Component Models for Reasoning

Cristina Seceleanu and Ivica Crnkovic, *Mälardalen University, Sweden*

Component models with a rich specification—that is, component models built for reasoning—facilitate the use of different analysis and prediction techniques that simplify a system's design while increasing trust in its correct functioning.

The recent boom in hardware development has helped developers create software that can manage sophisticated processes and applications. With fewer hardware-related performance constraints as a result of Moore's law, it might seem that software development would somehow become simpler, but that is not the case. Consider, for example, the automotive industry's latest attempt to increase safety: cars that can detect a pedestrian crossing the road and then stop themselves to avoid collision. The hardware involved consists of different computing units to operate the various software components—namely, a field-programmable gate array (FPGA) to handle the image frames produced by the stereovision camera system, a GPU to run object recognition, and a CPU to handle system control.

To realize such a system, its developers must meet several challenges that go beyond pure functionality:

- performance beyond the system simply processing information in a given time frame;
- dependability, reliability, availability, and synchronization with real-time requirements so that the system does not react improperly, or too early or too late;
- optimized use of memory and other electronic resources due to manufacturing savings without compromising software correctness; and
- control over software development and maintenance costs.

A trial-and-error implementation that includes unit and system testing is a possible solution for ensuring that the implementation fulfills these mixed requirements, but it is far from a good one. Any technology that comes equipped with analysis techniques for assessing basic software properties such as functionality, as well as extrafunctional properties (EFPs; they are also known as nonfunctional or quality attributes) such as safety, timeliness, reliability, and resource usage, has higher trustworthiness, due to its ability to uncover potential trouble spots before actual system implementation.

The most common method of software and system development today is component-based; systems are built from existing components. By reusing hardware or software components, developers can use knowledge of their properties to predict the new system's properties. Component models with a rich specification—that is, component models built for reasoning—facilitate the use of different analysis and prediction techniques that simplify a system's design while increasing trust in its correct functioning. In other words, the better information about components we have, the better we can reason about the system.

WHY COMPONENTS?

Componentization is a basic software engineering principle inspired by the ancient Greek and Roman strategy of

 Published by the IEEE Computer Society

"divide and rule." In the computational context, it means breaking down complex systems and managing the smaller pieces separately. Components and the relationships between them are the basic elements of a top-down architectural analysis.[1]

But components are not only the result of top-down analysis: they are also building blocks. The idea of software components is as old as software engineering itself. At the first software engineering conference in 1968, M. Douglas McIlroy's keynote referred to "mass-produced" software components[2] and "software component factories," the enterprises that had started developing components for future systems. These components are not designed according to system requirements—rather, the requirements are adapted to the already existing components.

If a developer chooses to reuse existing components, the development process becomes a combination of a top-down and a bottom-up approach, with the top-down perspective driven by an overall architectural analysis and the bottom-up using the components themselves as a starting point. The overall function of a system built from components is a result of component function composition; similarly, the system's EFPs are the result of composing component EFPs.

A component can be a large subsystem, a set of classes in an object-oriented approach, a single class providing a service, or a simple, directly translatable function to an implementation in a programming language. This generality gives components their flexibility and wide applicability, but it also precludes them from providing substantial support in automated analysis due to difficulties in their precise specifications.

A significantly more efficient and simpler way to develop software systems is to specify components more precisely and formally, but doing so requires specifying a component beyond its architectural units—for example, it is not enough to specify the interface; specification of different EFPs is also required, such as component execution time, memory usage, and so forth. The component-based software engineering (CBSE) community has hosted several discussions about component specifications and produced many different definitions—for instance, varying opinions of whether a "component" is an executable unit or a model of an executable unit eventually led to the realization that a unique definition is impossible. Instead, a model can define the standards for properties that individual components should have and the methods for composing or combining them. Here, the concept of "component properties" comprises both the functional properties and the EFPs of individual components.

