



Analyzing your way to a better system

UNRESOLVED PROBLEMS CAUSE PROJECT SLIPS, RESULT IN POOR QUALITY, AND LOWER ENGINEERING PRODUCTIVITY. SOFTWARE-ANALYSIS TOOLS STRIVE TO HELP YOU TO AVOID, IDENTIFY, ISOLATE, AND FIX THEM.

At a glance52
Using generated code.....52
Using analysis features.....54
For more information56

ASK EMBEDDED-SYSTEM developers what kinds of tools they use to analyze developing systems, and you will get a good idea of where they focus their efforts during

design projects. Each set of analysis tools is best brought to bear at a different stage in the design process. In each case, these tools help developers better understand and visualize the system they are creating and validate that it will behave properly. Analysis tools can allow you to explore your system's behavior with different configurations, assist your debugging efforts, and identify where to focus your optimization efforts. They can also help you predict your project's health, understand how legacy code operates and can fit into your design, and design your system to accommodate future evolutionary changes. None of these tasks is yet fully automated. Many kinds of analysis tools exist to assist your development efforts. They can improve system performance and robustness and increase your productivity. The tools you choose depend on the type of system you are designing and what part of the design cycle you support. Although some tools are useful throughout a design effort, each tool is strongest at a different point of the design cycle (**Table 1**).

During the system and requirements analysis of a project, much about the system is still unknown, but decisions you make at this time usually have the greatest impact on the project's final cost, feasibility, and timeliness. Host-based modeling tools, such as those from The Mathworks, support algorithm exploration

and the refinement of system requirements. Model-based analysis allows you to detect errors earlier in the design cycle, because it is part of system requirements and preliminary design analysis.

Model-based analysis compares a model specification, under simulation, with the expected system behavior. It can identify inconsistent behaviors that occur when the model cannot—because of interacting and interdependent components—simultaneously satisfy two or more requirements. Model-based analysis can provide visualization of the model and internal data structure, the relationships between objects, the time sequences of events, and the sequence of state changes during execution. Once you verify the model with the system requirements, the model can support all phases of the design cycle, and it can act as the foundation for defining test strategies, test cases, and critical areas that require focused or more extensive testing.

The continuing evolution of model-based design tools allows mathematical models to act as executable specifications for an increasing range of system designs. You can apply model-based development to all or part of a system; for complex systems, you might target only the critical areas for analysis. Although model-based analysis can apply to any design, it is more productive when you use it in systems that involve real-time constraints, concurrency, distribution, or complex interactions among components. Model-based analysis is more effective when the

AT A GLANCE

- ▶ Analysis tools exist for and offer value throughout each phase of development.
- ▶ Intrusiveness and cost are balancing considerations for each type of analysis tool.
- ▶ Use of analysis tools can help you avoid subtle problems before they become manifest.

requirements include sufficient detail to create meaningful models.

Advances in code-synthesis engines enable automated code generators to produce directly from the model descriptions source code meant to execute on the target processor (see **sidebar** “Using generated code”). However, model-based tools do not eliminate the need for you to understand the system under consideration, especially when the modeling environment ignores diagnostics, error handling, calibration support, process scheduling, and other target-specific considerations. You must more completely and correctly define a model specification that is the source for a code generator than you do for a model specification supporting rapid prototyping. Tarragon, for example, supplies model-level static checkers that can apply coding guidelines for model architecture and programming style to reduce the risk of errors when the model specification drives a code generator.

Architectural generator/estimator tools, such as those that Tensilica and ARC provide with their cores, are valuable during the preliminary design or system-partitioning phase for systems that use configurable or extendable processors. Celoxica also provides tools that support system partitioning and exploring hardware and software trade-offs but does not tie them to a specific processor-core architecture. These tools allow you to explore hardware configurations under simulation as well as the impact of implementing tasks as software or hardware blocks. However, you need a software implementation in hand for the blocks that you want to perform these trades, creating a potential chicken-or-the-egg situation. An offsetting opportunity for these systems is that their configurable nature allows you to re-evaluate your architecture to make strategic adjustments to your processing architecture.

