Project Plan (2nd Draft)

Team Dec 15-12: @PaniniJ
April 2nd, 2015

Advisor	Dr. Rajan	
Client	Dr. Rajan	
Team Members	Dalton Mills	*Webmaster*
	David Johnston	*Team Lead*
	Kristin Clemens	*Communication Lead*
	Trey Erenberger	*Key Concept Holder*

Table of Contents

Abstract: This project's main goal is to provide Java developers with a simple way to safely write concurrent programs using the Panini capsule-oriented programming paradigm. This is achieved by producing a deliverable `.jar` file that will easily integrate with standard Java workflow and toolchain. The annotation processor in this `.jar` will be triggered by any standard Java compiler to generate capsules and other artifacts needed for their operation. These capsules will run concurrently in a capsule system.

Project Statement

Modern multi-core CPUs provide powerful support for concurrent programs, but it is very hard to build correct programs and systems which take full advantage of this hardware. PaniniJ is a Java-like programming language with strong support for the capsule programming abstraction. The language has been designed such that programs written in the language cannot include certain common concurrency bugs.

Though `panc`, the existing PaniniJ compiler, is an extension of the standard Java compiler, PaniniJ code is not easily integrated into existing Java projects. Our project, @PaniniJ, intends to make Capsule-Oriented Programming more accessible to Java programmers than the existing PaniniJ tools can provide.

Our project will implement an alternative mechanism for generating capsules, one which can be used by a Java developer within the Java programming language and built using standard Java build systems (e.g. Ant, Maven, Eclipse and `javac`). Our solution will use annotation processors to easily generate capsule classes which are functionally equivalent to PaniniJ capsules but usable from within a Java development ecosystem.

Prior Work: Background on Capsule-Oriented Programming

TODO: Existing concurrent programming techniques require a developer to ...
TODO: Actor frame of reference?
TODO: Our system will extend existing theoretical work which has been done on Capsule

Usage: Capsule Generation via Annotation Processor

Annotation processors provide a simple way to hook into a standards-compliant java compiler, inspect annotated Java code, and then automatically construct additional Java source files. These machine-generated Java source files are then compiled and included for use in other source code in the project.

Our solution will be distributed as a `.jar` file which users can easily include in their java project. The user can then write a Java class annotated with `@Capsule` and suffixed with `Template`. This original `@Capsule`-annotated class is referred to as a *template class* because it serves as the template from which a capsule class is generated. When this class is compiled, `javac` will automatically call on our annotation processor, which will then inspect the template class and generate a variety of Java source artifacts.

Arguably, the most important of these generated source artifacts are the capsule classes, of which there are four types (Thread, Task, Monitor, and Serial). Of these four types, the Thread-based capsule is the canonical example (and the focus for our implementation in the first term). Every thread capsule instance wraps an instance of the corresponding template type. Then in order for a capsule instance to process a procedure invocation, the capsule delegates to the appropriate method of the template instance.

Additional information about the design of capsules and the our system which is used to generate them can be found the the design document.

Additional information on Java annotation processors can be found at (insert reference here, probably using a footnote).

Requirements

The requirements listed here are expanded on in the Design Document.

Functional Requirements

- Working annotation-based frontend for the generation of PaniniJ-compatible capsules.
- Support for four basic execution strategies (i.e. Thread, Task, Monitor, and Serial).
- Reference-counting garbage collection.
- Translate PaniniJ programs to @PaniniJ programs for use as
    - capsule-generation test cases
    - semantic analysis test cases
    - benchmarks comparing PaniniJ and @PaniniJ performance.
- Intra-capsule confinement analysis.
- Analysis of other yet-unidentified safety/correctness properties.
- Automated selection of appropriate execution policy based on system topology and capsule performance.

Nonfunctional Requirements

- Project Requirements
- Capsule template syntax is simple and declarative.
    - Does not introduce boilerplate code
- Fits Java developer’s expectations by supporting common language features
- Validation and Acceptance Test for each functional requirement.

Solutions Considered

We investigated three possible solutions that would allow us to make capsule programming more accessible:

Option 1. Eclipse Plugin / Modify Eclipse

Many languages create an IDE quickly by extending Eclipse to support custom language syntax and features. The process of extending eclipse is made easier through the use of XText[1]. XText is a library that facilitates creation of Eclipse plugins and language development simpler.

This option would require more maintenance in order to stay relative. Since we would be building on top of Eclipse, our forked editor or plugin would be locked with the version of eclipse we started with, and we would need to perform some merges when Eclipse is updated.

This option would require end-users to download the custom editor or plugin. This could be a potential deterrent from software developers using PaniniJ to create multithreaded programs.

Option 2. Modify Eclipse JDT for PaniniJ

This option was only briefly explored. However, it would be similar to option 1 such that maintenance would be required in order to keep up with the latest changes in Java.

Option 3. Annotation Process

Java 1.8 introduced the ability to create custom annotation processors. Java developers are already familiar with annotations such as `@override` and `@suppress`. We would develop a new annotation along the lines of `@capsule`. When compiled, any user code containing our custom annotation would get processed and we would generate code similar to the Java source artifacts produced by panc.

