Summary of Deliverables and Technical Approach

The Component-Based Lifecycle

In the future, systems will primarily be constructed by composing functionality provided by pre-existing components. These systems will be adapted to changes in requirements and available components by adding new functions, eliminating functions that are no longer needed, and substituting functions provided by new, improved components for those provided by obsolete components. The infrastructure required to support this component-based lifecycle is currently emerging. Intercomponent communication mechanisms, such as CORBA and DCOM, and data interchange formats, such as XML and DOM, are now widely used in both the defense and commercial sectors. Component discovery mechanisms, such as Sun's Jini and HP's e-Speak, while not yet in widespread use, are growing rapidly in popularity. And higher-level mechanisms for collaboration among components and delegation of service requests, such as SRI's Open Agent Architecture, are becoming available.

This new component-based lifecycle presents a host of new software engineering challenges. Traditionally, high-quality software is a product of a rigorously controlled, carefully monitored development process. Confidence that a system will satisfy its requirements is based on confidence that its subsystems satisfy their requirements. If those subsystems are, say, closed-source COTS components, what is the foundation for confidence in quality? Experience shows that developers of COTS components tend to value the addition of features more highly than quality, because most of their customers also value features more highly than quality. How, then, can such components be used in mission-critical systems that have strict dependability requirements?

The increased dynamism of the component-based lifecycle is another source of software engineering challenges. The life span of systems is measured in decades, while the life span of typical components can be as little as a few weeks. Frequent new releases of COTS components in particular must be expected: this is the principal source of revenue for the developers. Thus, even if system requirements never change, the system itself must be expected to undergo constant evolutionary change. And, of course, requirements do change, as a result of changes in missions, additional service demands from the users, needs for higher dependability, and so on. Even limited, infrequent, carefully controlled maintenance of systems tends to result in decreased quality due to unanticipated effects of changes to code. Frequent replacement of components by others that offer a somewhat different interface to somewhat different functionality — and that contain somewhat different bugs — will greatly exacerbate the traditional problems of maintenance, partly because of the increased frequency of change and partly because some of the traditional tools for coping with change (such as extensive regression testing of components and source code flow analysis) will no longer be available.

Dependable System Architectures

SRI research on methods for developing system architectures with guaranteed dependability properties can be applied to the software engineering challenges posed by the component-based lifecycle in several complementary ways. The main focus of our research has been on developing techniques for establishing that architecture transformations preserve dependability properties. This technology can be applied to the design of dependable architectures, to evaluating the dependability of component-based systems, and to the dependable evolution of component-based systems.

Designing for Dependability

There are two approaches to guaranteeing that a system satisfies a dependability constraint. The difference between these approaches can be illustrated by a simple example. Consider a system composed from components with varying security levels and clearances. Security policies for such multilevel systems include constraints on communication. For example, a typical constraint is that "read-up" is not allowed. That is, a component must not read data whose classification is greater than the component's clearance. How can we guarantee that such constraints are satisfied if the system contains closed-source components whose security has not been verified? One approach is to monitor all communication among components at runtime, and check whether the security policy is satisfied in each case. In cases where the communication would violate the security policy, the communication is blocked. An alternative approach is to design a system architecture that restricts communication among components so as to reduce the need for runtime constraint checking. If the architecture is designed so that communication channels between components exist only when communication between those components is consistent with the system's security policy, then no runtime checking of the security constraints is needed.

Our research has explored the latter approach. One product of that research is SDTP, a dynamic architecture for secure distributed transaction processing. SDTP was designed by writing a simple, abstract description of the architecture, showing that the description guarantees the desired security properties, successively refining the abstract description until a directly implementable concrete description results, and showing that each refinement step preserves satisfaction of the security policy. Refinement steps were the result of applying reusable refinement transformations that codify implementation techniques. Thus, SDTP is an example of how a dependable (in this case, secure) system can be constructed from not-necessarily-dependable components, without the overhead of runtime constraint checking.

However, one key capability required for effective use of refinement transformations in design is not yet available. A transformation contains no information about when it *should* be applied. Our preliminary investigation suggests that dependability requirements influence choice of implementation transformations in a fairly straightforward way. For example, a requirement for secure communication between components suggests the introduction of modules for encrypting and decrypting message traffic between them, the use of certain key exchange protocols when new connections are established, and so on. A requirement for fault tolerance suggests the introduction of duplicate, functionally redundant structure in the architecture. And so on. Including such information as part of the refinement transformation would make our design tools much easier for nonexperts to use.