In the component-based approach, functionality is expressed in the form of an interface—a functional specification or entry point for the services that the component provides (the provided interface) or the services that the component uses (the required interface). The existence of both required and provided interfaces enables different types of reasoning that use component dependencies. For instance, if component A allows input values between 1 and 100, it is possible to check whether component B provides data to component A in the same range or smaller. The interface provides information at the syntax level that enables type checking in component compositions, which helps developers catch type incompatibilities when composing components.

A more advanced interface specification would include functional semantics such as an interface contract, which asserts both required and provided interfaces, thereby letting the developer reason about context correctness—here, context is the environment in which the component is executing. Interface contracts also enable reasoning about system-level properties by combining components using their contract-based interfaces. A component behavior specification can also be part of the interface; it typically consists of different variants of state machines, representing the component's internal states and state changes. If combined with the functional interface and contracts, the behavioral specification enables reasoning about component behavior in a specific context or about the entire system's behavior.

To simplify software development in a component-based approach, the first step is to encapsulate both functional and extrafunctional properties.

To simplify software development in a component-based approach, the first step is to encapsulate both functional properties and EFPs. This enables direct reuse of a component instead of developing and verifying a new one. The second step is to provide composition rules for both functional properties and EFPs. In an ideal world, following both of these steps would make it possible to easily construct complex systems and, through compositional reasoning, predict their properties. In the real word, this goal has yet to be achieved.

WHAT CAN WE REASON ABOUT?

Reasoning relies on two pillars of software development: composition and encapsulation. Composition increases development effectiveness by applying formally defined rules that facilitate analysis, while encapsulation improves development efficiency through the philosophy of "reason once, use many times."

Component composition

The act of composing components includes building the

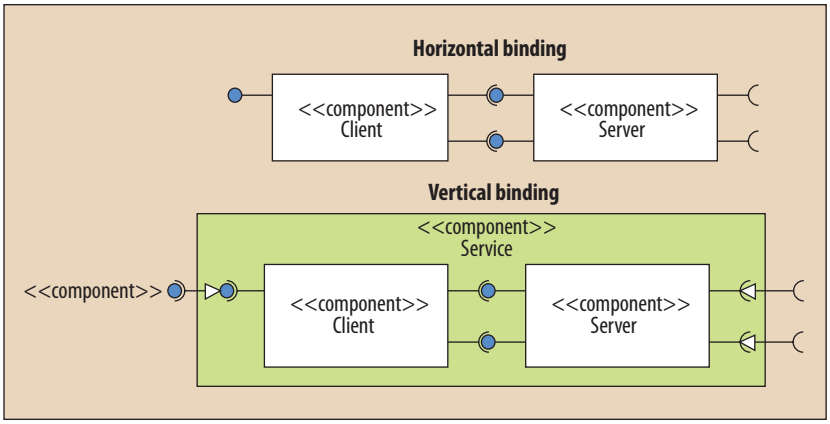

Figure 1. Horizontal and vertical binding of components. Vertical binding results in a new component: the new component Service has the same interface type as the constituent components Client and Server.

mechanism for component interaction. Such a mechanism results in an assembly that is characterized by a function (composed from the involved components' functions) and a set of EFPs (composed from the involved components' EFPs).

Component bindings, which are mechanisms to connect the components, enable component interaction by allowing one component's interface to be connected to another's. The component model's rules of composition define the types of components that are acceptable in a composition, their correct bindings, and the system's global properties.

We distinguish two types of bindings. A *horizontal binding* represents the connection of a component's provided interface with a subsequent component's required interface. This assembly does not necessarily constitute a new component; it is just an assembly of interacting components, and the resulting composition is called a horizontal composition. A *vertical binding* is an assembly that constitutes a new composite component that complies with the model's interface; the new composite component can be connected to other components in the same way as any other component complying with its model. Figure 1 shows this: here, the new component Service has the same interface type as the constituent components Client and Server. The sidebar "Generic Component Specification" gives formal definitions of some of the important terms in component-based development.

