Spring 2015

Formal Verification of Software Architecture

Ethan McGee
Clemson University

John McGregor
Clemson University

Strategic Software Engineering Group
Ethan McGee, John D. McGregor
{etmcgee, johnmc}@clemson.edu

Formal Verification of Software Architecture

Abstract

The majority of errors within a software project are introduced during the requirements and design phases of the project, yet these errors usually remain undetected until the implementation and test phases of the Software Development Life Cycle when they are most costly and difficult to correct. The use of Formal Methods during implementation allows many errors to be caught during the implementation phase; Formal Method use during the design phase of a project yields similar results allowing earlier detection of architectural inconsistencies and errors. The Architecture Analysis and Design (AADL) language and its verification tools, AGREE, BLESS and Resolute, provide an excellent platform for representing an architecture and its verification requirements. However, determining what proofs are necessary to declare a requirement as sufficiently verified is not well understood. Developing a set of guidelines or basic architecture proof techniques would better enable the adoption of Formal Methods at the architecture level as well as build a foundation on which future research could build.

Example

Consider a control interface for a security-critical system. The system has two levers and two buttons, one of each on either side of the interface. These devices should be connected to a computing element that can handle their events (enabled or disabled). When enabled, the system should store the non-negative time that the device's activation occurred, and both sides should be activated within two seconds of the other.

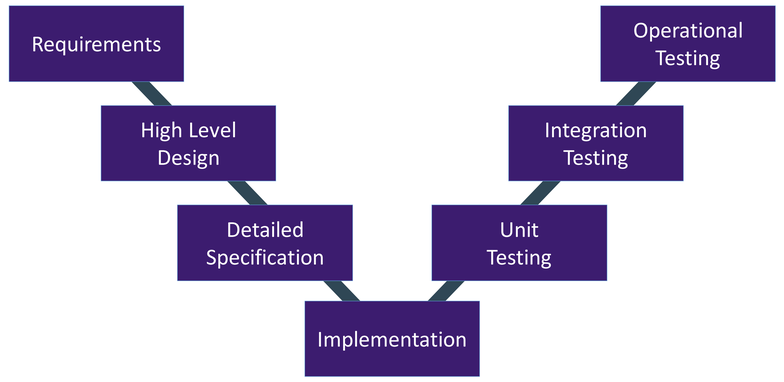

Figure 1: Softwa re Development Lifecycle

Consistency Verification

We have a requirement that our devices must be connected to a computing element that can handle their particular event signals. Without this connection, the devices will be effectively useless because we cannot determine their state. Resolute, a language for model consistency analysis, enables the performing of this type of verification.

```
device toggle_switch
    features
         toggled_up: out event port;
         toggled_down: out event port;
end toggle_switch;

device implementation toggle_switch.impl
    annex Resolute {**
        prove(instantiation_is_reachable(this, instance(launch_interface_thread)))
    **};
end toggle_switch.impl;
```

Figure 2: Model with Resolute Proofs

Figure 3: Sample State Diagram

Compositional Verification

We also have a requirement that our system's clock must produce a monotonically increasing sequence until a max value is reached inducing a wrap to 0. Our proof should affirm that the wrap occurs and that we do not enter an undefined state. AGREE, a language for compositional verification, allows us to perform this analysis by analyzing an abstract behavior representation of the model.

```
process clock_process
    features
        time: out data port Base_Types::Integer;
    annex agree {**
        guarantee "clock value will be greater than or equal to 0": time >= 0;
    **};
end clock_process;

process implementation clock_process.impl
    subcomponents
        time_interrupt_thread: thread clock_thread.impl;
    annex agree {**
        node Counter(init: int, incr: int, reset: bool) returns(count: int);
        let
            count = if reset then init
            else prev(count, init) + incr;
        tel;

        eq x1 : int = Counter(0, 1, prev(x1 = 86400000, false));

        assert time = x1;
    **};
end clock_process.impl;
```

Figure 4: Model with AGREE Proofs

Behavioral Verification

When our requirements deal with neither consistency or values but the behavior of the system, we can use AADL to specify the behavior of our models through the use of annexes. BLESS, a state machine verification language, can interpret these annexes to validate they meet the behavior requirements, such as the both sides of our interface having to be activated within two seconds of the other.