositional patterns that are used to explicitly architecting a system [7,21]. In architectural description, a role is abstracted as generalized role wrapper (GRW), a kind of abstract, generalized class in object-oriented philosophy [29,31,33,38]. The GRW of a role will reside in an architectural element known as patterned composer that promotes an interaction between two roles. Generalized role wrappers provide adherence to restricted, plug-compatible interfaces for both interaction and computation. The interfaces for interaction will be implemented as generalized procedures and the interfaces for computation are to be refined or overridden by the derivation of components that are assigned to play the role. In this way, compositional patterns provide a good level of abstraction not only for steering design inspection by localizing constraints on the patterns, but also for monitoring component derivation at run time.

Derivational Perspective. Derivational transformation provides the perspective for implementer whose main concern is **derivation** of components and their **connectivity.** This perspective describes physical components and their connectivity that will be instantiated to carry out the activities of the computational perspective and how to derive those components from the architectural environment (e.g., J2EE, COM+, CORBA [23-28]). Component derivation is embodied through subtyping and refinement. Derivational perspective addresses what kinds of physical components are needed to carry out computational activity and how to glue them to specific roles that provide adhere to different implementation constraints by capturing three kinds of formal arguments: physical *components* that derived from the assigned roles, connective *glue* by which that instantiated components are exactly attached to the roles, and *constraints* embodied in components, their styles and protocols. Its formula is given as follows

$$P_{\text{derivation}} = [R \supset COM_p, (C_p \mathrel{\rightarrow\!\mid} R_o) \xrightarrow{S/P} (R_i \mathrel{\mid\!\leftarrow} C_p), Cons\ (COM_p, S, P)]$$

Where $\boldsymbol{R \supset COM_p}$ is the set of physical components that are derived from the associated role, $\boldsymbol{C_p \mathrel{\rightarrow\!\mid} R}$ (its peer R $\mathrel{\mid\!\leftarrow} C_p$) is the set of instantiated components that are physically glued to associated roles, $\boldsymbol{Cons(COM_p, S, P)}$ is the set of constraints localized on physical components, styles, and protocols, respectively.

A set of compositional patterns relative to patternized composers acts as a bridge between computational activity (software requirement) and derivational transformation (system implementation) because they offer a means for evolving the operational concept or deriving (refining) physical components.

According to the conceptual framework in this proposal, library of compositional patterns provides a set of patterned composers that are always tangible from explicitly architecting a system to physical deriving component. Explicit composer in implementation is the expectation that software architecture separates interaction (connector) from computation (component). However, the current level of understanding and support for connectors has been insufficient, so that connectors are often considered to be explicit at the level of architecture, but intangible in the system implementation [17, 46]

Formulating dependability as localized constraints. The intersection of the prototype constructing and system architecting in Fig.1 becomes formulating loci of features against different perspectives. All of these features are focusing on the three aspects: computation, composition and derivation. *Granularity* comes towards taxonomy of computation that is embodied in computational responsibility and systematic decomposability, *heterogeneity* towards taxonomy of composition that is embodied in compositional coupling and architectural compositionality, and *transformationality* towards taxonomy of derivation that is embodied in derivation glue and componential evolvability. Fig.3 illustrates characterized perspectives associated with computational activity, compositional architecture and derivational transformation.

☞ COMPUTATIONAL PERSPECTIVE			
Well-grained Component	**Computaitonal Responsibility**	**Desired Property**	**Systematic Decomposability**
Computer Software Complex System (CSCS)	Collaboration	• Dependentability • Interoperability • Dynamic configuration	CSCS is the top-level component that undertakes global activity characterized as distributed and concurrent collaboration.
Computer Software Configuration Item (CSCI)	Mission	• Autonomous • Real-time • Platform-independent	CSCI is the 1st level component that undertakes specific mission, a part of composed top-level collaboration (CSCS)
Computer Software Common Component (CSCC)	Function	• Loosely coupled • Replacebility • Language-independent	CSCC is the 2nd level component that undertakes specific function, a part of composed 1st level mission (CSCI).
Computer Software Computing Unit (CSCU)	Task	• Well packaged • Tightly cohered • Evolvability	CSCU is the 3rd level component that undertakes specific task, a part of composed 2nd level function (CSCC).
☞ COMPOSITIONAL PERSPECTIVE			
Heterogeneous Interaction	**Compositional Coupling**	**Desired Property**	**Architectural Compositionality**
Interoperably-distributed Interaction (IDI)	Distributed	• Compatibility of roles • Consistency between roles, style, and protocol • Real-time constraints, such as maximum execution times (MET) on the assigned roles • Latency on tprotocol • Heterogeneous data, e.g., dataflow, event, knowledge, message	IDI is used for composing CSCS from CSCI components to enforce distributed interactive collaboration (maybe cross-platform),
Loosely-coupled Interaction (LCI)	Loose		LCI is used for composing CSCI from CSCC components to encourage flexible configuration with minimal communication between components (maybe cross-languages),
Tighlty-cohered Interaction (TCI)	Coherent		TCI is used for composing CSCC from CSCU components to emphasize independent partition of components, with low external and high internal complexity.
☞ DERIVATIONAL PERSPECTIVE			
Transformational Connectivity	**Derivational Glue**	**Desired Property**	**Componential Evolvability**
Inheritance	Extension	• Polymorphism • Dynamic binding • Incremental Evolution	it allows a system component to be derived from the corresponding role wrapper and then to extend its behavioral computation.
Association	Import	• Organic library • Simplicity • Information hiding	it allows a system component to be associated with the correpondent role wrapper and then to refine its behavioral computation.
Aggregation	Assembly	• Heterogeneity • Multiple roles • High-cohesion	it allows a system component to aggregate one more one more the correpondent role wrapper and then to refine its behavioral computation.
Configuration	Dynamism	• Replication • Insertion & Removal • Reconnection	Configurable glue means how the component is glued onto the specific role and this exhibits dynamism by allowing replication, insertion, removal, and reconnection of architectureal elements

Fig. 3 Perspective-based Characteristics

With respect with constraints localization, the characteristics in Fig. 3 can be considered as embodiments of dependability, for instance, integrity and maintainability could be translated into granularity, heterogeneity and transformationality [13,14], which will be associated with computation, composition and derivation, respectively. Furthermore, a CSCI (seen as a subsystem) can be composed from CSCC via LCI patterns, while a CSCS from CSCI via IDI, which could largely improve the maintainability and guarantee the integrity of the given system.

C.2.1.3 Automated Synthesizing approach

Starting with rapid prototyping (a computational model [1,6,8]), the synthesis approach is embodied as transformational process from computational activity, through compositional architecture, to derivational transformation. In next section, formal description for modeling architectural elements will be given that supports both component and composer evolution through subtyping and refinement. Under the support of automated software tools to be developed, two key mappings are applied to bridge the gaps between perspectives: explicitly architecting between computational activity and compositional architecture, physically transforming between compositional architecture and derivational transformation. Fig. 4 illustrates the transformational process among three perspectives.

