2002

Dependability-Assured Software Transformation

Luqi; Liang, Xianzhong

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/65080

NPS-SW-02-008

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Dependability-Assured Software Transformation

Technical Report

By

Luqi, Xianzhong Liang

July 2002

Prepared for: NSF

PROJECT SUMMARY

The proposed research is to create new paradigm of software transformation and analysis tools that will incorporate computer-aided prototyping system (CAPS) into dependability-assured software transformational platform (DAST) for highly dependable embedded systems (HDES). DAST extends CAPS with software architecting and composition technologies to transform macro dependability (global qualitative requirements) into micro dependability (quantitative constraints). Based upon rapid prototyping, the dependability-assured transformational process from a rapid-prototyped system to the highly dependable embedded system will involve quantitative constraint abstraction in multiple perspectives, software transformation, and formal method applied to verify the correctness of the eventual-evolved system. The proposal shows some distinguished features, however, the gap between software requirements and system implementations results in the following problems that must be solved:

- Perspective confusion problem: software-intensive systems inevitably involve different stakeholders: application customers, software architects / engineers and system implementers. Each kind of stakeholders will share a different perspective.
- Model construction problem: a formal model is needed to reflect different perspective, but how to, with dependability-assurance, transform one kind of perspective into the other becomes crucial, because there exists a big gap between software requirement and system implementation.
- Attribute identification problem: In macro view, the dependability of software systems is abstracted as availability, reliability, safety, confidentiality, integrity and maintainability. How to transform qualitative global requirement into quantitative constraints becomes the key.
- Software tool support problem: to transform intellectual models into automatic analysis will be the main challenge. Formal method and suitable representation for reasoning and manipulation by CASE tools hold the promise for mechanical analysis process.

In order to provide systematic solution towards dependability-assured software transformation from rapid prototyping to highly dependable systems, the proposed work is to develop new techniques for quantifying dependability and mechanical analysis tools. There are five research thrusts in our initiative:

- *Studying networked computing* via test-bed facilities. Software artifacts of significant scale provided by test-bed facilities represent mission-critical systems for both NASA and IT industry. As autonomous systems, they are characterized by networked distributed computing and communication, and are useful for the proposed work to collect requirements of desired properties and experiment with test-bed facilities.
- *Modeling embedded system* via multiple perspectives. Multiple perspectives are characterized by collection of computational activities (*customer's concern*), or a set of rules of compositional architectures governing the interdependencies and interactions among components (*architect's concern*), as well as evolutional implementation and component links for interoperability within the architecture (*implementer's concern*).
- *Architecting software-intensive system* via compositional patterns. As a system is decomposed into components, heterogeneous interactions among them are used to compose the system from components. Consistently engineering a system with software architecting is to capture such properties as granularity and heterogeneity, patterns to guide the composition and quantified constraints on the patterns.
- *Evolving rapid prototype* via dependability-assured transformation. A prototype is a hierarchy of networks of structured objects with semantics, which allows software transformation. The constraints provide formal method and means to *reason about* the validity of transformations. Both functional behaviors and non-functional properties are specified so that the system can be transformed in dependability-assured way.
- *Verifying software system integrity* via quantified constraints. System integrity refers to the correctness of the software-intensive system operation with respect to the interfaces and interactions among components. The system must satisfy the non-functional properties imposed by the compositional architecture so that application requirements concretely embody the high-level dependability.

A distinguished feature of the project is that we present a formal model for software transformation, on which three perspective are developed to reflect computational activities, compositional architectures and evolutional implementation, so that automated software process and mechanical analysis are to be processed as easily and as productively in engineering highly dependable embedded systems. The goal of the project is to develop new software transformation methods and tools to provide rigorous means and dependability-assured transformational process from a rapid prototyped system developed in CAPS into a highly dependable embedded system, and then verifying the eventual system with quantitative constraints dispensed on the compositional patterns.

PROJECT DESCRIPTION

We propose a four-year research initiative at Software Engineering Automation Center (SEAC), Naval Postgraduate School to create new paradigm of software transformation and analysis tools that will incorporate the computer-aided prototyping system (CAPS[Luqi88-1, Luqi88-2]) into the dependability-assured software transformational platform for highly dependable embedded systems.

C.1 Objectives and Significance

The main objective of the proposed work is to enable dependability-assured software transformation (DAST) for highly dependable embedded systems (HDES). Formal methods involve a number of technical activities, including formal system specification, specification analysis and proof, transformational development and program verification [Ians01]. We propose to develop a formal model for compositional architecture that is characterized by *collaborative roles* components play, *architectural styles* specified for interactions among them and *communicative protocols* used for data transportations. The quantitative constraints abstracted from rapid prototyping and software architecting are specified as measurable attributes that will be dispensed on the proposed formal model.

We also seek to develop methods for dependability-assured transformation from a *rapid-prototyped system* to the highly dependable embedded systems, and for reverse analysis from the existing systems to the formal model that is useful to verify the correctness of the *eventual-evolved systems*,

C.1.1 Significance and Benefits

New techniques, automated tools and methodologies that are proposed to develop will incorporate rapid prototyping, dependability-assured transformation and formal verification techniques into highly-automated, easily-applied CASE tools. There are five research thrusts in our initiative:

- *Studying networked computing* via test-bed facilities. Software artifacts of significant scale provided by test-bed facilities represent mission-critical systems for both NASA and IT industry. As autonomous systems, they are characterized by networked distributed computing and communication, and are useful for the proposed work to collect requirements of desired properties and experiment with test-bed facilities. Esp., distributed embedded computing entity for next-generation air traffic control is ideal target to study.
- *Modeling system* via multiple perspectives. Multiple perspectives are characterized by collection of computational activities (*customer's concern*), a set of rules of compositional architectures used to govern the interdependencies and interactions among components (*architect's concern*), as well as evolutional implementation and physical component links for distributed interoperability within the compositional architecture (*implementer's concern*).
- *Architecting system* via compositional patterns. As decomposition of a system into hierarchical-grained components, heterogeneous interactions among components are inevitably to be introduced to advocate system composition. Engineering a system with software architecting is to capture such architectural properties as granularity of components and heterogeneity of interactions, with well-designed patterns to guide the composition from components and quantified constraints to be dispensed on the patterns.
- *Transforming prototype* via dependability-assured evolution. A prototype is represented as a hierarchy of networks of structured objects with semantic (formal) constraints, which allows software transformation. The constraints provide formal method and means to *reason about* the validity of transformations. Both functional behaviors and non-functional properties are technically specified, based on which a system can be evolved in dependability-assured way.
- *Verifying system property* via quantified constraints. System integrity refers to the correctness of the software-intensive system operation with respect to the interfaces and interactions among distributed components. The system must satisfy non-functional properties, such as compositionality, autonomous synchronization, timing and resource constraints. Especially, non-functional properties imposed by the compositional architecture and application requirements concretely embody the high-level dependability.

C.1.2 Technical Barriers

To transform a rapid-prototyped system into the highly dependable system will involve formal methods that are used to model systems and to quantify desired properties, such as granularity of components, heterogeneity of interactions and quantified constraints on the interfaces among components. A rapid-prototyped system can be described in PSDL[Luqi88-2] as a hierarchy of networks of structured objects with semantic constraints. Theoretically, the structure allows us to transform a software and the semantic constraints, including the extensional and intentional behavior of the system, provides formal method and means to *reason about* the validity of transformations. Software transformations and formal methods have been extended towards performance and reliability evaluation[Clar96], however, the big gap between requirements and implementations results in following issues that must be solved.

