Master Thesis

YETI 2 - TinyOS 2.x Eclipse Plugin
http://tos-ide.ethz.ch

Benjamin Sigg

Distributed Computing Group - ETH Zurich

Supervisors
Nicolas Burri
Roland Flury
Prof. Roger Wattenhofer

September 10, 2008
## Contents

1 Introduction ........................................ 2
   1.1 Notation ......................................... 2

2 Requirements ........................................ 2

3 Features ............................................ 2
   3.1 Core ............................................. 3
      3.1.1 Build ......................................... 3
      3.1.2 Editor ......................................... 3
      3.1.3 Outline ....................................... 5
      3.1.4 Graph ......................................... 5
   3.2 Parser ............................................ 5
      3.2.1 Preprocessor ................................. 5
      3.2.2 Model ......................................... 6
   3.3 Environments ..................................... 6
   3.4 Wiki .............................................. 6

4 Implementation ...................................... 6
   4.1 Preprocessor ..................................... 7
      4.1.1 Input ......................................... 7
      4.1.2 Output ....................................... 7
   4.2 NesC 1.2.x Parser ............................... 10
      4.2.1 AST ........................................... 11
      4.2.2 Handling syntax errors ..................... 12
      4.2.3 AST-Model ................................... 13
      4.2.4 Bindings ..................................... 13
      4.2.5 Hyperlinks .................................. 13
      4.2.6 Quickfixes .................................. 13
      4.2.7 Code Completion ............................. 15
      4.2.8 Error detection .............................. 15
   4.3 Environments ..................................... 16
   4.4 Core ............................................. 16
      4.4.1 Project Organization ......................... 16
      4.4.2 AST-Model ................................... 16
      4.4.3 Build Chain ................................ 17
      4.4.4 Caches ....................................... 18
      4.4.5 Editor ....................................... 19
      4.4.6 Outline View ................................ 19
      4.4.7 Graph View ................................ 19

5 Future Work ......................................... 20

6 Conclusion .......................................... 21
1 Introduction

A lot of effort has been spent on writing development tools for nesC and TinyOS. Given the widespread use and extensibility of Eclipse [14], it is only natural that several TinyOS-plugins were written for it [1, 2, 3, 4]. Many plugins however support only nesC 1.1 [11], and with the introduction of new features in nesC 1.2 [12] they became obsolete.

The goal of this work was to take one of the existing plugins (YETI, written by Roland Schuler [1]) and extend it such that it supports nesC 1.2. This new plugin should do more than just syntax highlighting, it should provide the developer with accurate error messages and completion proposals.

1.1 Notation

This document is designed to help new developers working on YETI 2. It uses some special formatting to mark important things:

- “New concepts are introduced with italic text and in quotes”.
- Already known concepts will be italic.
- Classes, interfaces, methods and applications will be written monospaced.

2 Requirements

After working with YETI, the requirements for YETI 2 were found to be:

Understanding of NesC 1.2 Means that the plugin is able to parse any code that is handled by the "official" tools [13], understand and apply preprocessor directives, handle any construct that is available in the c programming language [8, 7], and associated tools [9]. The plugin should be able to perform, or at least to simulate, all steps of ncc (the nesC compiler).

Error detection The plugin should be able to detect (potential) errors in the program and inform the developer. Errors can range from the inclusion of a non-existing file or calling a function with the wrong arguments to an overflow caused by implicit type casts of integer types.

3 Features

Most of the features of YETI are still available in YETI 2. This chapter will focus on new features.

While just called “TinyOS plugin for Eclipse”, YETI 2 is actually a set of plugins. The “core” plugin handles functionality which is independent of time and place, like editing files or listing the “make-options”. The “parser” plugin
parses the source files and provides features which depend on the version of nesC. The “environment” plugins handle the interaction between core, parser and tools like ncc or bash. They handle those features which depend on the operating system.

3.1 Core

The most important duties of core is to provide the graphical user interface and to handle the “build system”. It is a buffer between parser and environment hiding them from each other.

3.1.1 Build

The build system has been completely rewritten. The build system tells when which file has to be checked for errors such that no error messages are out of date. The build system takes every event that may invalidate error messages into account and marks files with invalid error messages as “unbuilt”. Later it searches for all unbuilt files, analyzes them, and once valid again marks them as “built”. A restart of Eclipse will not delete these flags.

The build system can be canceled or restarted without loosing work that is already done. That allows to react on events during a build and prevents generating messages which will be invalidated anyway.

3.1.2 Editor

The main text-area where the user enters source code is called the “editor”. The editor supports a variety of (new) features:

Context sensitive syntax highlighting If a variable has the same name as a typedef, it will still not have the same color. This is shown in figure 1.

Some parts of this feature are implemented in parser.

