Institute of Informatics
Ludwig-Maximilians-Universität München

A Language Server and IDE Plugin for CPAchecker

Bachelor Thesis

Adrian Leimeister

Supervisor	Prof. Dr. Dirk Beyer
Mentor	Thomas Lemberger

Submission Date	March 30th 2020

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum Unterschrift

Abstract

Formal verification of software is an useful addition to testing when trying to eliminate unwanted behavior and errors. Multiple approaches for formal verification exist, such as model checking or program analysis, for which a multitude of different tools exist.

CPAchecker is a tool that makes it possible to combine advantages of both model checking and program analysis, and can be extended to include new verification ideas. With modern powerful, feature rich Integrated Development Environments (IDEs), most development tools are easy to integrated into a development workflow, but most verification tools are only available as commandline tools or via web interfaces.

The solution proposed is an implementation of formal verification into IDEs with the help of the Language Server Protocol (LSP). This is realized by implementing the interfacing with CPAchecker as a LSP server, with accompanying plug-in for an IDE, for which Eclipse CDT for C/C++ Developers was chosen. The implementation is based on the Microsoft LSP, which means it can be used to implement the functionality for other IDEs with less effort. Feedback from an evaluation of the resulting tool, which was done by conducting a survey among potential users, was included in the final tool. In conclusion, the resulting language server and IDE plug-in are sufficient for basic use of formal verification.

Contents

1 Introduction

Modern society is always changing, and software is integrated into everyday life. This leads to always changing or new usecases and requirements for modern software. To accommodate this, software developers constantly have to implement new features or change existing ones, resulting in a larger and more complex codebase. As the complexity increases, programming errors become unavoidable, and can lead to undesired behavior. Especially in safety related applications like self-driving cars or medical equipment, this can lead to catastrophic results and has to be prevented.

Testing is a common way for developers to find errors and wrong behavior. Large collections of tests are run against the software during development, testing a components behavior with different inputs in varying situations. Based on the results of the tests, errors can be found and removed, but testing cannot find every error, nor guarantee correctness of the program.

As concluded by Beyer and Lemberger in "Software Verification: Testing vs. Model Checking" [BL17], formal verification can and should be employed in addition to testing. For using formal verification, desired behavior must be specified, and a verification tool or verifier tries to proof that the program is fulfilling the specification. The specification describes properties of the program, like liveness or unreachability of certain code locations (error labels), that must hold for the program to be verified as correct regarding the specification. If a property is violated, the verification tool may also output a violation witness, describing the program path that led to the violation. With this information, the developer can trace the violation in the program, fix the issue, and verify again the program against the witness.

One such verifier is CPAchecker, a tool for configurable program analysis, developed by Software and Configurable Systems Lab at the Institute of Informatics at the Ludwig Maximilian University of Munich. CPAchecker allows the use of different approaches on model checking and verification within a single framework. It is written in Java and can be used as a commandline tool or via a web service called VerifierCloud.

The objective of this thesis is to make CPAchecker, and thus formal verification, easier to integrate into developers workflow by integrating it into an Integrated Development Environment (IDE) plugin. By implementing the functionality in form of a Language Server Protocol (LSP) server and Eclipse IDE LSP client plug-in, it can be ported to other IDEs with less effort. This way, CPAchecker can be executed from within the IDE, and the results can be displayed within the editor of the IDE as well, eliminating the need to leave the workflow of IDE to run CPAchecker in a commandline oder upload source files with a web interface.

This thesis will first give a short overview and brief comparison of other projects and works that allow for integration of formal verification into a development workflow. The following chapter will introduce and explain the software components selected and used to aid in fulfilling the objective of this thesis. After that, the software architecture will be explained, in conjunction with explaining the issues that were encountered while implementing it. Before coming to a conclusion, the feedback received from a survey, conducted to evaluate the result of this thesis, will be analysed.

2 Related Work

The objective of this thesis is the integration of CPAchecker into a LSP server, and provide a corresponding client plug-in for the Eclipse IDE. There are multiple tools and/or toolchains for formal verification, which also strive to provide integration into a development workflow.

One such tool is CBMC [CKL04], developed by the System Verification Group, consisting, among others, of Members of the University of Oxford. CBMC is model checker for C and C++ programs, and employs bounded model checking to verify memory safety, user specified assertions, pointer safety, and other common pitfalls of programming with C and C++. CBMC is available for a wide variety of operating systems, including Linux, MacOS X and Windows. For integration into an IDE, CMBC is providing an Eclipse CDT plugin called CProver[1], which can be installed from the update site linked on its web page. The plug-in makes use of the Eclipse debugger to navigate counter examples, and uses launch configurations to configure CMBC. It uses a local installation of CMBC, which needs to be installed explicitly before being able to be used. The Eclipse version which it is based on, Eclipse Luna[2], is not up to date, as well as the plug-in itself, which has seen no changes since late 2014. In contrast, the Eclipse IDE plug-in implemented in this thesis relies on the latest Eclipse Features, and already includes an installation of the used verification tool, CPAchecker[3].

Predator [DPV11] is another tool for formal verification, and is developed by the VeriFIT Research Group of Brno University of Technology. Predator is specialized on the verification of sequential C programs working with low-level memory manipulation operations, using algorithms based on graph theory. Predator supports analysis of operations such as pointer arithmetic, handling of invalid pointers and reinterpretation of memory, and aims to be

[1]`https://www.cprover.org/eclipse-plugin/`

[2]`https://www.eclipse.org/downloads/packages/release/luna/sr2/eclipse-ide-cc-developers`

[3]`http://cpachecker.sosy-lab.org/`

able to handle complicated techniques and tricks commonly used in system-level code. Predator aims to be usable for large projects like the Linux kernel, but is not ready for this task yet[4]. Predator can be used as a GCC[5] or LLVM plugin[6], both being actively maintained, providing easy integration of analysis within existing build processes, but is not as compatible as those two toolchains, supporting only Linux and Darwin. As the integration is at the level of a compiler toolchain, presentation of results depends on the level of integration of said compiler into the used IDE. The interfacing with GCC and LLVM is realized with the help of Code Listener[7], an API for building analysis tools which require code parsing, which was presented in [DPV12].

Ultimate[8] is a program analysis framework that is developed by a Division of the Department of Computer Science of Albert-Ludwigs-University Freiburg. Ultimates architecture is split into multiple plugins. Each Plugin is responsible for specific steps of program analysis, and the plugins can be linked together to form toolchains capable of complex tasks. Ultimate Automizer [HCD$^+$13] is one such toolchain, capable of verifying certain properties of a program written in C . Ultimate is available for Linux and Windows, and there is a Eclipse IDE plugin available for integrating into the development workflow, with plans to eventually giving access to all Ultimate toolchains. The Eclipse plug-in relies on the Eclipse CDT feature, and its configuration supports choosing from the available Ultimate toolchains, with views for results provided in an Ultimate specific Eclipse perspective. Development of the plug-in seems to have slowed down to only updates for supporting newer Ultimate versions, and it seems that no online update site for simple installation into the Eclipse IDE is available. A release version of Ultimate Automizer has to be downloaded, unpacked and added as a local update site in order to install the plug-in.

3 Used Technology

The following chapter is focused on introducing and describing the protocols used, and explaining the major software components of this thesis.

[4]"The analysis itself is, however, not ready for complex projects yet." `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`, visited on 24.03.2020

[5]`https://gcc.gnu.org/`

[6]`https://llvm.org/`

[7]`http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/`

[8]`https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/`

The Problem: a Matrix

	Go	Java	TypeScript	...	Language N
Emacs	Plugin 1	Plugin 2	Plugin 3	...	Plugin N
Vim	Plugin 1+N	Plugin 2+N	...	...	Plugin N*2
VSCode	...				
...					
IDE M					Plugin N*M

Figure 1: Illustration of the complexity of language support without LSP.[9]

3.1 Microsoft Language Server Protocol

IDEs offer a lot of convenience and useful features for writing code: refactoring, jumping to a definition, linting, auto complete, just to name a few. The implementation of these features usually differs for every supported programming language, and requires a deep language specific knowledge. Even worse, most implementation are specific to an IDE in addition to being specific for a programming language. This means that providing language support for N languages to M IDEs requires N times M different, language and IDE specific implementations, as illustrated on the left side of figure 1.

The LSP[10] tries to solve this issue. Although originally created by Microsoft for their Visual Studio Code editor, it since has become an open standard for providing language support to IDEs. The core concept of the LSP is to separate the code into two parts, called ”language server” and ”language client”, which then communicate using the LSP .

The ”language server” part contains the code that is specific for the language that the server is providing features for, like finding a declaration, renaming/refactoring an object or calculating suggestions for auto completion. These functionalities are then made available via the interfaces specified by the LSP . The ”language client” contains the code that is responsible for adapting user requests, for example for an auto completion suggestions, from the IDE specific format, to the interfaces specified by the LSP .

The communication protocol used by LSP is based on JSON-RPC[11], which is transport agnostic and thus enables different transport options between server and client, like sockets, pipes or via HTTP. Transport agnostic

[9]`https://langserver.org/`. Visited on 08.03.2020

[10]`https://microsoft.github.io/language-server-protocol/`

[11]`https://www.jsonrpc.org/specification`

The Solution: Clients and Servers

Language	Server	IDE	Client
Go	Server 1	Emacs	Client 1
Java	Server 2	Vim	Client 2
TypeScript	...	VSCode	...
...		...	
Language N	Server N	IDE M	Client M

Figure 2: Illustration of the best case complexity of language support with LSP.[15]

communication enables running language server and language client in different processes, which in turn enables the use of a different programming language for each of them. A language server for C++ might be also written in C++, but can provide its features to an IDE or Editor with a language client written in Java.

With this concept, in theory, it is possible to reduce the aforementioned complexity for supporting N languages in M IDEs to N plus M different implementations, as illustrated in figure 2. In reality, while many generic (as in "not specific to a certain language server") language client implementations, such as LSP4E[12] for the Eclipse IDE, can connect to any language server, they may not be able to use all features of every language servers. Features such as custom settings that are sent to or requested by the server, additional user interface elements or commands, require the development of a language client specific to a language server. While this means that the complexity is still N times M implementations, the work required for the specific language client implementations are still greatly reduced. Many generic language clients such as LSP4E for the Eclipse IDE or vscode-languageclient[13] for Visual Studio Code are made to be a base for extension. By using them as a base, only these specific features need to be implemented.

Nonetheless, this dramatically reduces the amount of work required, as the language server part can be reused, and the generic language clients simplify the language client implementation.