- Perspective confusion problem: software-intensive systems inevitably involve different stakeholders: application customers, software architects / engineers and system implementers. Each kind of stakeholders will share a different perspective.
- Model construction problem: a formal model is needed to reflect different perspective, but how to, with dependability-assurance, transform one kind of perspective into the other becomes crucial, because there exists a big gap between software requirement and system implementation.
- Attribute identification problem: In macro view, the dependability of software systems is abstracted as availability, reliability, safety, confidentiality, integrity and maintainability. How to transform qualitative global requirement of dependability into quantitative constraints becomes the key.
- Software tool support problem: to transform intellectual models into automatic analysis will be the main challenge. Formal method and suitable representation for reasoning and manipulation by CASE tools hold the promise for mechanical analysis process.

C.2 Technical Approach

In order to construct highly dependable embedded systems, DAST try to quest for such questions: how to *assuredly capture* the macro dependability from stakeholder' s informal needs, how to *accurately quantify* micro dependability in the formal model, and how to *mechanically analyze* those attributes and *reason about* the correctness of the eventual-evolved system via software transformation.

C.2.1 Strategic Approach to Highly Dependable Embedded Systems

The proposed research is to provide a systematic solution toward dependability-assured transformation. The solution is characterized as **one formalized core** that specifies quantitative constraints of micro dependability, **two evolutionary cycles** that transform software requirements into system implementation via rapid prototyping and software architecting, and **three perspective tiers** that shift the focuses on different perspectives, stated in Fig. 1.

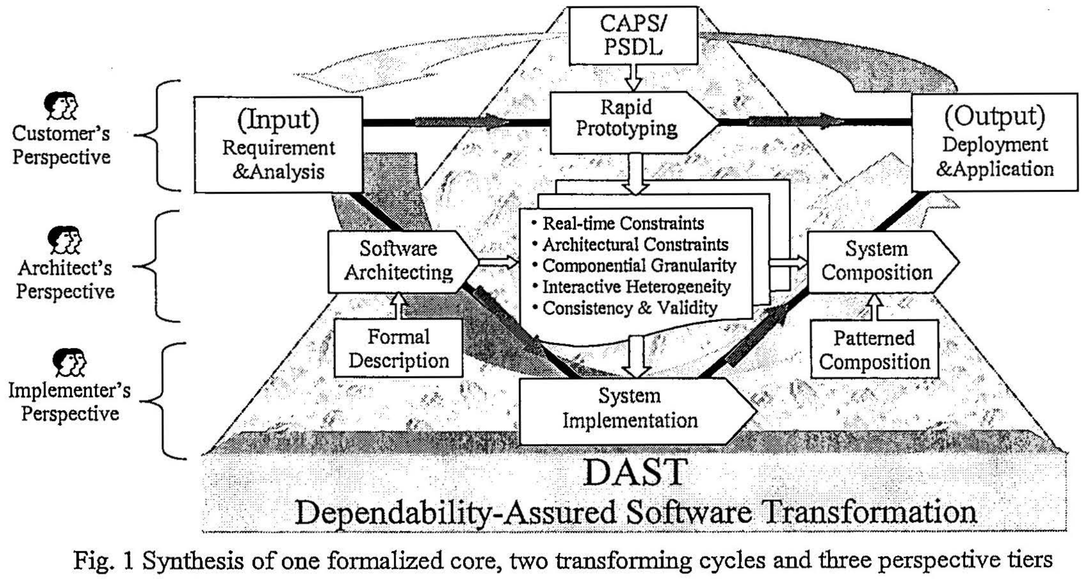


Fig. 1 Synthesis of one formalized core, two transforming cycles and three perspective tiers

C.2.1.1 One Formalized Core

The *formalized core* is closely associated with a formal model that is used to identify the components from which the system is built, and interaction among those components, patterns to guide their composition, and constraints on these patterns. The dependability of software-intensive systems is viewed as *qualitative requirements* and *quantitative constraints*. The former brings forth concepts such as availability, reliability, safety, confidentiality, integrity and maintainability with following the path of system decomposition, while the latter are associated with such quantitative constraints as real time constraints, architectural constraints, componential granularity, interactive heterogeneity, consistency and compatibility of interfaces among components and so forth. Furthermore, the formal model is used to collect quantitative constraints from both rapid prototyping and software architecting [Luqi88-1, Luqi88-2, Kram93, Luqi93,Lxzh01-1,Lxzh02], so as to provide patterns to guide system implementation and composition.

C.2.1.2 Two Evolutionary Cycles

The *rapid prototyping cycle* comprises rapid prototyping for reiterating requirement and analysis, specification and validation [Luqi88-1, Luqi88-2]. Rapid prototyping is particularly effective for ensuring that the requirements accurately reflect the user's real needs, increasingly reliability and reducing costly requirements changes. The *incremental evolutionary cycle* comprises rapid prototyping revision, software architecting and composition, and system implementation for building compositional architecture that identify such architectural constraints as componential granularity, interactive heterogeneity, and consistency and compatibility of interfaces among components.

C.2.1.3 Three Perspective Tiers

From the point of view of system engineering, consistently engineering a software-intensive system involves customer' s requirements, technical management and engineering implementation [DoDJ00, Andr98], which is inevitably associated with customer, software architect or engineering, and system implementers. DAST allows different personnel's concerns located on different perspective tiers with formalized core centered.

Customer's perspective in functional tier emphasizes **computational activities**, i.e., what activities are needed and how their interactions are associated with workflows, networking and plans necessary to support customer's operations [DoDJ00, Luqi88-1]. *Architect's perspective* in technical tier focuses on **compositional architectures**, i.e., a set of technical factors or rules are needed to govern the arrangement, interaction and interdependencies among components[DoDJ00,Shaw96].*Implementer's perspective* in physical tier concentrates on **evolutional implementations** [DoDJ00, Lxzh01-4], i.e., how to fulfill physical components and make them autonomous and independent, undertaking the compositional architecture with the constraints on the interfaces and interaction among components.

C.2.2 A Formal Model for Compositional Architecture

In order to provide a formal model for compositional architecture that is used to capture quantitative constraints, semantics of concurrent behavior, and architectural properties, DAST initiatively introduces architectural design entities known as Computer Software Compositional Pattern (CSCP). Assuredly, a CSCP involves three kinds of important factors with constraints dispensed on: ***role*** that component plays in system composition, ***style*** by which interaction is specified, and ***protocol*** used for communication during building interconnection among components.

C.2.2.1 Patterned Compositions

The explicit treatment of software architecting and composition makes CSCP first-class design entity [Lxzh01-1, Lxzh01-4] that is designed for specifying collaborative roles, architectural styles and communicative protocols. A CSCP is used to build interconnection between two collaborative roles via specific architectural style while complying with specific communicative protocol, so that the interconnection between real components can be built by gluing the related role with the component.

Fig. 2 illustrates that: for a given interaction between two components, these two components will play specific roles ***r***, architectural style ***s*** specifies how one component interacts with the other, while communicative protocol ***p*** builds specific communication channel for transporting data during the interaction. In order to construct the components as autonomous and independent entities, the CSCP assigns two roles as the representatives for the concrete components. On behalf of the components, the roles will deal with interactive behaviors (non-functional properties) and let the components have more flexibility in implementing their functional activities.