Automatic code completion When doing the "small stuff" like entering an open bracket, the editor will automatically insert a close bracket. That works for multi line comments as well.

Completion proposals The plugin guesses what the developer wants to do and shows him several proposals from which he can choose one. Proposals can range from filling in the name of some field to creating a missing included header file. Figure 2 shows how this can look like.

Hyperlinks When the ctrl-key is held down, Eclipse behaves like a browser. Elements in the source code behave like hyperlinks and lead to the place or file where they were defined. Hyperlinks are a great help for analyzing a project. This feature can only work if parser can handle files accurately. Otherwise they lead to wrong places, which is worse than no hyperlinks at all.
Figure 1: Italic fonts for types, but standard fonts for variables even if they have the same identifier.

call Timer0.

Figure 2: A list of proposals which command to call of the interface Timer0.
3.1.3 Outline

The “outline view” shows the contents of a source file as a tree. The nodes of that tree represent elements such as interfaces or commands. The new outline view not only shows content of one file, but can expand into other files. While the top nodes of the tree come from the current source file, other nodes come from included files. This view updates its content asynchronously, allowing the developer to continue working even while the view is trying to figure out how to open some specific node.

3.1.4 Graph

The “graph view” shows a graph for a module, configuration or an interface. In case of a configuration it shows the wiring of its modules and interfaces. The graph view loads its content asynchronously and is linked to the outline view. Figure 3 shows how a simple graph looks like.

Figure 3: A graph of the configuration MainC. The highlighted edge is selected in the outline view.

3.2 Parser

The parser for YETI 2 was completely rewritten. The new version is a plugin, while the old one remains in core as backup used only if “parser” is not installed.

3.2.1 Preprocessor

There is also a new “preprocessor” plugin. Preprocessor executes directives like the inclusion of a header file, macros and other directives defined in the c-standard. It also tracks the location of each character, a requirement to position
any message or hyperlink at their correct location. Parser adds a view which can be used to look at a file in its preprocessed state.

### 3.2.2 Model

Parser builds a model of the analyzed nesC-application. Each file has its own view of this model. A view can fill some gaps that are only visible from one file, e.g. the actual parameters of a parametrized interface. Parser can use the model or the views to perform complex tasks like type checks, execute static initializers or resolve indirect wiring of interfaces.

### 3.3 Environments

YETI 2 is prepared to work in different environments (Linux, Windows...). Currently there is an environment for TinyOS 2.x on Linux and on Windows, using cygwin for the later. An environment is able to find all important header files which are needed to preprocess and parse a file.

### 3.4 Wiki

Since YETI 2 is intended to be used by many people a small webpage with some basic information how to install and use the plugin was created. This page is located at [http://tos-ide.ethz.ch/wiki/index.php](http://tos-ide.ethz.ch/wiki/index.php).

### 4 Implementation

YETI 2 actually consists of several plugins which are connected with each other through the extension point mechanism provided by Eclipse. This chapter will go through each plugin and describe the most important concepts and features of them.

Figure 4 lists the 6 plugins of YETI 2, the arrows mark dependencies between the plugins.

![Figure 4: Plugins and their dependencies](Image)
4.1 Preprocessor

A preprocessor is an application that takes a source-file as input and outputs another file with some text replaced by rules defined in the input-file itself. The statements in the input can be divided into two groups: "directives" and ordinary text. Directives tell the preprocessor what to do, and ordinary text gets processed.

The preprocessor’s name is TinyOS Preprocessor. While it is an Eclipse plugin, it does not depend on any other plugins. Clients might be interested in the class Preprocessor which does all the setup and offers methods to easily convert a file into a preprocessed stream.

4.1.1 Input

Preprocessors are most often used together with C-source files. Theoretically they are able to process source files of any language, unless that language contains a statement that looks like a directive.

In fact, parser uses preprocessor to generate two different input files for CUP, the parser generator.

There are many different directives and vendors of C-compilers often define their own new directives. Fortunately the number of directives which are actually used by developers is much smaller. The three important groups are:

Inclusion directives To include the content of another file. Can be applied recursively, preprocessor however limits the number of recursive inclusions.

Conditional directives To include or exclude some parts of the input depending on variables set outside the file.

Macro directives To define macros which replace text.

Preprocessor recognizes and executes these kinds of directives. Some other directives are understood but not executed, and anything that remains will marked with a warning message "unknown directive".

4.1.2 Output

There are restrictions which make a preprocessor a complex piece of software:

- while the input is just text, directives still have a specific syntax which requires a parser to resolve.
- macros can change any text, even text in a directive.
- hence a preprocessor cannot just read the file, then parse it, apply the directives and finally generate the output. The first three steps have to be performed at the same time.
• if the preprocessor replaces text then the new text has to be preprocessed as well.