Figure 3 describes a possible communication sequence between language client and language server. The first type of communication are simple notifications. When the user opens a document associated with a language

[12] `https://projects.eclipse.org/projects/technology.lsp4e`

[13] `https://github.com/microsoft/vscode-languageserver-node`

[15] `https://langserver.org/`. Visited on 08.03.2020

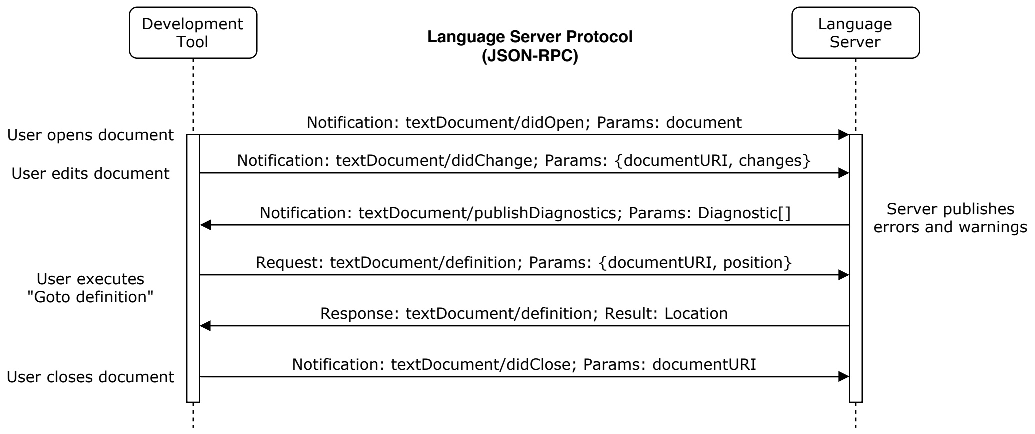

Development Tool
Language Server Protocol (JSON-RPC)
Language Server
User opens document
Notification: textDocument/didOpen; Params: document
User edits document
Notification: textDocument/didChange; Params: {documentURI, changes}
Notification: textDocument/publishDiagnostics; Params: Diagnostic[]
Server publishes errors and warnings
User executes "Goto definition"
Request: textDocument/definition; Params: {documentURI, position}
Response: textDocument/definition; Result: Location
User closes document
Notification: textDocument/didClose; Params: documentURI

Figure 3: Example communication between language client and language server.[17]

server, the client sends a "textDocument/didOpen" notification with information about the document to the server. The server may then already start actions, like processing the document to find errors. Assuming that the document did not contain any errors, the server does not need to send information to the client. The user now edits the document, adds an error to his source code. As a result of the editing, the client sends a "textDocument/didChange" notification to the client. Again, the server now process the document, and finds the error. The server now prepares diagnostics information, like a description and the location of the error, and sends a "textDocument/publishDiagnostics" notification to the client. The client adapts the information contained in the notification, to display an error marker in the editor of the document.

The second type of communication are requests. When the user issues a command, e.g. "Goto definition", a "textDocument/definition" request is send to the server. The request contains the URI of the document, and the position inside the document for which the user requested the definition. The server now, as it posses the necessary information about the language, will search for the definition, and will return the location as the return value of the request. The client can now make the editor display this location.

[17] `https://microsoft.github.io/language-server-protocol/overviews/lsp/img/language-server-sequence.png`. Visited on 02.03.2020

3.2 CPAchecker

CPAchecker[18] is an extensible framework, and new verification approaches can be added by implementing the configurable program analysis interface, or CPA for short, and its required component interfaces, as defined in "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis" from 2007 [BHT07]. The operation of CPAchecker revolves around the CPA algorithm. Figure 4 shows a simplified version of the CPAchecker architecture. The program code to be verified first needs to be converted into control flow automata, or CFA for short. The CPA algorithm uses a CPA interface, which depending on the configuration, may be a composite of multiple CPAs, to check this CFA against the specification. This allows the combination of two of the main approaches in formal verification, program analysis and model checking. Additionally, techniques such as CEGAR [BL13], short for counterexample guided abstraction refinement, or k-induction [BDW15] can be used to further mitigate disadvantages of certain approaches and contribute to a more accurate verification result and faster verification. It can verify different properties depending on the configuration, like unreachability of certain code locations, absence of null pointer dereferencing, memory safety and deadlocks, to just name a few. If CPAchecker is started by issuing "scripts/cpa.sh -preprocess -setprop analysis.machineModel=Linux64 -default example.c", it will report a property violation if the program in example.c can reach an ERROR label.

3.3 Eclipse IDE

The Eclipse IDE[20] is an IDE, written in Java. Eclipse is very extensible, almost every component included by default is implemented as one or multiple plug-ins. Adding support for a programming language to Eclipse is normally done by developing a Eclipse Feature, which is an aggregation of plug-ins. Eclipse Features for some programming languages, like Eclipse CDT for C/C++ or Eclipse JDT for Java, are officially developed under the umbrella of the Eclipse Foundation, and a lot of other languages and tools are supported by plug-ins developed by 3rd parties. These Eclipse Features need to implement convenience features on a very deep level, specific to the language in question. Fortunately, there is a language client implementation already available for the Eclipse IDE .

[18] `https://cpachecker.sosy-lab.org/index.php`

[19] `https://www.sosy-lab.org/research/prs/Current_CPAchecker.pdf`, page 22. Visited on 08.03.2020

[20] `https://www.eclipse.org/eclipseide/`

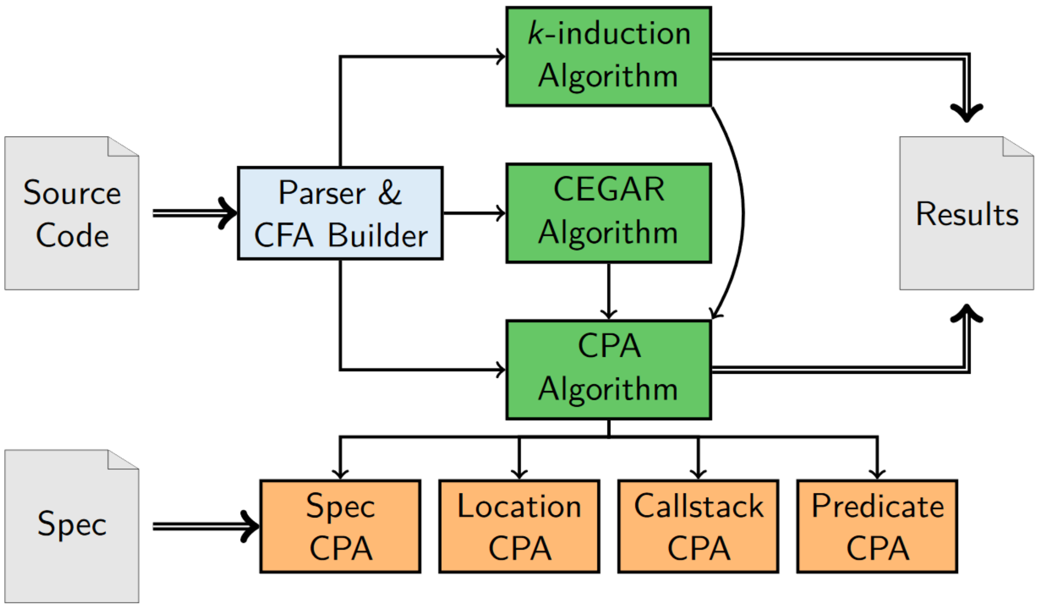


Figure 4: Simplified illustration of the CPAchecker architecture.[19]

As mentioned in chapter 3.1, most generic language client implementations are not meant for standalone use, but as a base for extending. In general, this is also true for LSP4E[21], the language client implementation for the Eclipse IDE. Associating a language server to an editor other than the generic editor, or making preferences available to language server requires a plugin extending LSP4E . However, basic usage, i.e. associating a language server with a file extension and the generic editor so that the server is started and connected to when a file with this file extension is opened, is possible. This means that Eclipse with LSP4E can be used as a development tool for testing a language server during early development, when no specialized plugin is existing yet. Later in the development process, LSP4E can be used as a base for a specialized language server plug-in, that is associated with the right editor and allows for more features. CPAchecker is mainly used for analyzing source code written in C, and are mentioned above, a fully featured Eclipse Feature for C/C++ development exists. With extensibility, a base plug-in for the LSP, and support for C development, and developed in Java, the Eclipse IDE is a good match for the requirements for the implementation of this thesis.

[21]`https://projects.eclipse.org/projects/technology.lsp4e`

4 Implementation

This chapter is focused on elaborating the details of the implementation of the tool that is the result of this thesis, called cpachecker-lsp. Both a language server and language client for integrating CPAchecker into the Eclipse IDE . Starting with describing the interfaces required for using the LSP on the server side, to integrating CPAchecker. The issues encountered will be described, as well as the solutions that were found to solve or work around said issues.

4.1 Language Server

This section will describe the implementation of the language server part of cpachecker-lsp. The language server for cpachecker-lsp is written in Java, and is thus able to interface directly with CPAchecker. In addition, it is also able to submit verification tasks to the VerifierCloud[22], an online service for running verification task on a computing cluster, provided SoSy-Lab of the Chair for Software and Computational Systems at Ludwig-Maximilians-University of Munich. Apache Maven[23]is used as the build and dependency management tool for this thesis, the reasons as to why are explained in section 4.2.3.

4.1.1 Interfacing with the VerifierCloud

As mentioned, one of the configuration possibilities of the cpachecker-lsp language server is making use of the VerifierCloud for running verification jobs. The provided API is available via HTTP requests, and is described on the VerifierCloud help page[24]. Figure 5 shows a simplified UML Diagram for the classes used to interface with the VerifierCloud. CpaCheckerRunnerCloud is the main class of this component, and is initialized with an URI of the document to be tested, an URI of the destination for the result of the verification task. It also receives a Configuration object, containing the settings to use, which are injected into the corresponding member variables. It is responsible for coordinating the steps needed to submit a task, and getting the results, with the help of specialized classes for each request type.

The first step is uploading the document to the VerifierCloud. This job is handled by the FileUploadRequest class, which sends it to the file upload endpoint as HTTP POST request with content type "application/octet-stream".