Co-design, or partitioning-assist, tools can provide you with estimates for system resources—such as the clock speed, power, and gate count needed for each configuration you explore—that can help you size caches and memory. These types of tools may support event-sequence analysis to identify events that occur often and tag sequences of instructions as candidates for a specialized instruction. They can help you understand the data flow between the hardware and the software blocks; however, partitioning between hardware and software remains a basically trial-and-error process of tag-

USING GENERATED CODE

Software-development tools change the way developers write and debug much of a system's code by abstracting the basic processor architecture. Developers use compiled source code to replace most assembly-level code efforts, except for functions that require hand-optimized code, to meet tight performance requirements. Source-level debuggers abstract register contents and address ranges and correlate them to source variables and data structures. Similarly, RTOS-aware profilers correlate system resources with

threads and contexts.

Model-based design and simulation tools provide an abstract view of a system and its components that can improve the effectiveness and efficiency for embedded controls development. Just as today's source-level debugging and profiling can translate system resources to source variables and thread contexts, model-level debugging needs to translate these resources to high-level descriptions. Using automated code-synthesis tools to produce target code directly from the model-

based tools may represent the next practical level of abstraction of the system for developers. If this situation occurs, future development tools will need to support the implementation and testing for the synthesized code with a correlation back to the original models.

An implication of using model-derived generated code in the target system is that the project team must maintain the models throughout the development cycle and during maintenance and field-troubleshooting efforts. The environment should

be able to identify where generated code is used without modification so the tools can leverage model-based analysis versus source- or object-code analysis and be able to flag areas where the model-based analysis may disagree with the object-code analysis of the system. The model specification for the code generator should support user-defined constraints that describe the important features of the system, including instrumentation constraints for validation, because embedded systems are typically resource-constrained.

TABLE 1—TOOLS FOR PHASES OF DESIGN

Development life-cycle phase	Applicable tools
System and requirements analysis	System-level modeling tools
Preliminary design/System partitioning	Architectural generator/estimators, hardware/software co-design tools
Software development and testing	Static code analyzers, simulators, debuggers, profilers
System integration/maintenance	Target-based debug/profiler with software agent, integrated on-chip debug/profiling blocks, external hardware tools

ging each block as software or hardware and manually comparing the profiling snapshots of the configurations.

A target system is often unavailable when you are developing the software. Static source-analysis tools are a cost-effective way to examine your code before you have a platform on which to execute it. These host-based tools work with the source code in a nonrunning mode; they infer information about your system by examining the relationships defined in the source code. Compilers with adjustable warning levels are common tools that can provide lintlike correctness analysis on your source code to identify possible logical errors. More sophisticated source-code analyzers, such as offerings from Programming Research, can apply stronger checking on source code and expose complexity issues, suggest testing information, and help you identify and prevent problems before you execute your code on a target. These tools can go beyond mere stylistic issues by applying coding standards that capture lessons learned and link relevant documentation with warning messages.

Static code analyzers can help you understand how your code fits together, especially if you are including legacy code or other software of which you lack intimate knowledge. Code analyzers are valuable in helping you identify what code affects what data, isolating the code into computational threads, presenting class hierarchies and call structures, visualizing code cross-dependencies, measuring code-size and resource allocations, comparing program versions, and exposing how proposed changes will impact other code in the project. Static code analyzers provide value by flagging inadvertent coding errors earlier in the development cycle, especially in more complex projects; however, they can't tell you how the code modules will interact in a running, multitasking, real-time environment.

Unlike static-analysis tools, dynamic-analysis tools characterize your system while it is running and can present you a picture of how the code interacts in a multitasking, real-time environment. Dynamic-software-analysis tools operate at the processor-execution level, so they

can operate on object code and compiled source code, regardless of the programming language. However, dynamic-software-analysis tools depend on the quality of your test cases. If your test cases are incomplete, your analysis results will be incomplete, too.