A component model that enables hierarchical compositions is a *hierarchical component model*, a powerful design option that enables functionality encapsulation on several levels via the same composition mechanism. Hierarchical component models allow the description of finely grained complex behaviors in the form of subcomponents nested within the same "host" composite component. A composite component can be a complex composition unit, containing many interconnected subcomponents. A specification of such components, which includes both the functional and EFP specifications, must be derived from the underlying components and then analyzed for both consistency and correct binding, as well as with respect to properties related to port access and data-flow semantics. Once its correctness is ensured, the component can be reused without repeating the analysis process.

For particular system architectures, the component compositions could be associated with port and data-flow requirements, which can be assessed by formal analysis at the architectural design level. Think back to the automatic pedestrian-detection system example from the introduction: given that the components and their connectors benefit from a formal semantic description, the associated reasoning framework should support the analysis of the system's architectural properties with respect to port access to ensure that specific data is not missed. For instance, the developer could check that the pedestrian-detector component's input signal is never written twice without being read between writings. This kind of safety-related analysis helps the designer by uncovering problems that could otherwise propagate to later design stages, where detecting them comes at extra cost.

Functional properties of components and compositions. Verifying functionality is an essential yet classic endeavor in system verification. Usually, functional properties describe the relationships between component and system variables and constrain the values associated with system operations or state changes.[3] For instance, a functional property of an antilock braking system (ABS) might require that for certain values of the slip rate, the brake actuator always releases and no brake is applied; to prevent the car from hitting a pedestrian, the developer would need to ensure that the brake is activated when a person crossing the road is detected at a specified distance from the car's front bumper.

At the binding level, the designer can reason about functional properties by checking interface contract correctness—for example, in the client-server horizontal binding in Figure 1, verifying interface contract correctness amounts to checking that the client component's provided interface is equivalent or contained in the server component's required interface. The contract specification has the major advantage of simplifying the verification of binding correctness, by reducing it to proving simple Boolean conditions. We exemplify the concept below.

Consider a well-defined (that is, one that has a specified interface and consistent properties) component model *CM* and two components C_1 and C_2 that comply with *CM*, as

specified by their required-provided interfaces, $\{preC_1\}$, $\{postC_1\}$, $\{preC_2\}$, and $\{postC_2\}$, respectively (we use $\{\ldots\}$ to denote an assertion):

$$\{preC_1\}\ C_1\ \{postC_1\},\ \{preC_2\}\ C_2\ \{postC_2\}$$

The correctness (consistency) of their horizontal binding entails discharging the following proof obligation: the precondition of C_1, $preC_1$, must satisfy the contract that establishes the postcondition of C_1, $postC_1$, which should imply the precondition of C_2, $preC_2$.

Extrafunctional properties of components and compositions. The composition of certain EFPs is well understood in principle. For instance, in most cases, composing the static memory and energy used by interconnected components, respectively, ends up in adding the values of the resources used by each component—hence, static memory and energy consumption are additive, meaning that the semantics of the composition operator ($\oplus$) is known (see the "Generic Component Specification" sidebar). However, in many situations, the composition is not straightforward and cannot be easily generalized. One challenge, for instance, is that the composition operator depends on the context in which the component is used—for instance, a composite component's response time does not depend solely on each subcomponent's response time but also on the underlying scheduling mechanisms. EFPs can vary depending on several factors and choices, including the overall system architecture and the underlying platform's characteristics.

To be able to reason about component properties in general, we need an unambiguous form of component semantics—essentially, behavioral semantics—as a way of expressing what to check, next to which we also need to provide a formalized property and tools to support the reasoning. An analysis can equate to a verification that returns a yes or no answer, or it can come in the form of quantitative analysis that includes statistical or probabilistic techniques.