[22]`https://vcloud.sosy-lab.org/cpachecker/webclient/run/`

[23]`https://maven.apache.org/`

[24]`https://vcloud.sosy-lab.org/cpachecker/webclient/help/`

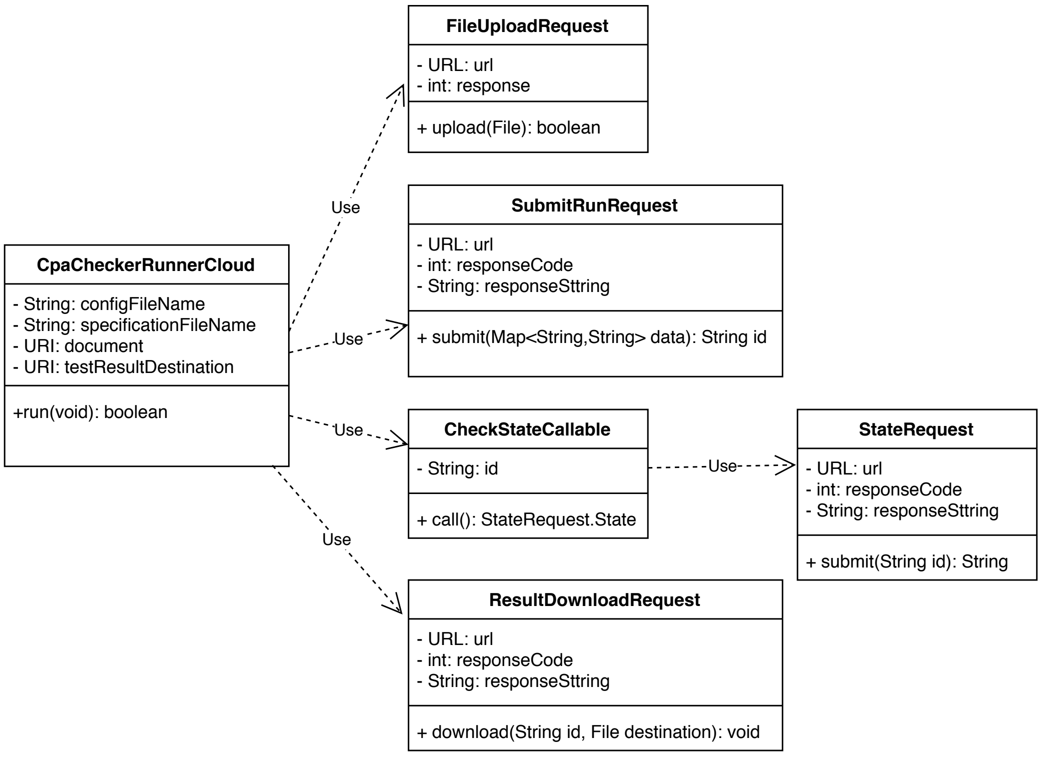


Figure 5: Simplified UML Diagram for the Interface with the VerifierCloud.

Next, the SubmitRunRequest class is used to submit the run to the run endpoint as HTTP POST request, with content type ”application/x-www-form-urlencoded”. This request contains the settings that the VerifierCloud should use, and document to test, which is the previously uploaded document, referenced by its sha256-hash value. The response string of this request is the unique ID of this verification task on the VerifierCloud.

An instance of the CheckStateCallable class is submitted to a ThreadPoolExecutor. It uses the StateRequest class to check the result endpoint if the verification task with this unique ID is finished, cyclically, until the result is available for download.

The archive containing the results is then downloaded to a temporary directory by the ResultDownloadRequest class. After that, the results are extracted from the archive and copied to the location received during initialization of CpaCheckerRunnerCloud.

4.1.2 Interfacing with CPAchecker locally

There are two different possible approaches for running CPAchecker locally.

1. Starting CPAchecker by using the classes directly:

   In this approach, the main CPAchecker class is directly used to perform the verification task. This makes it possible to integrate CPAchecker more seamlessly, and possibly having a greater amount of detail in the presentation of the results, being able to access to output messages and results directly.

2. Starting CPAchecker in a new process:

   This approach makes use of CPAchecker by creating a new process, and starting CPAchecker using the provided shell scripts. Options like configuration file, specification file or processor architecture will be configured via parameters given to the shell script. This allows for integration with less effort.

The cpachecker-lsp language server uses the first approach, starting CPAchecker by using the classes directly. In order to seamlessly integrate CPAchecker into the language server, three main issues have to be solved:

1. Handling of the CPAchecker configuration

2. Handling of messages which are normally sent to the stdout stream

3. Processing of CPAchecker results

CPAchecker uses the configuration system provided by the SoSy-Lab Java common library[25]. It works by building a Configuration object containing a key-value map of options and an instance of an implementation of the org.sosy_lab.common.log.LogManager interface. When using CPAchecker normally, via commandline and the provided shell scripts, the entry point is the org.sosy_lab.cpachecker.cmdline.CPAMain class. Parsing of commandline parameters into a Configuration object is then done with help of the other classes in the org.sosy_lab.cpachecker.cmdline package. The goal was to reuse as much code as possible, to avoid introduction of new bugs when translating from commandline options to options in the Configuration object, and to avoid duplicating code.

This proved to be difficult, as the required classes are not public and thus only available to other classes in their own package. This problem was worked around by implementing a new factory class, called ConfigurationFactory, to be also a member of the org.sosy_lab.cpachecker.cmdline package. Only half of the problem was solved by this though, as a lot of code needed was

[25] `https://github.com/sosy-lab/java-common-lib`

implemented in private methods of the CPAMain class, which could neither be accessed by inheritance, nor could the CPAMain class be configured to work as needed. In the end, parts of the CPAMain class had to be copied to the new factory class. Adaptions where only made when necessary, the input of the create method of the factory is received in a commandline argument string style. The produced Configuration object can then be used as input for creating an instance of the org.sosy_lab.cpachecker.core.CPAchecker class.

CPAchecker uses an implementation of the LogManager interface for outputting messages to stdout stream and/or a log file. As mentioned above, this is added to the Configuration object before instantiating a CPAchecker class. The SoSy-Lab Java common library provides a few different implementations of the LogManager interface, e.g. org.sosy_lab.common.log.BasicLogManager, which is the one normally used by CPAchecker. This implementation uses a java.util.logging.Logger instance. The destination of logged messages can be changed by supplying a class inheriting from java.util.logging.Handler. The org.sosy_lab.cpacheckerlsp.logging.MessageHandler class inherits from java.util.logging.Handler , and sends log messages to the language client via the API defined by the LSP. The instance of BasicLogManager is created, and configured to use the MessageHandler class that is using the LSP protocol, during creation of the Configuration object by the ConfigurationFactory.create method mentioned above.

Because the org.sosy_lab.cpachecker.core.CPAchecker class is used directly, processing of the results is simple. Verification is started by calling the CPAchecker.run method, which returns a CPAcheckerResult object, containing the verification result and information about the violated properties.

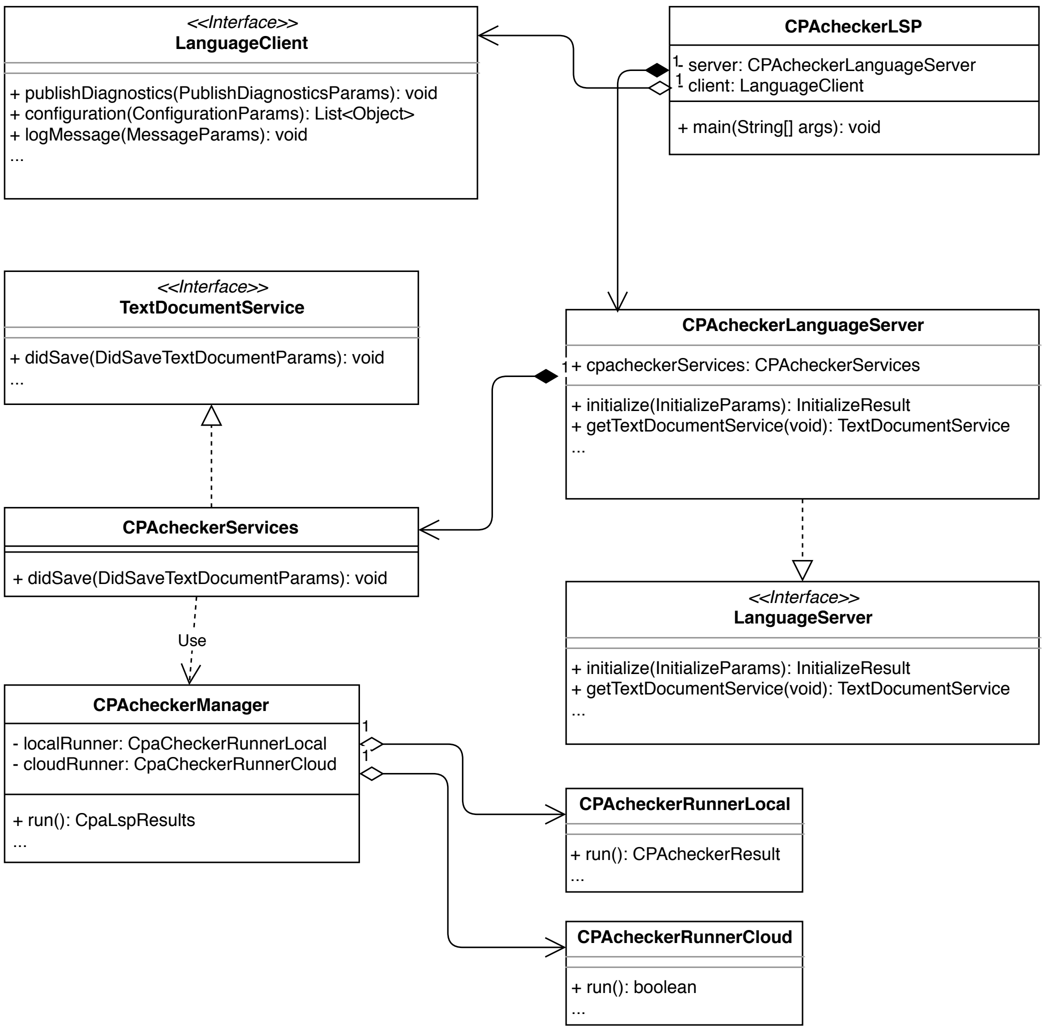
CpaCheckerRunnerLocal is the main class of this component, which is responsible for executing CPAchecker locally, with the help of the classes described before.

4.1.3 Interaction with the Language Server Protocol

Implementation of the Language Sever Protocol is done by implementing interfaces provided by the LSP4J[26] library, a java implementation of the LSP.

Figure 6 illustrates the structure of the language server. The main class for the cpachecker-lsp language server is CPAcheckerLSP in the org.sosy_lab-.cpacheckerlsp package. It is responsible for creating the CPAcheckerLanguageServer instance and connecting to a language client. CPAcheckerLanguageServer implements the LanguageServer interface provided and required

[26]`https://github.com/eclipse/lsp4j`



<<Interface>>
LanguageClient
+ publishDiagnostics(PublishDiagnosticsParams): void
+ configuration(ConfigurationParams): List<Object>
+ logMessage(MessageParams): void
...
CPAcheckerLSP
1
server: CPAcheckerLanguageServer
1
client: LanguageClient
+ main(String[] args): void
<<Interface>>
TextDocumentService
+ didSave(DidSaveTextDocumentParams): void
...
CPAcheckerLanguageServer
1
+ cpacheckerServices: CPAcheckerServices
+ initialize(InitializeParams): InitializeResult
+ getTextDocumentService(void): TextDocumentService
...
CPAcheckerServices
+ didSave(DidSaveTextDocumentParams): void
Use
<<Interface>>
LanguageServer
+ initialize(InitializeParams): InitializeResult
+ getTextDocumentService(void): TextDocumentService
...
CPAcheckerManager
- localRunner: CpaCheckerRunnerLocal
- cloudRunner: CpaCheckerRunnerCloud
+ run(): CpaLspResults
...
1
1
CPAcheckerRunnerLocal
+ run(): CPAcheckerResult
...
CPAcheckerRunnerCloud
+ run(): boolean
...

Figure 6: Simplified UML Diagram for the cpachecker-lsp language server.

by LSP4J, containing the initialize method which is used to negotiate the capabilities and initial settings between language server and client. Capabilities are features that are supported by either client or server, like code completion, which obviously both sides need to know if the other side supports it. Settings are for example that the client should notify the server when a document is changed or saved, which is one of the settings that the cpachecker-lsp languageserver uses.