Supposing that there are three sets: ***R*** { role pairs that interact with each other }, ***S*** {architectural styles in which interaction performs}, and ***P*** {communicative protocols specified for given interaction}:

R = {		S = {	P = {
(Caller,	Definer),	Explicit-invocation,	Parameter-passing,
(Announcer,	Listener),	Implicit-invocation,	Access-memory,
(Outflow,	Inflow),	Pipe-filter,	Dataflow-stream,
(Source,	Repository),	Rep-knowledge,	Sampled-stream,
(Request,	Reply),	Interoperable distribution,	Datagram-stream,
… …		… …	… …
}		}	}

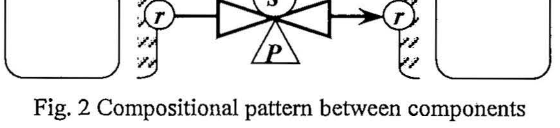


Fig. 2 Compositional pattern between components

A composition is defined by that there exists an interaction between role pairs via a style while complying with a protocol, so cross product $\boldsymbol{R} \times \boldsymbol{S} \times \boldsymbol{P}$ collects all possible compositions:

$$\text{Compositions} = \boldsymbol{R} \times \boldsymbol{S} \times \boldsymbol{P}$$

A computer software compositional pattern will involve some quantitative constraints in order to guarantee desired properties of the interaction between two collaborative roles. The constraints on the pattern have several aspects: consistency, validity and effectiveness. For example, an interaction between two collaborative roles should be done via specific architectural style while complying with specific communicative protocol, which states both consistency and validity. With respect to effectiveness, for instance, there is hard real time constraint on role *Producer*: *Producer* must provide dataflow stream within *maximum execution time*[Luqi93], to satisfy needs for role *Consumer* to consume the data. So compositional patterns are compositions with quantitative constraints dispensed on:

$$\text{CSCP} = \{R \times S \times P \mid \text{<constraints on role pairs, style and protocol>}\}$$

That is, a CSCP is defined as a relation on $R \times S \times P$ with quantitative constraints that are reasonably dispensed on roles, styles and protocols. Typical compositional patterns are listed as follows:

$\text{CSCP}_{\text{procedural-call}}$	= { (*Caller*, *Definer*),	Explicit-invocation,	Parameter-passing	}
$\text{CSCP}_{\text{Event-driven}}$	= { (*Announcer*, *Listener*),	Implicit-invocation,	Access-memory	}
$\text{CSCP}_{\text{pipeline}}$	= { (*Outflow*, *Inflow*),	Pipe-filter,	Dataflow-stream	}
$\text{CSCP}_{\text{Repository}}$	= { (*Source*, *Repository*),	Rep-Knowledge,	Sampled-stream	}
$\text{CSCP}_{\text{Interoperable}}$	= { (*Request*, *Reply*),	Int-Distribution,	Datagram- stream	}

The most important factors in CSCP are collaborative roles that will be on behalf of the components to interact with the other. Collaborative roles are formally abstracted as *generalized role wrappers* (GRW) -- a kind of generic and abstract class (in object-oriented philosophy), so that concrete components can be wrapped for building interconnections among them via the wrappers. The formal CSCP is defined as Fmt0.

Fmt0: Computer Software Compositional Pattern

$$\text{CSCP}(\boldsymbol{R}, \boldsymbol{S}, \boldsymbol{P}) = \{(r_i, r_j) \in \boldsymbol{R},\ s \in \boldsymbol{S},\ p \in \boldsymbol{P},\ \text{grw}(r_i) \longrightarrow\!\overset{S}{\underset{p}{\bowtie}}\!\longrightarrow \text{grw}(r_j) \mid \text{Const}(r_i, r_j, s, p)\ \}$$

Where

- grw(r): role r is wrapped by generalized role wrapper, which represents the separation of non-functional behavior (GRW performs) from the functional behavior (role component performs)
- $\longrightarrow\!\overset{S}{\underset{p}{\bowtie}}\!\longrightarrow$: the interaction among role-wrapped components is specified via specific architectural style while complying with specific communicative protocol for data transportation.
- Const(r_i, r_j, s, p): constraints dispensed on roles, styles, and protocols are used to build a compositional relation on the cross product $\boldsymbol{R} \times \boldsymbol{S} \times \boldsymbol{P}$.

Compositional patterns provide a good level of abstraction not only for reasoning about the desired properties of highly dependable embedded systems, but also for dependability-assured transformation from a rapid-prototyped system into the eventual-evolved system.

C.2.2.2 Generalized Specification

The CSCP (compositional patterns) in DAST are treated as reusable architectural entities that involve generalized role wrappers to enforce interactions among components via specific architectural styles while complying with specific communicative protocols. Furthermore, generalized role wrappers need more genericity so that physical components can be evolved with ease. The formal specification of compositional patterns provides excellent reusability with parameterized template, abstracted class and actively glued collaboration, stated as follows:

- **Parameterized template:** a compositional pattern is treated as a generic module (e.g., generic package in Ada or template in C++) and allows generic parameters to be substituted via instantiation, such as constants, types and procedures to ensure the reusability for generalized role wrappers.
- **Abstracted class:** a generalized role wrapper is defined as an abstracted class that is designed for adherence to restricted, plug-compatible interfaces for composition; some of interfaces that are characterized as functional behaviors of components are expected to override during evolving the components.
- **Actively glued collaboration:** the glued collaboration is only concerned with the instances of generalized role wrappers that are autonomously and concurrently executed [Lxzh98]. Computational behavior of generalized role wrappers is described by CSP-based notation[Lxzh89, Lxzh01-4] that is suitable for automatic generation of concurrent control in threads (Java) or tasks (Ada95) and deadlock detection.

The typical CSCP is ***Pipeline*** that exhibits excellent architectural properties (e.g., loose component coupling, asynchronous communication, possible data buffering), and is used to enforce interaction between components with dataflow stream, shown as Example 1. So, the parameterized CSCP provides well-defined template for generalized role wrappers, and we can instantiate different kinds of pipelines (see Example 2).

Example 1. Pipeline pattern

```
compositional Pipeline is
generalized
   type Data is private;
   Size : Integer : = 100;
   procedure Consume(d: Data);
style as <#pipe-filter#>;
protocol as <#dataflow-stream#>;
wrapper as CSCI
  role Outflow is
  port
    procedure Output(d: Data);
    procedure Produce(d: Data) is abstract;
  computation
    Produce (d);
   *[ Output (d) → Produce (d) ∇  met() →exception; ]
  end Outflow;
  role Inflow is
  port
    procedure Input(d: Data);
    procedure Consume(d: Data) is abstract;
  computation
     *[ Input (d)→ Consume (d)∇  mrt() →exception; ]
  end Inflow;
collaboration
   Outflow• Produce(d);
  *[Outflow•Output(d)            → Outflow•Produce(d)
   ▯ not Buffer•Full             → Buffer•Put (d)
   ▯ not Buffer•Empty            → Buffer•Get (d)
   ▯ Inflow•Input(d)             → Inflow•Consume (d)
   ]
end Pipeline;
```

Example 2. Instances

```
-- Instance Pipeline for integer data item:
compositional Int-Pipeline is new Pipeline (
    Data => Integer,                -- dataflow with integer
    Size => 300,                    -- 300 items buffered
    Consume=> I-Consume             -- specific procedure
 );


-- Instance Pipeline for Adt data item:
compositional Adt-Pipeline is new Pipeline(
    Data => Adt,                    -- dataflow with Adt
                                    -- default buffer size
    Consume=>A-Consume              -- specific procedure
 );
```

With respect to computational activity of components, CSP-based semantics provides not only synchronous constraints but also asynchronous control transit, for instance: *[Output (d) → Produce (d) ∇ *met*() →**exception;**]

Both Output(d) in *Outflow* (a role wrapper) and Input(d) in *Inflow* are treated as execution guards that coordinate concurrent synchronization. For instance, with considering real-time constraints, the role *Outflow* is subjected to *maximum execution time* (*met*) and the role *Inflow* is subjected to *maximum response time* (*mrt*). Both *met*() and *mrt*() are transformed as asynchronous control transit for hard real time constraints. That is, when outputting a produced data onto the given **Pipeline**, the role *Outflow* requires to be synchronized within *met*(), otherwise the synchronization is considered failure and MET_EXCEPTION is triggered (∇ represents asynchronous select). Similarly when inputting a data from the pipeline, the role *Inflow* requires to be synchronized within *mrt*(), otherwise the synchronization is considered failure and MRT_EXCEPTION is triggered.