Preprocessor solves these issues with a sequence of modules, each module using the output of its predecessor as input. The modules at the end of this stream can influence the modules at the beginning. The architecture of the involved libraries JFlex [5] and CUP [6] allows that the number of characters in the stream is reasonably small. As a result the last module can react fast enough to change the stream before wrong characters wander into modules where they do not belong.

The modules can be put into three groups:

Stream A “stream” reads characters from a source (e.g., a file or a macro) and provides an interface between source and the other modules of preprocessor. To inject characters into a stream (e.g., a macro replacing text) one can temporarily disable a stream and use another stream. This is called “pushing” a stream over another stream. There are about 10 different streams implemented.

Lexer/Filter The “lexer” looks out for tokens (identifiers, keywords, ...) like any other lexer. But there is also a “filtering system” which suppresses forwarding of some tokens that the lexer finds. A good example is the “conditional-filter” which throws away any token that was found within an unused if/else-block. Another would be the “macro-filter” which pushes a macro-stream over the current stream when a macro-identifier is found. All these filters are implemented as little state machines with no more than 10 states.

Parser The “preprocessor-parser” finally takes the tokens of the lexer and puts them together. If the preprocessor-parser finds a directive it informs the other modules of it. The other modules will then apply the directive. Given the fact that CUP has a lookahead of one token, and the newline token always terminates a directive, the preprocessor-parser recognizes directives before any stream starts reading the next line. If the preprocessor-parser finds tokens which do not form a directive, then these tokens are just stored in a list which later becomes the output.

In addition to the mechanisms just mentioned, there is a feature which stores for each character of the output the “origin”. The origin not only includes the original input file and the exact location in that file but also how the character came into the output. Whether it was by applying a macro or including a file. And if so, the location of the identifier or directive that was responsible for the inclusion is stored as well. All this together allows preprocessor and parser to give the precise location of each error or warning that was issued.

Figure 5 shows how data flows through preprocessor.
Figure 5: Dataflow within preprocessor.
4.2 NesC 1.2.x Parser

The parser plugin is designed to handle any file written in nesC 1.2.9. Tests have shown that parser can also handle the new nesC 1.3.0, but there is no way to tell how long it will be supported.

The parser for (preprocessed) nesC files is a plugin as well.

The project TinyOS Parser is the standard parser. It is connected to the core plugin through the extension point TinyOS.Parser. The project also is responsible for syntax highlighting by fulfilling the extension point TinyOS.Reconciler.

It adds the “preprocessor view” (PreprocessorMultiPageEditorPart) to the editor. It can add an “AST-view” (ASTMultiPageEditorPart) and a “Binding-view” (BindingMultiPageEditorPart, for inspecting the high level “bindings”) to the editor. All of this is done through the extension point TinyOS.Editor.

The libraries JFlex [5] and CUP [6] were used to generate parser. JFlex is a lexer generator, CUP a parser generator. The two libraries work together excellently. Both libraries contain a tool that reads an input file in a grammar specified by the library and writes Java-code as output. These tools can easily be called by a build script. Both libraries are several years old and tests indicate that they contain a smaller number of bugs than younger libraries.

The problem of JFlex and CUP is, that they were not designed to handle languages with a big grammar like NesC. While JFlex just needs a lot of memory to generate the lexer, CUP actually produces illegal code, several methods and String-initializers exceed the code size limitation of 64KB. To overcome this limitation parser contains an additional tool which takes the output of CUP and splits these big methods into several classes, also String-initializers are written into external files.

That tool is represented by the class ParserCreation. This class is able to perform all the necessary steps to read the input files for JFlex and CUP, apply preprocessor to the input files, and generate two different versions of lexer and parser (one version is the “main parser”, the other the “collector”). The input file for JFlex is tokens.jflex, the input file for CUP is parser.cup.

Actually it is wrong to speak of just parser, because there are three different parsers in the project.

- The “initializer” is a very fast parser only capable of finding interfaces and components within a file. These elements can be seen in other files without the need to explicitly include the files in which they were declared.
- The collector is a medium parser. It skips many parts of an input file, but reports all declarations of elements that can be seen in another file.
if the input file is implicitly included (which happens if an interface or component is used in another file).

- The main parser is the heavyweight parser which understands everything. This parser is responsible for anything that the initializer and the collector can’t handle, e.g. creating the full abstract syntax tree (\textit{AST}) or error detection. When speaking of parser, most times this main parser is meant.

The initializer is implemented by the class NesC12Initializer. The collector by the class IncludingParser. Finally the main parser is represented by the class Parser.

