Project Overview
The MISRA-C Compliance Checker will be an open-source system that is used to identify software on embedded medical systems that does not comply with the set of MISRA-C standards. Code that does not comply with these standards is potentially harmful to users of those embedded medical systems. This system will check well-formed, compilable C code against a subset of the MISRA-C standards. This subset will consist of MISRA-C rules that have been deemed statically checkable. These rules shall be prioritized in order of complexity and implemented in order of least- to most-difficult, in order to provide the maximum possible coverage in the time available. The system will provide the user with a list of rule violations and the locations of these violations. Also provided will be a list of possible consequences arising from these violations. By finding these violations, noncompliant code can be identified and fixed, averting failures which could possibly be life-threatening to users.

Basic Requirements
The system’s input consists of the file system path of a directory that contains a set of files that consist of well-formed, compilable C code. In addition to specifying the path to this directory, the user is able to configure which file extensions (.c, .h, etc.) are to be checked by the system. The user is also able to configure the set of MISRA-C rules that will be checked by the system.

When the system is run, the input files are checked for violations of the specified MISRA-C rules. As violations are discovered, the system collects information regarding their nature and location within the input files. Along with this information, a list of known consequences that could possibly arise due to the presence of each violation is printed to an output file in XML format.

Constraints
Five constraints were placed upon the project by our sponsor. Firstly, it was stated that the system must be implemented in Java. This aids us in satisfying the constraint that the system must be independent of operating environment and platform. A third constraint is that the system must not require any external software to run, aside from a JVM. The
fourth constraint demands that the MISRA-C copyright is not violated. The final constraint is that the system must be designed as an open-source and, as such, must only contain freely distributable code.

**Development Process**

The team's process for developing the system mirrored our planned technical process. The process methodology that was used for this project was a slightly modified version of Scrum. Scrum allowed for a high level of transparency in the project's development, which helped mitigate the risk of falling behind. Incremental releases (sprints) were planned for every two to three weeks, this was because the scope of the project is 6 months, and every thirty days was too long.

Overall requirements were captured in a product backlog. For each sprint, a sprint backlog was created that consists of a subset of requirements from the product backlog that was met by the end of the sprint. Scrum's product and sprint backlogs helped focus on a smaller set of features for each release; ensuring they got done.

Our sponsor approved this process, as he wanted an agile technique to getting the project done, however it was not mandated. We chose Scrum because we saw our system mostly as a series of rules to implement, which seemed perfect for breaking down into backlogs. As stated before we also were interested in delivering software frequently to keep the sponsor in the loop as much as possible and to keep our progress visible. We met with our sponsor once a week on Thursday via teleconference.

**Project Schedule: Planned and Actual**

Our team developed the project schedule around our development process. The Scrum sprints were set for every two weeks, so it was a matter of deciding when to release. Initially we laid out release plans for every 2 weeks, such as Friday of week 2, week 4, etc. This proved illogical as we ran into issues right away. We started development in the first week of spring quarter, and so starting and getting development kicked off took a lot longer than simply a regular release kickoff. We had then been mandated that we release a half release right away (at the end of week 1) to make everyone feel comfortable. By rushing to release right away, we actually took longer in getting a feel for what we needed to do. At the end of week 3, we finally got out our first release, with only a few of the rules that we had planned to release. The schedule was adjusted to be every two weeks from the first release. By the time release 2 was due to be released, we discovered by pushing the release back from a Thursday to a Tuesday, we could produce a more complete release. From there the team chose to release every 2 weeks, or if the release could be made more meaningful by releasing a few days later, we opted to push releases back accordingly.

**System Design**

For our system, we originally considered two possible designs. The first idea was to develop our system as a plug-in for an existing IDE (Eclipse). This idea would have taken advantage of Eclipse’s existing static analysis tools. Our other concept was to
develop a stand-alone system based off of parsing technologies. Due to our sponsors request for a stand-alone system, we went with the second idea.

Once we decided on an overall system structure, we based the design off of our key non-functional requirements. The system needed to be modifiable to include additions of rules in the future. Aside from that, the rules needed to be testable to determine the level of accuracy in rule checking. Due to this, we decided to go with a simple system design that implemented the rules in a pipe-and-filter architecture.

The base system consists of a Compliance Checker class, a Properties class, a ViolationList class, and an XMLWriter class. Additionally, each rule is implemented as a separate class. The Compliance Checker consists of a set of steps for checking the compliance of source code. Each of these steps is delegated to another class in the system.