C.2.2.3 Constraints on Patterns

A CSCP transforms the interaction between physical components into the interaction between generalized role wrappers via specified style while complying with specified protocol. In this way, many non-functional constraints can be quantified as measurable attributes and dispensed on roles, styles and protocols, respectively. For instance, a given CSCP involves role pairs, e.g., (*Producer*, *Consumer*), specific style and protocol. The interconnection between *Producer* and *Consumer* holds some significant attributes suitable for automated analyzing and reasoning:

- Consistency: a CSCP provides adherence to restricted, plug-compatible interfaces for composition, checking some attributes[Lxzh01-4,Lxzh01-5] can reveal consistencies, including *style consistency* (for interaction), *protocol consistency* (for communication), *port-computation consistency* (for gluing collaboration) and so forth.
- Compatibility: a CSCP is also associated with *port-role compatibility* (for wrapper and components), *granularity-responsibility compatibility* (for composition), *heterogeneity-coupling compatibility* (for architecting), *parameter-substitution compatibility* (for generalization-instance).
- Timing constraints: GRW can be subjected to timing constraints which are specified by giving bounds on the durations of various kinds of time intervals[Luqi88-2, Luqi93], such as *maximum execution time* (met), *maximum response time / minimum period* (mrt / mp), period and finish within, and *precedence constraints* (for time-critical schedule).
- Synchronous constraints: most embedded systems are autonomous systems that are constructed as loosely coupled sets of concurrent entities. *Synchronization* between concurrent entities is needed to ensure proper functioning of the system. It is difficult to track down the sources of synchronization bugs in using traditional code inspection techniques.

- Deadlock free: interoperability between autonomous components will result in asynchronous control transit and restricted scheduling process, especially the computing is associated with timing constraints[Luqi88-2, Luqi93]. Deadlock detection involves *roles deadlockfree* (for wrapper computation), *glue deadlockfree* (for collaboration)[Lxzh02, Lxzh89]

C.2.3 Architectural Properties with Software Architecting

A software-intensive system will involve hierarchical-grained *components* from which the system is built, and *interaction* among those components, *patterns* to guide their composition, and *constraints* on these patterns[Shaw96]. Engineering a system with software architecting and composition is to capture such architectural properties as componential granularity and interactive heterogeneity, with well-designed patterns to guide the composition from components and quantified constraints to be dispensed on the patterns. DAST will specify components (operators in PSDL) with granularity, and interconnections (edges in PSDL) among operators with heterogeneity, because both granularity and heterogeneity are metric factors for consistently engineering software-intensive systems. By specifying architectural properties, we can assess the quality and reliability of software and predict future maintainability of the product.

C.2.3.1 Granularity via Hierarchical Decomposition

Introducing *Collaboration-Mission-Function-Task* responsibility schema shown in Fig. 3, we can decompose the system into hierarchical-grained components. The components of different granularity will undertake different responsibility and require specific interaction with other components. Good granularity is a key factor for increasing productivity because it improves the understandability, reliability and maintainability.

As the system is hierarchically decomposed, granularity naturally adheres to the decomposed components according to the responsibility the component undertakes in the system. The significant granularities for hierarchical components are described as follows:

Granularity = { CSCS, CSCI, CSCC, CSCU | <*according to responsibility schema*> } or

```
Granularity={ CSCS,  -- Computer Software Complex System is the top-level component and undertakes global
                     -- activity characterized as distributed and concurrent collaboration.
              CSCI,  -- Computer Software Configuration Item is the 1st level component and undertakes specific
                     -- mission as a part of the CSCS collaboration.
              CSCC,  -- Computer Software Common Component is the 2nd level component and undertakes
                     -- specific function as a part of the CSCI mission.
              CSCU   -- Computer Software Computing Unit is the 3rd level component and undertakes specific
                     -- task as a part of CSCC function, essentially supported by programming facilities such as
            }        -- Ada packages, C++ / Java classes.
```

From the point of view of system-of-systems, each of CSCI components can be seen as a subsystem (embedded system), so that CSCS composed from those CSCIs is a system-of-systems. In this way, the componential granularity will determine important properties, for instance, the component of CSCI granularity should be developed as an autonomous and intelligent entity that takes autonomous responsibility, while the component of CSCU granularity will perform relatively-simple task, so that it does not need to be autonomous any more.

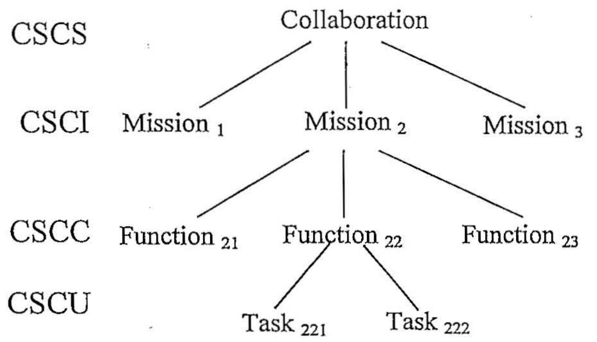


Fig. 3 Hierarchical Responsibility Schema

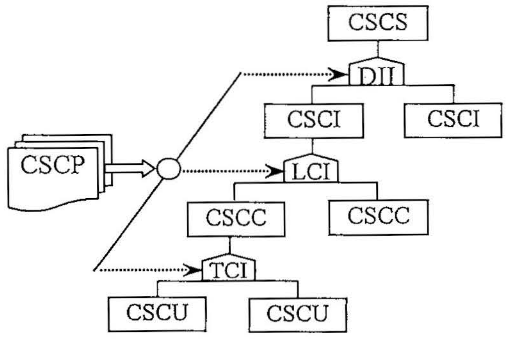


Fig. 4 Heterogeneous composition

C.2.3.2 Heterogeneity via Taxonomic Composition

Interactive heterogeneity is inevitable, because most fundamentally, different architectural styles have different strengths and weaknesses, and a system architecture should be chosen to fit the problem at hand [Shaw98]. Just as inevitably, diverse components and systems will have to work together. Distinct architectural styles often require different component packaging and interactions; these complicate the interoperation problem.

Since components of different granularity will inevitably require specific interaction with other components, we classify interconnections among hierarchical-grained components into heterogeneous-coupled interactions, according to *Distributed-Loose-Tight* coupling schema, shown in Fig. 4. The interaction of the heterogeneity will have different performance, and allow the same component interacts with other components in proper ways. For instance, a component playing role *Announcer* and role *Producer* interacts with a *Listener* component via event, and with a *Consumer* component via dataflow, respectively. Interactions via event or dataflow embody the interactive heterogeneities. As the system is decomposed into components of different granularity, interactive heterogeneity will adhere to the interconnections among components according to the heterogeneous coupling needs in the system. The significant heterogeneities for taxonomic interactions are described as follows:

Heterogeneity = [DII, LCI, TCI, HEI | <*according to coupling schema*>] or

Heterogeneity={DII, -- Distributed Interoperable Interactions are used for CSCS-composition from CSCI
-- components to enforce distributed interactive collaboration (maybe platform-cross
-- computing)

LCI, -- Loosely Coupling Interactions are used for CSCI-composition from CSCC components to
-- encourage flexible configuration with minimal communication between components (maybe
-- language-independent programming).

TCI, -- Tightly cohesive interactions are used for CSCC-composition from CSCU components to
-- emphasize independent partition of components, with low external and high internal
-- complexity

HEI -- Heterogeneous interactions are used for the composition from different-grained components
-- to advocate heterogeneous interaction among them, e.g., the same component interacts with
other components via *Implicit-invocation* and *Pipe-filter*

}

In order to enforce interaction among hierarchical-grained components, some architectural facilities as well as the way in which how interactions are performed are needed. For instance, ***Pipe*** in Unix is the special facility that enforces two more processes to interact with the others so as to systematically compose the new software from the processes. The reusable architectural entity library built from all kinds of compositional patterns can ensure the heterogeneity to meet the needs for different interaction. Fig. 4 states that interactive heterogeneities can effectively enforce heterogeneous composition from those hierarchical-grained components.