$P_{\text{computation}}$	Architecting	$P_{\text{composition}}$	Transforming	$P_{\text{derivation}}$
COM_c	Explicit Architecting via Compositional Patterns	$COM_c \Rightarrow R$	Physical transforming via Patternized Composers	$R \supset COM_p$
INT		$R_o \xrightarrow{S/P} R_i$		$(C_p \mapsto R) \xrightarrow{S/P} (R \mapsfrom C_p)$
$Cons(COM, INT)$		$Cons\ (R, S, P)$		$Cons\ (COM_p, S, P)$

Fig. 4 Transformational process

Explicit architecting the computational activity starts with that components are assigned as specific roles. According to the characteristics of roles, the related architectural styles and communication protocols can be determined so that suitble compositional patterns can be applied to govern those interconnections among components. Physical transforming components and establishing their connectivity result from that chosen compositional patterns are mapped into patternized composers (architectural elements). Accoding to assignment that components play specific roles, the components will be derived from the associated role facility. After being derived, the components will be instantiated and then glued to the associated roles [32].

C.2.2 Synthesis of Architecting from Compositional Patterns

Architecture is recognized as a critical element in the successful development and evolution of software-intensive systems [44]. Generally, the architecture is the fundamental organization of a system that is embodied in its *components*, their relationships (*interactions*) to each other and to the environment and the principles (*patterns*) guiding its design and evolution. Precisely, explicit architecting involves interactions among interactive roles (components are specified to play) via architectural styles (specifying the interaction) while complying with communication protocols (specifying the information transportation).

C.2.2.1 Formulating Compositional Patterns

Compositional patterns provide a set of rules for governing the interactions among components and the constraints on them. They are associated with three kinds of formulated factors: interactive *roles*, architectural *styles*, and communication *protocols*. In previous work [7,21,22], a compositional pattern is designed as patternized composer, a kind of architectural element, which is used to promote heterogeneous interactions between components.

Example of Compositional Pattern. Fig. 5 shows an example of a compositional pattern: for a given interaction between two components, these two components are assigned to play specific roles r_1 and r_2, architectural style s specifies how r_1 interacts with r_2, while communication protocol p builds specific transporting channel for data, event or message during the interaction. In order to construct the components as autonomous entities, roles in the compositional pattern are deputized for the components in dealing with interaction while the components are only concerned with their functional activities (separating computation from interaction). The pattern also provide ameans for gluing specific component to the role.

Compositional Patterns. Formal compositional patterns via rigorous mathematics will provide a means for property reasoning and analyzing, and automated manipulation by CASE tool. Supposing that there are three sets: ***R*** {interactive roles}, ***S*** {architectural styles}, and ***P*** {communicative protocols}

R = {
Caller, Definer,
Announcer, Listener,
Outflow, Inflow,
Source, Repository,…
}

S = {
Explicit-invocation,
Implicit-invocation,
Pipeline,
Rep-knowledge,…
}

P = {
Message-passing,
Access-memory,
Dataflow-stream,
Sampled-stream,
}

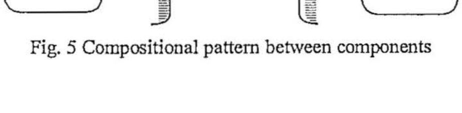


Fig. 5 Compositional pattern between components

Without considering any constraint, a composition is defined as an interaction between two roles (e.g., *Caller* and *Definer*) via an architectural style (e.g., *explicit-invocation*), while complying with a communicative protocol (e.g., *message-passing*), so the Cartesian product $\boldsymbol{R} \times \boldsymbol{S} \times \boldsymbol{P} \times \boldsymbol{R}$ enumerates all possible compositions $\boldsymbol{C}$

$$C(\boldsymbol{R},\boldsymbol{S}\ ,\boldsymbol{P}) = \{\ r_o \xrightarrow{s/_p} r_i \mid \mathrm{r_o}, \mathrm{r_i}\ \in\ \boldsymbol{R}, s \in\ \boldsymbol{S}, p \in\ \boldsymbol{P}\ \}$$

Where $r_o \xrightarrow{s/_p} r_i$ represents interaction between r_o and r_i via a style s while complying a protocol p.

Compositional patterns refer to the constraints, that are applied on the compositions so that sophisticated patterns are formed. For instance, architectural style *Pipeline* can only applied to roles (*Outflow* and *Inflow*). In architecture design, a role is formally abstracted as a generalized role wrapper (GRW), a kind of generic and abstract class in object-oriented philosophy. A GRW provides adherence to restricted, plug-compatible interfaces for interaction and template of behavior for computation, while components derived from the GRW will specify, refine or override the template. In this way, interactions are separated from computations (performed by components). So compositional patterns *CP* are the relation on Cartesian product of compositions with the constraints reasonably localized on roles, styles and protocol, respectively

$$CP(R, S, P) = \{\mathrm{GRW}(r_o) \xrightarrow{s/_p} \mathrm{GRW}(r_i) \mid r_o, r_i \in\ R, s \in\ S, p \in\ P, \mathrm{Cons}(\ r_o, s,p\ , r_i)\ \}$$

Where $GRW(r)$ abstracts role r as generalized role wrapper that separates interaction (GRW provides) from computation (assigned component refines), $\xrightarrow{s/_p}$ represents interaction between r_o and r_i via a specific style s while complying a specific protocol p,$Co\ ns(r_o, s, p,r\ _i)$ represents constraints localized on concrete factors.

Constraints Localized on Patterns. The term "localization" represents abstraction of dependability, its translation to quantitative constraints, and handling these constraints applied (localized) in the design, construction, deployment and evolution of HDSIS. The dependability of HDSIS reflects the quantitative global requirements and should be applied as quantitative constraints in architecture design. In macro view, the dependability is abstracted as availability, reliability, safety, confidentiality, integrity and maintainability. How to translate qualitative global requirements into quantitative constraints becomes the key. HDSIS involve important aspects of architecture, such as *well-grained components* from which the system is built, heterogeneous *interactions* among them, *compositional patterns* to guide their composition, and *quantitative constraints* on these patterns [9,10]. There are two points that are considered crucial: what dependable properties need transforming to quantitative constraints, and how they are localized on compositional patterns. In this research, the transformation from dependable properties to quantitative constraints, and localization of the constrains on compositional patterns can be embodied in the following framework shown in Fig. 6

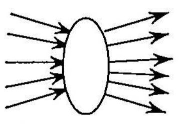
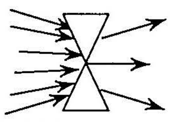
Dependability	Translation	Constraints	Localization	Patterns
• Availability / usability • Reliability/Confidentiality • Safety / Security • Integrity / maintainability • Flexibility/interoperability		• Consistency • Compatibility • Granularity • Heterogeneity • Real time • Synchronization		• Role • Style • Protocol

