bitreactive

The Secret Twists to Efficiently Develop Reactive Systems

The three most powerful principles to develop reactive systems are the following:

- Divide and conquer, then reuse effectively
- Visualize your system
- Automatically and formally verify everything, always

These are such fundamental principles that one can hardly call them new. It is surprising, however, that they do not live up to their potential in programmers' daily routines. In this whitepaper, we present the secret twists to make these principles work for you and your organization.

Bitreactive Whitepaper 2012-12-1

Your domain probably requires applications that are more reactive than you think.

Introduction

Reactive simply means to react appropriately to events, keeping an eye on time, and handling events as parallel as possible, but in an ordered way. The ability to react correctly is important for several reasons:

- If your application has a user interface, it should quickly react on user input and deliver feedback to users instantly.
- If your application is heavy on communication, it needs to process incoming messages quickly, and react appropriately on network troubles without introducing any errors.
- If your application decomposes tasks into subtasks, it should execute subtasks in an optimal way, as concurrent and independent as possible, so that the overall result can be provided as quickly as possible.

Reactive systems typically pose strict requirements on security, dependability, responsiveness and efficiency. They must work robustly even if the network is down or if there are transmission errors. In addition, many systems must work without any intervention from the outside environment.

To meet these requirements requires a careful design of reactive behavior.

It is difficult to develop reactive systems right with pure programming only.

The problem is that reactive behavior is hard to get right by pure programming: Concurrent behavior becomes very complicated even for small systems, resulting in synchronization problems. Describing concurrency and synchronization using pure code, in a tangible manner is therefore almost impossible. To understand a system, a programmer must jump around in the code in an endless number of possible scenarios. Ad hoc and last-minute-fixes add more lines of code and obscures the system further. As a result, the business-specific code is buried down under thousands of lines of code. Valuable logic is therefore hidden, not only from the project stakeholders, but also from developers. How the system is designed will affect the potential of its evolution and the cost of change. Therefore, it is important that the system is designed correctly from the start. A successful software design for reactive systems follows three design principles:

- Divide and conquer, then reuse effectively.
- Visualize your system.
- Automatically and formally verify everything, always.

Though these principles seem intuitive, doing it right is not. In this whitepaper, we present the necessary twists to make them work for you and your organization.

Principle 1:
Divide and Conquer, then Reuse Efficiently

Why: Manage complexity and develop flexible systems that support teams of experts.

Challenges: Interfaces of standard programming languages are surprisingly weak, and do not guarantee that all components work together correctly.

Solution: Reactive blocks have unique contracts that ensure that the combination of blocks work together perfectly.

It is possible to create applications from building blocks if it is done right.

There are several well-known benefits associated with a system that is divided into smaller units. For one thing, the complexity of the system is reduced resulting in a more flexible and maintainable system. Separate people can work on each component of the code and gain expert control over certain units. With this follows the possibility of reusing code. Effective reuse means to reuse knowledge, working designs and analytic results in such a manner that one does not have to understand all details of the work already done.

Building a system block by block and reuse code that already works is a naive approach. After all, most programming languages have some mechanisms for handling modules and interfaces (also called Application Programming Interface, API). However, as easy and practical this approach sounds, it is rarely implemented right. Traditional interfaces are surprisingly sparse, since they most often only cover method signatures of interfaces, that is, their name and which types are passed. This is not sufficient information to connect modules correctly. Interfaces do not describe in which sequence certain operations may be used. Neither do they formally express from which thread an

	Traditional APIs	Reactive Blocks
Method names	✔	✔
Types	✔	✔
Sequence of calls	-	✔
Execution times	-	✔
Thread compatibility	-	✔
Truly bi-directional	-	✔

Reactive blocks have an interface description that extends APIs. This ensures that the combination of blocks work together perfectly.

operation may be called, or if an operation is blocking. A wrong assumption of this kind may deadlock an entire system. Of course, some information may be provided by the documentation of an interface. But since the description is given in free text, it may be inconsistent and incomplete, and a compiler may not detect and inform about violations. As a consequence, developers must study internal code or use time with experimenting when using a new API. Moreover, most APIs are clumsy for truly bi-directional connections, which are daily business in reactive systems.

In contrast, reactive blocks are software modules, especially suitable for reactive systems. These blocks are similar to APIs, but they contain an additional behavioral contract that ensures that reactive blocks are correctly connected. A behavioral contract covers information about the sequence in which parameters must be provided, when parameters are expected and simple but effective timing information.