C.2.3.3 Compatibility between Granularity and Heterogeneity

In quantifying componential granularity and interactive heterogeneity, there exists compatibility between granularity and heterogeneity. That is, a given interaction of specific heterogeneity can only applied to promote the composition from the components of specific granularity, which means compatibility between granularity and heterogeneity.

According to the responsibility schema, componential granularity taking different responsibility reflects not only the complexity of internal structure but also complexity of external interconnection. Obviously, the interaction of DII is more complex than that of LCI. Architecturally, interactive heterogeneity is one of important factors by which the granularity is determined.

For instance, Pipeline and Event-based invocation belong to loosely coupling interactions but which embody different interactive heterogeneities, because of Pipeline via *Dataflow* while implicit-invocation via *Event.* This can provide guide that CSCC could be involve asynchronous access buffer, event trigger mechanism and so forth.

C.2.4 Real-Time Constraints with Rapid Prototyping

The requirements of hard real-time systems include timing constraints that must be met in the worst case for the system to be considered correct [Luqi93]. Rapid prototyping can be accomplished using CAPS/PSDL to aide the designer in handling hard real-time constraints[Luqi88-1, Luqi88-2]. The time critical operations are modeled with *maximum execution times* (MET), *maximum response times* (MRT), *minimum periods* (MP) and *period and finished with* (PFW), all of which are considered measurable attributes for highly dependable embedded systems.

C.2.4.1 Quantitative Constraints on Operators

Timing constraints for operators include maximum execution times, maximum response times and minimum period, and so on. According to the firing way by which the operator is triggered, operator will be sporadic and periodic. Any component can be subjected to timing constraints that are specified by giving bounds on the durations of various kinds of time intervals. The significant timing constraints are described as follows:

Timing-constraint = {MET, MRT, MP, PFW | <*according to firing way*> }

C.2.4.2 Data Stream Properties on Edges

In order to build interconnections among operators, PSDL exploits data stream that carries instance of abstract data type associated with the edges. Data stream is categorized into dataflow and sampled stream [Luqi93]. Considering the demand for network-centric and concurrent distributed computing, we will extend a new stream for PSDL: datagram stream. In distributed computing environment, the stream over the internet / intranet is only a stream packet without the meaning of data value. According to the properties that data stream is transported and used on edges, the significant data streams are described as follows:

Data-Stream = {Dataflow, Sampled, Datagram | < *transportation and usage*>}

Data streams on edges in PSDL can be translated into architectural properties, that is, a data stream can be specified as communicative protocol with specific architectural styles in the formal model for compositional architecture.

C.2.4.3 Constraints Dispensed on Compositional Patterns

In order to specify real-time constraints on operators and data streams on the edges with compositional patterns, a formal model is created for compositional architecture, so that those properties in PSDL should be dispensed on the three important factors in the compositional pattern: generalized role wrapper, architectural styles and communicative protocols.

Because the semantic behavior of components in a distributed embedded system is concerned with real-time constraints, the generalized role wrappers are subjected to real-time constraints in their semantic behavior. Example 1 shows the idea about how to specify timing constraints in the computational behavior in CSP-based description.

Compared with specifying real-time constraints as semantic behavior, data streams are easier to be specified by communicative protocols associated with the specific architectural style. For instance, below compositional patterns can more conveniently establish "edges" (interconnections) among operators:

CSCP $_{\text{pipeline}}$	= {	(*Outflow, Inflow*),	Pipe-filter,	Dataflow-stream }
CSCP $_{\text{Repository}}$	= {	(*Source, Repository*),	Rep-Knowledge,	Sampled-stream }
CSCP $_{\text{Interoperability}}$	= {	(*Request, Reply*),	Interactive-Distributed,	Datagram-stream }

C.2.5 Formal Method for Dependability-Assurance

The term "formal method" includes a number of different activities, including formal specification, specification analysis and proof, transformational development and program verification[Iams01]. All of these activities depend on a formal specification of the software that holds quantitative attribute for dependability.

This proposed research is to develop the formal specification to create a formal model for compositional architecture, and based on which the three perspectives can be transformed so as to reflect different personnel's concerns. Because of well-defined inherent relationships among those perspectives, it is practical for different perspectives to be evolved by means of software transformation and analysis tools[DoDJ00,Andr98].

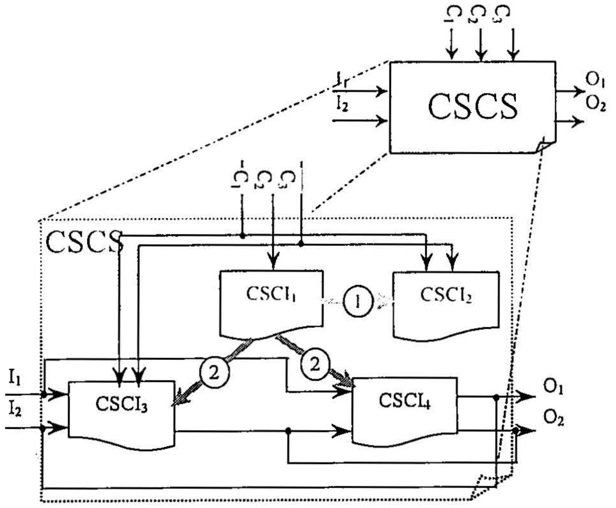
C.2.5.1 Multiple Perspectives

Functional Perspective (FP)

FP is a useful tool to acquire stakeholder' s needs, based on which requirements and analysis is developed. FP is represented as hierarchical networks of components with specific responsibility and semantic constraints. For highly dependable embedded systems, FP is concerned with macro dependability such as availability, reliability, safety, confidentiality, integrity and maintainability. The formal FP, involving three elements: components from which the system is built, interconnections enforcing interactions among components, and constraints on both components and interconnections, is defined as Fmt1.

Fmt1: Functional Perspective:: FP = [COM, INT, Const(COM, INT)]
Where:
- COM: Components from which the system is built
- INT: Interconnections that enforce interactions among components
- Const(COM, INT): Constraints on components and interconnections

As the system is decomposed into low-level components, interconnections need specifying as interactions among those components. Since decomposition follows the uniform method in PSDL, the data stream (edges) among components are mapped into interactions among the components, which is shown as Fig. 5.



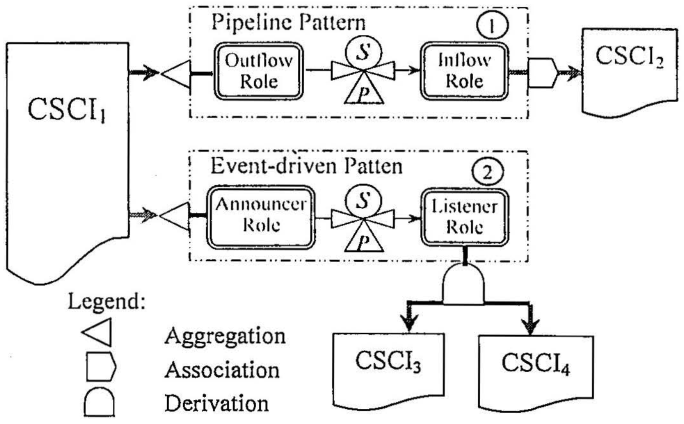


Fig. 5. Decomposition and Functional Perspective

Fig. 6. Technical and Physical Perspective

Technical Perspective (TP)

TP provides rigorous formal specification for specifying quantitative constraints on the compositional architecture. For a given system, TP is essentially a set of CSCP (compositional patterns) that are characterized as collaborative roles, architectural styles and communicative protocols. For highly dependable embedded systems, TP is concerned with micro dependability (quantitative constraints) such as componential granularity and interactive heterogeneity (architectural properties), timing constraints (real-time properties) and synchronous constraints (distributed concurrent properties). Some of these properties can be abstracted as measurable attributed in temporal logic that fits with model checking techniques [Clar96]. The formal TP, represented as hierarchical-grained *components*, from which the system is built, and *interactions* among components, *patterns* to guide their compositions, and *constraints* on these patterns, is defined as Fmt2.