Given a well-defined behavioral model M of component C, some environment assumptions Γ, and an initial state s_0 from which the model starts executing, we can formally express the verification of C with respect to property ρ as follows:

$$\Gamma, M, s_0 \models \rho,$$

which means that the property ρ should hold for all executions of M, within the context specified by Γ, and starting from s_0.

Encapsulation

We have seen that a component interface encapsulating functional behavior (including functional properties), together with well-defined rules of component compositions form the prerequisite for high-level predictions that include analysis of proper component bindings. We argue further that a component model that goes beyond such interfaces enables diverse analyses at various levels of abstraction, simplifying component selection, development, and system design.

Generic Component Specification

Some formal definitions of important terms in component-based development give an indication of this model's power and flexibility.

Component. A component C is specified by a set of functional properties expressed as an interface I and a set of extrafunctional properties P:

$$C = \langle I, P \rangle$$

Here, we abstract away other possible mechanisms that can be otherwise encapsulated in the component model.

Interface. In its complete form, an interface I specifies the provided interface I_p, the required interface I_r, the contract specification I_c, and the behavior specification I_b:

$$I = \langle I_p, I_r, I_c, I_b \rangle.$$

Compliance with a component model. If a component $C = \langle I, P \rangle$ complies with a component model CM, then both its functional and extrafunctional properties comply with the component model:

$$C \models CM \Rightarrow I, P \models CM.$$

Component composition. Assuming that the component function is expressed by interface I, and extrafunctional properties by a set P, a composition (denoted by $\oplus$) of two components $C_1 = \langle I_1, P_1 \rangle$ and $C_2 = \langle I_2, P_2 \rangle$ is defined as follows:

$$C_1 \oplus C_2 = \langle I_1 \oplus I_2, P_1 \oplus P_2 \rangle.$$

Binding. A binding defines a connection between components, realized as a composition of component interfaces. Assuming two components $C_1 = \langle I_1, P_1 \rangle$ and $C_2 = \langle I_2, P_2 \rangle$ that comply with a particular component model CM, that is,

$$C_1, C_2 \models CM \Rightarrow I_1, I_2, P_1, P_2 \models CM,$$

we can define two types of bindings (interface compositions):

Horizontal binding: $H = C_1\ o\ C_2 \Rightarrow I_H = I_1\ o\ I_2$

Vertical binding: $V = C_1\ o\ C_2 \Rightarrow I_V = I_1\ o\ I_2$, where $I_V \models CM$.

Vertical composition. The result of a vertical composition of C_1 and C_2 is component $V = \langle I_V, P_V \rangle$, which complies with the component model CM with respect to both functional and extrafunctional properties:

$$V = C_1 \oplus C_2 \Rightarrow$$

$$V = \langle I_V, P_V \rangle \mid (I_V = I_1 \oplus I_2, I_V \models CM) \text{ and } (P_V = P_1 \oplus P_2, P_V \models CM).$$

The two pillars on which any component-based design should rely are *reuse* and *predictability*. The first principle can be achieved by allowing hierarchical compositions of components, meaning that components and their connectors are nested (up to arbitrary application-dictated depths) within host or top-level components. Predictability can be increased by requiring a read-execute-write semantics from the component model, meaning that once a component is triggered, its execution is functionally independent of any concurrent activity (the execution cannot be interrupted). This also facilitates analysis: component executions can be modeled analytically by input-output functions. However, such semantics can be rather strict, preventing parallel computing, so one option is to specify a component at various levels of abstraction by using different communication styles. One example of such specification on more than one abstraction level is ProCom,[4] a component model for designing vehicular and telecom systems.

The more encapsulated reasoning is supported in component models, the more predictable software solutions can become.

To address the issue of reuse, ProCom consists of two layers: ProSys, which allows loosely coupled, coarse-grained components that communicate via message passing, and ProSave, which relies on time-triggered, pipe-and-filter communication and strict run-to-completion execution semantics. The two layers enable analysis at the earliest development stages, giving developers insight into system behavior, whether the design includes fully developed components or unformed ones. The analysis can be performed at various design stages, so the component model can consequently store a plethora of analysis results.

