OPTIMIZING PORTABLE SDR SOFTWARE

John Hogg (Zeligsoft, Gatineau, QC, Canada; hogg@zeligsoft.com);
Francis Bordeleau (Zeligsoft, Gatineau, QC, Canada; francis@zeligsoft.com).

ABSTRACT

Software-defined radios have two requirements that are usually assumed to be diametrically opposed: high performance and portability. Standards such as the SCA focus on the second requirement for control logic, but SDR data paths in these systems tend to be optimized at the expense of flexibility. This is the result of two fundamental constraints on optimizing component-based SDR software in the traditional development lifecycle. The first is the ordering of the steps in the lifecycle itself: deployment information isn't provided until after it's needed. The second is the conflating of the component model contract with the code contract. This paper examines past practices. It then explains how to overcome these constraints to deliver components that are both portable and optimized for the context in which they're used.

1. THE PROBLEM

Component-based development is widely accepted as a powerful way to deliver high-quality software-defined radio (SDR) waveforms quickly. Components have clear interfaces, simplifying interactions between development teams and reducing the task of verification teams. This leads to increased quality. Component encapsulation also effectively reduces the size of a waveform, since the number of interactions (and hence the complexity of a system) increases exponentially with the number of interacting elements. This decreased problem size leads directly to decreased resource requirements and decreased waveform time-to-market.

Another key perceived benefit of component-based development is increased portability of waveforms. Since interfaces are tightly specified, a waveform developed for one physical platform can be ported to another platform with minimal need to rewrite platform-specific code. (Components may also be reused in different waveforms, but a deeper discussion is beyond the scope of this paper.)

For these reasons mainstream SDR development today is component-based. Military software-defined radios use the Software Communications Architecture (SCA) standard and the SCA is increasingly used in non-military radio and waveform development as well.

All is not positive, however. In addition to the perceived benefits of component-based SDR development, there is a perceived problem as well: decreased performance. Component-based development is widely viewed as delivering slower and larger waveforms than hand-optimized, non-component development. The key question is, "How can we maintain the benefits of components without paying a performance cost?"

The remainder of this paper is organized as follows. Section 2 briefly outlines the concepts of components and component-based development, including an explanation of how components are deployed to platforms. Section 3 reviews traditional development lifecycles. This includes optimized development not using components and conventional component-based development. Section 4 identifies the fundamental drawback with conventional component-based development: the ordering of steps in the lifecycle precludes fully optimizing code. A reordered lifecycle is described that avoids this shortcoming and Deployment-Aware Generation™ (DAG™) is described. Finally, Section 5 looks at other potential optimizations through DAG.

2. COMPONENTS

2.1 Component Elements

A *component* is a composable unit of software with a precisely-specified interface. A component *definition* is the specification of an interface and a (usually less formal) specification of the behavior behind that interface. A component *implementation* is a realization of the definition. Depending on context, this realization may be an executable file, the source code from which the executable was generated or one or more models from which the source code was generated. A component *instance* is the runtime manifestation of an implementation. In the SCA, the term is also applied to component *roles*, or the use of a component within a waveform definition.

The composition of a set of interacting component instances is a *waveform* in SDR (or more generally, an *application*). This is the architecture of the waveform, or "who speaks to whom". A waveform specifies the connection between the interfaces of its component roles and the component definitions and implementations that play those roles.

In general systems a waveform is simply another type of component and components may be hierarchically composed to arbitrary levels. In standards such as the SCA there is a strict component/waveform dichotomy, where waveforms have internal structure but no behavior and components have behavior but no internal structure. This paper covers the general case but space precludes a full discussion of it here.

Component and waveform definitions, implementations, roles and instances may all have aspects or properties that are *configured* or set to specific values in the contexts in which they are used. For instance, the cardinality of a replicated interface may be configured when a component role is defined in a waveform.