We opted for the third option of creating a library which uses Java 1.8's annotation processing. The other options were discarded fairly quickly in favor of this approach for the reasons outlined below:

[1] https://eclipse.org/Xtext/

1. Annotation Processing is now standard in Java 1.8. We would not be creating a non-standard solution.
2. Creating a standard solution caters to our ultimate goal of having accessible capsule-oriented programming.
3. It will be easier to integrate into existing projects.
4. Java developers are already familiar with annotations.

Risk & Feasibility Assessment

Risk reduction was the ultimate deciding factor in why we chose to use annotation processing for this project. Moving forward, however, we still have many concerns regarding potential and inherent risks of our proposed solution, as well as questions about the feasibility of achieving our goal of accessible capsule-oriented programming.

Performance

1. How will multiple stages of code generation and type checking affect compiler performance?
2. Will code generated by the annotation processor be as performant as `panc` generated code?

Workflow Integration

1. Will @PaniniJ provide sufficiently familiar development workflow for the average developer programming concurrent solutions?
2. @PaniniJ template and capsule classes are just Java code, but what will be required from the user to make common Java IDEs work seamlessly with @PaniniJ?
3. Will IDE debugger integration be possible? If so, will it work consistently across all IDEs for which @PaniniJ is implemented?

Attainment of Project Goals

1. Will @PaniniJ lower the barrier to entry for developers new to programming concurrent solutions?

Semantic Analysis

1. Will @PaniniJ be able to provide the semantic analysis expected by experienced IDE users? If not, how much will this negatively impact the user's ability to program effectively using @PaniniJ, and thus their desire to use @PaniniJ at all?

Schedule

By the end of the Spring 2015 semester:

- Generation of Capsule and Signature artifacts from annotated template classes
- Generation of Duck Future artifacts to support generated capsules
- Support for Thread capsule execution strategy.
- Benchmark programs comparing PaniniJ and @PaniniJ performance

By the end of the Fall 2015 semester:

- Translate more existing PaniniJ benchmark programs into @PaniniJ for use as tests and benchmarks comparing the performance of PaniniJ and @PaniniJ systems.
- Implement intra-capsule confinement analysis.
- Implement semantic analysis of additional safety/correctness properties.
- Support for Task, Serial, and Monitor capsule execution strategies.
- Automated selection of appropriate execution policy based on system topology and capsule performance.

Cost

This project does not have any hardware components and does not require the purchase of any commercial software. By building on top of research undergone at Iowa State University we receive a vast pool of resources for no cost. We therefore expect our project budget to be $0 and since we are working for free, the cost of development will be $0.

Market Survey/Literature Review

Our final product will compete with existing PaniniJ tools created within the Laboratory for Software Design at Iowa State University, in particular, the `panc` compiler. Our products are comparable because they both provide support for capsule-oriented programming. However, the existing solution has several restrictions that negatively affect its usability for the average Java programmer. Our project seeks to resolve some of these shortcomings.

PaniniJ currently inhibits the programmer from using standard Java toolchains. A client needs to use the special `panc` compiler. However, our solution is easily integrated via the classpath by hooking into the standard Java compiler produced by Oracle. `panc`, on the other hand, is a fork and extension of the Java compiler produced by Oracle. Our product decouples the capsule-generation process from a specific version of compiler allowing for portability across toolchains, compilers, and compiler versions.

Both systems produce comparable capsule systems. The code which our system generates is directly modeled after the code generated by `panc`.

Project's Future

After our work is completed, the project will be released as an open source project on Sourceforge or Github. It is expected that it will then be maintained and contributed to by the ISU Laboratory of Software Design's Panini research group.

If our project is successful and we implement enough functionality to make our product functionally competitive with the existing `panc` compiler implementation, it is expected that @PaniniJ will replace PaniniJ and `panc` in future Panini capsule oriented programming research. This would mean that the Panini research group would build on @PaniniJ in order to add additional features to @PaniniJ as the Panini programming model expands.

Glossary

Panini	A capsule-oriented programming language whose goals are to ease development of correct, scalable, and portable concurrent software.[2]
PaniniJ	A capsule-oriented extension of the Java programming language that runs on the standard JVM platform.[3]
Capsule	Similar to a process, it defines a set of public operations and also serves as a memory region for some set of ordinary objects.[4]
Capsule-oriented Programming	A programming paradigm that aims to ease development of concurrent software systems by allowing abstraction from concurrency-related concerns; entails breaking down program logic into distinct parts called *capsule* declarations and composing these parts to form the complete program using *system* declaration.[5]
Java annotations	A form of metadata, provide data about a program that is not part of the program itself, having no direct effect on the operation of the code they annotate.[6]
Java type annotations	Annotations that are applied to *type use*, which support improved analysis of Java programs way of ensuring stronger type checking.[7]
Pluggable type checker	A custom module used in conjunction with the Java compiler to ensure the state of a variable, with the goal of preventing or detecting all errors of a given variety.
Checker Framework	A pluggable type checker tool developed at The University of Washington[8] that comes with predefined type checkers and provides a framework for developers to build their own custom type checkers.

[2] http://www.cs.iastate.edu/~panini/about.shtml#what
[3] http://www.cs.iastate.edu/~panini/about.shtml#what
[4] http://www.cs.iastate.edu/~panini/about.shtml#capsule
[5] http://www.cs.iastate.edu/~panini/docs/faq.shtml#q_l_whatis
[6] http://docs.oracle.com/javase/tutorial/java/annotations/index.html
[7] http://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html
[8] http://types.cs.washington.edu/checker-framework/