Anything that can be written in code can be encapsulated in a reactive block. A block may for instance represent a component, a function, parts of a system or a protocol. Even platform-specific functionality and legacy code can be encapsulated by a block.

Principle 2: Visualize Your System

Why: Graphics are the most efficient way to get an overview of a system, and graphics are the best way to specify concurrent behavior and intricate synchronization.

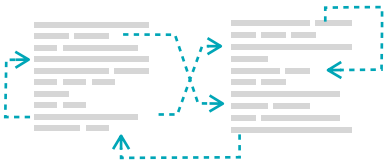
Challenges: Graphics are not always better than code. Keeping code and graphics in sync are difficult.

Solution: Visualize what works best in graphics and code what belongs in code. Reactive blocks make it possible to have a seamless integration between graphics and code.

Understanding how the system works is the number one prerequisite for a maintainable design.

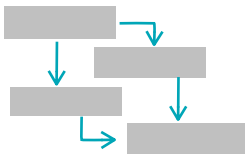
Code is essentially one-dimensional in nature; it follows the sequential order of a program counter. Since reactive systems are not linear but highly concurrent and branch, documentation is especially difficult when using code only. By just looking at the code, it takes a long time to understand what a system does. Even if only small changes need to be done, the entire context has to be studied to ensure that the changes are correct. Often, only a small fragment of all code lines contains your specific and relevant application logic. In fact, typically only 10% of code is differentiating code. The rest is only a technical necessity that obscures the important stuff. Business-critical assets are buried down under thousands of lines of code. As a result, the implementation is not only hidden from project stakeholders but also from the developers.

Graphics provide a visual model of the system and are inherently better suited to document concurrency and synchronization. In most applications, about 60% of the code represents such coordinating code. This is also the code that is most 'boring' to develop: It is difficult to understand, difficult to get right and introduce a high risk of undiscovered errors.



Code is better for

- Detailed algorithms
- Content of operations
- Determine data types
- Keep detailed data



Graphics is better for

- System overview
- API contracts
- Interaction states
- Keep the state of the system

A consistent combination of code and graphics is key for high productivity.

However, using graphics to describe a system is not all-or-nothing: To make graphics practical, they must be used only where graphics work best, namely getting an overview of the system structure and its concurrent behavior. Code is better at detailed operations, contents of operation, determining data types and keeping detailed data.

Due to the behavioral contracts of the reactive blocks, it is possible to automatically generate the code that represents the reactive behavior. Documentation is thus always updated, neatly in sync with graphics that everyone can understand.

Instead of piles of code, your system consists of a hierarchy of building blocks.

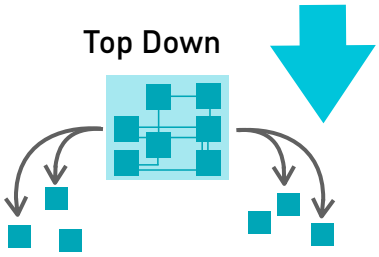
When code is encapsulated by reactive blocks, functionality is made understandable not only for system developers but also domain experts. Reactive blocks become a solid foundation for collaboration and reuse across disciplines and rapid prototype development. The system can be maintained and evolve block by block without risking the immature lock up to choice of technology.

Reactive blocks make development flexible. Systems can be built iteratively with a combination of top-down or bottom-up strategy.

Iterative, Agile

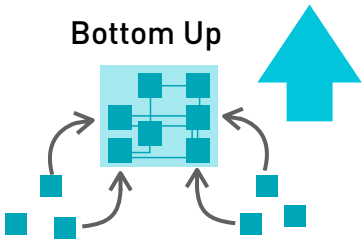
- Bottom-up or Top-down can be applied to all blocks in a system separately

Top Down



- Develop the contracts for the blocks first
- Refine with inner details later

Bottom Up



- LEGO-like, starting with existing blocks
- Perfect for prototyping

Principle 3: Automatically and Formally Verify Everything, Always

Why: Reduce errors by avoiding them in the first place and verify that the system works correctly for all scenarios.

Challenges: Formal verification is difficult to begin with, and not always scalable.

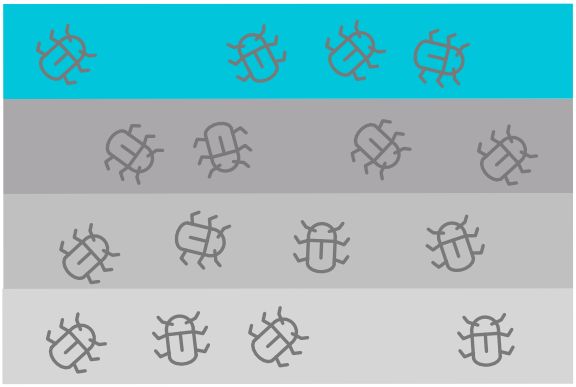
Solution: Reactive blocks can be analyzed separately, since they are encapsulated by their contracts. The automatic analysis can be started with a simple action.