Fig. 6 Framework of dependability translation and constraints localization

With respect to translating dependability and localizing semantic constraints on the patterns, handling of real-time constraints indicates an easily understood example. Reliability of the time-critical system may be embodied as an immediate reply of a discriminant component, under a given request, within *maximum execution time* (MET), or data stream communication between components within *latency* (LATENCY) [4,8]. First of all, this time-critical reliability should be translated into timing constraints maximum execution times MET and LATENCY, two quantitative constraints. Both MET and LATENCY can be associated with the patterns referring to the role and protocol, respectively. MET requires computation of the role (component is assigned to play) must be executed within the amount of time (hard real time); LATENCY constrains the maximum delay of data transportation communicating on the protocol. On the other hand, timing constraints can be verified not only by static design inspection via static scheduling [5], but also by runtime monitoring correctness assurance [42,43]. The typical examples of constraints localized on the patterns are listed as follows

- Compositional roles: a specific role can only interact with the other (consistency)
- Interactive roles should be consistent with architectural style (consistency)
- Appropriate glue of a role with the corresponding component acting as multiple roles (compatibility)
- Suitable protocols for a specific style that could be accommodated with a few protocols (compatibility)
- Hierarchical composition from well-grained components via heterogeneous composers (granularity)
- Autonomous components running on the distributed environment via datagram protocol (granularity)
- Heterogeneous interaction of the component that is assigned to play several roles (heterogeneity)
- A component interacts with one via *implicit invocation* and with the other via *pipeline* (heterogeneity)
- Max execution times (MET) specified for a consumer role relative to components (timing constraints)
- Max call times (MRT) specified for a producer role relative to components (timing constraints)

- Latency of communication protocols, potential starvation in distributed aspects (timing constraints)
- Synchronization between concurrent entities that ensures proper functioning (synchronous constraints).
- Rendezvous of the producer role with the consumer role (synchronous constraints)

C.2.2.2 Generalized Architectural Description

Compositional patterns are strongly associated with three essential aspects: *functional components*, *patternized composers* and *architectural configuration*. A functinal component is a unit of computaion, a loci of computation and state. A patterned composer is an architectural building block used to model interactions between components and rules governing the interaction. Both composer and component are reusable architectural elements. Architectural configuration is a connected graph of composers and components that describe the structural organization of HDSIS. In our research, components have three views: a decomposed unit in computational activity, a role in compositional architecture, and a derivative entity in derivational transformation. Modeling a component starts with hierarchical operators described in PSDL, assigns a component to play a specific role, identifes a suitable compositional pattern and derives the coppréspondent component with the selected composer. So in our formalism, the patternized composer is the centric entity that involves two interactive roles, an architecturel style and a communication protocol. The composer is described in generic package that can provide excellent reusability with parameterized genericity, abstracted class definition and active glued collaboration [22,33]. The roles in the composer are abstracted as generalized role wrappers that provide the adherence to the restricted, plug-compatible interfaces for interaction and the template of behavior for computation that components is expected to refine.

Framework for Architectural Description. The framework states that a set of formal description for modeling architecture should support what kinds of modeling features and their function embodied in the description. Both component and composer are architectural building blocks used to model computation, interaction and rules governing those interactions. Unlike connectors in other architectural description languages, composers in our research are always explicit entities that run through all architectural levels and implementation [21,22]. Surveying current architectural description languages [17], these following aspects are considered essential for modeling both component and composer, such as types, interfaces, semantics, constraints, evolution and hierarchical composition.

1) **Types.** Architecture-level communication is often expressed with complex protocols. To abstract away these protocols and make them reusable, formal description should model architectural entities as types. Abstract class are considered valuable generalized types which can be specialized to produce instances.
2) **Interface.** Interface of GRW or component is the type of interaction between it and the external world. The interface enables proper connectivity of components and their communication in an architecture. It also enables reasoning about the well-formedness of a configuration. On the other hand, an interface is modeled with ports and determined by (potentially dynamic) interfaces of its attached components. This added flexibility might prove a liability when analyzing for interface mismatches between components.
3) **Semantics.** To perform analyses of component interactions, consistent refinements across levels of abstraction, and enforcement of interconnection and communication constraints, architectural descriptions should provide processing semantics. A set of formal description generally uses a single semantic model for both components and composers. For instance, CSP-based concurrent semantics can be easy to map to Ada95 task mechanism such rendezvous, protected buffer access and asynchronous control transfer.
4) **Constraints.** In order to ensure adherence to restricted interactive roles, architectural styles and communication protocols, several kinds constraints should be specified, such as consistency, compatibility, timing constraints, etc.
5) **Evolution.** Component interactions are governed by complex and changing protocols. Maximizing architectural entity reuse is achieved by inheriting and refining existing ones. The formal description support entity evolution with subtyping and refinement.
6) **Hierarchical Composition** For HDSIS, architecture will be required to describe a system at different levels of details, where complex computation are either explicitly represented or abstracted away into single components (sub-systems). In the latter situation, an entire architecture becomes a single component in another, large architecture. Therefore, support for hierarchical composition is crucial.

Formal Architectural Description. In oreder to support perspective-based approach for architecture development and evolution, the formal architectural description must explictly model components that perform behavuoral computation, composers that promote interactions among components, and their configuration that are connected graphs of components and composers. Ada-like notation is taken to create the formal architectural description, because it could easily be mapped to current excellent object-oriented programming languages, such as C++, Java and Ada95 [30-33]. The description framework of composers and components associated with the above-mentioned aspects are presented as follows

Patterned Composer

```
composer PattName is generalized                            -- 1)
   -- generic parameters
style      as    ...;
protocol as    ...;
wrapper     -- role wrappers with generic parameters
   role W_Role1 is [associate | inherited | aggregate ] <grw> -- 5)
    port                                                     -- 2)
      ... -- computing states and service operations
    computation                                              -- 3)
      ... -- autonomous behavior and semantics              -- 4)
    end W_Role1;
    role W_Role2 is ... end W_Role2;
collaboration (P : W_Role1 ; C : W_Role2)
    ... -- collaborative behavior and semantics
end PattName;
```

Functional Component

```
component ComName is [ associate | inherited | aggregate ] <grw>
   ... -- wrapped roles in composers                      -- 1), 5)
port                                                      -- 2)
      ... -- computing states and service operations:
      ...      -- some computing states
      ...      -- some are overridden from the wrapped role
      ...      -- some are refined and extended
computation                                                -- 3)
      ... -- autonomous behavior and semantics including
      ...      -- control constraints                      -- 4)
      ...      -- task scheduling and synchronous coordination
      ...      -- asynchronous task scheduling
architecture                                                -- 6)
     ... -- hierarchical decomposition
end ComName;
```

Basically, a composer is represented as generalized template via generic parameters that can express complex protocols reusable different situation in concrete application. Two generalized role wrappers (GRW) in the composer are typical abstract classes that provide restricted, plug-compatible interfaces for interaction and template of behavior for computation. Collaboration part provides behavioral semantics during the interaction between roles that will be glued with physical components. A component is represented as a computational entity that derived from the associated GRW and it also provide an architectural decomposition which means the current component is composed from the entire architecture. Configuration in the research mainly focuses on instantiating components and composers and then gluing the instance components to the relative roles.