CPAcheckerLanguageServer.getTextDocumentService returns an instance of CPAcheckerServices, which implements the TextDocumentService interface provided by LSP4J. The methods of this interface get called via JSON-RPC by the language client. Verification tasks are started from the CPAcheckerServices didSave method, which is called when the language client notifies the cpachecker-lsp language server that a C source document was saved. CPAcheckerManager is responsible for getting the configuration via the LanguageClient interface configuration method, and dispatching it to either the CPAcheckerRunnerLocal or CPAcheckerRunnerCloud classes, which were already described.

4.1.4 Sequence of Operations

The components described before in this section 4.1 together form the cpachecker-lsp language server. Figure 7 shows a simplified overview of the sequence of operations. The cpachecker-lsp language server is started by a language client when the user opens a C source document. The language client issues a initialize request to the server, telling the server its capabilities. The server answers this request by providing a list of its own capabilities.

After changing the configuration as to use CPAchecker locally, the user edits and saves the document. The client now send a didSave notification to the server, which now requests the configuration from the client. After receiving the configuration, the server uses the component for running CPAchecker locally. The main class of this component, CPAcheckerRunner-Local, now creates a configuration that can be passed to the CPAchecker classes, mostly in the same way that using CPAchecker via commandline does. The LogManager interface provided to this configuration is configured to output to the language client via logMessage notifications. The verification result is processed and returned. The server now sends the processed result to the client via the publishDiagnostics notification, after which the client displays the results in the editor of the document.

If the user configures the usage of the VerifierCloud, the process is the same, but now the component for cloud verification is used. The main class of this component, CPAcheckerRunnerCloud, now uploads the document to the VerifierCloud. The verification request to the VerifierCloud contains the chosen configuration values, and a hash value identifying the previously uploaded document. After cyclic checking of the job state until it reports that it has finished, the result is downloaded, processed and returned. Same as before, the server now sends the results to the client via a publishDiagnostics to the client for displaying. When the user closes the document the language server is stopped.

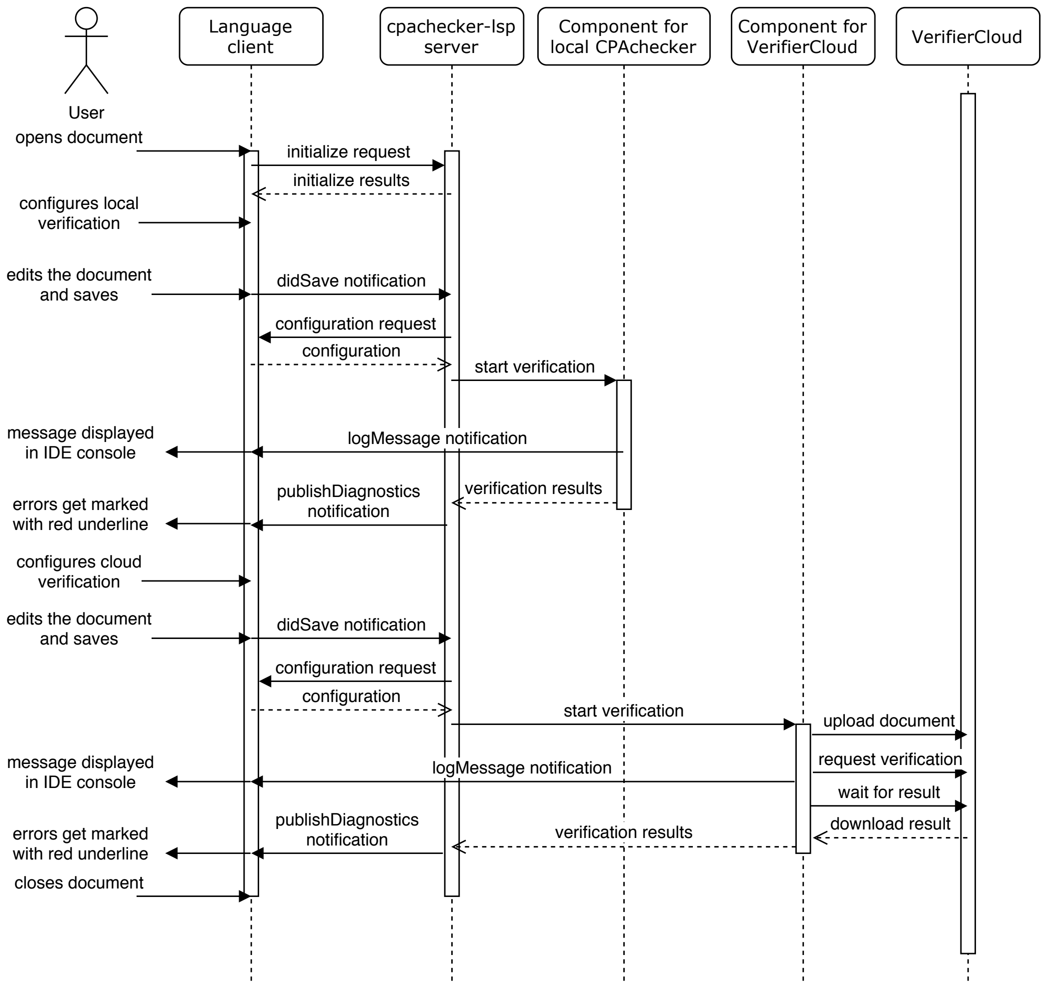


Figure 7: Simplified illustration of the sequence of operations for the cpachecker-lsp language server.

4.2 Client for Eclipse IDE

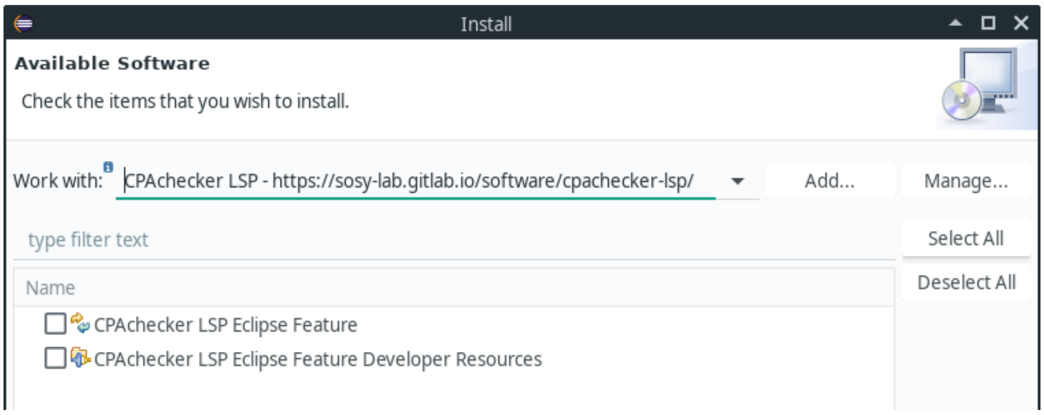
This section will first describe the issues that were faced during the development of the build process of the Eclipse IDE client, and then explain the implementation of said client. The Eclipse IDE language client for cpachecker-lsp is implemented as a plug-in. As mentioned in chapter 3.3, almost every component of the Eclipse IDE is a plug-in. The platform runtime core, which is the base of the Eclipse IDE, finds and loads plug-ins. The plug-in system is based on an implementation of the OSGi framework specification[27], the OSGi specification describes a system for modular platforms. An Eclipse

[27] `https://www.osgi.org/developer/specifications/`

plug-in is thus also an OSGi Bundle.

4.2.1 Description of the Language Client

The Eclipse plug-in for cpachecker-lsp is based on the generic language client LSP4E. It supports running CPAchecker either locally, or by making use of the VerifierCloud, with results displayed in the editor and console output. It can be easily installed by adding the update site[28], which is a result of the build process described in section 4.2.3, in the "Install New Software.." Dialog in the Eclipse IDE, and selecting it from the list for installation as shown in Figure 8. This installation also includes the files for running CPAchecker locally, no separate installation is required. Settings, like local execution or verification with the help of the VerifierCloud, can be changed in the settings section of Eclipse, as shown in Figure 9. Available settings are also the configuration and specification that will be used by CPAchecker, and in case of local execution, additional commandline parameters can be added as well. Execution of the verification task is triggered after a C source code document is edited and saved, described in more detail in section 4.1.4. The result of the verification is then shown on the console view of Eclipse, and as diagnostics information as seen in Figure 10. A directory containing CPAchecker output files is also created.


Install

Available Software

Check the items that you wish to install.

Work with: CPAchecker LSP - https://sosy-lab.gitlab.io/software/cpachecker-lsp/

type filter text

Name

☐ CPAchecker LSP Eclipse Feature

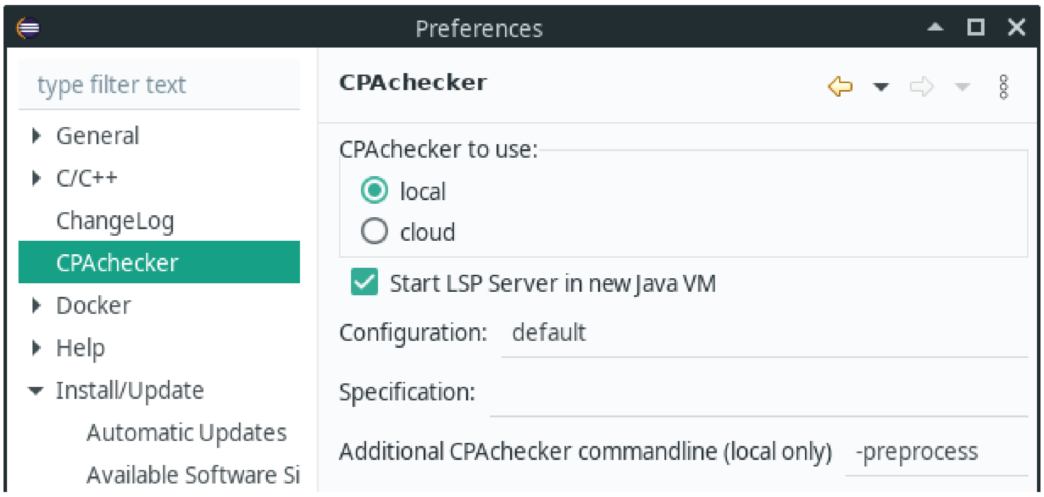
☐ CPAchecker LSP Eclipse Feature Developer Resources

Figure 8: Installing via update site

4.2.2 Implementation of the Language Client

As mentioned before, the implementation of the Eclipse IDE language client for cpachecker-lsp relies on LSP4E, but also needs to use extension points

[28]https://sosy-lab.gitlab.io/software/cpachecker-lsp/


Preferences

type filter text
- General
- C/C++
- ChangeLog
- CPAchecker
- Docker
- Help
- Install/Update
  - Automatic Updates
  - Available Software Si

CPAchecker

CPAchecker to use:
[✓ local] cloud

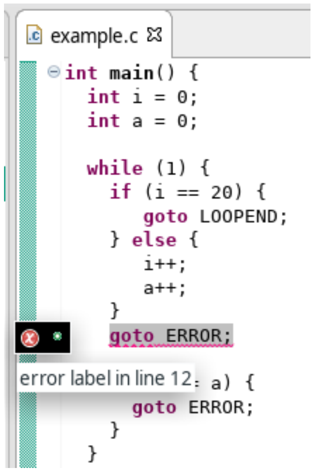
[✓] Start LSP Server in new Java VM

Configuration: default

Specification:

Additional CPAchecker commandline (local only) -preprocess

Figure 9: CPAchecker LSP configuration


example.c

```
int main() {
  int i = 0;
  int a = 0;

  while (1) {
    if (i == 20) {
      goto LOOPEND;
    } else {
      i++;
      a++;
    }
    goto ERROR;
error label in line 12: a) {
      goto ERROR;
    }
  }
```

Figure 10: Property violation information hover

from other plug-ins. Extension points are used for registering functionality with other plug-ins, e.g. registering classes that are then used by those plug-ins.