Generating Dependability Gauges by Abstraction

Formal methods, such as model checking and theorem proving, can be effective for determining whether systems have desired dependability properties. Less formal methods, such as simulation, also provide useful information about dependability, even when incapable of providing definitive answers. But, for complex systems, these methods cannot be applied directly. An abstract system model, designed specifically for the purposes of the particular dependability analysis, must be created. As much system detail as possible is abstracted away, in order to make the analysis more tractable.

So far, our application of abstraction techniques has been at design time. In the component-based lifecycle, where the system is constantly evolving, there are obvious advantages to applying them at runtime as well. For each dependability property of interest, an abstract system model that can be analyzed to determine whether the system has the property will be generated. The abstraction transformation steps will demonstrably co-preserve dependability. In other words, if the generated abstract model is dependable, then the more concrete input to the abstraction step must be dependable as well. As the system evolves, the abstract models will be updated. The dependability of the system is dynamically reassessed whenever there are relevant changes in the environment or within the system itself. Thus, these abstractions will serve as *dependability gauges* that can be used to monitor the dependability status of the system, as shown in Figure 1.

Dependable Evolution through Perpetual Design

Much as transformations can be used at design time to create a baseline architecture that has desired dependability properties, transformations can be used at runtime to ensure continued dependability as requirements, the system, and its environment evolve. We are currently developing a simple first approximation to a "perpetual (re-)design" capability for DARPA/ISO's ANTS program.

For ANTS, we are working with Sanders to develop a dynamic, agent-based radar scan scheduler. Our primary contribution will be to ensure that the resource negotiation protocol used in scheduling has certain required dependability properties (e.g., rapid convergence). The architecture for the scan scheduler will be dynamic, in that every detected emitter is represented by an agent and the number of detected emitters varies dynamically. Agents will be coordinated by one or more centralized facilitators. Since a facilitator's performance degrades as the number of agents it must coordinate increases, facilitators are potential performance bottlenecks. So, as the number of agents increases, the number of facilitators should be increased as well, and "meta-facilitators" added to coordinate the facilitators. This evolution of the negotiation architecture will be accomplished by having the addition of emitter agents trigger application of functionality-preserving, dependability-preserving, performance-enhancing transformations to an architectural model that is maintained at runtime. The transformations add facilitators and meta-facilitators to the model, and, at an appropriate later time, the architecture of the running system is upgraded to match the optimized model.

The architecture evolution of the ANTS scan scheduler is a special case of a more general dependable evolution capability. The basic idea of changes triggering application of appropri-