Formal semantics and constraints. For highly dependable software-intensive systems, the architectural description should be concerned with quantitative constraints such as componential granularity and interactive heterogeneity, timing constraints and synchronous constraints. Some of these properties could be abstracted as quantitative constraints in temporal logic that fits with model checking techniques [34,40,52]. In linear temporal logic (LTL), real-time constraints can be added to future time temporal operators to become metric temporal logic [52]. And the components and their interactions are treated as formal factors so that granularity and heterogeneity can be used to modify them. For instance,

$$\textit{Eventually}_{\text{FINISH-WITHIN=1000 ms}}\ (\textbf{\textit{Control\_Feeder}}_{\in\ \textit{cscc}}\ \textit{deliver FeedingAmount} > 50\text{ lb}),$$

which illustrates that component ***Control_Feeder*** belonging to the 2nd level component *CSCC* and undertaking feeding function, will deliver FeedingAmount greater than fifty pounds within 1000 ms.

A Typical Example. Pipeline is the typical composer that exhibits excellent architectural properties (e.g., loose component coupling, asynchronous communication, possible data buffering). Pipeline can be used to enforce interaction between components with dataflow stream. Two-side roles interconnected by Pipeline are *Outflow* and *Inflow*, respectively. The *Outflow* is deputized for the producing component to output stream, while the *Inflow* for the consuming component to input stream through the Pipeline. The formal Pipeline composer provides four generic parameters for enhancing reusabbility, such as transported ***Data*** (basic item for dataflow), buffer ***Size*** (data transportation buffer), timing constraints ***Vmet*** and ***Vmrt*** (the ammount of MET and MRT [4,5]). The derciption of the Pipeline composer and component derivations are described as follows

Pipeline Composer

```
composer Pipeline is generalized
      type Data is private;
      Size : Integer : = 100;
      vmet: Time;
      vmrt: Time;
style as   <#pipe-filter#>;
protocol as   <#dataflow-stream#>;
wrapper
   role Outflow is
    port
      procedure Output(d: Data);
      procedure Produce(d: Data) is abstract;
    computation
       Produce (d);
      *[ Output (d) → Produce (d) ◊ met(vmet) →exception; ]
    end Outflow;
    role Inflow is
    port
      procedure Input(d: Data);
      procedure Consume(d: Data) is abstract;
    computation
      *[ Input (d)→ Consume (d) ◊ mrt(vmrt) →exception; ]
    end Inflow;
collaboration (P : Outflow; C : Inflow)
     P•Produce(d);
   *[ P•Output(d)→ P•Produce(d)   C•Input(d) → C•Consume (d)]
end Pipeline;
```

Component derivation & Configuration

```
composer My_Pipe is specialized Pipeline (
       Data => Adt,   Size => 300,   Vmet=>80, vmrt=>100 );
——————<<< Component Derivation >>>——————
component Source_COM is inherited My_Pipe.Outflow
port
    procedure Produce(d: Data) is overridden;
computation
     Produce (d);
   *[ Output (d) → Produce (d) ◊ met(80) →exception; ]
architecture ... -- hierarchical decomposition
end Source_COM;
component Sink_COM is associate My_Pipe.Inflow
port
   procedure Consume(d: Data) is redefined;
computation
    *[ Inflow•Input (d)→ Consume (d) ◊ mrt(100) →exception; ]
end Sink_COM;
———————<<< Configuration >>>——————————
configuration CONFIG is
     Producer :   component Source_COM;
     Consumer:    component Sink_COM;
     My_Pipe•Collaboration(Producer, Consumer);
     ... -- Roles are glued with the instances of producer & consumer
     ... -- Instances assigned to play roles are concurrently executed
     ... -- Producer keeps producing data and then outputting it
     ... -- Consumer keeps inputting data and then consuming it
end CONFIG;
```

The example provides well-defined template for generalized role wrappers. With respect to behavioral comutation of components, CSP-based semantic description [30] provides not only synchronous constraints but also asynchronous control transit. For instance, both *Output* and *Input* are designed implemented as exclusive procedures. They can be treated as *execution guards* to coordinate concurrent synchronization. For instance, on timing constraints, the role *Outflow* is subjected to max execution time (met) while *Inflow* to max response time (mrt). Both met(Vmet) and mrt(Vmrt) are transformed as asynchronous control transit for runtime monitoring of real time constraints. That is, when outputting a produced data onto the given pipeline, *Outflow* requires to be synchronized within met(Vmet), otherwise the synchronization is considered failure and MET_EXCEPTION is triggered (◇ represents asynchronous select). Similarly Inflow has the similar situation with mrt(Vmrt).

Colaboration part in the composer description will automatically generated as architectural configurations that are connected graphs of components and composers. In concert with models of components and conposers, configurations enable assessment of autonomous and concurrent aspects of an architecture, such as potential for deadlocks, and starvation, performance, reliability, security, etc.. Configurations also enable concurrent execution immediately after the roles are glued with the instances of corresponding components.

C.2.2.3 Substantiated Interconnections

The interconnections among components are the central argument because they embody the substantial architecture of a system. In early time, the architecture of a software system was nothing except for graph of "box-line-box". Substantiating interconnection is a must for an explicit architecture of HDSIS. It deals with three aspects: explicit architectural entity by which interaction among component are promoted, compositional relationships that guide topologic connectivity of components, and heterogeneous information forms by which communication among components can be built. With respect to explicit architectural entity, we have thoroughly discussed compositional patterns and the associated composers in this proposal. Only compositional relationships and heterogeneous information form will be discussed here.

Compositional Relationship. Since the asymmetry of an interaction among roles, a producer role could interact with one more other components that are assigned to play the same role. For instance, an *Announcer* of event might have one more *Listener* that is listening to the same *Announcer.* In contrast, one more *Source* of knowledge might have the same *Repository,* which means shared memory. So there exist compositional relationships for transforming interconnections to patternized composer, which is illustrated as follows

- Fork (1~N): Fork allows a producer to interact with multiple consumers that play the same role via a composer.
- Merge (N~1): Merge allows multiple producers that play the same role to interact with a consumer via a composer
- Unique (1~1): Unique allows a producer to interact with the other consumer via a composer.
- Hierarchy: Hierarchy allows the external[1] producer to interact with the internal[1] consumer, and vice versa.

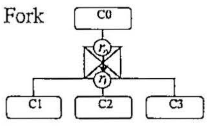
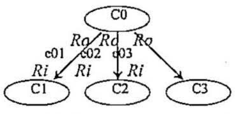
Types	PSDL-like description	Formal Description
Fork 	Graph Vertex C0, C1, C2, C3 Edge E01: C0 as Ro -> C1 as Ri Edge E02: C0 as Ro -> C2 as Ri Edge E03: C0 as Ro -> C3 as Ri  C0 Ro Ro Ro c01 c02 c03 Ri Ri Ri C1 C2 C3	Architecture Fork: (C0 ⇸ Ro)—s/p→ (Ri↤C1, C2, C3)

Figure 7. Typical compositional architecture: Fork relationship

Fig. 7 illustrates the example about how to explicitly architect the computational model described in prototyping language with patterned composer. The related outcome is automatically generated compositional architecture. In prototyping description, a ***Vertex*** denotes a component and an ***Edge*** denotes an interconnection. The vertices are tagged as roles for explicit architecting. Generally, since a patternized composer is designed as an aotonomous entity, it dyanamically maitain necessary state space in order to coordinate the interaction among more than 2 roles. For instance, the event-bsaed composer will maintain a event-listening list that is used to register multiple Lisenters. Once an event is announced, all Listeners in the list will be triggered by the event [22]. In this way, Fork crelationsip greatly simplifies interconnections among components, that is, several edges could share the same one composer if the composnents are supposed to paly the same role.