The following extension points are needed:

1. org.eclipse.core.contenttype.contentTypes

   - content-type:
     This extension point is needed to register a new content type for CPAchecker, with the id ”org.sosy_lab.lsp4e.cpa”. It is a type de-

rived from "org.eclipse.cdt.core.cSource", and thus binds to common C file extensions.

2. org.eclipse.ui.editors

   - editorContentTypeBinding:

     This extension point is needed to associate the new content type "org.sosy_lab.lsp4e.cpa" with the editor plug-ins. This means that files with this content type will now be opened with the associated editors. The ids of those are "org.eclipse.cdt.ui.editor.CEditor" for the C/C++ editor provided by Eclipse CDT, and "org.eclipse.ui.genericeditor.GenericEditor" for the generic text editor.

3. org.eclipse.lsp4e.languageServer

   - server:

     This extension point is for registering an implementation of the StreamConnectionProvider interface, which is used to start the actual cpachecker-lsp language server, and connect the input and output streams for communication. Also, a specialized language client class is registered, for handling of the cpachecker-lsp language server configuration.

   - contentTypeMapping:

     This extension point is used to associate the content type "org.sosy_lab.lsp4e.cpa" with the cpachecker-lsp language server, so that it gets started when a document of this type is opened.

4. org.eclipse.ui.preferencePage

   - page:

     This extension point is used to register the preferences page, an implementation of the "IWorkbenchPreferencePage" class, to the Eclipse IDE.

5. org.eclipse.cdt.ui.textHovers

   - hover:

     This extension point is used to register a class that can be used to display information when hovering over a code location, and is needed for displaying LSP based hover information inside the C editor.

The Eclipse IDE has a concept for getting and setting configuration values in form of a key-value store, called preference store, where plug-ins can store values in different scopes, e.g. different settings for each workspace or project. The preference page registered contains several field editors to set values in the preference store. The configuration values are wrapped in setters and getters by the ConfigurationAdapter class.

As mentioned beforehand, a specialized language client class is necessary. It inherits from the language client class provided by LSP4E, and implements the "workspace/configuration" request[29] specified in the LSP. It returns the list of values requested by the cpachecker-lsp language server, using the getters provided by the ConfigurationAdapter.

The implementation of the StreamConnectionProvider interface mentioned above is actually using the proxy design pattern, which is also accessing the configuration via the ConfigurationAdapter class. Depending on how it is configured to start the cpachecker-lsp language server in a new process or not, a different class is used.

CEditorTextHover, the class registered to the org.eclipse.cdt.ui.textHovers hover extension point, is an adapter class extending the LSBasedHover class provided by LSP4E by implementing the ICEditorTextHover interface required by Eclipse CDT.

4.2.3 Build Process

The cpachecker-lsp language server and Eclipse IDE plug-in builds are managed with the help of a build tool. The description of the build process, the issues that occurred, as well as the solutions to these issues will be described in this section. Before explaining the issues that occurred during the setup of the build process, additional information regarding details about the structure of Eclipse plug-ins, how dependencies are specified and how they can be build by an automated process, is required.

In order to be found and loaded by the platform runtime core, a plug-in has to provide an OSGi manifest file and a plug-in manifest file. This plug-in manifest file contains information about extensions points defined by other plug-ins that this plug-in uses, as well as extension points that this plug-in provides for others. The OSGi manifest contains information about the plug-in, such as the name, version, a class file which to execute upon loading the plug-in, required Java Runtime Environment, and a list of additional paths to be added to the Java Classpath. It also specifies the required dependencies, which in turn must also be OSGi Bundles.

[29] `https://microsoft.github.io/language-server-protocol/specifications/specification-current/#workspace_configuration`

There are two ways to build an Eclipse plug-in. The first way is to build from within the Eclipse IDE, with the Eclipse Feature for RCP and RAP Developers. This is obviously not a solution that allows for automated builds, and is thus not suitable. The second way is to build with Apache Maven[30], a software project management tool that can manage builds, dependencies and more. Although plug-ins can be build with Maven, they have to be build by using a Maven plug-in called Eclipse Tycho[31]. There are some pitfalls regarding Maven projects using Tycho.

Maven projects are defined by a POM, short for "Project Object Model", which is stored in XML format. The POM normally contains all information on how to build the project, like dependencies, but dependencies for Eclipse plug-ins are also defined in the OSGi manifest. Tycho is using both the dependencies specified in the POM, and the dependencies specified in the OSGi manifest, during compilation. Dependencies specified in the OSGi manifest are expected to be in repositories that use "p2" repository format, which is are also used by the Eclipse IDE when installing Eclipse Features and plug-ins. This type of Maven project is called "Manifest-first".

The issue with this is that the dependencies that specified in the POM are not specified in the OSGi manifest, so they are not automatically discovered and loaded by the Eclipse IDE platform runtime core later, and thus not available during runtime. This issue occurs with cpachecker-lsp language server from chapter 4.1. The cpachecker-lsp eclipse plug-in has a POM dependency on the cpachecker-lsp language server, as the language server is not a OSGi bundle.

There are two possible solutions to this problem:

1. Configure Maven to just add the dependencies to the build output of the Eclipse plug-in before packaging the artifact:

   For this approach, the cpachecker-lsp language server is added as a POM dependency, and is thus used during to resolve dependencies during compilation, but not during runtime. By using the Apache Maven Dependency Plugin [32], Maven can be configured to copy dependencies to specified locations during the build. This can be used to add the cpachecker-lsp language server to the build output of the Eclipse plug-in before it is packaged. To work around the problem which is that the dependencies are still not loaded automatically by the Eclipse platform runtime core, the paths of the dependencies have to be added manually

[30] `https://maven.apache.org/`

[31] `https://www.eclipse.org/tycho/`

[32] `https://maven.apache.org/plugins/maven-dependency-plugin/`

to the "Bundle-ClassPath" section of the OSGi manifest, so they will be available on the Java Classpath during runtime.

2. Change the project type of the cpachecker-lsp language server to a "POM-first" Maven project:

   By using another Maven plug-in, Apache Felix Maven Bundle Plugin[33], it is possible to generate an OSGi manifest from the POM . A Maven project where an OSGi manifest is generated from the POM is called a "POM-first" project, and the resulting project build artifact is an OSGi Bundle. There are some restrictions though.

   If a Maven project has sub-projects, it is called a "multi-module project", and the sub-projects are called modules. It is impossible to have "POM-first" and "Manifest-first" modules in multi-module project, because the OSGi manifest for the "POM-first" projects are only generated during the actual build step. Dependency resolution happens very early in the build process, and during this process, Tycho needs to read the OSGi manifests of all dependencies, which are not existing yet, resulting in a failed build. With this approach, cpachecker-lsp can not be build in a single step in a multi-module project.

   Moreover, Maven Bundle Plugin is not associated with Eclipse Tycho, and does not provide any Tycho or "p2"-repository specific information. As mentioned before, Tycho tries to resolve dependencies from the OSGi manifest from "p2" repositories. Tycho can be configured to consider Maven repositories for resolving dependencies. For this to work, the "POM-first" Maven project needs to be added as a dependency to both the POM, and the OSGi manifest, of the Eclipse plug-in Maven project. Tycho then generates "p2"-repository metadata, but only for OSGi Bundle dependencies, so non-OSGi Bundle dependencies of the "POM-first" Maven Project, like CPAchecker, are still not resolved.

As solution number 2 still does not fully solve the problem, and has additional drawbacks, the decision was made for solution number 1.

To make easy installation of the Eclipse cpachecker-lsp language client plug-in to an Eclipse IDE installation possible, some additional projects are necessary. Firstly, an Eclipse Feature project is needed, which defines which plug-ins are included. It also specifies the software license that it is released under, the URL of a repository or update site from where it can be obtained

[33]https://felix.apache.org/documentation/subprojects/apache-felix-maven-bundle-plugin-bnd.html

from, and, again, also contains information about other Eclipse plug-ins that are required for operation. Secondly, an Eclipse Update Site project, from where aforementioned Eclipse Feature project can be obtained from, is required. Both Feature project and Update Site project can also be build with the help of Tycho.

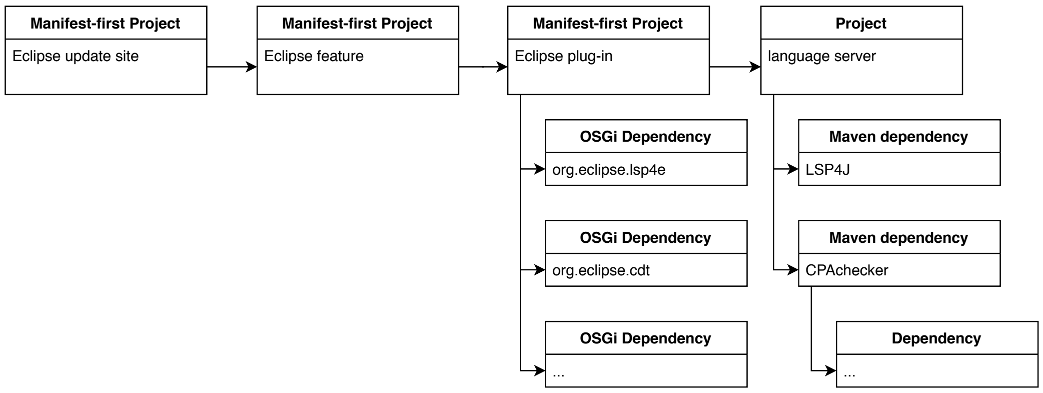


Figure 11: Illustration of dependencies and project types

In the end, the 4 projects were aggregated into a multi-module Maven project, as illustrated in Figure 11:

- Language server:

  A normal Maven project, described by its POM.

- Language client plug-in:

  "Manifest-first" project, build with Tycho, with dependency to the language server project defined in its POM, configured to be added during build and manually added to the classpath.

- CPAchecker LSP Eclipse Feature:

  Build with Tycho, containing a reference to the language client plug-in project.

- CPAchecker LSP Eclipse Update Site:

  Build with Tycho, containing a reference to the Eclipse Feature.