2.2 Component Behaviors and Interfaces

A component implementation has two distinct aspects. The first is the *functional behavior* of the component. This is the component's reason for existence: it is the logic that the component is intended to deliver, and would exist in a non-component-based implementation.

The second aspect is the *component behavior*, or the logic that controls how a component interacts with the external world. It is the code that makes a component a component and includes communication behavior, startup and shutdown of the component and its connections, and fault management.

Functional and component behaviors frequently come from different sources. Either or both may be manually written as code or generated from a tool. Regardless of source, the two must be merged. Furthermore, no interesting component is written in a single pass, so the evolving functional and component behaviors must be synchronized through each iteration of the development lifecycle.

Components have two types of interface. The obvious one is a component's *outer API*, or set of interfaces to the external system. The component communicates with other components as part of a waveform through external APIs, and also with the system outside of the waveform. Developing against the outer API requires a knowledge of the communication system used by the API.

The second type of component interface is the *internal API*, or the interface between the functional behavior and the component packaging. Developing against this interface requires knowledge of the messages, but not the external communication system through which the component behavior will interact with other components.

Both the internal and external APIs provide opportunities for integrating generated code. Therefore, both APIs provide opportunities for optimizing code, as we will see.

2.4 Platforms

A waveform and its components do not execute in a vacuum. Each component runs on some *node* and nodes are connected to each other to form a *platform*. Nodes and platforms are a component-based representation of an execution environment; they are the architectural analogs to components and waveforms respectively. More general systems treat a platform as another type of node and allow nesting to arbitrary levels.

A platform may exist at several levels. At the bottom is physical hardware. Above that, a logical platform can be composed of software execution environments such as RTOSes and BIOSes. Different levels of granularity are possible: an RTOS can contain processes which contain threads which contain logical or lightweight threads.

2.4 Deployment

An application and a platform come together through *deployment*. A component deployment is the assignment of a source component instance in a waveform to a target node instance in a platform. A waveform deployment is a set of component deployments.

Source, target, and indeed platform are relative terms. An application implementing the parts of a logical platform may be deployed to a physical platform. A waveform application may then be deployed to the logical platform. In one case the logical platform is the source; in the other it is the target.

3. TRADITIONAL DEVELOPMENT LIFECYCLES

With these definitions in hand, we can look at two traditional approaches to waveform definition. The first is the development lifecycle of optimized waveforms not based on components. The second is the conventional component-based development lifecycle for waveforms.

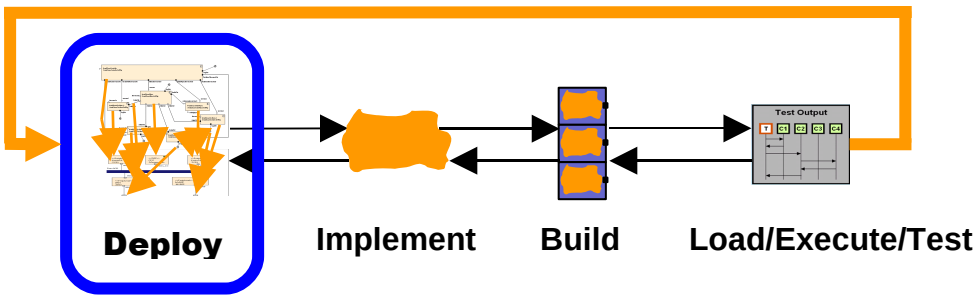

Figure 1: the traditional hand-optimized non-component lifecycle

3.1 Hand-Optimized Lifecycle Without Components

SDR waveform development is now primarily component-based, but radios based on or using significant amounts of software existed well before the adoption of the SCA. An idealized lifecycle for development of these radios is as follows:

- Define the broad architecture of the hardware and software and the deployment of software to hardware.
- Implement the software (i.e., write code).
- Build the executables.
- Load, execute and test the software.
- Iterate until the required functionality and quality has been achieved.

See Figure 1 for a graphical view of the lifecycle.