Heterogeneous Information Forms. Heterogeneous information is strongly asscoaited with the architectural style and communication protocol. The way of information transportation will be refered to communication protocols in compositional patterns. The homogengeous information form in CAPS [1] is data stream, by which a producer interacts with the consumer via FIFO. In our research homogeneous information is extended to following heterogeneous information forms which are associated with the interaction between interactive roles via specific architectural style while complying with specific communication protocol:

[1] Both external and internal are referred to the hierarchical decomposition. For a given hierarchical level of decomposition, a component in current level is the *external* to the component in low level, while the latter is the internal to the former..

- Dataflow: *Outflow* interacts with *Inflow* via *Pipeline* while complying with *Dataflow*
- Sampled: *Sampler* interacts with *Reader* via *Sensored-Status* while complying with *Sampled*
- Knowledge: *Sourcer* interactis with *Repository* via *Knowledge-Repository* while complying with *Knowledge*
- Event: *Announcer* interacts with *Lisenter* via *Implicit-invocation* while complying with *Envent*
- Message: *Caller* interacts with *Definer* via *Remot-procedure-call* while complying with *Message*

Patternized composers of heterogeneous information transportation can support heterogeneous interaction among components and hieratrchical composition from decomposed components, which increases flexibility of modeling computational activity, especially enhances suitability for Software-intensive systems development.

C.3 General Plan of Work

C.3.1 Plan Overview

We envision the computational model that we currently have as the start point for additional research.

Prototyping language and its equivalent graph are suitable for capturing computation and interconnection. Relevant ambiguities and missing attributes in PSDL will need to be identified and appropriate extensions made to correct these shortcomings for perspective-based architectures. For instance, convenient representations of the roles' assignment and roles' selection are likely needed, because interactive roles in compositional patterns is a critical factor that used for adherence to restricted, plug-compatible interfaces for both interaction and computation; meanwhile, architectural styles and communication protocols are strongly dependent on the assignment of roles to specific components.

We must investigate compositional patterns that are expressly useful for explicitly architecting. They are embodied by a set of rules governing interactions among components and characterized as interactions between roles via architectural styles while complying with communication protocols. Finding and abstracting sophisticated roles for compositional patterns likely makes the proposed research more practical and applicable. For instance, highly abstracted role pairs RMI_Caller / RMI_Definer will largely simplify remote method invocation for distributed architecture, because many complicated concepts and details such as marshaling of calling parameters for RMI_Caller, Datagram via TCP/IP unmarshaling of invoked arguments for RMI_Definer are all hidden from the physical components by the tangible RMI composer.

We should thoroughly study optimized object models represented in UML, Ada95, Java and C++ so as to support template, concurrent and exclusive object design, because this is the foundation of the implementation of formal description. We also need to study formal semantics for behavioral concurrency and their mapping with sophisticated concurrent distributed object-oriented programming languages such as Java, Ada95, CSP. Automatically generated behavioral concurrency is likely useful in runtime monitoring of component evolution, because concurrent bugs is hard to detect by static design inspection. These and many more patterned features will be needed in the formal description for supporting generalized architectural framework.

Many of the tasks can be undertaken in parallel. Each of fundamental approach can be considered independently of each other. For instance, computational activity depends on prototyping model extension, while compositional architecture on the formal description that models components, composers and configurations. Students will be assigned services for research and implementation. For our purposes, being able to certify correctness form a diverse set of patternized composers with multiple implementations is a goal of this research.

C.3.2 Broad Design Activities

Responding to the requirement in this proposal, three fundamental approaches are proposed to implement HDSIS: computational model for rapid system prototyping, compositional patterns for explicitly architecting, and optimized object model for component object evolution. All of them can be synthesized to characterize different perspectives of a HDSIS, so that a transformational process can be developed to support the perspective-based architecture design. The synthesis approach calls for explicit treatment of software composition and architecting that involves (1) **three perspectives of architecture** characterized as computational activity, compositional architecture, and derivational transformation, (2) **two mappings** between three perspectives, embodied in explicitly architecting the system via compositional patterns and physically deriving the component from architectural role wrappers, (3) **a set of formal description** that is used to model well-grained components, heterogeneous / hierarchical composition, and flexible configuration, and (4) **associated support tool** that makes the synthesis approach more applicable in design, analysis, evolution and executable system generation.

C.3.3 Deliverables

1) Automated tool for analyzing prototyping model into computational perspective
2) Rigorous taxonomy of interactions to enhance compositional patterns for compositional perspective
3) A set of formal description tools that explicitly models components, composers and configurations
4) An automatic inspection tool for inspecting constraints localized on the architectural description.

5) Generalized framework for monitoring component derivation at run time.
6) A group of associated technical papers and publications

C.3.4 Description of Procedure

Building on our strengths, we will perform the following:

1) Three perspectives of architecture characterized as computational activity, compositional architecture, and derivational transformation,
2) Two mappings between three perspectives, embodied in explicitly architecting the system via compositional patterns and physically deriving the component from architectural role wrappers,
3) A set of formal description that is used to model well-grained components, heterogeneous interactions and hierarchical composition, and
4) The associated support tool that makes the synthesizing approach more applicable in design, analysis, evolution and executable system generation.

C.3.5 Evaluation Factors

C.3.5.1 Perspective-based Models

Perspective-based models are built to reflect the concerns of differing stakeholders, which will obviously improve understandability, maintainability, and reliability of the modeled system, reduce re-certificate. Clearly modeling stakeholer's concerns as computational activity will reduce re-certification effort required after each requirement change that stays within the envelope of some invariants. Model built for architect's concerns hold promise to keep invariant with respect to system requirements changes.

C.3.5.2 A set of compositional patterns supporting explicit architecture

Compositional patterns capture architectural factors such as role components are assigned to play in the architecture, styles the architecture will performs, and protocols by which information are transported during interactions. Because of undertaking coherent tasks that are well designed and frequently reused, computational complexity will be largely reduced and analysis of the systematic properties will be dramatically simplified.

C.3.5.3 Dependability translation and localization

Dependability of software-intensive systems as well as their affordable flexibility is crucial factors. Flexibility can be embodied by sophisticated compositional patterns that hold invariants with respect to system requirements changes, while dependability will be translated to quantitative constraints and then localized on the patterns, so that the effectiveness of requirements validation will be increased

C.3.5.4 Agility and flexibility for adaptation to rapid changes

The agility and flexibility is heavily concerned by C4ISR systems because the adaptation to rapidly changing circumstances is needed to bring about desired outcomes that provide an unprecedented level of interoperability in software-intensive systems to support the various units of a coalition (the "plug and play" concept).