The final build artifact of this multi-module Maven project is an Eclipse Update site that can be either added as a local update site, or hosted on a web server, and can be used to install the cpachecker-lsp Eclipse plug-in.

5 Evaluation

For evaluating the result of this thesis, an online survey among potential users was conducted. The questions that were asked, the additional information that was given to the participants, as well as their answers are available in appendix A. During the survey, the participants were asked to install the CPAchecker LSP Eclipse Feature into their Eclipse IDE installation. After an explanation of the configuration options, they were asked to follow instructions for a basic usage scenario, and to try the tool on their own. The questions inquired about the participants experiences.

The survey included questions about the installation process, configuration, general usage and presentation of results. This section will list the feedback refined from the questions, and the changes that were implemented as a result.

5.1 Analysis of the Results

This section will guide through the results and the analysis of the survey. "Question 1: Installation and Known Issues" served as a an introduction to the survey, and to assure that the participants did read the given instructions beforehand.

5.1.1 Installation

The first part of the survey were questions regarding the installation process. Out of the six participants, five answered "Question 2: Did the installation process work without problems?" with a "yes". The participant that answered with "no" was asked two additional questions, question 3 and 4, which served the purpose of finding the issue and possible workarounds. The answers given pointed to an issue with the version of the Java Runtime Environment (JRE) that is used to start the Eclipse IDE and the cpachecker-lsp language server, which could be worked around by switching the system default JRE to version 11.

5.1.2 Configuration

"Question 6: Are the configuration options enough to cover your use case?" asked the participants to rate the available configuration options on a scale from 0, meaning "Totally unusable", to 10, meaning "Everything covered". This question was asked to asses if the provided configuration options were sufficient. Three of the participants answered with the maximum of 10, and the other three answered with 5, 7 and 8, for an average of 8.33. The participants were given the opportunity to state their usecase and additional configuration options that they would like to see, in question 7. The options requested were:

- options regarding handling of witnesses
- selection of the machine model used for testing
- configuration of conditions for when to run the verification
- configuration of output files
- configuration of resource limits

The last question of the the configuration section was "Question 8: Did you encounter any bugs within the configuration process?". The answers indicated issues with verification with the VerifierCloud and the validation of configuration input.

5.1.3 Usage

"Question 9: Did the plugin do its job for you?" asked the participants if they could use CPAchecker LSP to start verification jobs, and get a visual representation of results, which all participants answered with "yes". The answers to the followup, "Question 10: Did you encounter any bugs or unexpected behaviour?", raised the following issues:

- starting a verification task is signaled by a "DidSave" notification instead of something more descriptive
- error markers do not disappear after fixing the problem
- manual starting and canceling a verification task is not possible
- no notification if an invalid configuration is used
- a not reproducible situation where CPAchecker LSP stopped working

5.1.4 Presentation of Results

The answers to question 12 and 14 showed that the visual error markers and output files appeared for all participants. In "Question 13: Are you satisfied with the presentation of the results?", the participants were asked to rate their satisfaction on a scale from 0, meaning "Unusable", to 10, meaning "Totally satisfied". Two of the participants answered with 3, the other four answered with 4, 7, 9 and 10, for an average of 5.83.

5.1.5 Ideas for Improvement and Comments

The last question of the survey gave the participants the option to voice suggestions, which were as follows:

- improvements to the error marker position
- better presentation of counterexamples
- per file configuration and specification
- manual starting of a verification task

5.2 Implementation of Feedback

This section will cover the issues that were fixed as a result of the feedback, as well as why specific suggestions were not implemented.

1. Wrong Java Runtime Environment (JRE) version:

   CPAchecker LSP requires the Eclipse IDE to run with at least JRE version 11, but if the participants configured only Eclipse to be run with the required version, the plug-in might still not work. The cause of this issue was that the cpachecker-lsp language server was started with the system default JRE, which might have been still, for example, JRE version 8. This issue could be worked around by changing the default JRE to version 11. The feedback was implemented by using the JRE used by the running Eclipse IDE to start the cpachecker-lsp language server. This way, as long as Eclipse is using the right JRE, the language server will also work.

2. More configuration options:

   Selection of the machine model used for verification was only possible to be selected for local execution, via the additional commandline parameters. An additional option to select the machine model was added to the configuration options.

3. Input sanitation:

   It was possible to enter line breaks in the string input fields, which resulted in a corrupted configuration. This issue was fixed by sanitizing the input.

4. Different configuration for each file, possibly via drop-down menu in the toolbar:

   As different files can need different specifications or configurations, a different configuration for each file would be useful. Implementing this feedback was not possible due to time constraints of this thesis.

5. Configuration of output files:

   Configuration of output files while keeping the handling of cloud and local verification the same from the user point of view was not possible due to time constraints of this thesis.

6. Configuration of resource limits:

   The resource limit concept that CPAchecker uses is not easily applicable to the cpachecker-lsp server, as it sets these limits on the process executing the verification task. This would possibly include the resources used by the Eclipse IDE, which would render the limits useless. Further investigation of this problem was not possible due to time constraints.

7. Manual starting of verification tasks:

   Manual starting of verification tasks would be useful to be able to start verification with a different configuration or specification, without the need to change and save the program. This would require a lot of work on the language client side, which was not possible due to time constraints of this thesis.

8. Handling of witnesses:

   Testing against a witness is cumbersome and only possible with local verification. Additional configuration options for easier handling and cloud verification would be helpful. Implementing this feedback was not possible due to time constraints of this thesis.

9. Notifications:

   Some confusing log messages were removed or renamed.

10. Cloud verification:

    There are cases where submitting a run to the VerifierCloud resulted in a timeout. This is a shortcoming of the API used to communicate with the VerifierCloud, where it does not give a response in time when CPAchecker needs to be build on the VerifierCloud before starting a run.

11. Error marker:

    The squiggly underline that marks the line with a property violation was starting at the beginning of the line, and was too short. It now marks the actual content in the line. The issue of error markers not disappearing after fixing the problem seems to be an issue with LSP4E, and is not fixable by the author in the timeframe of this thesis.

12. Better presentation of witnesses:

    The presentation and handling of witnesses could be improved. Instead of only marking the violating line, the path leading to the violation could be highlighted. Implementing this feedback was not possible due to time constraints of this thesis.

6 Future Work and Conclusion

In the future, implementation of language clients using the cpachecker-lsp language server can be implemented for other IDEs, such as Visual Studio Code[34] or CLion[35]. Possible improvements for the language client would be increasing flexibility of configuration, like separate settings for each source file, and providing easier configuration for testing with witnesses. These improvements would be mostly implemented on the language client side, and thus would need an implementation for each language client supporting the cpachecker-lsp language server, while the language server would only require minimal changes. Presentation of results could also be improved, like analysis of witnesses to show the path leading to a property violation. This could be realized on the language server side with the Debug Adapter Protocol (DAP)[36], a sister protocol to LSP for providing IDE independent debugging implementations. Using DAP, the path of the witness through the program could be visualized step by step, like debugging a running program.

Taking the user survey into consideration, the cpachecker-lsp language server and the corresponding Eclipse IDE plug-in can be successfully used to integrate formal verification into a graphical development workflow. It can be installed into the Eclipse IDE easily via the update site[37], and includes everything necessary to start using CPAchecker. Even though some issues exist regarding presentation of results and configuration, they can be worked around, and the result of this thesis is sufficient for basic usage.

[34]`https://code.visualstudio.com/`

[35]`https://www.jetbrains.com/clion/`

[36]`https://microsoft.github.io/debug-adapter-protocol/`

[37]`https://sosy-lab.gitlab.io/software/cpachecker-lsp/`

Overall, the cpachecker-lsp language server and CPAchecker LSP Eclipse Feature are usable for formal verification in a graphical development workflow, and can be used as a base for improvement, extension, and integration into other IDEs.

List of Figures

References

[BDW15] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 622–640, Cham, 2015. Springer International Publishing.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 504–518, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BL13] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on cegar and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 146–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BL17] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 99–114, Cham, 2017. Springer International Publishing.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[DPV11] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 372–378, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[DPV12] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. An easy to use infrastructure for building static analysis tools. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *Computer Aided Systems Theory – EUROCAST 2011*, pages 527–534, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[HCD$^+$13] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. Ultimate automizer with smtinterpol. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 641–643, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Appendix A CPAchecker LSP User Study

A.1 Introduction

Please go to `https://sosy-lab.gitlab.io/software/cpachecker-lsp/` and follow the instructions to install the plugin into your Eclipse with CDT.
A short introduction is also availabe at the linke above. Please try it out and spend a little time experimenting before continuing the survey.

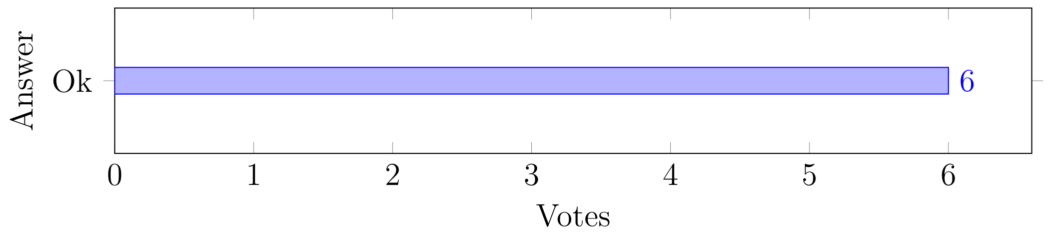
Known Issues/Pitfalls

There are five things to keep in mind:

1. Eclipse has to be run with at least Java 11. As the plugin directly uses CPAchecker classes and CPAchecker requires Java 11, the plugin also requires Java 11 to run.
2. You need your Eclipse version needs to be at least 2019-09, otherwise the installation will fail.
3. The plugin will not work on *.c files that are automatically reopened by Eclipse after a IDE restart. You have to close the editor and open the file again. This is an issue with the CDT editor for .c files. It does not occour with the generic editor.
4. After getting results, refresh your project in Eclipse to make the result directory appear in your project explorer.
5. If you are getting a Message "Submitting run configuration timed out after 15 seconds" when using cloud verification, this is not a problem of CPAchecker LSP but of VerifierCloud failing to give a response code when the most current CPAchecker is not compiled on the Verifier-Cloud.

Question 1: Installation and Known Issues

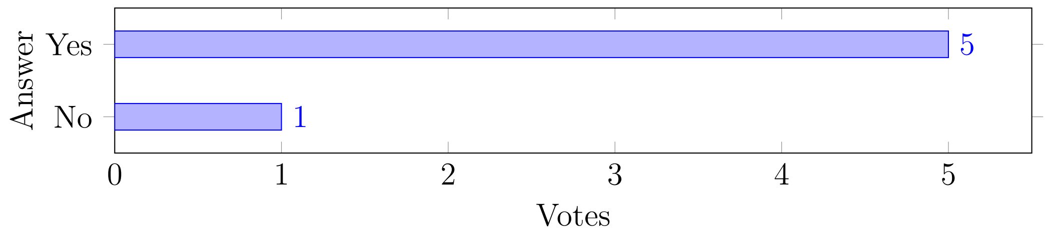
required
I have installed the plugin and i have read the known Issues.
6 responses:



A.2 Installation

Did you have Problems during installation? Did you notice something wrong?