Host-based simulators, such as instruction-set and cycle-accurate simulators, are common dynamic-analysis tools that you can use to characterize your systems' behavior before a target system is available to test your code. An instruction-set simulator provides low-cost, high-speed simulation but costs you accuracy. It performs instruction execution, maintains the processor state, and provides approximate timing data, but it does not generate accurate timing data. A cycle-accurate simulator has the capabilities of the instruction-set simulator and provides accurate timing data, but it sacrifices simulation speed. Instruction-set simulators can typically perform millions of processor cycles per second; cycle-accurate simulators typically perform several orders of magnitude slower.

Software-based simulation is unsuitable for validating system development because the interactions between external, real-time events and the execution of the software on real hardware are difficult to exactly simulate. However, a simulator can be useful for narrowing the focus of your effort with the target system when exploring behavior that results from subtle interactions between software and hardware. The simulator provides a mechanism to examine in detail the state of the

USING ANALYSIS FEATURES

Tool vendors do not include analysis features, especially advanced features, in their tool offerings lightly. Interestingly, common observations across the software-tool-vendor industry find that these advanced analysis features have the poorest adoption rates by the developer community. A few developers use these features, but not nearly the number of developers that tool vendors expect should be able to benefit. Tool vendors attribute this low adoption rate for these features to developers' low awareness. Tool vendors are

expending efforts to increase tool awareness and better disseminate information about the newer capabilities by expanding the documentation with tutorials, walkthroughs, and examples of how to use each feature.

Another issue affecting why developers do or do not adopt certain features is the complexity of using those capabilities. The initial learning curve for a developer to use a capability might be longer than the time it takes to directly perform the task. The complexity issue is a function of the effort to set up and maintain

an analysis capability. Simple tools can address simple problems. Standard debugger capabilities enjoy a high adoption rate, but they usually address simple questions and data relationships. Complex problems, by definition, require developers to design the system a specific way or more completely describe the system to the tool for the analysis to perform useful work.

The payoff for these features occurs when they support an iterative process that amortizes the developer's setup effort, are easy to configure between itera-

tions, save significant time, and provide better visibility into the system behavior than other instrumentation methods. Efforts are ongoing to develop ways for these tools to better automate the capture or translation of the system description to address the complexity issue. In the meantime, tool providers note that developers tend to use the simplest features due to time pressures to complete tasks, and developers adopt these advanced features more as part of a lesson learned from a previous project.

processor while your code is executing, but the simulator itself is effectively hidden behind the other tools you can hook to it, such as debuggers and profilers.

Software-profiling tools can gather dynamic data about execution times, code frequency, calling patterns of functions in your code, and code coverage. You most often use them to pinpoint where to optimize code. Software-development-tool vendors offer RTOS-aware profiling tools that, with an instrumented kernel, provide visibility into operating-system events, including context switching, messages, exceptions, interrupts, and user events. Unfortunately, profilers require a level of instrumentation that is usually intrusive to the target system and can materially change the system's behavior. Because the simulation instrumentation runs outside the simulated system, the simulator affords you a more detailed, nonintrusive instrumentation that may be too costly to implement on the target hardware.

INSTRUMENTATION AND INTRUSIVENESS

Managing the intrusiveness of your data-collection instrumentation is critical in real-time systems. Werner Heisenberg, a German physicist, proposed the "Uncertainty Principle," which states that you cannot simultaneously determine both the location and the velocity of a particle. In short, by measuring a parti-

cle's location, you alter its position. The act of measuring disturbs what you are trying to measure. This uncertainty concept applies to embedded designs when your system's behavior is time-sensitive and relates directly to the interaction between the interrupts and application code.

