Over the past several years, interest has grown considerably in new techniques and technology for improving the task of creating and maintaining high-quality software. These efforts have arisen in response to a growing sense among application developers that traditional approaches are inadequate. Such new methods for improving software efficiency and predictability include intentional programming, evolutionary programming, model-based programming, and self-adaptive software—the last a novel approach sponsored by the Information Technology Office of the US Defense Advanced Research Projects Agency.

Software creation, lifetime management, and quality have always been a nearly intractable set of engineering problems. Practitioners have approached these problems with a specific set of engineering techniques, specialized to the software domain: problem and tool abstraction, modularity, testing, and standards, among others. Examples of tool abstraction include high-level languages, operating systems, and database systems; examples of modularity include structured and object-oriented programming. Despite these efforts, and despite significant improvements in software tools and technology, software is still hard to produce, hard to support, and generally of significantly lower quality than we would like.

These more traditional approaches have not been worthless in improving our ability to produce better code more affordably. Rather, the problem has been that our reach always exceeds our grasp. As hardware capabilities improve and our understanding of...
how to apply computation to problems
improves, we continually try to solve more
difficult problems, driving up the complex-
ity of solutions and overrunning the ability
of our tools to manage the complexity.
Needed now is a fresh approach that lets us
build software in a new way and that offers
better methods for significantly enhancing
robustness. The authors included in this spe-
cial issue believe that self-adaptive software
is a promising new approach (or family of
approaches) that responds appropriately to
these new requirements.

Self-adaptive software

According to the DARPA Broad Agency
Announcement on Self-Adaptive Software
(BAA-98-12, December 1997; see www.darpa.
mit/ito/Solicitations/PIP_9812.html):

Self-adaptive software evaluates its own behav-
ior and changes behavior when the evaluation
indicates that it is not accomplishing what the
software is intended to do, or when better func-
tionality or performance is possible.

This implies that the software has multiple
ways of accomplishing its purpose and has
enough knowledge of its construction to make
effective changes at runtime. Such software
should include functionality for evaluating its
behavior and performance, as well as the abil-
ity to replan and reconfigure its operations to
improve its operation. Self-adaptive software
should also include a set of components for each
major function, along with descriptions of the
components, so that system components can be
selected and scheduled at runtime, in response
to the evaluators. It must also be able to impedi-
ance match input/output of sequenced compo-
ents and generate some of this code from spec-
fications. In addition, DARPA seeks this new
basis of adaptation to be applied at runtime, as
opposed to development/design time, or as a
maintenance activity.

The key aspects of this definition are that
code behavior is evaluated or tested at run-
time, that a negative test result leads to a run-
time change in behavior, and that the runtime
code includes the following items not cur-
cently included in shipped software:

• descriptions of software intentions (goals
  and designs) and of program structure and
• a collection of alternative implementa-
tions and algorithms (sometimes called a
reuse asset base).

Those of us working to shape and define
this area have largely been informed by two
metaphors: coding an application either as a
dynamic planning system or as a control sys-
tem. In the first metaphor, we imagine that

the application doesn’t simply execute a spe-
cific set of algorithms, but instead plans its
actions. Such a plan is available for inspec-
tion, evaluation, and modification. Replanning
can occur at runtime in response to a
negative evaluation of the plan’s effective-
ness or execution. The plan would treat
computational resources such as hardware,
communication capacity, and code objects
(components) as resources that the plan can
schedule and configure.

In the control-system metaphor, the run-
time software is treated like a factory, with
inputs and outputs, and a monitoring and
control facility that manages the factory.
Evaluation, measurement, and control sys-
tems are layered on top of the application
and manage system reconfiguration. This
regulated behavior derives from explicit
models of the application’s operation, pur-
pose, and structure. It is significantly more
complex than standard control systems,
because the effects of small changes are
highly variable and because the filtering and
diagnosis of results before they can be
treated as feedback or feed-forward mecha-
nisms are also very complex. Despite the dif-
culties of applying control theory to such
highly nonlinear systems, there appears to
be a very valuable set of insights to be ex-
plained from control theory, including, for
example, the concept of stability.

