 By Robert Cravotta, Technical Editor

EMBEDDED DEVELOPMENT TOOLS CONTINUE TO INCORPORATE HIGHER LEVELS OF ABSTRACTION TO BALANCE THE INCREASE IN DESIGN COMPLEXITY.

Driving out complexity with ABSTRACTION TOOLS

Complexity is relative. It can prevent you from solving a problem if you can't characterize and understand the problem within a specific context. At the same time, it provides an opportunity to differentiate your product offerings. Furthermore, if you can surmount it, the complexity becomes the basis for you to differentiate your designs and acts a barrier, however temporary, to your competitors. Without sufficient complexity, it is difficult to offer unique, value-added, differentiating features because your competitors can easily duplicate your development effort and add those features to their own products.

At a glance 30
Insatiable appetite 30
For more information 34

An engineering rule of thumb states that a design team can deliver a product with the best feature set, in the shortest development time, for the lowest cost, as long as the team cares about only two of those elements at the expense of the third. In a competitive world, it is critically important to address all three of these elements, which in turn drives up the overall complexity of the design effort (see **sidebar** "Insatiable appetite"). Fortunately, you need not exhaustively understand every detail of something to use it. Using tools that support domain-specific abstractions or that enable you to create ad hoc abstractions for the system you are working with is a technique for reducing the complexity of your design effort and improving your development productivity.

Abstracting is the process of or-

ganizing the details of a system so that you can focus on the important elements of the big picture. Abstracting does not reduce—and can even increase—the number of details you must address for a design. However, abstracting enables you to organize the details so that you can reduce the number of details you must consider for any portion of your design. A challenge for building useful abstractions is deciding which properties are essential and which you can ignore based on the context and who will be using the abstraction. Abstractions are intellectual frameworks, and it may be useful to abstract a system in multiple ways. A designer and a user may need to use different abstractions of the same system. For hierarchical abstractions, a designer working at each level of the abstraction may need exposure to a different set of details.

Appropriately abstracting a system can reduce its complexity for each member of a design team because you can hide and remove the irrelevant details from each designer's consideration; in effect, you are simplifying and bounding the scope of conditions each designer must address. Abstracting a system can facilitate aggregating a system's micro behavior so you can better expose, conceptualize, address, and manipulate the behavior at a macro level. It is sometimes unnecessary to understand the detailed sequence of events to extract data from storage or to perform an arithmetic operation on that data; however, understanding and controlling the details of these two mechanisms is sometimes critical to the success of a project, such as when your system performs computationally intensive operations on a continuous stream of data in real time.

AT A GLANCE

- ▷ Abstracting reduces complexity by hiding details.
- ▷ Abstracting does not reduce the number of details for a design.
- ▷ Automated development tools simplify and commoditize formerly innovative design ideas and implementations.
- ▷ Abstraction hides details, but it should not prevent you from examining them.

Abstracting a system can help you identify, isolate, and manage the computational and interface dependencies between logical components of a system so that you can solve the larger problem in parts. If you lack the appropriate vocabulary and other symbolic tools to express a problem, you face a severe handicap in solving that problem. Appropriately abstracting a system can provide you with the vocabulary and symbolic set to meaningfully communicate among the design team members the parts and their relationships with each other. An appropriate vocabulary and symbolic set is application- and domain-specific. A standard or generic symbolic set may not enable you to sufficiently express the system and problem that you are trying to solve, and it may bias you toward a suboptimal approach.

TOOLS, TOOLS, TOOLS

Product designs are incorporating more complex features that consume more processing performance with each generation, and the product-design cycles are shorter than ever before. To keep pace with the growing complexity in these designs, development teams may rely on reusing legacy designs, licensing third-party IP (intellectual property), and using an assortment of tools that abstract or hide implementation details of the target platform. Assemblers, compilers, modeling tools, code generators, operating systems, and peripheral drivers hide details of the target-processing platform and generally improve designer productivity. These types of tools do not usually provide application-level abstractions; rather, they operate in a hierarchical fashion to abstract and automate target-implementation details without necessarily understanding the application's behavior.

Another type of code generator, such as from Celoxica and Stretch, help automate the translation of C source code into hardware blocks. These tools allow you to maintain a consistent software-de-

INSATIABLE APPETITE

The marketing pitch for many embedded development tools is that product complexity is rising, product design cycles and resources are shrinking, and it is strategically important that you release new features before your competitors do. At first glance, the long-term implications of such a trend would mean impossibly short design cycles would soon be the norm.