A key point is that the deployment decision is made before the functionality is implemented. The implementation can therefore be optimized with respect to the chosen deployment.

However, there is a corresponding drawback: the early optimization severely limits component portability. Since each executable is hand-crafted to be optimized for a given context (chip and logical environment) within a platform, the cost of redeploying it to another context can be very high. It may also require the rewriting of other software that is not redeployed, but which communicates with the redeployed software.

Hand-optimized, non-component-based waveforms are therefore potentially optimized but non-portable.

3.2 Conventional Component-Based Lifecycle

Most SDR waveform development today is based on the SCA. It is therefore based on components and component models. An idealized lifecycle for the lifecycle development is as follows:

- Model the platform and waveform architectures.
- Verify the model consistency and correctness.
- Implement the components in code; where possible, generate component behavior from the models.
- Build the component executables.
- Deploy the waveform component executables to the platform nodes.
- Load, execute and test the software.
- Iterate until the required functionality and quality has been achieved.

See Figure 2 for a graphical view of the lifecycle.

Here, the key point is that deployment is performed *after* generation, implementation and building. A component executable can be deployed without change to any compatible node. The deployment information is not used in the generation and implementation tasks.

There are very good reasons why a component-based approach has become the norm in SDR development. Among other benefits, the resulting components are highly portable across different platforms.

However, there is a cost. Opportunities for component optimization are limited because generated components rely on the existence of a specific hardware abstraction layer or middleware. Communication code is invoked through a generic, portable interface.

This does not mean that no optimization is possible. On the contrary, optimization strategies are based on middleware, and middleware vendors have finely tuned their products over the years. A modern ORB greatly reduces the number of instructions executed when communicating components are collocated. However, there are limits to this type of optimization. A hand-coded direct call will always beat an optimized message.

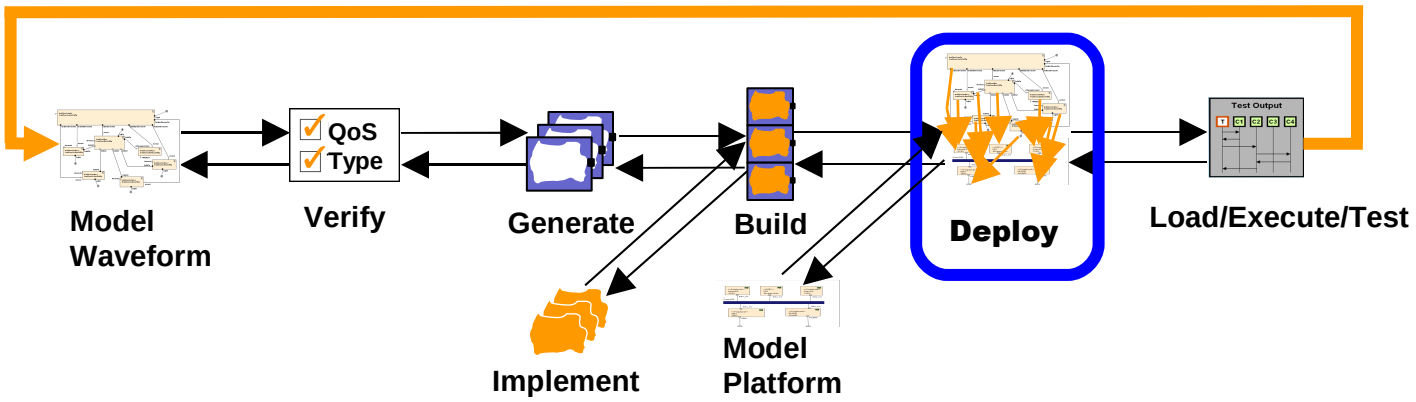

Figure 2: the conventional component-based lifecycle

4. REORGANIZING THE LIFECYCLE

In moving forward to component-based development we have seemingly lost the opportunity to deliver hand-optimized performance. But is this really an unavoidable tradeoff? In this section we will see that it is not—that we can have our cake and eat it too.