The “Articles” sidebar on the next page
shows the interplay between the dynamic
planning and control motifs in current self-
adaptive software work.

Problems, future issues

We don’t expect self-adaptive software
technology to develop immediately. Indeed,
difficult problems remain. The first is run-
time performance. Evaluating outcomes of
computations and determining if expecta-
tions are being met takes time. Expectations
will be met in most cases, and the checking
will seem to be purely overhead in those
cases. On the other hand, comprehensively
evaluating what algorithms and implemen-
tations to use is an advantage if it lets us
select the optimal or near-optimal algorithm
for the input and state context we have at run-
time, rather than making a preselected de-
sign-time compromise. Additionally, hard-
ware performance keeps increasing, but
perceived software robustness does not. Even
so, we will need to find ways to eliminate
unnecessary evaluation cycles and possibly
develop hierarchical systems with escalating
amounts of evaluation needed for particular
problems. That will clearly take effort.

Software creation effort is a second per-
formance problem. The kind of program-
ning that delivers code capable of evaluat-
ing and reconfiguring itself is difficult to
build by hand. Although such coding can be
accomplished in principle by having pro-
grammers and system designers write addi-
tional code (over and above the code needed
for straight functionality), we would prefer
automated techniques. Specifically, we think
that approaches that stress automated gener-
ation of evaluators from specifications of
program requirements and detailed design
are likely to significantly reduce the addi-
tional burden of producing evaluators.

Advances in computer hardware provide
both opportunities and problems. The pos-
sibility of rapidly reconfigurable hardware,
in the form of field-programmable gate
arrays, is one opportunity. DARPA’s Adap-
tive Computing Systems program is inves-
tigating how to build and program larger,
denser, and faster FPGAs. The program’s
researchers are considering how to switch
hardware programming as a computation’s
phases or modes change. Self-adaptive
software supports runtime management of
software components, including schedul-
ing alternative components and managing
I/O data mismatches. The runtime support
also exploits opportunities identified by
evaluators and replanning operations to
restore and improve functionality or per-
formance. Clearly, such reconfigurable
software could provide a useful top-level
application-programming level for recon-
figureable hardware. In such a system, the
hardware configuration would simply be

Robert Laddaga is a research scientist at MIT’s AI Lab and, until
recently, a program manager at the Information Technology Office of the
US Defense Advanced Research Projects Agency. His research interests
include intelligent systems and software, software-development tools, and
semantically based collaboration. He has a PhD in philosophy from Stan-
ford University and an MA in philosophy and a BS in mathematics from
the University of South Carolina. Contact him at the MIT AI Lab, Massa-
chusetts Inst. of Technology, Cambridge, MA 02139-4307; rladdaga@
ai.mit.edu.
another resource to be scheduled and planned. Under hardware problems, DARPA’s Data
Intensive program is addressing the increasing mismatch between processor speed and
memory-access speeds. One approach involves moving portions of processing to
memory modules, to reduce the requirement to move all data to the main microprocessor.
Self-adaptive and reconfigurable software can play a role in dynamically determining
the distribution of components in such highly parallel architectures.

Evaluation presents one of the hardest problems for self-adaptive software. It is
not always intuitively clear how to evaluate functionality and performance at runtime. It
might help here to consider three classes of application:
• constructive,
• analytic with ground truth available, and
• analytic without ground truth.

Consider a robot attempting to reach a specified goal, around a series of partially
mapped obstructions. A constructive task would be to make a plan to reach the goal.
Clearly, we can evaluate the plan statically based on its merits, in comparison to con-
straints and goals known beforehand. The harder case is the task of actually moving
closer to the goal. If obstructions prevent us from making progress, we can still evaluate
performance if we know our position relative to the goal, for example, by having GPS
or inertial guidance sensors on board. This is an example of an analytic problem with
ground truth (our position relative to the goal) known.