For analysis purposes, developers will want to be able to associate attributes with components and subsystems for specifying different functional and extrafunctional characteristics.[5] Some attributes might be represented by single numbers—for example, worst-case execution time (WCET) or static memory usage. However, for more complex functional and extrafunctional behavior such as timing, resource consumption, and reliability, the number-based annotations are not enough. If we consider resource usage, the attributes must represent various rates of resource consumption for continuous resources such as energy—or edge probabilities to analyze component reliability—to understand the causes of potential trouble spots and prepare for model refactoring. Thus, possibly complex behavioral models of components' and compositions' internal workings need to be encapsulated within the component model itself. One such resource-aware behavioral model that works for component-based development is the dense-time, state-based hierarchical modeling language REMES,[6] which is used with ProCom. REMES describes ProCom component behavior by representing possible behavioral modes and their function (as assignments or conditionals), timing constraints (as mode invariants), and resource usage (as mode annotations in terms of linear differential equations or mode assignments).

The encapsulation that simplifies component-based system design and facilitates the prediction of possible errors at early design stages can be represented by three levels of component "knowledge" required to equip the component model. The first is the functional specification (in the form of a signature), and contracts connected to the interface, including execution semantics such as read-execute-write; the second is component behavior, or the internal states and transitions serving as the semantics of a state-based model; and the third is made of EFPs that can be expressed in different forms, such as a value or a model. The sidebar "Rich Interface Component Model Example: ProCom" shows examples of architectural designs built with ProCom; the sidebar "Extrafunctional Property and Behavior Encapsulation in ProCom" shows the specification of EFPs and behavior in ProCom.

Today's software is characterized by its continuous and rapid evolution, increasing complexity, dynamic runtime environment, and intensive interaction. For most software, a bottom-up approach with support for dynamic composition and adaptation is the key to successful development. A component-based approach that enables the reuse of black-box software components with encapsulated functionality and well-defined properties in the form of contract-based interface specification supports this type of development. The more encapsulated reasoning is supported in component models, the more predictable software solutions can become.

However, this approach also has serious challenges: compositional reasoning is complex and, in many cases, not achievable; rich component specification requires additional efforts that do not pay off immediately, and overspecified components are less reusable. Complexity in compositional reasoning is especially tricky—the response time, for example, of two combined components does not depend just on each component's response times but also on the environment, such as the underlying scheduling policy.

Formalisms are getting simpler and system behavior more predictable when more rules are enforced on component models. Strict read-execute-write component execution semantics, the separation of a component's

Rich Interface Component Model Example: ProCom

ProCom is an example of a component model that, due to its rich interface, enables the design of a large variety of products, but also the modeling and analysis of important properties of embedded systems. The interface richness is realized through different interaction types, precise execution semantics, and component-level specification of extrafunctional properties.

Figure A shows the upper-level ProSys for overall system design, and Figure B shows the lower-level ProSave for subsystem design.

The upper-level ProSys components, such as Stability Control System, are coarse-grained, with message-type interface and connectors implemented as asynchronous named communication—for example, Yaw angle.

The lower-level ProSave components such as Computing Actual Direction are fine-grained with synchronous communication, separation of control flow (modeled by trigger ports) from data flow (modeled by data ports), and using read-execute-write execution semantics. This enables modeling and predictions of timing properties such as execution and response times.[1]

Reference

1. J. Carlson, "Timing Analysis of Component-Based Embedded Systems," *Proc. 15th Int'l ACM SIGSOFT Symp. Component-Based Software Eng.*, ACM, 2012, pp. 151-156.