4.1 Identifying the Problem

The basic problem in the component-based lifecycle described above is that generation is performed before deployment information is available. As a result, the waveform components must interact through some consistent, homogeneous communication system (middleware or hardware abstraction layer).

Taking a slightly different view, we can observe that a component-based approach is generally identified with a model-based approach—the components are modeled before they are implemented. However, the model perspective is not complete. Most critically, *the "contract" of the generated code is identical to the contract of the model*. The model and code interfaces are treated as different representations of the same thing. There is an unstated assumption that a model will be mapped to a single code pattern for any given target platform. Despite the initial modeling, this approach is implicitly code-centric—or equivalently, binary-centric. The model is simply a graphical representation of the code.

With these insights we can take another look at the SDR development lifecycle and reorder it to maintain both portability and optimized performance.

4.2 The Optimized Lifecycle

The optimized development lifecycle for developing component-based SDR waveforms is close to the conventional one, but there is a vital reordering. The steps are:

- Model the platform and waveform architectures.
- Verify the model consistency and correctness.
- Deploy the waveform component executables to the platform nodes using a precise model.
- Implement the components in code; where possible, generate component behavior from the models.
- Build the component executables.
- Load, execute and test the software.
- Iterate until the required functionality and quality has been achieved.

See Figure 3 for a graphical view of the lifecycle.

This is exactly the conventional lifecycle, except that the deployment is done *before* generation and building. The key benefit is that no universal HAL or middleware is needed. In fact, several middleware or HAL facilities can coexist in the same system, and the optimal one (or a minimal internal communication mechanism) can be used for each communication path.

The reordering described here supports *deployment-aware generation* or DAG, the use of deployment information to generate code. DAG enables simultaneous component portability and optimization.

4.3 Deployment-Aware Generation

Full use of DAG requires a full and precise specification of the deployment target platform and a precise specification of the deployment itself.

Target platform information has been part of software development based on Model-Driven Architecture® or MDA® for at least a decade and a half. However, this *platform-aware generation* is based on knowledge of the *generated component's* target location only. The important information in platform-aware generation is the target environment: the RTOS or BIOS, middleware and the like. The generated code is context-independent. The same code will be generated regardless of other components' locations.