The Properties class consists of a set of methods for loading files to be checked and rules to be checked against. The files are loaded based on a path entered by the user and a list of acceptable file extensions. The rules are loaded based on a list of rules to be used by the system. These are loaded using Java ClassLoaders.

The ViolationList represents a list of violations that have been found in the system. This class is responsible for storing, organizing, and sorting these violations. The XMLWriter is responsible for converting these violations to XML and outputting them to a file that is viewable by the user.

Each rule class is responsible for checking the inputted source code against one specific MISRA-C rule. These classes are responsible for scanning a representation of a source code file and checking that representation for violations. These rules are created using a tool called ANTLR and each rule class consists of a subclass of a lexer, a parser, or a tree-parser. There also exist a generic lexer class, a generic parser class, and a generic tree-parser class that are used by the rules.

The files are passed into the system via a FileStream. This FileStream is piped into three separate sets of filters. The first set is a generic lexer/parser pair. The second set is a list of lexer/parser rules. The third and final pair is a list of multi-stage rules.

In the generic lexer/parser pair, the code is first filtered by a lexer which converts the FileStream to a TokenStream (a stream consisting of a bunch of tokens representing
character and character combinations found in the code). This TokenStream is then converted into an Abstract Syntax Tree (AST) by a parser. The AST is then piped into a set of tree rules which consist of rule-specific tree-parser. These tree-parsers are subclasses of the generic tree-parser class that have some modified functions to check for violations in certain structures in the tree. These violations are then sent to the ViolationList for storage and eventual output by the system.

In the second set of filters, the FileStream is first passed to a rule-specific lexer, which may check the contents of certain tokens for violations to a specific MISRA rule, in addition to performing the same functions as the generic lexer. The TokenStream created by this lexer is then piped into a rule-specific parser, which checks certain token combinations for violations to the specified MISRA rule. As each of these filters finds violations, they are passed on to the ViolationList class.

In the last set of filters, the FileStream is passed to a rule-specific lexer/parser pair, just like the previous filter set. This pair, in addition to checking for some possible violations and converting the FileStream into an AST, also stores some information about the code that is checked so that this information may be used at a later stage. Next, the AST produced by this pair of filters is piped into a rule-specific tree-parsers work just like those in the first set of filters, except that they also take advantage of information found in the lexer/parser pair that is associated with them. Finally, violations found by these filters are passed on to the ViolationList class.

The choice of using a pipe-and-filter style architecture was mostly due to the nature of our rule-checking process. Due to the complexity of some of the rules, it was deemed that parsing was necessary, in contrast to some simpler methods such as regular expression checking. The parsing process consists of several data transformations (FileStream to TokenStream to AST), which is what the pipe-and-filter architectural pattern was designed for. Additionally, our system needed to be modifiable, with different rules be added or removed with ease. Designed each rule as a separate filter allowed for them to be easily added or removed from the system without needing to change the operation of the rest of the system. Lastly, as each rule was designed as a filter, allowing for them to be easily tested. Testing consisted of simply attaching the appropriate input pipe to a rule filter and then checking the output pipe of the filter to see if the appropriate output was produced.

To add to the modifiability of the system, the rules were designed to implement the RuleLexer, RuleParser, or RuleTreeParser interface, depending on which type of filter each one is. Each rule is loaded by Java’s Classloader in a fashion similar to a factory pattern. The system loads a list of rules to be loaded, and then asks the Properties class for an instance of each. The Properties class loads the Classloader for each of the requested rules and then returns the rules as an instance of their implemented interface. This design was also chosen for its additional benefit of reducing coupling. The actual concrete class need never be known, by the system. The system merely has a string representation of a rule, but it only knows that the class it loads is a RuleLexer, a RuleParser, or a RuleTreeParser.
Process and Product Metrics

One of the metrics we employed was time effort per rule complete. The results were as follows:

Ian - 9 - 41 hours = 4.6 hours per rule
Phil - 7 – 31.5 hours = 4.5 hours per rule
Jon - 6 – 21 hours – 3.5 hours per rule
Tracy - 6 - 36 hours = 6 hrs per rule
Bryan - 2 – 6 hours = 3 hours per rule
Overall – 30 rules – 135.5 hours = 4.5 hours per rule

We initially figured our time per rule was going to be around five hours per rule. We were pretty close to that number. The average was 4.5 hours per rule. However this was just for rule work alone, not including failed attempts at other rules, or other work that was unexpected. Knowing that, there was no chance that the same amount of effort could be put into each release as estimated. This helped us cut our scope back a bit and commit to fewer rules per release to make sure that we knew how much was going to get done.