A third type of problem is like the second, except that here we have evidence about our
position, but not practically certain knowl-
edge. In this case, we have an analytic prob-
lem without ground truth available. In this
third and hardest class of applications, we
can still use weight of evidence and proba-
bilistic reasoning to evaluate performance at
time. The Robertson and Musliner arti-
cles present examples of this.

The foregoing example makes it clear that
there is great variability in the ease with
which evaluation can be accomplished, and
that even in the hardest case, it is still possi-
bility to evaluate. However, there is still much
work to be done in determining what classes of
application require what forms of evalua-
tion, which tools will provide better evalua-
tion capability, and to what extent such tools
will need to be specific to particular applica-
tion domains.

The lack of adequate metrics for degree of
robustness and adaptation is another weak-
ness of current self-adaptive software re-
search. It is difficult to determine the effec-
tiveness of self-adaptive software or its
degree of adaptiveness, or even how many
dimensions are required to measure robust-
ness or adaptiveness.
Our notions of reconfigurability and self-knowledge raise interesting questions about the unit of modularity and where the structural and requirements knowledge should be held. We need to determine whether we should be able to evaluate individual lines of code and functions or only larger-scale modules. We must also determine where the knowledge used to inform evaluation should be held—centrally, at the module, or in individual functions or lines of code. Although it is not currently clear that there is a right answer to these questions, let alone what that right answer might be, there is an interesting implication in the notion that the knowledge and self-evaluation should be at the level of modules. Modules with self-knowledge and self-evaluation capability are one way to characterize intelligent, intentional agents. In fact, several of the articles we have gathered organize their self-adaptive programs around intentional agents. It might be that there is a useful confluence of ideas between intentional agents on the one hand and self-adaptive software on the other.

**SOFTWARE DESIGN IS LARGELY** the task of analyzing the cases that a system will be presented with and ensuring that the software meets its requirements for those cases. In practice, providing good coverage of cases is difficult and ensuring complete coverage is impossible. Furthermore, because program behaviors are determined in advance, the exact operating conditions and inputs are not used in deciding what the software will do. The state of the software art is to adapt to new operating conditions “off line”—through the efforts of designers, coders, and maintainers. The requirement for human intervention means that needed change is delayed. The premise of self-adaptive software is that the need for change should be detected, and the required change effected, while the program is running (at runtime).

Self-adaptive software’s goal is the creation of technology to enable programs to understand, monitor, and modify themselves. Self-adaptive software understands what it does, how it does it, how to evaluate its own performance, and thus how to respond to changing conditions. I believe that self-adaptive software will identify, promote, and evaluate new models of code design and runtime support. These new models will let software modify its own behavior to adapt, at runtime—when exact conditions and inputs are known—to discovered changes in requirements, inputs, and internal and external conditions.

Self-adaptive software will provide swifter response, improved performance, and ease of update. Programs with self-knowledge, which use that knowledge to adapt to changing circumstances, are also an interesting avenue of exploration in AI. It is surprising that in an area like AI, which has provided so many novel tools for software development, so little has been done to apply AI techniques such as planning, probabilistic reasoning, and knowledge representation to the problem of producing and managing software applications.

Finally, many interesting tasks lay ahead in the development of technology for self-adaptive software. Undoubtedly, we’ll learn several interesting lessons along the way.

---

**Special Focuses**

**Intelligent Information Retrieval...**

Intelligent information retrieval—the problem of delivering truly relevant documents matching user needs—has become increasingly central in recent years. This special issue will present creative approaches to solving intelligent information retrieval problems using AI or ML techniques.

**...and Constraints**

A wide variety of problems can be naturally modeled as constraint-satisfaction and optimization problems, including scheduling, planning, resource allocation, routing, design, configuration, and diagnosis applications in the engineering, manufacturing, and service sectors. The special issue will provide insight into the increasing popularity of constraint technology.