Fmt2: Technical Perspective:: TP = [GRW(***R***) $\xrightarrow[P]{S}$ GRW(***R***), CSCP(***R, S, P***), Const(***R, S, P***)]
Where:
- GRW(***R***): under compositional architecture, components play specific roles abstracted as generalized role wrappers (GRW) and GRW provide adherence to restricted, plug-compatible interfaces for composition.
- $\xrightarrow[P]{S}$: interactions among role-wrapped components are specified via specific architectural style while complying with specific communicative protocol for data transportation.
- CSCP(***R,S,P***): compositional patterns is used to guide system composition from components and essentially characterized by three factor sets R, S and P,
- Const(R, S, P): constraints mapped on composition patterns will be naturally dispensed on three essential factors: collaborative roles, architectural styles and communicative protocols.

By identifying interactive heterogeneity among the decomposed components, the TP will capture the architectural features via compositional patterns [Lxzh01-1,Lxzh01-4]. Furthermore, TP provides generalized role wrappers with adherence to the restricted, plug-compatible interfaces for composition, which is shown as Fig. 6. Since architectural properties are based on compositional patterns, semantic constraints (e.g. timing constraints and data stream) will be dispensed on generalized role wrappers, architectural styles and communicative protocols (see Example 1).

Physical Perspective (PP)

PP is useful tool for system implementers to evolve physical components by means of generalized role wrappers. PM is represented as a set of physical components and object-oriented relationships with GRW, such as association, derivation and aggregation, For highly dependable embedded systems, PP is mainly concerned with generalized role wrapper and redefinition or override of restricted, plug-compatible interfaces under the support of reusable architectural facilities provided by compositional patterns (see Fig. 6). The formal PP, closely associated with generalized role wrappers, is defined as Fmt3.

Fmt3: Physical Perspective:: PP = [COM u GRW($\boldsymbol{R}$), INT($\boldsymbol{R}$, $\boldsymbol{S}$, $\boldsymbol{P}$), Const($\boldsymbol{R}$, $\boldsymbol{S}$, $\boldsymbol{P}$)]

Where:

- COM $\sqcup$ GRW(R): COM and GRW(R) fulfill the physical components to meet the requirements of their externally visible behaviors and the interconnection among them.
- INT(R,S,P): interconnections are represented as interactions between collaborative roles via architectural styles while complying with communicative protocols under the compositional architecture.
- Const(R,S,P): constraints mapped on composition patterns will be naturally dispensed on their essential factors: collaborative roles, architectural styles and communicative protocols.

The evolutional process from technical perspective (the formal model for compositional architecture) to physical perspective involves three relationships with the generalized role wrappers: association, inheritance and aggregation. Three evolutional methods are stated as follows:

- Association allows components to be associated with architectural properties (from GRW) in order to refine their own functional behaviors.
- Inheritance allows components to be derived from correspondent GRW (for architectural properties) in order to extend their functional behaviors.
- Aggregation allows components to aggregate more than one set of architectural properties (from GRW) in order to refine their own functional behaviors.

C.2.5.2 Transformations among Perspectives

Based on the formalized core (Fig. 1) and formal specification provided by compositional patterns, the transformations among three perspectives create visual scenarios by software tools, that is, our focus on compositional architecture (TP with constraints specified) can be zoomed-out on functional activities (FP with qualitative requirements analyzed) or zoomed-in on physical implementation of components and their links (PP with constraints implemented). Fig. 7 illustrates software transformation among multiple perspectives.

Collecting micro dependability from rapid prototyping and software architecting process, the technical perspective specifies them as formal semantics, measurable attributes in tangible compositional patterns. As an intermediate perspective, TP is not only the evolutional basis from which PP is transformed but also the formal foundation on which model checking is performed to verify the correctness of the eventual-evolved system.

Architecting Extension for Rapid Prototyping

DAST extends PSDL with compositional patterns to create a compositional architecture which is characterized as collaborative roles, architectural styles and communicative protocols. Generalized role wrappers are used to glue the components between which interactions are made. Following formula states the scenario from FP to TP.

$$\text{Architecting}^{\text{FP}\to\text{TP}} = \begin{bmatrix} \text{COM}, \\ \text{INT}, \\ \text{Const(COM, INT)} \end{bmatrix} \xrightarrow{\text{compositional patterns}} \begin{bmatrix} \text{CSCP}(\boldsymbol{R},\boldsymbol{S},\boldsymbol{P}), \\ \text{GRW}(\boldsymbol{R}) \overset{S}{\underset{P}{\longleftrightarrow}} \text{GRW}(\boldsymbol{R}), \\ \text{Const}(\boldsymbol{R},\boldsymbol{S},\boldsymbol{P}) \end{bmatrix}$$

By specifying tangible compositional patterns, the components and interconnections among them are abstracted as general role wrappers, and interaction among collaborative roles via architectural styles while complying with communicative protocols, respectively. And the constraints on components and interconnections are reasonably *dispensed* on the roles, styles and protocols.

Forward Evolution via Software Transformation

By means of object-oriented evolutional relationships with generalized role wrappers, e.g., association, derivation and aggregation, PP can be evolved from TP via software transformation. That is, the focus is shifted from compositional architecture to evolutional implementation. Following formula states the scenario from TP to PP:

$$\text{Evolution}^{TP\rightarrow PP} = \left(\begin{array}{l} \text{CSCP}(\boldsymbol{R},\boldsymbol{S},\boldsymbol{P}), \\ \text{GRW}(\boldsymbol{R}) \xrightarrow[P]{S} \text{GRW}(\boldsymbol{R}) \\ \text{Const}(\boldsymbol{R},\boldsymbol{S},\boldsymbol{P}) \end{array}\right) \xrightarrow{\text{object-oriented evolution}} \left(\begin{array}{l} \text{INT}(\boldsymbol{R},\boldsymbol{S},\boldsymbol{P}), \\ \text{COM u GRW}(\boldsymbol{R}), \\ \text{Const}(\boldsymbol{R},\boldsymbol{S},\boldsymbol{P}) \end{array}\right)$$

Under the support of reusable architectural facilities provided by compositional patterns, physical components wrapped by generalized role wrappers only need to extend or refine their own functional activity. That is, the evolved components will be associated with GRW, or will be derived from GRW, or will aggregate one more GRW, and then refine (extend) their own functional activities.

Reverse Extraction via Software Analysis

By means of source code analysis, many properties can be extracted from PP so that the formal model for compositional architecture can be built. Those properties include topological structure among synchronization (concurrent properties), precedence constraints (real-time schedule), components and class relationship among them (association, inheritance and aggregation). Following formula states the scenario from PP to TP:

$$\text{Extraction}^{PP\rightarrow TP} = \left(\begin{array}{l} \text{INT}(R,S,P), \\ \text{COM u GRW}(R), \\ \text{Const}(R,S,P) \end{array}\right) \xrightarrow{\text{source code analysis}} \left(\begin{array}{l} \text{CSCP}(R,S,P), \\ \text{GRW}(R) \xrightarrow[P]{S} \text{GRW}(R) \\ \text{Const}(R,S,P) \end{array}\right)$$

By identifying three kinds of factors for compositional patterns, generalized role wrappers can be extracted from the physical implementation, which is the crucial step to specify the interconnections among components.