C.3.6 Schedule

1) Study current and future releases of UML, ADLs; Investigate their appropriateness for HDSIS (3 months)
2) Formulate stakeholder' s concerns as key aspects for development, evolution and inspection (6 months)
3) Create generalized architectural template for explicit architecting software-intensive systems (12 months)
4) Formulate dependability to quantitative constraints, and study their localization (3 months)
5) Develop a set of formal description with accompanied support tools for architecture design (8 months)
6) Improve CSP-based semantic description for concurrent and synchronous constraints (4 months)
7) Build reusable library of compositional patterns, and create a set of patterned composers (6 months)
8) Abstract generalized role wrappers as easily derived templates for framework generation (12 months)
9) Study current component middleware, and abstract architectural facilities for interoperability (3 months)
10) Incorporate rapid prototyping & explicit architecting into an automatic synthesis approach (8 months)

C.3.7 Comparison with Other Research

In present, software research community focuses on software-intensive systems development. Since it involves many perspectives and each perspective appropriately deals with different concerns, the three perspectives for software-intensive systems development are requirement engineering, architecture design and systematic implementation. In this proposal, these three perspectives correspond to computational activity, compositional architecture and derivational transformation, respectively. Architecture is recognized as a critical element in the successful development and evolution of software-intensive systems. Explicitly architecting a system bridges the gap between software requirement and system implementation to attain the benefits of reduced costs and increased quality such as usability, flexibility, reliability, interoperability and so forth. Software-intensive systems will inevitably involve different stakeholders such as customer, architect and implementer. For instance, increased uncertainty about requirements, flexible configuration in organizational structures, and rapid development of application demand diversified perspectives to reflect stakeholders' concerns.

Work related to the topics discussed in this proposal includes research in the areas of rapid system prototyping, software architecture and component middleware, all of which focus on composing software systems from coarser-grained components. Although these efforts present complementary, often overlapping approaches, they come down to the areas of requirement acquisition, architecture design, systematic implementation. HDSIS development also involves two accompanied research areas: design inspection and perspective-based architecture framework. The former provides dominating methods for detecting errors in software systems, whereas latter specifies multiple perspectives (views) of software-intensive system architecture.

Rapid system prototyping (e.g., CAPS [1]) has been found to be an effective technique for clarifying requirements and eliminating the large amount of wasted effort currently spent on developing software to meet incorrect or inappropriate requirements in traditional software life cycle. A prototype is an executable model or a pilot version of the intended system. Lack of agreement on the requirements as specified by the customer and as analyzed by the designer causes inconsistencies between the delivered system and customer expectations, leading to expensive rebuilding. generally, rapid prototyping is effective to capture uncertainty about requirements. This kind of technique is seriously concerned of computational activity so that rapid consistent response is given to customer, with less consideration of both explicit architecture and derivational implementation, because a prototype is usually a partial representation of the intended system, used as an aid in analysis and design rather than as production software [6,7].

System architecting (e.g., software architecture and the associated architectural description languages (ADLs) [9-11,13,15,17,18]) holds promise in attaining benefits of reduced costs and increased quality because software architecture forms the backbone for building highly dependable software-intensive systems. A system's quality attributes are largely permitted or precluded by its architecture. Architecture represents a capitalized investment, an abstract reusable model that can be transferred from one system to the next. Architecture represents a common vehicle for communication among a system's stakeholders, and is the arena in which conflicting goals and requirements are mediated [44,46]. The software architecture research community, essentially academics, has focused on the creation and improvement of special-purpose languages, known as architecture description languages (ADLs). In the past years, numerous ADLs have been specially designed to represent different aspects of architectures of software-intensive systems. ADLs have the advantage of being mathematically founded, facilitating analysis of architectural models, but they have also the disadvantage of lacking adequate support for separating various kinds of stakeholders' concerns along different viewpoints. Due to their too rigorously formal nature, ADLs can be hard to understand and to use, as developers in need of ADL-based software architectures will have to learn the mathematical models of systems. Especially, software architecture approach typically separates computation (components) from interaction (connectors) in a system. Despite this, connectors are often considered to be explicit at the level of architecture, but intangible in the system implementation [15,46].

Using UML extension to model software architecture is a new effort for architecture deign. The UML [19,20] is a family of design notations that is rapidly becoming a de facto standard software design language. Unified system modeling constructs large enterprise applications in a way that enables scalability, security, and robust execution under stressful conditions. UML provides a variety of useful capabilities to the software designer, including multiple, interrelated design views, a semiformal semantics expressed as a UML meta model, and an associated language for expressing formal logic constraints on design elements. In supporting architectural concerns within UML, one approach involves using UML "as is," while the other approach incorporates useful features of existing ADLs as UML extensions. The assessment of UML's expressive power for modeling software architectures indicates that UML currently lacks support for capturing and exploiting certain architectural concerns whose importance has been demonstrated through the research and practice of software architectures. In particular, UML lacks direct support for modeling and exploiting architectural styles, explicit software connectors, and local and global architectural constraints [16,18,46].

Component middleware platform (e.g., J2EE, CORBA, COM+ [23-28]) enables frequently composing software systems from prefabricated, heterogeneous components that provide complex functionality and engage in complex interactions. Existing research on component-based development has mostly focused on component structure, interfaces, and functionality. Recently, software architecture has emerged as an area that also places significant importance on component interactions, embodied in the notion of software connectors. However, the current level of understanding and support for connectors has been insufficient. This has resulted in their inconsistent treatment and a notable lack of understanding of what the fundamental building blocks of software interaction are and how they can be composed into more complex interactions. Middleware platform assumes a homogeneous architectural environment in which all components adhere to certain implementation constraints (e.g., design, packaging, and runtime constraints) and it is unalterably associated with derivational transformation because its primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure [15,28].

Design inspection and testing are the dominating methods for detecting errors in software systems. However, testing has the drawback that errors are detected very late in the development process. As a result errors that have their origin in the system design may impose substantial costs even if the error is detected during testing. Design inspection, on the other hand, facilitates early detection of errors. Design inspection also known as verfication and validation (V&V) is the checking and analysis processes that ensure that software conforms to its specification and meets the needs of the customers who are paying for that software. [34,35,37]. Formal verification and automated analyses are static V & V techniques, as they do not require the system to be executed. However, even if a design has been formally verified, it still does not ensure the correctness of an implementation of the design. This is because the implementation often is much more detailed, and may not strictly follow the formal design. So, there is possibility for introduction of errors into an implementation of a design that has been verified. Runtime assurance based on formal specification is the dynamic V & V techniques, as they require the system to be executed under the designing monitors to check for the correctness of the system [42,43]. Automated source code analysis and formal verification used to check the correspondence between the implementation system and its specification (verification). Translating source code to highly-abstracted finite-state automaton for model checking is a practical approach [36,40,41], which is insuring correspondence between the properties of the source code that are to be reasoned about and the properties of the automaton to be checked. All of these efforts rarely connect formal specification with architectural elements such as components, composers and configurations.