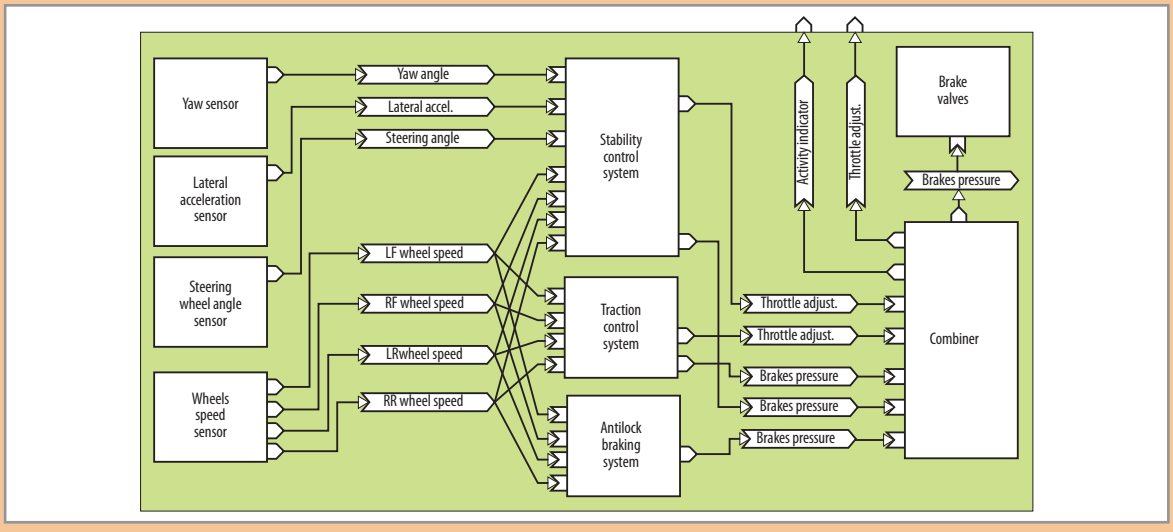

Figure A. Car stability system architectural design using ProSys components that enable asynchronous communication using messages, which is convenient for subsystems distributed in a network.

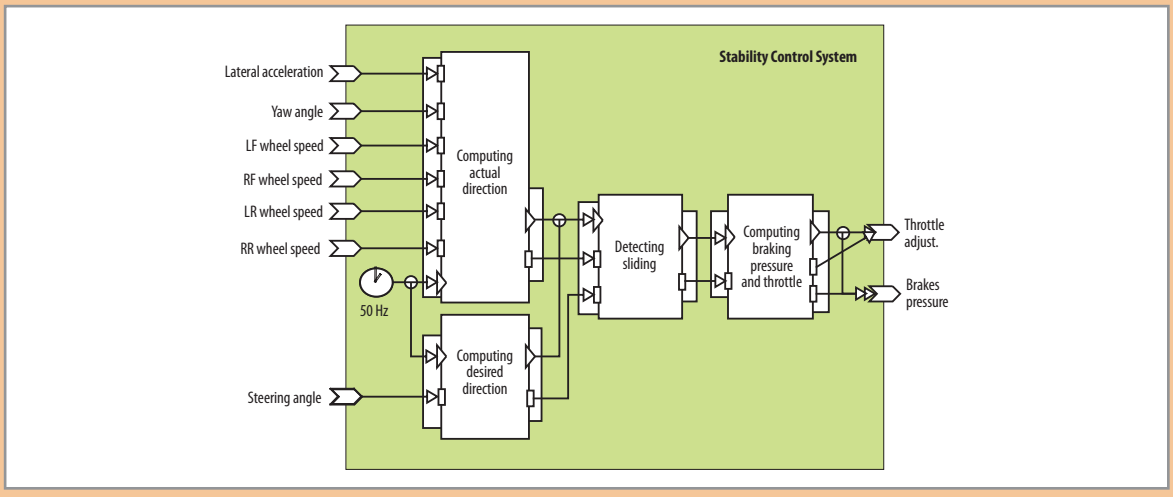

Figure B. Car stability subsystem architectural design using ProSave components that enable synchronous communication using pipe & filter architectural style and a separation of control flow from data flow. This simple style is suitable for local communication and for a precise prediction of timing properties.

