The Software Engineering and Network Systems (SENS) Laboratory supports research in software engineering, formal methods, distributed computing, fault tolerance, and computer networking. In addition to projects within each area, the diversity in faculty expertise enables SENS researchers to address challenging problems that span multiple areas. Examples include: automated development of software for embedded systems in automobiles and aircraft; adaptive software for protecting critical infrastructures from cascading failures and cyber-attack; robust software for networks of sensors used in military reconnaissance; adaptive protocols for pervasive computing via handheld and wearable computers; and providing security and privacy in peer-to-peer computing systems. Current SENS research projects are supported by grants from NSF, ONR, and DARPA. Several projects also involve industrial support and collaboration; partners include Motorola, Lucent, Cisco, Siemens, NASA/JPL, and Texas Instruments. The Lab is equipped with interconnected high-end workstations and PC platforms, an NSF-supported cluster of symmetric multiprocessors, and a mobile computing testbed that includes wireless networks and a variety of handheld and wearable computing devices.

Model-based software engineering
Software engineers have long used models and model-based design techniques to address the many sources of complexity in software construction. Unfortunately, powerful value-added applications of modeling (e.g., formal verification, code generation, and evolvable and demonstrable connections between model and source code) have yet to mature into a form that is usable by practitioners. Consequently, software models, if they are developed at all, are of little use to anyone but the modeler and are soon lost or outdated after the initial release of a product. Our research is concerned with how to bring models, modeling methods, and model-based design tools into mainstream software development. Projects in this genre apply diagram-based modeling, rigorous correctness analysis, software reuse, automated code generation, and software visualization.

Integrating informal and formal development techniques
Errors in software are most likely to be introduced in the requirements phase of development. This problem is largely due to ambiguous or inconsistent requirements descriptions. One potential solution is to capture requirements information in the form of formal specifications that can be checked for consistency and completeness using automated techniques. In general, it is difficult to construct formal specifications from scratch. In contrast, many developers find it more intuitive to create diagrams that serve as “blue-prints” of their systems. To bridge the gap between these formal and informal approaches to software development, we are investigating how to generate formal specifications (and code) from object-oriented diagrams of systems. Our focus has been on UML diagrams, the de facto standard in object-oriented modeling. The formal specifications enable us to “execute” the diagrams to check for consistency and visualize the behavior of the system.

Assurance Patterns
In order to help developers model and analyze requirements, we have developed a number of assurance patterns, with a focus on the embedded systems domain. First, after evaluating artifacts from numerous industrial projects, we have developed a number of object analysis patterns to assist in the construction of structural and behavioral models in terms of templates for UML diagrams. Models created from these patterns can be translated into formal specifications, which can then be analyzed using simulation and model checking. Second, we have identified a number of claims patterns representing commonly occurring properties to be analyzed against the object analysis models. Finally,
we have developed a number of real-time specification patterns based on the evaluation of real-time requirements for industrial projects. These patterns complement the qualitative specification patterns developed by Dwyer et al. In order to make specification patterns more accessible to developers, we have developed a natural language grammar and tool support for constructing the specifications, including seamless integration to existing analysis tools. In general, we have focused on developing industrial automotive embedded systems applications, such as smart cruise control, electronic steering, diesel filter systems, integrated starter systems, etc.

Generation of integrated analysis tools
Automatic verification capability tends to be packaged as stand-alone tools (e.g., model checkers, behavior simulators). Such packaging complicates integration with other tools in a software development environment, resulting in numerous translations to and from different internal representations. The Amalia project combines work on object-oriented frameworks, formal specification and verification methods, and generative programming to package verification capability as assemblies of light-weight components, which are easily customized to different internal representations. Amalia components allow tight integration with other tools in a software development environment.

Software assurance through contract-aware programming
One approach to integrating modeling into practice is to use of state-based modeling languages to declare and associate contracts with modules. A contract specifies invariant properties that a module will maintain, assumptions the module makes about the values of parameters passed to its operations, and guarantees that each operation will provide when invoked in an environment that satisfies those assumptions. A module that is designed by contract may assume (without containing code to verify) the rights specified in its contract, and this ability affords dramatic reductions in the amount of code required to implement a module along with a subsequent increase in assurance. We have developed a model of synchronization contracts, which allow developers to declaratively specify synchronization needs without having to program the logic that fulfills them. The model, called Szumo (Synchronization Units MOdel) enables the specification and run-time negotiation of contracts that guarantee client objects exclusive and conditional access to one or more servant objects.