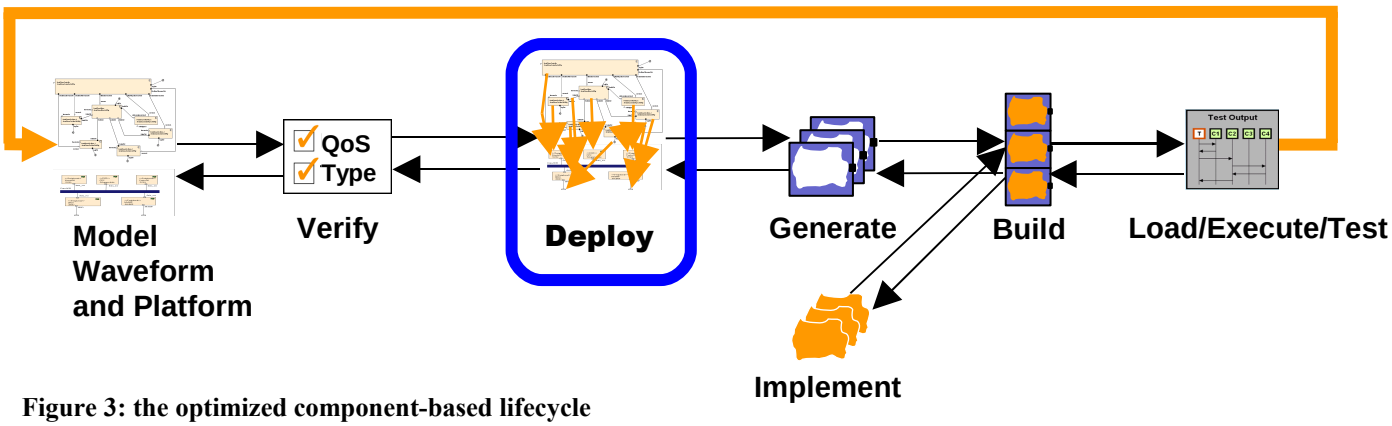

Figure 3: the optimized component-based lifecycle

By contrast, DAG is context-sensitive. The generated code depends not only on the location of the generated component, but also on the locations of generated cooperating components. If two components are collocated they may interact through direct function calls (for synchronous communication) or shared memory (for asynchronous communication). If they are not collocated the communication mechanism may be through middleware.

DAG of a communication path can even be affected by the location of a third component. If a service is only used in one collocated place, it may be provided through a local API. If it must be available globally, a remote interface may be provided to all clients.

Up to this point we have only discussed the deployment of waveform components to platform nodes. However, DAG code can also depend on the deployment of individual communication paths between components. For example, control and data paths between a single pair of components may have completely different quality-of-service (QoS) requirements. DAG generation can map each path to the best communication mechanism. This powerful capability has been largely ignored in past because conventional development lifecycles have not been able to make use of communication path deployment specifications.

DAG applies most naturally to component-based behavior as described in the introduction, not functional behavior. Component-based behavior is normally tightly stereotyped and well-adapted to code generation. Optimization (whether automated or by hand) requires knowledge of the deployment context. By contrast, functional behavior varies considerably in structure between components. While high-quality generation is possible, it is most often hand-generated. Since the entire rationale of component-based development is to encapsulate component internals from their environment, it is independent of component context knowledge.

This distinction between the treatment of component-based and functional code is enabled by distinguishing between the contracts of the modeled component and the generated code. The contract of the (possibly hand-written) functional behavior is expressed in terms of the internal API. The generated component behavior presents the external API that is the contract of the modeled component.

The functional behavior developer works against the minimal, efficient internal API. The generated component code uses the optimal communication mechanism for each communication path. This is the same approach that is used to hand-optimize non-component-based waveforms.

4.4 Iteration

Any interesting piece of software is developed over multiple iterations, and this is especially true of SDR waveforms. DAG fully supports iterative development—in fact, it enables iterative development.

When any change at the global level carries a heavy cost at the detailed level, development managers will naturally be reluctant to explore too far. DAG-based automated optimization of communication details frees the developer to experiment and explore at an architectural level. Since locally optimized code can be generated at the click of a mouse, the investment in a specific deployment is low.

This leads to global optimization. The goal is not to avoid making mistakes; the goal is to make mistakes (i.e., experiment) quickly and cheaply. DAG automation drives down the cost and time required for optimized deployment evaluation.

5. CONCLUSION

5.1 Other Applications of DAG

This paper concentrated on the application of Deployment-Aware Generation to optimizing communication between components. While communication is an obvious aspect of waveforms that can be optimized with respect to deployment information, it is far from the only one.

For instance, timing, log and other services can be implemented and accessed in different ways depending on component locations. Encryption may be optimized out depending on the exposure of communications.

At a more complex level, fault management behavior can be deeply affected by deployment decisions. If two components are guaranteed to fail together, they do not need to manage failures in each other. Of course, this generalizes to control of startup, shutdown and reconfiguration.

5.2 Summary

Hand-coding maximizes optimization, but limits portability. Conventional component-based SBC development maximizes portability, but limits optimization. Deployment-Aware Generation maximizes portability and optimization. The cornerstones of this approach are the right ordering of deployment and generation in the development lifecycle, and precise models of the waveform, the platform and the deployment of the former to the latter. When these come together, software-defined radio developers can deliver waveforms that are both optimized and portable.

REFERENCES

[1]JTRS Standards, Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS), *Software Communication Architecture Specification Version 2.2.2*, 15 May, 2006. http://sca.jpeojtrs.mil/downloads.asp?ID=2.2.2.