From the perspective of designing a product from scratch without tools, product complexity is rising, but is each designer's per-product design-cycle complexity increasing? Development tools that continue to integrate more capabilities as bundled features and deliver automated code generation at higher levels of abstraction are balancing the level of complexity that each designer must handle.

Although the time between product releases has over the last few years been shrinking, has the relative design complexity that a designer must deal with changed? Probably not: Designers are working with a comparable, effective complexity per product releases; again, the evolution of reusable blocks and development tools is balancing the scale of complexity that a designer must handle.

Adopting a method to quantify and normalize design complexity between product releases can help you avoid committing too many or too few resources and provide an indicator of how well your high-level development tools are balancing the growing complexity. It can also assess the risk of having your development team burn out. Quantifying the complexity is application-specific, and you could derive it by tracking the number and complexity of component and interface changes.

You should normalize the complexity by tracking how much reuse and code maintenance your abstractions and tools support, how much team turnover you experience, and how well your development tools and processes capture and abstract changes from earlier releases.

One possible reason that product-release cycles are shrinking is that customers demand shorter cycles. However, they may be shrinking so that you can respond rapidly to any innovations that your competitor manages to announce before you do.

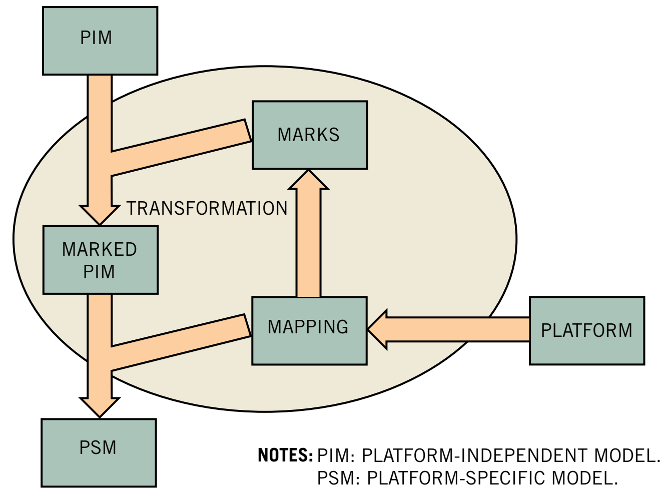


Figure 1 UML forms the foundation of OMG's model-driven architecture. The platform-independent model encompasses the application behavior, and the platform-specific model encompasses the implementation specifics based on platform-specific mapping rules and platform-independent-model markups.

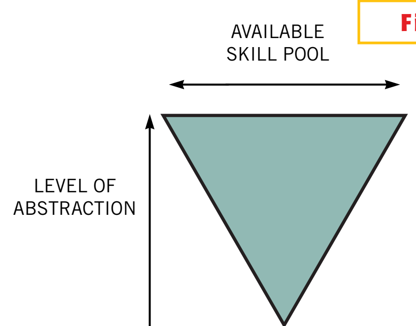
velopment methodology and still benefit from hardware acceleration of processing that exhibits parallel elements. Code generators from AccelChip and Catalytic operate at an even higher level, working directly from MatLab files to generate synthesizable RTL or to convert floating-point models into fixed-point target code.

Application abstraction tools attempt to decouple application behavior from implementation. Companies such as Aonix, The MathWorks, National Instruments, and Teja offer tools that provide domain-specific abstractions for safety-critical, signal-processing, or highly parallel multiprocessor application systems. Domain-specific abstraction tools include appropriate symbolic tools and structures that enable you to more naturally express and explore design approaches within that domain, but their usefulness is usually limited outside their target domain. Generic abstraction tools, such as UML (unified modeling language) tools from I-Logix and IBM, enable you to create your own abstractions and are useful across a wider range of applications, but they rely on you to apply your engineering expertise to build meaningful domain-specific abstractions.

MDA (model-driven-architecture)-development tools decouple modeling the application-behavior details from the implementation details by separating them into UML platform-independent and platform-specific models (**Figure 1**). The platform-independent model includes no technical-implementation details and focuses on describing the application's function and behavior. The MDA development tools apply a mapping template to produce the platform-specific model. To perform the conversion, you must fine-tune and annotate the platform-independent model to include semantics and rules to generate code. The MDA-development tools produce machine-generated code from the models; you can test the generated code against the behavioral models for correctness. If you change either the behavioral or the implementation models, the regenerated code consistently applies the changes.

By abstracting and automating the creation of the lower level implementa-

Figure 2



As development tools adequately support higher levels of abstraction, the size of the pool of skilled designers increases.

FOR MORE INFORMATION...

For more information on products such as those discussed in this article, contact any of the following manufacturers directly, and please let them know you read about their products in *EDN*.