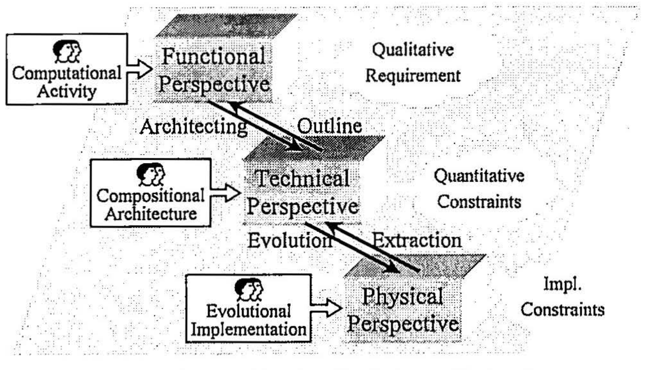


Fig. 7 Transformation: Architecting, Evolution and Extraction

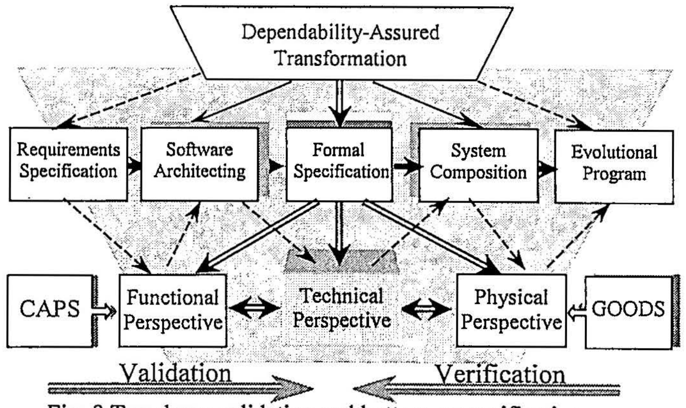


Fig. 8 Top-down validation and bottom-up verification

C.2.5.3 Formal Verification on Formal Model

Verification 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. V & V is an expensive process for highly dependable systems because which are associated with complex non-functional constraints[Iams01].

Based on the formal specification for compositional architecture, we are to develop a set of software tools that provide automatic analysis of the source text of a system or associated document in order to analyze and check system representations such as the requirements document, design diagrams and the program source code, which is illustrated in Fig. 8.

Validation is well stated as "Are we building the right product" [Boem81], which refers to a different set activities that ensure that the software that has been built is traceable to customer requirements[Iams01]. Requirements validation is concerned with showing that the requirements actually define the system which the customer wants. CAPS[Luqi88-1] is a software engineering environment that provides necessary tools for engineers to quickly develop, analyze, and refine real-time software systems. The environment is useful for requirements analysis, feasibility studies, and the design of large embedded systems.

Formal verification and automated analyses are static V & V techniques as they do not require the system to be executed. Automated source code analysis and formal verification used to check the correspondence between the eventual-evolved system and its specification (verification). Translating source code to highly-abstracted finite-state automaton for model checking is a practical approach [Jame00,Davi00,Matt00], 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.

GOODS[Lxzh91] is a Graphical Object-Oriented Development and analysis environment that provides two-way software transformations: topological structures, programming units, and implementation body described by conceptual diagram can be automatically generated into source code in Ada (C/C++); reverse analyzer of source code is useful to extract unit specification, body and topological structure into conceptual diagram.

Both CAPS and GOODS are extendable for dependability-assured transformation. The strategic approach is accompanied with multiple perspectives, rapid prototyping (CAPS), software analysis (GOODS), and formal verification to check that the system meets its specified functional and non-functional requirements (constraints).

C.3 General Plan of Work

C.3.1 Plan Overview

We envision the CAPS that we currently have as the starting points for additional research and extension of CAPS with software architecting via compositional patterns follows. Using rigorous mathematical formalism, we define formal model as compositional patterns on which three perspectives are constructed; using patterned composition facilities, we thoroughly study dependability-assured transformation among multiple perspectives; using rapid prototyping techniques, the formal description from functional perspective to technical perspective embodies requirement validation; using software analysis, the automated extraction of formal model from physical perspective to technical perspective will advocates formal verification of correctness of the eventual-evolved system. Related enhancements include:

1) Taxonomic interactions among components are crucial for patterned compositions that are treated as reusable architectural facilities, furthermore, the identification of interactive roles and specification of architectural styles and communicative protocols are key aspects in building heterogeneous interconnection among large-grained components.
2) Measurable attributes abstracted from rapid prototyping and software architecting are specified as timing constraints, architectural properties such as granularity for components, heterogeneity for interactions and restricted and plug-compatible interfaces for composition. And they are the basis on which a predictable level of dependability is achieved.
3) Three perspectives such as computational activities, compositional architectures and evolutional implementation derived from the formal model are used to promote dependability-assured transformation and system property analysis, so that formal verification of the correctness of the eventual-evolved systems can be performed based on the formal model with constraints dispensed on.

Test-bed facilities provide software artifacts of significant scale that represent mission-critical systems for both NASA and IT industry. As autonomous systems, they are characterized by networked distributed computing and communication, and are useful for the proposed work to collect requirements of desired properties and experiment with test-bed facilities. In our proposed research, we will use test-bed facilities in following ways:

1) Collaboration infrastructure is used to support information transportation, coordination among autonomous and concurrent components, and quantitative constraints performance checking.
2) Software artifacts of significant scale that represent mission-critical systems are used to demonstrate real-time constraints that are dispensed on the formal model for compositional architectures.
3) The background domain knowledge is used to perform experiments of micro dependability for highly dependable embedded systems on the artifacts that are well documented, packaged, and configurated.

C.3.2 Broad Design Activities

The steps in implementing dependability-assured software transformation from rapid prototyping are:

1) Study current architectural approaches, and abstract useful facilities for interoperability.
2) Build reusable architectural entities, and Create PSDL extensions with compositional patterns,
3) Study to dispense current timing and data streams in PSDL on the patterns
4) Add CSP-based semantics to describe synchronous constraints on the restricted interfaces.
5) Abstract generalized role wrappers as easy-derived and parameterized templates for code generation.

C.3.3 Deliverables

1) Rigorous formal model for compositional architecture
2) PSDL-extension language for compositional patterns and automatic transformation
3) Architecting taxonomy to enhance compositional pattern library
4) Measurable attributes adherence to technical model and automatic checking procedure
5) Prototype of software tools for software transformation and model extraction

C.3.4 Description of Procedure

Building on our strengths, we will perform the following:

1) Further develop the formal basis of reusable architectural facilities to compositional patterns, semantic formal constraints, and generalized representations based on excellent programming languages.
2) Construct prototype software tools that realize the method enabled by 1)
3) Perform formal analysis for domain-specific application
4) Design reusable architectural facility library for compositional architecture
5) Seek the mapping rules between formal specification and highly-abstracted mechanism in programming languages

The software transformation and analysis tools will aid designer to fulfill transformation among multiple perspectives. This capability will be demonstrated for typical rapid prototype of real-time embedded systems.

C.3.5 Evaluation Factors

C.3.5.1 Dependability-Assured Transformation

Presently, people recognize that non-functional properties become crucial for successful construction of highly dependable embedded systems, and computational activities can be only guaranteed under consistent and compatible interconnections among components with the quantitative constraints dispensed on the compositional patterns.

C.3.5.2 Measurable Dependability Attributes

Many quantitative attributes such as timing constraints (real-time), resources constraints (embedded applications), synchronous constraints (for concurrent distributed systems), and so on, are treated as measurable dependability attributes which are predictable via mechanical analysis.