Question 2: Did the installation process work without problems?
required
If yes, skip to question 6.
6 responses:



A.3 Problems during Installation

Question 3: What was the Problem?
required
1 response:

It is not sufficient to select Java 11 during Eclipse installation, if the default command-line Java is 8.

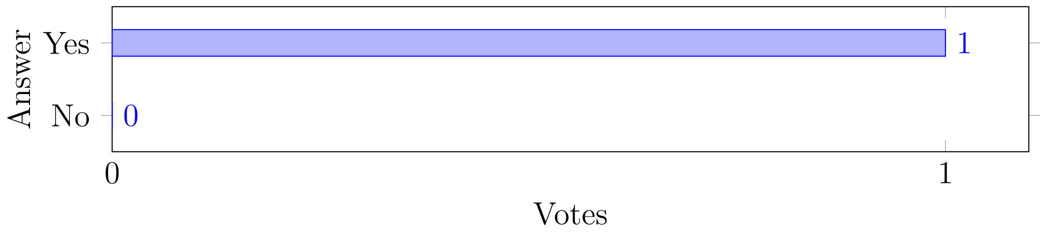
Question 4: Did you find a Workaround?
required
1 response:

Switch Java versions. Is this a problem of some Eclipse component or of the plugin? Could you find out the currently running JVM and use that one?

Question 5: Do you want to continue with the survey?

required

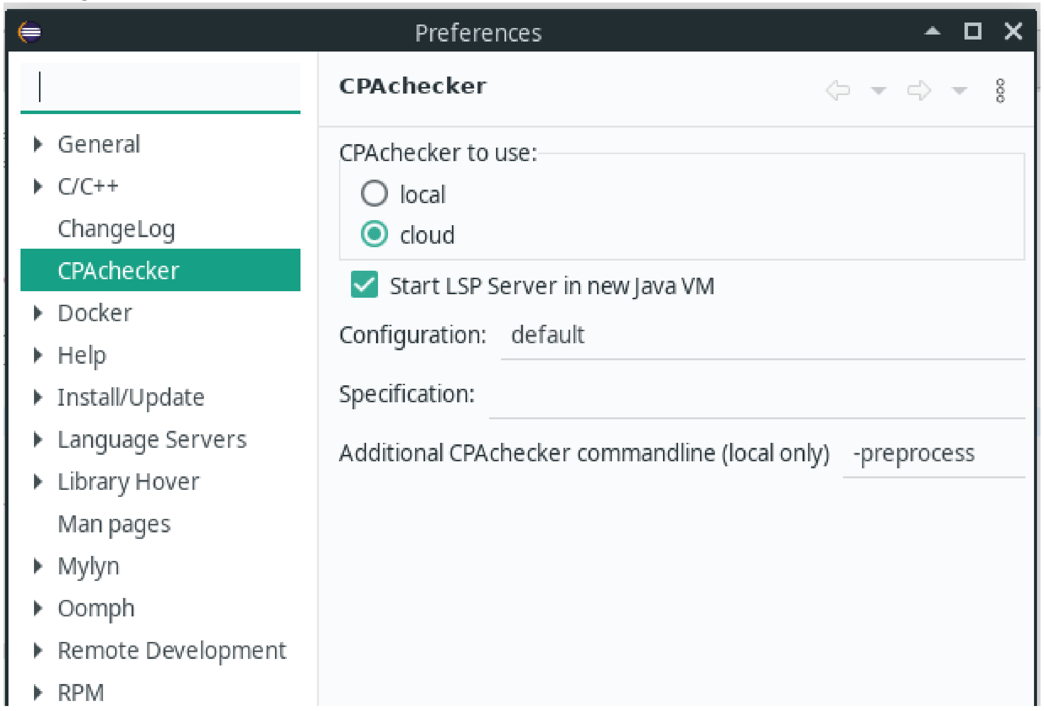
1 response:



A.4 Configuration

An explanation of the configuration is available at `https://sosy-lab.gitlab.io/software/cpachecker-lsp/`

Configuration Options for CPAchecker LSP:



CPAchecker

CPAchecker to use:

local [✓ cloud]

✓ Start LSP Server in new Java VM

Configuration: default

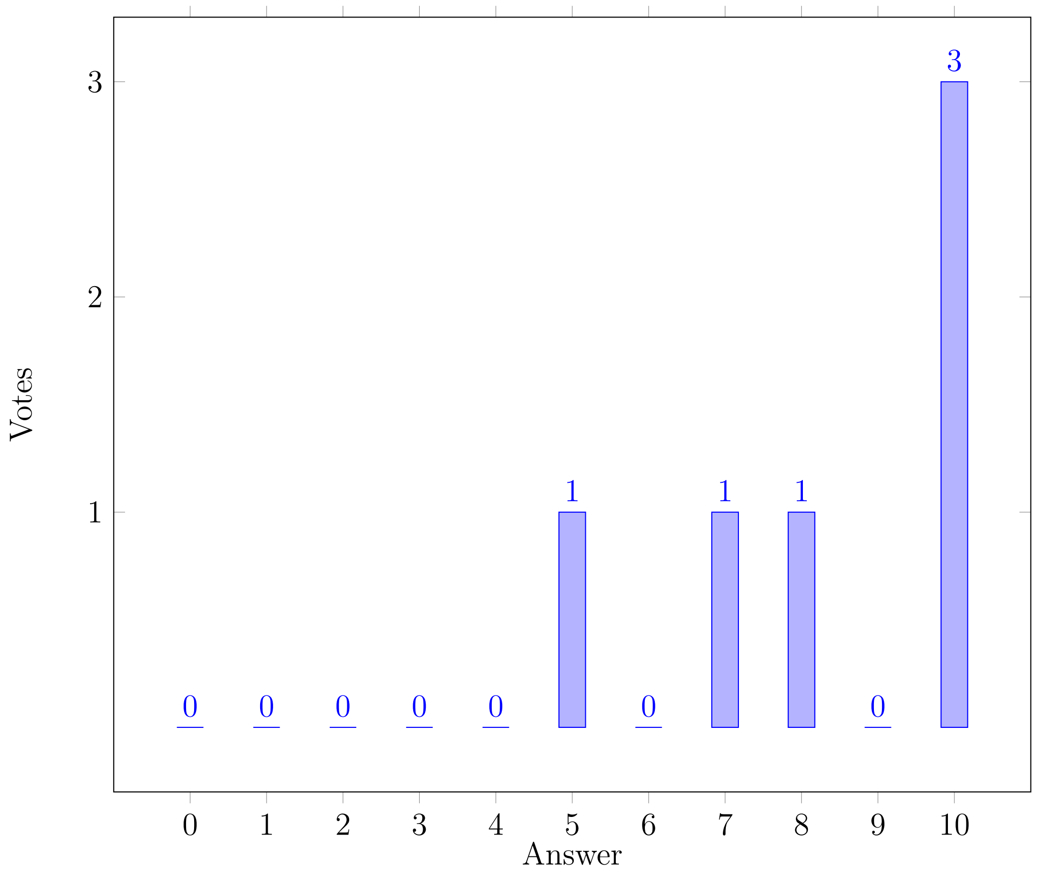
Specification:

Additional CPAchecker commandline (local only) -preprocess

- You can chose if the verification tasks will be done locally or by sending a request to the VerifierCloud at https://vcloud.sosy-lab.org/cpachecker/webclient/help/
- The option "Start LSP Server in new Java VM" should stay selected for normal use. Disabling it is mostly useful for debugging.

- You can chose a configuration file for CPAchecker. Any of the configurations shipped with CPAchecker by default should work.
- You can chose a specification file for CPAchecker. Any of the specifications shipped with CPAchecker by default should work.
- If you selected to use local verification, you can add additional commandline arguments to verification tasks. By default, this includes "-preprocess".

Question 6: Are the configuration options enough to cover your use case?
required
0 = Totally unusable, 10 = Everything covered
6 responses:



Question 7: What is your use case? What is missing, which configuration option would make the plugin more usable for your use case?

6 responses:

A separate field for witness validation would be nice. This is a common usecase for CPAchecker, but at the moment, I would have to write all of the parameters for witness validation in the quite small text field for local parameters. This also means that witness validation doesn't work with cloud execution.
In addition, radio buttons for 32bit/64bit analysis would be good to have. At the moment, this is also only possible through local execution and manually providing '-32' or '-64' as locla parameter.

Testing JavaSMT Solvers

Disabling auto-run of verification job on file save.

To play around

I do my bachelor thesis on CPA-checker as well. The plugin simplifies testing my c benchmarks. The configuration options are sufficient for my use case.

configuring output files, resource limits

Question 8: Did you encounter any bugs within the configuration process?

3 responses:

Cloud execution did not work.

No

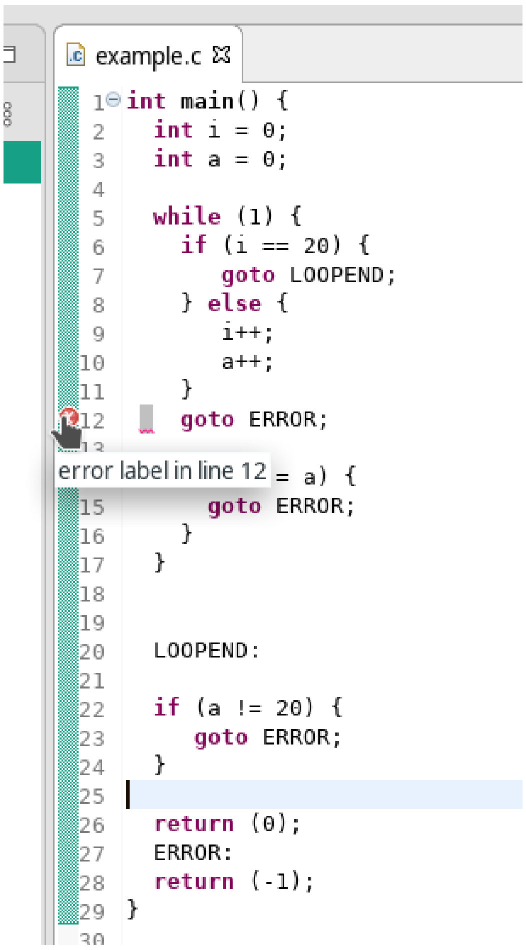
There is no feedback when invalid config/spec/args are specified. Furthermore, there is no input validation, even line breaks can be entered into the text fields (e.g. by pasting text). This leaves 'cpals.cfg' in a corrupt state. Manually editing 'cpals.cfg' is possible, but content is overwritten when a new verification task is started.