Extrafunctional Property and Behavior Encapsulation in ProCom

The ProCom component model enables specification of extrafunctional properties as particular values that can be obtained from measurement, calculation, or estimation. One part of the extrafunctional property specification is the model of the context for which the specification is valid.

Formally, we specify an extrafunctional property as an attribute *A* of type T_A with a set of values *V*. The value itself includes data *D* (a measured value), metadata *MD* (value metrics or some other additional data), and *Ct* (a context or a set of conditions valid for the considered extrafunctional property; *Ct* can, for example, represent the CPU type, the scheduling algorithm, or another condition that has influence on the attribute value):

$$A = \langle T_A, V^+ \rangle ; V = \langle D, MD, Ct^* \rangle.$$

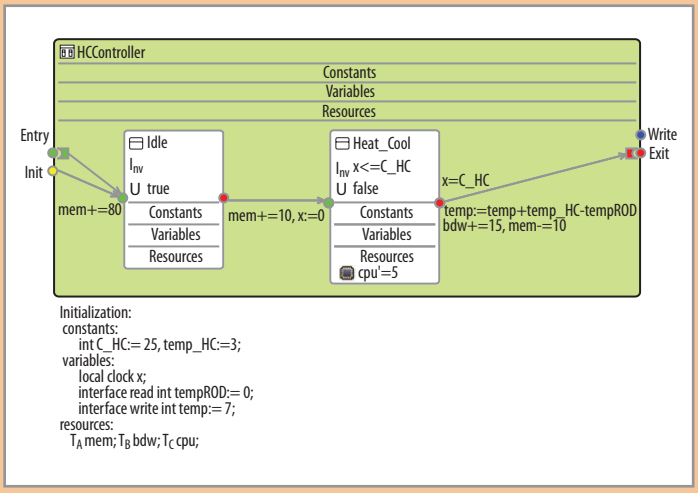

Figure C. REMES model of a ProCom component controlling temperature. The resources are declared as "resource" variables (mem, bdw, cpu), and the model describes the component's behavior modes (Idle, Heat_Cool), each mode's function (temp:=temp+temp_HC-temp ROD), timing constraints (x<=C_HC), and mode behavior with respect to resources (cpu' = 5, mem+=10).

Note that the same attribute can have different values for different contexts, as well as in the same context—for example, we can have different values of an attribute defined in different phases of the component life cycle, as an attribute can be estimated or predicted during the development phase but measured during runtime.

ProCom provides mechanisms to specify certain component attributes such as WCET and static memory usage, and enables specification of additional attributes of components or of any other architectural element such as interface elements, connectors, and primitive or composite components.

An example of behavior encapsulation in ProCom is resource usage. We can model resources such as CPU, memory, energy, and bandwidth together with the component behavior by using the REMES modeling language. Figure C shows an example of a REMES model describing the behavior of a ProCom component that controls the temperature in a nuclear tank via rods for cooling.

control interface from the data interface, and time-triggered scheduling, as implemented in the ProSave level of ProCom, enable predictable timing properties.[7] However, strict execution semantics limit the design space—for example, a loop, regularly used in feedback controls, cannot be implemented directly using ProSave.

A proper tradeoff between flexibility and predictability could lead to an effective and predictable component-based development. A component model does not need to support all possible concerns—different domains have different issues, a fact reflected in the design of various component models that encapsulate different formalisms that enable reasoning related to component properties and their compositions. The sidebar "Component Models for Reasoning" provides some examples.[8,9]

Some of the reasons why it is difficult to embed simplicity in software design are beyond components—for example, the unwillingness to replace proprietary software with off-the-shelf components or software from the open source community—but perhaps by integrating component-based ideas into their builds, developers can start simplifying the overall process.