IEEE 1471 [44] defines architecture as "the fundamental organization of the system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution". In addition, it refers to an architecture description as "a collection of products to document an architecture". By these definitions, the distinction between the architectural description and architecture of a software-intensive system. IEEE 1471 is a recommended practice, but it does not prescribe any particular views and concerns, that is, IEEE 1471 does not provide a standard architecture, or architectural process or method. The United States Department of Defense has mandated Joint Technical Architectures (JTA) to support the acquisition of systems that are interoperable and will meet the needs of military coalitions. JTA Framework [45] specifies three perspectives (views) of an architecture and defines a set of products that describe each view. The three views of JTA architecture framework are supposed to promote a great help in consistently engineering HDSIS, but this framework does not provide a procedure for developing the artifacts that used in the description [47-49]. The lack of adequate support for separating various kinds of stakeholders' concerns along different viewpoints is the collective weakness of software architecting with architecture description. IEEE 1471 and C4ISR architecture framework specify multiple perspectives for software-intensive systems, which provides complementary efforts for architecture description. However, the lack of a definitive process has posed a challenge to those who are responsible for developing architectures that are compliant with the framework. Some of software research groups have set their focus on developing a process for the framework design, for instance, modeling software architectures in the UML [18], bridging the gap between IEEE1471, ADLs and UML [46], structured analysis approach and object-oriented approach for architecture design [47-50], object-oriented model for interoperability via wrapper-based translation [51]. However, these researches towards JTA and IEEE 1471 do not support explicit treatment of software composition and architecting.

Most of the previous work focuses on either specific profiles, for instance, computational perspective (rapid prototyping, UML), compositional perspective (software architecture, ADLs, UML extension), derivational perspective (component middleware), or architecture framework of multiple perspectives without a definitive process for architecture design (JTA, IEEE 1471). This proposed research not only develop synthesis approach for highly dependable software-intensive systems, but also founds a set of compositional patterns that support explicit architecting throughout the whole procedure from computational activity, through compositional architecture, to evolutional transformation among multiple perspective models. This research provides the formulating loci of desired properties features against different perspectives in supporting design inspection and run-time correctness assurance. This research also provides a means of translating macro dependability into quantitative constraints and localizing them on compositional patterns embodied as generalized template. Indeed, the issue of how to implement architecture framework for HDSIS with explicit treatment of architecting has been an area of proposed research.

C.4 Broader Impact

If effective constructing HDSIS can become a highly automated procedure then the following will take place:

1) The agility and flexibility will be acquisitioned to adapt to rapidly changing circumstances and bring about desired outcomes that provide an unprecedented level of interoperability in software-intensive systems to support the various units of a coalition (the "plug and play" concept).
2) HDSIS can be constructed by using multiple perspectives of architecture to address increased

uncertainty about requirements, rapid changes in technology, changes in organizational structures, and a widening of spectrum of mission and operations.

3) The dependability of software-intensive systems as well as their affordable flexibility will be improved by integrating the proposed research with requirements validation techniques, and by integrating differing stakeholder perspectives and concerns into the models and methods to be used.
4) We put high confidence on a sound objective basis, via a systematic method for expressing dependability objectives via measurable localized constraints associated with the subsystems of the architecture. We also enable an assurance processes into independent tasks associated with each subsystem to the extent possible, thus simplifying analysis and reducing computational complexity.
5) We integrate rapid prototyping, explicit architecting and consistent engineering into a synthesis approach for HDSIS based on multiple perspectives of an architecture, and this approach is embodied in the transformational procedure that will be automatic procedure by CASE tool support.

C.4.1 Transition of Technology

Technology transfer will be addressed by integrating the proposed new capabilities with an existing code developed under CAPS projects. By re-using commonly used language like PSDL and Ada-adapted architectural description instead of creating fully new languages, general acceptance of our approach is enhanced. Publish results in ACM, and IEEE sponsored conferences and making toolkit available can facilitate acceptance.

The Software Engineering Group at the Naval Postgraduate School offers M.S. and Ph. D degrees. The students as NPS will contribute to this research and development effort. Their involvement will facilitate information transfer into the DoD further. We also plan to integrate emerging technologies into the courses we teach.

C.4.2 Experimentation and Integration Plan

The faculty of the Software Engineering Group at the Naval Postgraduate School and their Ph. D and M. S. students will perform the work. The principle investigators will be responsible for coordinating the following plan previously stated in section C.3.6 for schedule:

1) Study current and future releases of UML, ADLs; Investigate their appropriateness for HDSIS
2) Formulate stakeholder' s concerns as key aspects for development, evolution and inspection
3) Create generalized architectural template for explicit architecting software-intensive systems
4) Formulate dependability to quantitative constraints, and study their localization
5) Develop a set of formal description with accompanied support tools for architecture design
6) Improve CSP-based semantic description for concurrent and synchronous constraints
7) Build reusable library of compositional patterns, and create a set of patterned composers
8) Abstract generalized role wrappers as easily derived templates for framework generation
9) Study current component middleware, and abstract architectural facilities for interoperability
10) Incorporate rapid prototyping & explicit architecting into an automatic synthesis approach

C.5 Related Work

In order to support effective construct software-intensive systems, we have undertaken research efforts to address three problems; (1) can an explicit architecture be extracted from existing prototyping model, (2) can a minimal set of compositional patterns be founded to govern explicitly architecting and hierarchically composing the system from well-grained components, and (3) can an transformational process be developed to support the perspective-based approach. In paper [2,3,6,8], computational model for rapid prototyping is represented as a hierarchy of networks of structured objects and interconnections among them with semantic (formal) constraints. That work established the fact that an architecture could be implicitly developed using prototyping model and that the resulting description could be mapped to the requisite software-intensive system products (problem 1); In papers [21,22], compositional patterns and the associated composers treated as first-class architectural elements to promote hierarchical composition from components were investigated. Especially, the patternized composer for implicit invocation can be designed as an explicit event broker decentralized in Ada95 [22]. So the initiative solution toward perspective-based approach is to build a library of reusable composer to support explicitly software architecting (problem 2). In papers [31-33,21,22,51], object-oriented model for interoperability via wrapper-based translation [51], object-oriented design with different programming languages were thoroughly studied, especially, the optimized object model based on Ada95 can be used to not only describe both functional component and patternized composer but also support such architectural element evolution through subtyping and refinement so that smooth transformation between architecture design and physical implementation could be made (problem 3).

REFERENCES CITED

1. Luqi, M. Ketabchi, A computer-Aided Prototyping System, IEEE Software, March 1988

2. Luqi, V. Berzins, R. Yeh. A Prototyping Language for Real-Time Software, IEEE Trans. on SE, Vol. 14(10), October 1988

3. Luqi, Robot S., *et al.* CAPS as a Requirement Engineering Tool. Proc Tri-Ada'91, Oct 22-25, 1991, San Jose, USA

4. Luqi, Real-Time Constraints in a Rapid Prototyping Language, Computer Language, Vol 18(2), 1993

5. M. Shing, Luqi, Real-Time Scheduling for Software Prototyping, Journal of Systems Integration, 6, 1996

6. Luqi, Ying Qiao, Lin Zhang, Computational Model for High-Confidence Embedded System Development, Accepted by Monterey workshop 2002 -- Radical Innovations of Software and Systems Engineering in the Future, Venice, Italy, October 7-11, 2002.

7. Luqi, Xianzhong Liang, Transformational Model for Highly Dependable Systems, Technical report #NPS-SW-02-008, NPS Monterey, CA: July 2002.