Adaptive Middleware for Critical Infrastructure Protection
The RAPIDware project addresses the design of adaptive, component-based middleware to support robust computing services in dynamic, heterogeneous environments. Adaptability is especially important to systems that must continue to operate correctly during exceptional situations. Examples include systems used to manage critical infrastructures, such as electric power grids, telecommunication systems, nuclear facilities, and command and control systems. Such systems require run-time adaptation, including the ability to modify and replace components, in order to survive hardware component failures, network outages, and security attacks. RAPIDware addresses a unified methodology and related tools for constructing dynamically adaptive middleware services.

Transparent Shaping
Driven by the emergence of pervasive computing and the demand for autonomic systems, increasingly, software is required to adapt to a changing environment. The foundation of an adaptive software system is the ability to modify the behavior of software components at run time, namely, to insert and remove software sensors and actuators in active components. In many cases, it is also desirable to add new adaptive behavior to existing programs. As part of the RAPIDware project, we have developed a new programming model called transparent shaping, that enables new adaptive behavior to be woven into existing programs without modifying the application source code. We have developed transparent shaping tools that operate at the programming language level (for Java, C++, and C# programs) and at the middleware level (for CORBA, Java RMI, and Microsoft .NET programs). We have used these tools to realize several types of adaptive behavior in mobile computing environments.
Safe and Assured Adaptation
In order to provide assurance before, during, and after adaptation, we are developing a number of techniques to be used in conjunction with the dynamic adaptation techniques. Specifically, we have developed techniques to ensure that a dynamically adapting system does not enter into an inconsistent state during adaptation, which may require guiding the system to a safe state before performing the adaptation. We are also exploring how system invariants can be monitored and be satisfied during and after adaptation. This work includes creating A-LTL, an extension to linear temporal logic (LTL) with an adapt operator. We use A-LTL to specify the wide range of adaptation semantics, which is necessary when specifying the adaptation requirements and invariants.

Fault-tolerant and secure sensor networks
We are developing methods that simplify the design of fault-tolerant systems by separating the concerns of functionality and fault-tolerance. Using this method, we have developed several fault-tolerant programs for distributed systems, computer networks and parallel systems; these programs are more efficient and more fault-tolerant than previously known programs. We are applying these methods to the design of fault-tolerant and secure sensor networks, specifically to enable dynamic composition of software on sensors so that the mission of the sensor network can be changed dynamically. We are also studying automation of fault-tolerance, namely, the transformation of fault-intolerant systems into systems that can tolerate several types of faults.

Kernel-Middleware eXchange (KMX) for pervasive computing
Pervasive computing systems, including smart spaces and sensor networks, require software that can tolerate highly dynamic operating conditions. Necessary functionality includes providing quality-of-service for multimedia traffic, changing security policies for mobile users, and managing energy in battery-powered devices. The KMX project investigates the interaction between middleware and the operating system kernel to support adaptable pervasive computing. To validate the proposed methods and facilitate technology transfer, the project includes experimental case studies with Linux-based handheld and wearable computers on a mobile computing testbed.

Incentive Mechanisms for Peer-to-Peer Systems
In peer-to-peer systems, each user of the system is both a consumer and a service provider. For such systems to scale, users must cooperate in providing service to others. Unfortunately, actual systems often present users with the means and motive to act as free-riders—peers who consume but seldom provide service. Removing the means for misbehavior typically hobbles system functionality; we therefore focus instead on eliminating motive. The goal of this research is twofold. First, we are producing a suite of incentive mechanisms targeted at applications with differing characteristics. For example, we have developed a payment mechanism based on anonymous digital cash. Second, we are developing an analytical framework for evaluating the impact of such mechanisms on system performance and scalability.

Dynamic negotiation and enforcement of synchronization and security contracts
One of the fundamental challenges in building distributed software is the difficulty of specifying, programming, and reasoning about “non-functional” concerns, such as synchronization, adaptive quality of service, and access control. We are exploring the use of dynamically negotiable contracts as a paradigm for addressing these concerns with assurance. Unlike traditional pre- and post-condition style contracts, which a designer must verify at design time, our contracts are negotiated at run time. To date, we have developed frameworks and run-time systems that support the run-time negotiation of synchronization contracts, which guarantee clients exclusive and conditional access to one or more servants, and access contracts, which guarantee that the services of a component are accessed only by clients with sufficient privilege. To explore these ideas, we extended the Eiffel language with support for these contracts and implemented a run-time system that efficiently negotiates them. We are also working on extensions for C++.