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley 1997.
2. M.D. McIlroy, "Mass Produced Software Components," *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, October 1968*, P. Naur and B. Randell, eds., Scientific Affairs Division, NATO, 1969, pp. 138-155; http://cm.bell-labs.com/cm/cs/who/doug/components.txt.
3. J. Hatcliff et al., "Behavioral Interface Specification Languages," *ACM Computing Surveys*, vol. 44, no. 3, 2012, article 16.

Component Models for Reasoning

Although the overall principles of component models are quite well defined, in practice they can differ in both their support and design. The following examples offer some good places to start with component-based reasoning:

- AUTOSAR (AUTomotive Open System Architecture; www.autosar.org/index.php?p=3&) is the new standard in the automotive industry for component models with basic functionality.
- The BIP (Behavior, Interaction, Priority; www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html) framework is used for modeling heterogeneous real-time components.
- BlueArX is a component model used in the automotive control domain. It supports the modeling of timing, memory usage, and the generic specification of other properties.
- Fractal (http://fractal.ow2.org) is a modular, extensible component model that can be customized through the notion of a component control membrane.
- Palladio (http://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model) provides a domain-specific modeling language for component-based software architectures that is tuned to enable early life-cycle performance predictions.
- ProCom (www.idt.mdh.se/pride/?id=documentation-publications) is a component model for real-time and embedded systems that targets the domains of vehicular and telecommunication systems, enabling predictability of timing properties and resource utilization. It provides a framework for reasoning encapsulation.
- RoboCop (Robust Open Component Based Software Architecture for Configurable Devices; www.hitech-projects.com/euprojects/robocop) is a component model developed for the high-volume consumer device domain.
- The Rubus (www.arcticus-systems.com/index.php?pageId=11) component model is intended for small, resource-constrained embedded systems. It provides reasoning encapsulation and composition of real-time properties.

4. S. Sentilles et al., "A Component Model for Control-Intensive Distributed Embedded Systems," *Proc. 11th Int'l Symp. Component Based Software Engineering* (CBSE 2008), LNCS 5282, Springer, 2008, pp. 310-317.
5. S. Sentilles, "Managing Extra-Functional Properties in Component-Based Development of Embedded Systems," doctoral dissertation, IDT, Mälardalen Univ., Sweden, June 2012.
6. C. Seceleanu, A.Vulgarakis, and P. Pettersson, "REMES: A Resource Model for Embedded Systems," *Proc. 14th IEEE Int'l Conf. Eng. of Complex Computer Systems* (ICECCS 2009), IEEE CS, 2009, pp. 84-94.
7. J. Carlson, "Timing Analysis of Component-Based Embedded Systems," *Proc. 15th Int'l ACM SIGSOFT Symp. Component Based Software Eng.*, ACM, 2012, pp. 151-156.
8. I. Crnkovic et al., "Classification Framework for Software Component Models," *IEEE Trans. Software Eng.*, vol. 37, no. 5, 2011, pp. 593-615.
9. K.-K. Lau and W. Zheng, "Software Component Models," *IEEE Trans. Software Eng.*, vol. 33, no. 10, 2007, pp. 709-724.

Cristina Seceleanu *is a senior lecturer in the Embedded Systems Division at Mälardalen University, Sweden, where she co-leads the Formal Modeling and Analysis of Embedded Systems group. Her research focuses on developing formal models and verification techniques for designing predictable real-time embedded systems and service-oriented systems. Seceleanu received a PhD in computer science from the Åbo Akademi and Turku Centre for Computer Science, Åbo/Turku, Finland. She is a member of IEEE and ACM. Contact her at cristina.seceleanu@mdh.se.*

Ivica Crnkovic *is a professor of industrial software engineering at Mälardalen University, Sweden, where he is the administrative leader of the Software Engineering Laboratory and the scientific leader of industrial software engineering research. His research interests include component-based software engineering, software architecture, software configuration management, and software development environments and tools. Crnkovic received a PhD in computer science from the University of Zagreb, Croatia. Contact him at ivica.crnkovic@mdh.se.*