C.3.5.3 Replaceability of computational components

Generalized role wrappers within specific compositional pattern deal with almost all non-functional properties so that interoperability between components is encouraged with loose coupling and computational components are only concerned with their own activities. For a specific component to be replaced, once it keeps the same external behaviors, the other side components that are interacting with it will not care.

C.3.5.4 Formal verification enforcement

Development of new verification methods and tools is to provide a rigorous means for checking the integrity and correctness of designs for these systems before they are deployed on target platforms. The proposed formal model for compositional architecture will be extendable basis for formal verification via the extraction of existing systems and requirement validation via rapid prototyping. Abstracting desired properties from quantitative constraints is based on the formal specification and static analysis for these properties embodies automated verifying process.

C.3.6 Schedule

1) Study current architectural approaches, and abstract useful facilities for interoperability.
2) Build reusable architectural entities, and Create PSDL extensions with compositional patterns,
3) Study to dispense current timing and data streams in PSDL on the patterns
4) Add CSP-based semantics to describe synchronous constraints on the restricted interfaces.
5) Abstract generalized role wrappers as easy-derived and parameterized templates for code generation.

C.3.7 Comparison with Other Research

Work related to the topics discussed in this proposal includes research in the areas of software transformation, modeling dependable embedded systems from multiple perspectives, software architecture, component wrapping techniques and formal verification based on the formal model.

PROJECT DESCRIPTION

In present, software research community focuses on highly dependable systems. Since system development takes place at many tiers and each tier appropriately deals with different concerns, three **perspective tiers** for different personnel are *computational activity* (for customer), *compositional architecture* (for architect) and *evolutional implementations* (for implementer), respectively. The key **factors** for consistently engineering a highly dependable system are coarser-grained *components* from which the system is built and *interactions* among them, *patterns* to guide their composition, and *constraints* on these patterns. And the **solution** toward highly dependable systems is explicit treatment of *software architecting* and *composition*, as well as *support tools*[Shaw96, Medv00, DoDJ00, Andr98].

A number of techniques, frameworks and approaches have recently emerged to address the problem of engineering highly dependable systems. In general, the widely embraced efforts have been fallen into three categories: functional perspective (e.g., rapid prototyping [Luqi88-1]), technical perspective (e.g., software architecture [Shaw96, Perr92] and JTA Framework [Andr98, DoDJ00].), physical perspective (e.g., JavaBean, COM$^+$ and CORBA[SUNM00, Orfa96, Sess97]). And they all focus on composing software systems from coarse-grained components and some of which inevitably involve formal methods. Formal methods include a number of technical activities: such as formal specification, specification analysis and proof, transformational process and program verification [Jams01].

Functional perspective is mainly concerned with requirement validation. Rapid prototyping approach [Luqi88-1] uses a computer-aided prototyping system (CAPS) and its associated prototyping language (PSDL) to aide the designer in handling *hard real-time constraints* that must be met in the worst case for the system to be considered correct [Luqi93]. CAPS provide a useful environment for requirements analysis, and feasibility studies, but which needs extending in combining with compositional architecture and evolutional implementation.

Technical perspective is generally concerned with two aspects: what kinds of rules are used to govern the interdependencies and interactions among components, and how to specify the validated requirements as *architectural* and *quantitative constraints* dispensed on the architecture. Software architecture approach typically separates computation (components) from interaction (connectors) in a system. Architectural and quantitative constraints are represented as consistency and compatibility on interfaces and interactions among components, real-time constraints, as well as synchronization semantics [Medv00, Meth00, Garl97]. Despite of this, connectors are often considered to be explicit at the level of architecture, but intangible in the system implementation [Meth00, Lxzh01-4]. JTA Framework specifies three views of information architecture and defines a set of products that describe each view, but the framework does not provide a process for architecture design [Alex00].

Physical perspective is concretely concerned with how to evolve those components that represents activities undertaking the compositional architecture into the highly dependable system. Componentware (JavaBean) and middleware techniques (COM$^+$/CORBA), assume a homogeneous architectural environment in which all components adhere to certain *implementation constraints* (e.g., design, packaging, and runtime constraints) [10, 11]. They provide an unalterable architecting mechanism that does not create to be easily extended to support heterogeneous composition.

Some of above-mentioned approaches involve formal methods with specific constraints dispensed on different perspectives, but only CAPS / PSDL allows the designer to specify measurable attributes treated as timing constraints in software design specification, and this kind of dependability can be assuredly lasted in the implementation by means of code generation via reusable components [Luqi88-1, Luqi88-2]. Current formal verification techniques are used to check the correspondence between the eventual-evolved system and its specification (verification) by means of automated source code analysis that translates source code of existing systems to highly-abstracted finite-state automaton for model checking [Jame00,Davi00,Matt00], but which can not provide full support from requirement to implementation. DAST extends CAPS with software architecting and composition to transform macro dependability into micro dependability (quantitative constraints) and uses two evolutionary cycles to cover cover full life cycle of software development.

C.4 Broader Impact

If the formal model for compositional architecture works well then the following can be take place:

1) The gap between rapid-prototyped systems and the eventual-evolved systems will be bridged by reusable architectural entities which result in the reduction of development time

2) The gap between qualitative requirements of dependability and quantitative constraints on interfaces and interactions among components will be fulfilled by compositional patterns suitable for formal verification.
3) The focus of software development is shifted from computational activities onto compositional architectures with large-grained components replaceable, which supports flexible configuration for HCES.
4) The proposed formalized core collecting two aspects of measurable attributes from rapid prototyping and software architecting will make model-intensive verification practical and reasonable.

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 IEE 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 coordination of the following plan previously stated in section C.3.6 for schedule:

1) Study current architectural approaches, and abstract useful facilities for interoperability.
2) Build reusable architectural entities, and Create PSDL extensions with compositional patterns,
3) Study to dispense current timing and data streams in PSDL on the patterns
4) Add CSP-based semantics to describe synchronous constraints on the restricted interfaces.
5) Abstract generalized role wrappers as easy-derived and parameterized templates for code generation.

C.5 Related Work

C.5.1 CAPS: Computer-Aided Prototyping System

CAPS[Luqi88-1] is a useful software engineering environment for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the PSDL, which provides facilities for modeling timing and control constraints within a software system. The CAPS data flow diagram and PSDL program can be augmented with timing and control constraint information which allows the user to model the functional and real-time aspects of the prototype. In this proposed project, CAPS is located in the rapid prototyping cycle that is responsible for requirement analysis, measurable attribute capture and generation of a pilot version of intended software system.

C.5.2 GOODS: Graphical Object-Oriented Design System

GOODS[Lxzh91] is an Ada-based software development and analysis environment that provides two-way software transformations: topological structures, programming units, and implementation body described by conceptual diagram can automatically be generated into source code in Ada (C/C++); reverse analyzer of source code is useful to extract unit specification, body and topological structure into conceptual diagram. In this proposed project, GOODS is located in the bottom of the incremental evolutional cycle that is responsible for implementation of functional components and reusable architectural entities.

C.5.3 Architectural Styles with Ada95

Software development is shifting its focus from lines-of-code to coarser-grained components. Software architecture has been proposed to respond such a high level design that invokes the elements composing systems, interactions among those elements, patterns guiding their composition, and constraints on these patterns [Shaw96, Medv00]. Since ADLs need developing in their applicability, to adapt traditional programming languages to ADLs is a significant approach. Previous researches[Shaw96,Lxzh00,Lxzh02,Lxzh01-2,Lxzh01-3] shows the possibilities that by means of adding specific architectural patterns, traditional programming languages might be improved into architectural description languages. Since Ada has been broadly used for large-scale, embedded real time, mission-critical and high reliable systems, related work has been done to firstly unify object model[Lxzh01-2,Lxzh01-3] in Ada95 and then enhance Ada to support specific architectural patterns[Lxzh00,Lxzh02], which is considered valuable for proposed model.