Making the instrumentation part of the production system avoids the intrusion issue because it is part of the system behavior you verify during testing. You can leave the instrumentation in the production version, because your system resources and time windows have large enough margins that, although the instrumentation may slow your system or make it larger, it does not do so in a material fashion. An advantage of making instrumentation inherent in the production version is that your testing represents the production environment. Therefore, you may be able to use the instrumentation to troubleshoot your system in the field. The verification effort is also simpler for production-instrumented systems; it allows you to avoid a second verification effort, because verifying the instrumented version is the same as verifying the production version.

Instrumentation methods take many forms, and each method focuses on a different perspective of dynamic operations, sensitivity to intrusion, and cost to implement. Broadly, you can categorize

FOR MORE INFORMATION...

For more information on products such as those discussed in this article, go to www.edn.com/info and enter the reader service number. When you contact any of the following manufacturers directly, please let them know you read about their products in *EDN*.

Accelerated Technology
www.acceleratedtechnology.com
 Enter No. 327

Agere Systems
www.agere.com
 Enter No. 328

Analog Devices
www.analog.com
 Enter No. 329

ARC
www.arc.com
 Enter No. 330

ARM
www.arm.com
 Enter No. 331

Celoxica
www.celoxica.com
 Enter No. 332

Green Hills Software
www.ghs.com
 Enter No. 333

Intel
www.intel.com
 Enter No. 334

Mathworks
www.mathworks.com
 Enter No. 335

Metrowerks
www.metrowerks.com
 Enter No. 336

MIPS Technologies
www.mips.com
 Enter No. 337

NewMonics
www.newmonics.com
 Enter No. 338

OSE Systems
www.ose.com
 Enter No. 339

Programming Research
www.programmingresearch.com
 Enter No. 340

QNX
www.qnx.com
 Enter No. 341

STMicroelectronics
www.stm.com
 Enter No. 342

Tarragon Embedded Technology
www.tarragon-et.co.uk
 Enter No. 343

Telelogic
www.telelogic.com
 Enter No. 344

Tensilica
www.tensilica.com
 Enter No. 345

Texas Instruments
www.ti.com
 Enter No. 346

Wind River
www.windriver.com
 Enter No. 347

SUPER INFO NUMBER

For more information on the products available from all of the vendors listed in this box, enter no. 348 at www.edn.com/info.

these techniques as target-based software-agent instrumentation, integrated on-chip dynamic-analysis-support blocks, and external hardware tools. Target-based software-agent instrumentation includes many techniques and dynamic-analysis tools, because a target agent can intelligently collect analysis data at the speed of the processor more inexpensively than the other methods. Unfortunately, target software agents are usually more intrusive than the other methods

Source-code instrumentation, in which you explicitly insert instrumentation into your source code, is one of the simplest target-agent techniques. It can minimize intrusiveness by collecting only the most relevant, application-specific information. The code can be as simple or as complicated as necessary. However, the technique can be labor-intensive, does not scale well, and requires a recompilation every time you turn an instrumentation block on or off. Conditional compilation constructs help manage these blocks, and removing the instrumentation from the production system may increase the system-verification effort.

Compilers often include switches that direct them to add support for source-level debugging and profiling to the object code. Automated instrumentation insertion avoids the labor intensiveness of explicit code insertion, but you still need to recompile the code with the switches inactive to compile without the instrumentation code.

RTOS-kernel instrumentation is another target software-agent technique that enables a higher level abstraction of the system behavior in operating-system events. You can turn the instrumentation code on and off, and code left in the production image can be valuable for analyzing production systems in the field. Leaving the instrumentation code resident impacts the code size, and leaving the instrumentation active impacts execution speed, but for some systems, the benefits of these methods outweigh their intrusiveness. An advantage of kernel instrumentation is that it exists independently of the application code, so its inclusion or exclusion does not require a recompilation of your application code. You need not instrument your object code, and it can capture the activity of legacy and third-party object modules during analysis.