It is important to show that a software design works.

Extensive testing eliminates many errors in the system. However, concurrency will lead to unforeseen errors, such as race conditions, deadlocks and starvation that may not show up before the system is put in use. What is needed is a formal analysis.

An automatic verification computes all possible scenarios that the application can go through at once. While the principle is rather simple, there are two challenges that usually come with model checking:

- One needs a proper description of the system on the right abstraction level. To get such a representation from code, major manual effort is necessary, often even more than writing the actual code.
- Models of realistic applications are too complex and the analysis takes too much time.



A formal analysis can detects errors in the system that cannot be detected by other means.

Reactive Blocks make the formal analysis feasible by solving both these challenges. The automatic verification uses the graphical part of the application together with the behavioral contracts of the reactive blocks as input. This makes formal analysis easy, as no setup is required. The analysis can be started by a single action with just the push of a button.

What makes reactive blocks extremely powerful is that behavioral contracts make it possible to analyze each block separately. This reduces the computational effort drastically. The time it takes to complete a formal analysis will be governed by the time it takes to analyze one block, which is typically under 1 second in 95 % of blocks in a system. Having reactive blocks with self-contained units of work directly implies that the analysis is scalable, also for large systems.

Tools Support for Reactive Blocks

With the Reactive Blocks SDK, you can build systems from reactive blocks.

Feature	Description	Benefits
Reactive blocks that extends API interface descriptions	Reactive blocks have special graphical interfaces that serve as a contract and protect the logic inside.	▪ Always reuse correctly ▪ Support highly concurrent programming ▪ Protect legacy code ▪ Flexible design ▪ More efficient teamworks
Library of building blocks ready to use	Bitreactive offers libraries of reusable reactive blocks that provide access to all functionality that a reactive system may need.	▪ Build prototypes fast ▪ Reuse about 70% of code ▪ Quickly gain domain knowledge
Powerful graphical language with a consistent combination of code and graphics	A smart combination of code and graphics ensures that model and code are not redundant	▪ Customer centric development ▪ Make important design decisions ▪ Easy to maintain ▪ More involvement and engagement ▪ Shorter boarding times for engineers
Automatic code generation from graphics	Automatically generates code from graphics and ensures that graphics and code are aligned without redundancy	▪ Graphics and code are always in sync ▪ Documentation is always updated ▪ Code is event-driven and highly concurrent by construction. ▪ The resulting code runs efficiently
Formal Reactive Blocks SDK analysis and automatic verification	The analysis analyzes the actual system description not some abstraction of the system. Each block can be analyzed separately.	▪ Show that your system works ▪ Find a range of strange behavior like deadlocks, errors in synchronizations and race conditions. ▪ Reduce analysis time tremendously by analyzing block individually. ▪ Scalable also for large systems ▪ Input from graphics, no extra input necessary
Visual animation of the errors	The result of the analysis is projected back as animation into the editor. This makes it possible to stepwise simulate the scenarios and find out what is wrong and what can be done to correct an error.	▪ The analysis runs automatically and explains what is wrong. ▪ Easy to understand intricate errors ▪ Correct errors faster.

Conclusion

A successful software design for reactive systems follows the three design principles:

- Divide and conquer, then reuse effectively
- Visualize your system
- Automatically and formally verify everything, always

These principles are easy in theory. With traditional programming, however, it is extremely challenging to master them.

Reactive blocks, on the other hand, are a new type of software modules that make the three design principles a natural part of the design process. Due to the behavioral contracts of the reactive blocks, the result will be consistent and the blocks will work as expected in the complete system. Documentation is always updated, neatly with graphics that everyone can understand.

The result is a consistent combination of code and graphics, which is a key for obtaining high productivity.

Why Bitreactive?

Bitreactive has experience with building robust reactive systems such as M2M applications. This experience is accessible to you through the Reactive Blocks SDK and the library of reactive blocks.

The Reactive Blocks SDK is a software development kit in alignment with the three paramount principles of successful reactive software development.

Bitreactive delivers a consistent set of tools, methods, reactive blocks and training material, and it knows about the challenges with reactive systems.

For more information, visit bitreactive.com.