In estimating five hours per rule, we figured that we would get on average one rule implemented per week per person, as other commitments and senior project work was required, especially in the latter half of the second quarter. By being off by a lot, we couldn't reasonably expect to complete the forty five rules we committed to. The most we could've had was around 35, and that was with an optimistic schedule, no sidetracking, and no problems with trying to implement certain rules.

Another set of metrics we kept track of was personal and team effort. This effort can be viewed on our website under Documents, under the Progress Tracking section. This data can be used to see when the team got off track, sidetracked by additional documentation or other class's work. Individually, each rule's effort to completion can be used to see how widely rules can take to be completed (1-20 hours). When something that is estimated to take 5 hours takes 20, it throws everything off, and things get pushed back.

The documentation was a lot heavier than we had expected to, and the poster ate the middle of April for us, so we were far less productive than we had hoped.

Product State at Time of Delivery

At the completion of the product, 30 rules are completed. This was 15 rules short of our 45 rule goal. In terms of features, we decided not to implement one of our planned features, XML Error Reporting, as we did not encounter any system errors that would need to be reported to the user. The system is in a functional state, and should be extendable by the next senior project team. We added a database to manage rules (to make things easier to manage and organize), and line text for violations (as per our sponsor’s request). Finally, we added a graphical tool that the user can use to edit their rule configuration.
Project Reflection

Overall, our project turned out to be a success. Both our sponsor and faculty coach have been consistently satisfied with our deliverables. We believe our success is due to our choice of and adherence to scrum as a process methodology and ANTLR as our main technology. There are also things we could have done better.

The scrum process we adopted fit the project perfectly as we were able to use compliance checker rules as chunks of functionality that could be split up across sprints and across team members. By estimating how long each rule would take, we were able to estimate how much work could be done by each team member for each sprint, and by extension were able to estimate how much of the system would be completed by the end of the project.

Another feature of scrum which helped us greatly was the daily scrum meeting. When we started holding midnight meetings halfway through our first quarter, we saw our productivity increase. The scrum meetings forced us to set significant and measurable goals for the next day and we found out that things started getting done quicker. This helped end our initial slump where a lot of the team was stuck doing research, but research was too vague a goal. The meetings ensured that there was always communication between team members so if one team member got stuck, it would only be a day before that member got help from the others. The meetings were so successful that we believe we would have been even more productive if we started them earlier, perhaps the second week. We put it off until later in the quarter but there was no good reason for doing so.

Our faculty coach was very helpful in getting us a jump start on the project. Our faculty coach started us off in the right direction by telling us about ANTLR, giving us access to previous done work on a project similar to ours and giving us a lecture on grammars, the syntax which we would be dealing with most of the project. This guidance helped us avoid time that would have been spent trying to find the right technology and learn it.

The research approach used amongst ourselves, however, could have used improvement. Though it was a good idea to assign only some members of the team to gather research and learn ANTLR, generally only the research members practiced ANTLR examples and got to learn the syntax. There was minimal meeting time planned where research members were allowed to show ANTLR to the team. We should have planned team practice session or other activities that would have had members involved in practicing ANTLR earlier.

ANTLR turned out to be well-suited for our needs; many of the rules we implemented were not difficult thanks to ANTLR and the grammar we used. The design that we chose for implementing rules, as plug-ins, was also a good choice. By making rules independent of each other team members were able to work simultaneously on different parts of the system without too much dependence on each other. The plug-in design also makes it easy to continuously add new rules. This helped us attain the modifiability requirement from our sponsor.

Early on in our project we were faced with the major decision of upgrading our grammars to use ANTLR v3 or continuing the project in ANTLR v2. We came to the conclusion that the benefits of v3’s GUI tool and easier syntax did not outweigh the time required to
completely convert the grammars and having to regression test the grammar due to our changes. Though we did start on the conversion, we were able to reassess our risk early enough where it did not lead to too much time spent.

We believe we have made a great deal of progress on this project and it is ready for the next team to finish. Our maintenance manual has very detailed instructions that, combined with examination of our existing rules, will allow the team to quickly continue rule implementation, leaving them plenty of time to improve their process or implement other features.