A.5 Usage

Verification of a document is started after making a change, and then saving. The following explanation and known issues is also available at `https://sosy-lab.gitlab.io/software/cpachecker-lsp/`
First Steps:

- Create a new C Project
- Download `https://raw.githubusercontent.com/sosy-lab/cpachecker/trunk/doc/examples/example.c` and add it to your project
- Open "example.c", change something minor and save the file. Adding or removing a new line should be enough.
- Changing and saving a file triggers a verification task.
- You will see the progress and the result in the console window.
- Now try adding "goto ERROR;" in line 12, and save
- A new verification task should have been started. This time, the verification result should be "false", and you should see that the beginning of line 12 has a red underline, signifying the error.

Example of a marker for a property violation:



example.c

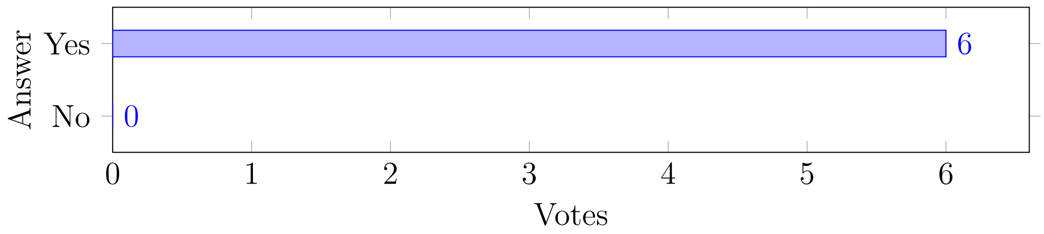
```
 1 int main() {
 2   int i = 0;
 3   int a = 0;
 4
 5   while (1) {
 6     if (i == 20) {
 7        goto LOOPEND;
 8     } else {
 9        i++;
10        a++;
11     }
12     goto ERROR;
13
14                   = a) {
15       goto ERROR;
16     }
17   }
18
19
20   LOOPEND:
21
22   if (a != 20) {
23      goto ERROR;
24   }
25
26   return (0);
27   ERROR:
28   return (-1);
29 }
30
```

error label in line 12

Question 9: Did the plugin do its job for you?

required

6 responses:



Question 10: Did you encounter any bugs or unexpected behaviour?

5 responses:

Cloud execution did not work.

No

You get a 'DidSave' notification after saving a file. If this is supposed to indicate that a new verification task started, a 'Verification started/running...' message might be more appropriate.
There is no indication that a task is still running. Also, there is no indication that a task has failed (e.g. configuration is invalid).
You cannot cancel a running task.
The error-marker does not disappear when the error has been fixed and the verification result is true.
You cannot manually start a verification task.

Issue 1:
The x that marks an error position (in this case: goto ERROR on line 12) did not disappear after fixing the problem (removing line 12). Refreshing does not fix the problem.
Issue 2:
1) I added the example.c file to my project
2) I made changes and saved it
3) Verification run locally without issues
4) Changed the setting to: cloud
5) Verification run on the cloud without issues
6) Changed back to local verification in the settings
7) I deleted the goto ERROR line (line 12)
8) I saved the file and nothing happens anymore. No matter what I change, the plugin won't react. (No error message either)
9) Restarting Eclipse fixes the problem.
10) BUT: I could not reproduce it

Unexpected notification bubbles with "DidSave"

Question 11: Did you encounter any Errors or Exceptions? If so, please provide a stack trace.

0 responses

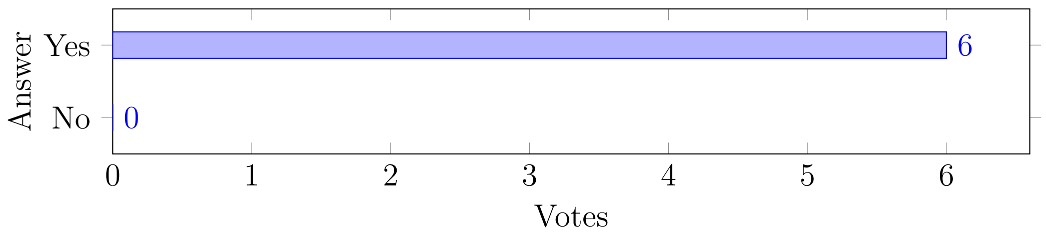
A.6 Presentation of Results

Example of a marker for a property violation

Question 12: Did the markers for property violations appear in the editor?

required

6 responses:

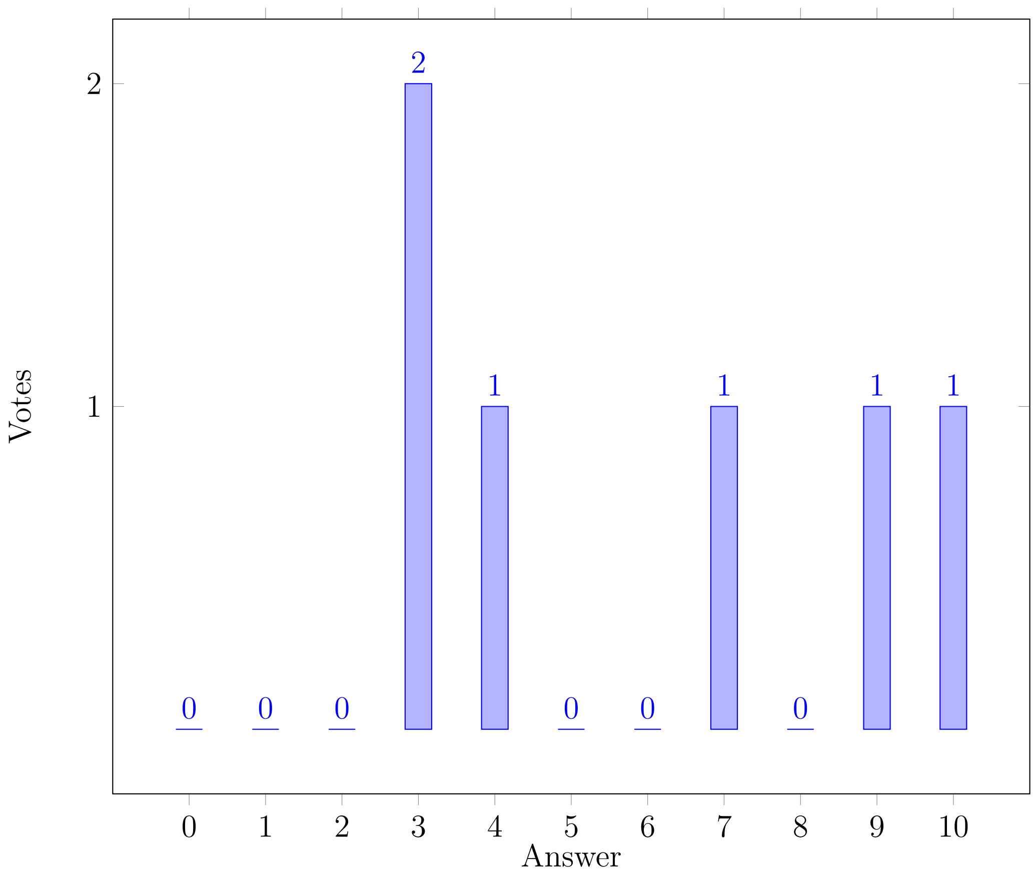


Question 13: Are you satisfied with the presentation of the results?

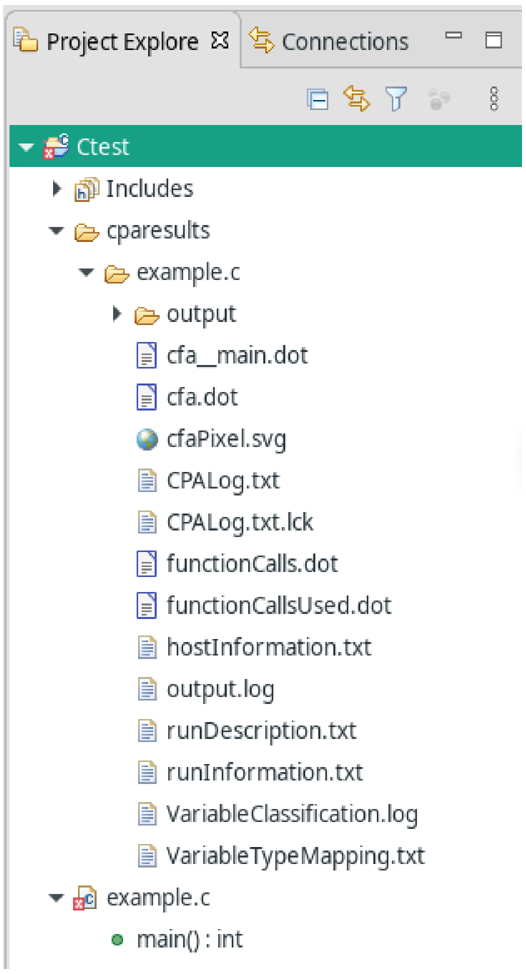
required

0 = Unusable, 10 = Totally satisfied

6 responses:



After executing a verification job, and refreshing your project in the project explorer, it should look something like this. The directory may look a little different, depending on if its a local or a cloud result:

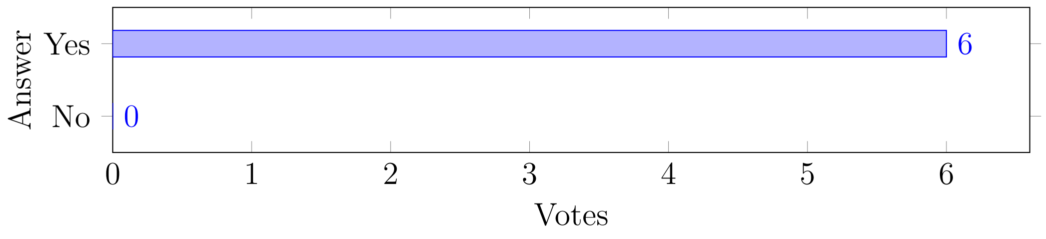


```
Ctest
  Includes
  cparesults
    example.c
      output
      cfa__main.dot
      cfa.dot
      cfaPixel.svg
      CPALog.txt
      CPALog.txt.lck
      functionCalls.dot
      functionCallsUsed.dot
      hostInformation.txt
      output.log
      runDescription.txt
      runInformation.txt
      VariableClassification.log
      VariableTypeMapping.txt
  example.c
    main() : int
```

Question 14: Did you see the result files generated by CPAchecker?

required

After refreshing your project in Eclipse, there should be a directory named "cparesults". This directory contains additional information about the verification run. It contains all output files normally generated by CPAchecker. 6 responses:



A.7 Ideas for Improvement and Comments

Question 15: Do you have any ideas for Improvement or suggestions?

3 responses:

It would be nice if the error indicator, i.e., the squiggly red underline, was not at the beginning of the line, but right after the ERROR encountered. If this is not possible, at least the whole line could be underlined. This may be more visually pleasing.

1) more informative error messages maybe 2) use the running JVM, not the default one for LSP (if that is possible)

Counterexamples need to be presented better (full path etc.), not only the violating line; Verification should not always done on each save, it is too expensive; needs a way to specify configuration & specification easily without changing the IDE-wide configuration, for example if I have different files that need different specifications; standard configurations and specifications should be available in a drop-down list; directory name ”cparesults” is not really useful for users