The second instrumentation method, using on-chip dynamic-analysis support blocks, encompasses debugging and profiling capabilities integrated with the processor architecture, such as hardware breakpoints, dedicated trace buffers, and profiling registers. This method allows systems to capture data at full speed without slowing execution or modifying the code, and it makes more internal system resources visible to a host system through a few pins. Too much data is available every clock cycle, so the on-chip system provides data-compression and -filtering mechanisms before transmitting the data to the host system. With intelligent data collection and filtering, the host system can correlate the data with high-level event and timing tracing. The trade-off for this support is a higher system cost, a larger silicon area and dedicated pins on the device, and the need for a connection interface on the board to connect to the host system.

External hardware-instrumentation tools, such as logic analyzers and in-circuit-emulators, are typically the least intrusive method. However, external hardware instrumentation is expensive, requires the connection of test fixtures that may preclude operation of the processor in the system, and requires significant expertise to determine the proper instrumentation points. These tools lose value when you compare them with on-chip analysis support for devices with cached architectures, integrated memories and peripherals, and complex instruction processing, including pipelines, superscalar, and VLIW architectures.

WHERE IS ANALYSIS GOING?

Analysis tools are still generally islands. They collect and present system data within a specific context. Cooperation among tools, especially when they cross company boundaries, generally involves nondisclosure agreements instead of public APIs. The Eclipse Consortium (www.eclipse.org) is, by providing common presentation and integration methods, trying to establish a universal tool platform that encourages public interoperability among tools and allows tool vendors to focus their energy on specialized tools. The Eclipse platform is most valuable to vendors with specialized tools and less valuable to larger vendors with competitive, comprehensive sets of inte-

grated tools—circumstances that may slow adoption of the consortium's efforts.

Cross triggering between tools is one way tools can operate together and enables multiple analysis tools to collect data around the same event. When one of the analysis tools signals a trigger, all of the connected tools can start and stop data collection in a coordinated manner. This ability helps you to receive time-correlated data from several perspectives and abstraction levels, such as RTOS events, application data, source-code execution, device architectural resources, and peripheral-signal-level activity. The debugger is a natural integration point for presenting these perspectives.

Profiler-assisted-compilation is an emerging example of using the results from an analysis tool to feed development tools further up the design process. Usually under simulation, a profiler analyzes the compiler output. The compiler then uses the profiler results to perform optimizations. By comparing iterations, the set of tools can home in on a set of optimizations.

Simulation is becoming more important as systems continue to increase in complexity, and even small levels of intrusiveness are unacceptable, such as for high-performance, multichannel, networking applications. Analysis tools for multiprocessor implementations are still in their infancy. An opportunity for simulators and chief among the current challenges of analyzing multiprocessor systems are synchronizing and accurately time-correlating data across the entire system. The multiprocessor-debugging and -profiling infrastructure must be able to negotiate the setup of each local island processor and direct collected data through an optimized interface to a unified collection system.

Tool vendors observe that developers often do not start using analysis tools until after a difficult problem appears in the system. Although analysis tools can provide system visibility that aid troubleshooting and optimization efforts, they

can also lower the risk of finding problems later in the design process, when corrective action is more costly. Tool vendors are focusing their innovation efforts to more strongly integrate analysis and profiling information across all levels of a design effort, so that developers will benefit from using analysis tools throughout the design process and eliminate subtle problems before they become manifest (see **sidebar** "Using analysis features").

Future analysis tools will incorporate expert-system technology and support to provide interactive assistance to analyze and quickly pinpoint the root cause of specific behaviors. These tools will need to draw on analysis data across the entire development cycle, because to be useful, they will need to intimately understand both the execution platform and the application characteristics. Maybe tools such as these will someday need only two buttons: find problem and fix problem. □

You can reach Technical Editor Robert Cravotta at 1-661-296-5096, fax 1-661-296-1087, e-mail rcravotta@edn.com.