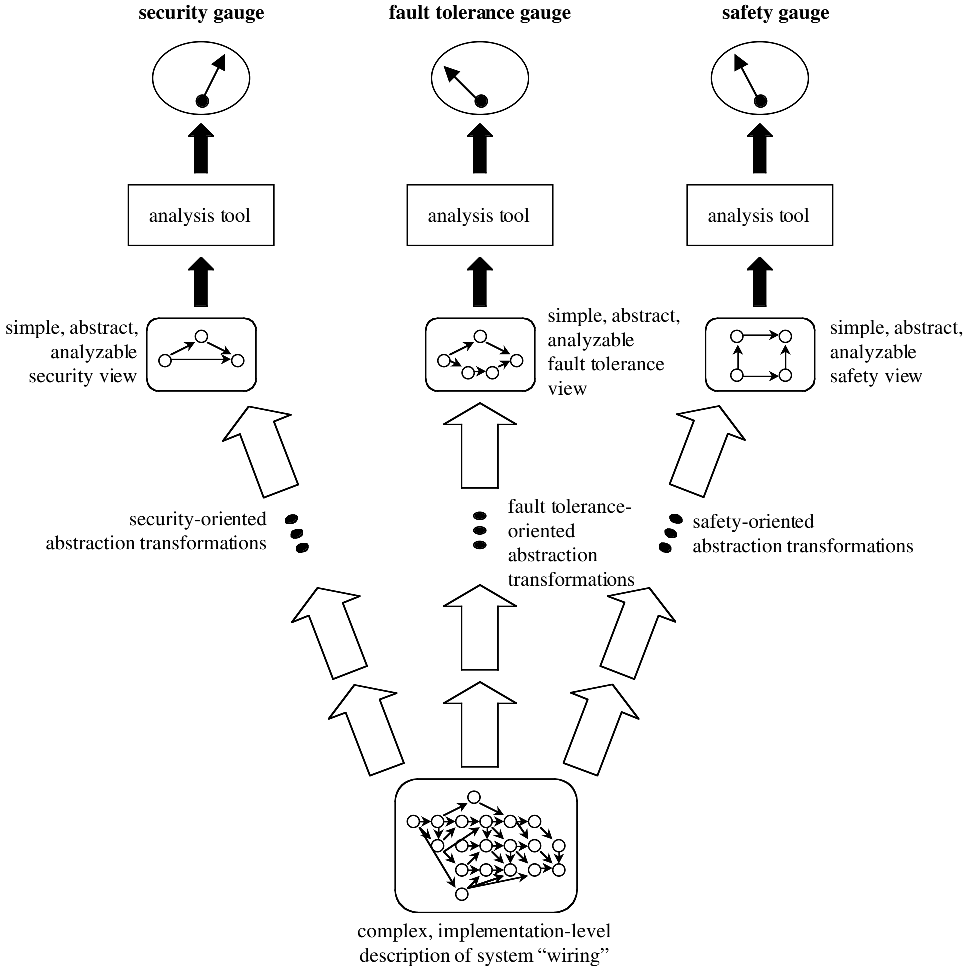

Figure 1: Dependability Gauges

ate functionality-preserving transformations is the same. But, instead of simply preserving dependability, these transformations enhance dependability. And, instead of re-architecting in response to only one type of directly observable event — addition of an agent — any of the following may trigger re-architecting.

- *A change in required dependability level.* The system administrator may decide that the system should be made more (or less) secure, more (or less) fault tolerant, more (or less) safe, etc. Architectural transformations expected to have the appropriate effect will be applied, and the results monitored using the appropriate dependability gauge, as shown in Figure 2. Other dependability gauges will be monitored as well, to ensure that other dependability constraints remain satisfied.

- *A change in desired functionality.* The system user may request that the system provide some additional service, requiring the integration of additional components or the replacement of components with other components offering additional functionality. If a new component is added to the system, or one component is replaced by another that proves less dependable, then the system architecture may have to be made more

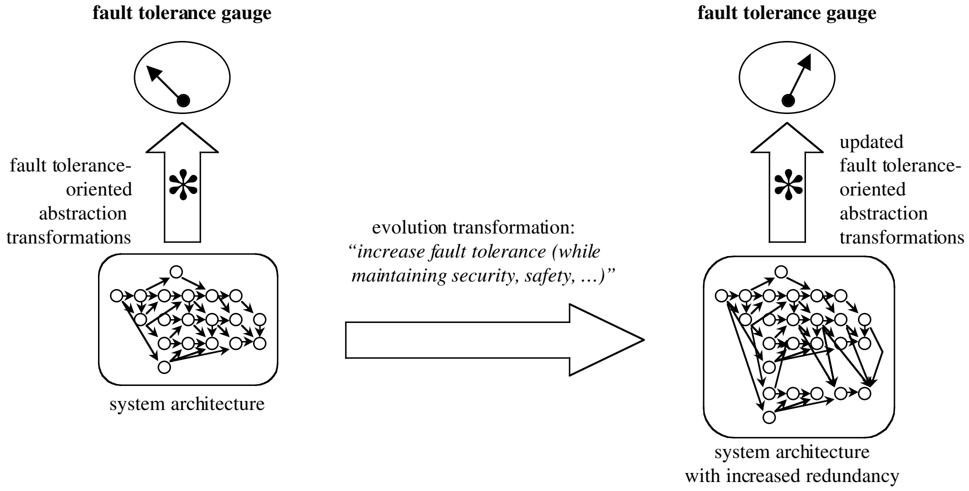

Figure 2: Re-Architecting to Improve Fault Tolerance

dependable to compensate. An example is provided by our SDTP work, where we showed that a very simple architecture can be used to securely integrate single-level databases that are all at the same level, but substantial architectural complexity must be introduced when a database at a different level is integrated into the system.

- *A change in component availability.* Changes in component availability can also necessitate addition and replacement of system components, and hence changes in system architecture.
- *A change in the system's behavior or performance.* Monitoring of dependability gauges may reveal dependability requirements that were being satisfied but are no longer being satisfied. Similarly, monitoring of performance gauges may reveal that system performance has declined to an unsatisfactory level. Architectural transformations can be applied to address such problems.

Proposed Research Tasks

1. Develop a language for annotating an architecture transformation with information about the effects — guaranteed and merely expected — of applying that transformation on system dependability gauges.
2. Extend the language in which architecture transformations are expressed to support inclusion of such annotations.
3. Extend our transformation toolset to make use of the annotations.
4. Define dependability gauges for several aspects of dependability.
5. Implement these dependability gauges for architectures described in the ACME architectural description language as a library of annotated abstraction transformations using the extended transformation toolset.

6. Evaluate the utility of this approach to the implementation and maintenance of highly dependable component-based systems by performing a case study based on a system of interest to ISO.

We also propose, in an optional third year, to develop *functionality gauges*, based on generation of abstract functional models of the system. We expect that the result will be a substantial advance in our ability to assemble dependable systems from an evolving collection of not-necessarily-dependable components.