8. B. Kramer, Luqi, V. Berzins, Compositional Semantics of a Real-Time Prototyping Language, IEEE Trans. on SE, Vol. 19(5), May 1993

9. D.E. Perry and A. L. Wolf, Foundations for the Study of Software Architectures. ACM SIGSOFT SE Notes, Oct 1992

10. M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Inc., 1996

11. Garlan D, Monroe B, Wile D. Acme: an interchange language for software architecture. Technical Report, CMU-CS-95-219, Software Engineering Institute, Carnegie Mellon University, 1997

12. R. J. Allen, Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, May, 1997.

13. Andrew P. and Charles L., Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives, System Engineering, John Wiley and Sons, Inc., 1998

14. M. Shaw, Coping with Heterogeneity in Software Architecture,

    http://www-2.cs.cmu.edu/afs/cs/project/compose/www/paper_abstracts/Coping.html

15. N. R. Mehta, N. Medvidovic. Towards a Taxonomy of software Connectors. Proc. ICSE 2000, Limerick Ireland

16. Garlan and Kompnek, Reconciling the needs of architectural description with objectmodeling notations. In Proceedings of the Third International Conference on the Unified Modeling Language (UML 2000, York, UK, October), Springer-Verlag, Berlin, Germany

17. N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, 2000, 26(1)

18. N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, Modeling Software Architectures in

the Unified Modeling Language, ACM Transaction on Software Engineering and Methodology Vol. 11(1), 2002

19. Object Modeling Group, Inc., Unified Modeling Language Specification, version 1.3, June 1999

20. G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language user guide, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 1999

21. Xianzhong Liang. Interaction-based Compositional Evolutionary Approach for Complex Software Systems, Institute of Software, Chinese Academy of Sciences, Ph. D. Dissertation, 2001

22. Xianzhong Liang, Zhenyu Wang. Event-based implicit invocation decentralized in Ada, ACM AdaLetters, March, 2002

23. R. Orfali, D. Harkey, J. Edwards, The Essential Distributed Objects Survival Guide. John Wiley & Sons, Inc., NY, 1996

24. Sessions N., COM and DCOM: Microsoft's Vision for Distributed Objects. John Wiley & Sons, Inc., NY, 1997

25. OMG/ISO Standard, CORBA: Common Object Request Broker Architecture, http://www.corba.org/

26. Sun Microsystems, Inc. Java 2 Enterprise Edition Specification v1.2. http://java.sun.com/j2ee/

27. COM+ (Component Services), Version 1.5,

    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cossdk/htm/complusportal_9o9x.asp

28. A. Gokhle, D. Schmidt, B. Natarajan, N Wang, Applying Model-Integrated Computing to Component Middleware and Enterprise Applications, Communications of the ACM, Vol. 45 (10) Oct., 2002

29. ANSI/ISO/IEC-8622: 1995, Ada95 Reference Manual: Language.

30. Xianzhong Liang. Ada task specification and CSP-based formal description for Concurrent Semantics. Computer & Digital Engineering, 1989.5

31. Xianzhong Liang, Zhenyu Wang, Ada-based Support for Abstraction, Encapsulation and Unit Hierarchy, Proceedings of Tri'91 International Conference, San Jose, USA, Oct. 1991.

32. Xianzhong Liang, Z. Wang, L. Xu, Java Interfaces and Dynamic Version Configuration for Object-Oriented Systems, Proceedings of TOOLS Asia'98 & OOT China'98, Beijing, China, Sep 22-25, 1998.

33. Xianzhong Liang, Z. Wang. Omega: A Uniform Object Model Easy to Gain Ada's Ends, ACM AdaLetters, June, 2001

34. E. M. Clarke (CMU), R. P. Kurshan (Bell Lab). Computer-Aided Verification, Feb. 17, 1996

35. E.M. Clarke, S. Jha, Y. Lu, H. Veith, Tree-like Counterexamples in Model Checking, *Logic in Computer Science (LICS 2002)*, July 2002.

36. Mattew B. Dwyer, Corina S. Pasareanu, Translating Ada Programs for Model Checking, Depart. of Computing and Info Sciences, Kansas State University

REFERENCES CITED

37. Ian Sommerville, Software Engineering, Addison-Wesley, 2001 (*6th ed*)

38. Meyer B., Object-Oriented Software Construction, Prentice-Hall International Inc., 1997

39. Boehm, B., Software Engineering Economics, Prentice-Hall, 1981

40. James C. Corbett, Matthew B. Dwyer, Bandera: Extracting Finite-state Models from Java Source Code, Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)

41. David Y.W. Park, *et el*, Java Model Checking, Computer Science Depart., Stanford University, June 22, 2000

42. S. Kannan, M. Kim, Insup Lee, Oleg Sokolsky and M. Viswanathan, Run-time monitoring and Steering based on Formal Specifications, Proceedings of the 200 Monterey Workshop on Modeling Software System Structures in a fastly moving scenarios, June 13-16, 2000, Santa Margherita Ligure, Italy

43. M. Kim, Insup. Lee, U. Sammapun, J. Shin, O. Sokolsky, Monitoring, Checking, and Steering of Real-Time Systems, 2nd International Workshop on Run-time Verification. Copenhagen, Denmark, July 26, 2002

44. IEEE Standard Board, Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE-std-1471 2000), September 2000.

45. DoD Joint Technical Architecture (JTA Version 4.0, 2002), http://www-jta.itsi.disa.mil/

46. M. M. Kande, Valentin Crettaz, Alfred Strohmeier, Shane Sendall, Bridging the Gap between IEEE 1471, Architecture Description Languages and UML,

    http://icwww.epfl.ch/publications/documents/IC_TECH_REPORT_200259.pdf

47. H. Alexander and W. Lee, C4ISR Architectures: I. Developing a Process for C4ISR Architecture Design. Systems Engineering, John Wiley and Sons, Inc., Vol. 3 No. 4, 2000, p225

48. W. Lee, S. Insub, *et al.* C4ISR Architectures: II. A Structured Analysis Approach for Architecture Design. Systems Engineering, John Wiley and Sons, Inc., Vol. 3 No. 4, 2000, p248

49. W. Lee, Haider, S. and Levis, A.H. Synthesizing Executable Models of Object Oriented Architectures. In Proc. Formal Methods in Software Engineering and Defence Systems 2002, Adelaide, Australia. Conferences in Research and Practice in Information Technology,

50. P. Micheal, *et al.* C4ISR Architectures: III. An Object-Oriented Approach for Architecture Design. Systems Engineering, John Wiley and Sons, Inc., Vol. 3 No. 4, 2000, p288

51. P. Young, V. Berzins, J. Ge and Luqi, Use of Object-Oriented Model for Interoperability in Wrapper-Based Translator for Resolving Representational Differences between Heterogeneous Systems, Monterey Workshop 2001 on Engineering Automation for Software Intensive System, Monterey, California, USA, 2001

52. Modeling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice, Paul Pettersson. Ph.D. Thesis, Technical Report DoCS 99/101, Department of Computer Systems, Uppsala University, 19 February 1999.