AccelChip
1-408-943-0700
www.accelchip.com

Accelerated Technology
1-800-468-6853
www.acceleratedtechnology.com

Aonix
1-800-972-6649
www.aonix.com

Catalytic
1-888-228-2549
www.catalyticinc.com

Celoxica
+44-0-1235-863656
www.celoxica.com

eg3
1-510-713-2150
www.eg3.com

Green Hills Software
1-805-965-6044
www.ghs.com

IBM
1-800-426-4968
www.ibm.com

I-Logix
1-978-682-2100
www.ilogix.com

Microchip
1-800-437-2767
www.microchip.com

National Instruments
1-888-280-7645
www.ni.com

Object Management Group
1-781-444-0404
www.omg.org

Stretch
1-650-864-2700
www.stretchinc.com

Teja
1-408-288-2560
www.teja.com

Texas Instruments
1-800-336-5236
www.ti.com

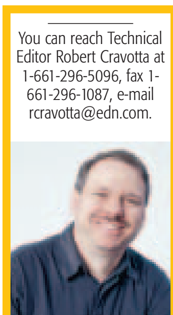
The MathWorks
1-508-647-7000
www.mathworks.com

tion details, these tools can provide a mechanism for portability of a design across target platforms and the reuse of design blocks. However, abstraction only enables a mechanism for portability; for example, if a target processor has no code generator, you must still manually port the design implementation. In addition to these benefits, the number of people who can maintain legacy-design details and materially participate in an embedded design increases as the higher level tools become more mature, usable, and effective at the target level (**Figure 2**). Fewer people possess the skills and knowledge to design using lower level tools than can design using higher level tools.

NO FREE LUNCH

High-level development tools promise many benefits, but embedded designs that are the first to market, deliver more powerful features, come in smaller packages, consume less power, and cost less often push a platform's resources to new limits. According to the e-mail-alert subscription requests and Web-page search traffic from eClips, the top three software keywords, overwhelmingly month to month, are "assembly," "C," and "C++." The fact that assembly ranks so high every month may be a surprise unless you consider that compilers and code generators cannot accommodate every novel application-specific data and resource optimization.

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



Development-tool features that enable you to complete your design as quickly as possible lag behind the first use of innovative techniques. Development tools such as compilers or code generators encapsulate the years of experience of the programmers that create them, but the input model or source-code language, even with proprietary extensions, cannot include specifications for every innovative and novel data representation and resource usage. A tool developer is unlikely to add and refine a capability unless it has a proven value and exists in some analogous form. When a compiler or code generator includes an automated or enabling capability, it commoditizes that capability that previously was a means of differentiation for someone.

For designers to differentiate their latest generation product offerings, they rely on the mix of features, size, power consumption, and cost to be sufficiently complex to act as barriers to competitors. As high-level development tools incorporate abstractions and automated tools that simplify that complexity, you need to push the complexity envelope that much further ahead of what the development tools abstract to maintain your competitive position.

Pain ultimately drives designers to adopt a higher level of abstraction for the non-differentiating portions of their designs. If you adopt a poor abstraction, you will probably get poor results, despite any heroic efforts, so it is important to be able to recognize as soon as possible when you are using an inappropriate and inefficient abstraction. Warning signs for using a poorly matched abstraction include the need for you to create workarounds for shortfalls in the abstraction, a lack of ability to evolve each layer of a hierarchical abstraction without significantly impacting the other levels, a lack of sufficient interfaces among components, or the necessity to repeat structures or operations among multiple components. If adopting a higher level of abstraction yields no productivity improvement, you are unlikely to adopt it.

Using abstractions is not free; by definition, when you abstract something, you hide some details to gain a complexity and development-productivity advantage, but you do so at the loss of your ability to control those details. If you need to control those details, you need a mechanism to break the abstraction. Compilers provide such a mechanism by supporting inline assembly. Not all modeling tools and the code generators that they rely on provide an analogous mechanism; in fact, modeling tools need to provide more of such mechanisms than simple inline assembly, such as back annotation to reflect low-level changes back to the model specification, to allow designers to control a detail at the right level of abstraction without further complicating design documenting and verification.

Sometimes, even useful abstractions fail. As an example, iterating over a large 2-D array can yield different performance depending on how you organize memory and whether you traverse the array horizontally or vertically. The only way you can effectively deal with these types of abstraction failures is for you to understand what the tool is abstracting and how it is implementing that abstraction. This necessity places a subtle learning burden on your development team and illustrates why, although abstractions may hide details from you, they should never prevent you from examining and understanding the details of the abstraction.□

TALK TO US

Post comments via TalkBack at the online version of this article at www.edn.com.