Figure 6 shows the data-flow between the different modules of parser.

\begin{figure}[h]
\centering
\includegraphics[width=1.0\textwidth]{figure6}
\caption{Dataflow within parser. Blue boxes mark algorithms, yellow boxes mark data structures. The orange box represents anything that is known about the project, including data from other files.}
\end{figure}

4.2.1 AST

Like most parsers, the main parser generates an abstract syntax tree. There are over 150 different kinds of \textit{nodes} in the \textit{AST}, each node is represented by its own class, called an ASTNode. Each node has two tasks to perform:
- Put constraints on its children to maintain type safety. The **null-node** (the non-existing node) and “error nodes” (nodes which are explicitly marked as being wrong) can however be put anywhere.

- Create a more abstract view of the *AST* where elements like a type or a function are represented by only one object rather than a whole tree. The elements of this abstract view are called “binding” and can be used for tasks like error detection.

The method `resolve` of `ASTNode` is called after the creation of the *AST*. This method does several tasks in one sweep: create the “AST-Model”, store “declarations” (IDeclaration) and error detection. The method can use `AnalyzeStack` to get information of other files and to store its output.

### 4.2.2 Handling syntax errors

Whenever a parser like CUP reads a token, it can either perform a “shift” (push the token on a stack and continue without further evaluations) or a “reduce” (read symbols from the stack and combine them to a new symbol, and then re-read the token that was just found). The input-grammar defines under which condition a `shift` or a `reduce` happens. But not all possible inputs are part of the grammar, so there are cases when the parser can do neither a `shift` nor a `reduce`. In such a case a syntax error is discovered.

CUP has an internal mechanism which tries to recover from such an error. The parser starts to throw away tokens and searches a rule which offers a special “error-shift”. First tests have shown that this mechanism is too simple and much valuable information gets lost while the error-shift is searched. Also the number of cases where this mechanism succeeds is very limited. Putting more rules with error-shift into the grammar is not a solution since it would not only enlarge the grammar, but also introduce many shift-reduce conflicts during parser creation.

To solve this problem CUP’s error recovery had to be modified. The modified mechanism tries first to find an error-shift. But in each unsuccessful try it tests also for a reduce. While the old version would immediately start to delete tokens, the new version can perform the reduce and hence changes the situation for another round. In the end, the new version just checks more rules for an error-shift, and thus it is more likely to find one. Additionally every reduce produces information that can be used for the *AST*. With the new approach chances to get a complete, or at least a non-empty, *AST* are more likely than before. Chances that the parser just stops and has no output at all are decreased.

There are still situations where the parser can’t finish its work. A missing parenthesis (like ‘}’) is an excellent candidate to kill the parser. If that happens, the parser may already have executed some productions, but the elements that were created are not yet put together. They are still waiting on the stack for a `reduce` which will never happen. The modified parser takes these elements and tries to guess how they can be put together. Not every guess is correct, but the algorithms of the parser plugin are built in a way that they can deal with an incorrect *AST*. Surprisingly this solution produces good results. The reason seems to be that the number of correct reduces is in almost any case much higher.
than the number of guesses. So the part of the AST that is incorrect is small compared to the whole AST.

4.2.3 AST-Model

The “AST-Model” is a concept of core. It is an abstract view of the AST and contains only those elements which can be shown in the outline view. The architecture of the model will be discussed in chapter 4.4.2.

The main parser sees the AST-Model as a summary of the AST. In theory the model could be used to get easier access to information (e.g. what commands an interface contains). In reality the AST-Model is too generic to be directly used.

4.2.4 Bindings

“Bindings” are the most advanced abstractions of the AST. Each binding represents a high level element like a module or a function. Bindings are type safe and use their own cache to ensure that nothing is calculated twice.

Figure 7 gives a small insight into bindings.

There are about 30 different bindings. They all implement the interface Binding. Bindings can be obtained through the AST or the AST-Model.

Each binding can be associated with other bindings, forming a graph of a whole application.

4.2.5 Hyperlinks

When holding down ctrl and clicking with the mouse, one can navigate within a file. Source and target of each “hyperlink” are determined by parser.

Hyperlinks are created by a set of “rules”. Each rule finds a certain kind of hyperlink, e.g. all hyperlinks regarding local fields. Most of these rules first analyze a part of the AST to find the name of the element they should lead to. Then they either use the “global index” of the project to find a target, or bindings associated with the AST.

Each rule implements the interface IHyperlinkRule. Parser offers the extension point nesc12.parseryperlinks to add additional rules.

4.2.6 Quickfixes

Each error message can be associated with some meta information. This meta information can later be read to create “quickfixes”. Parser offers a limited set of quickfixes. Either they just add some missing source code, or they can create new files when necessary. Quickfixes are created by rules just like hyperlinks.
Figure 7: The binding view, only visible when in “debug mode”. Currently the binding view shows how the module BlinkC sees the application. BlinkC uses an interface Timer but renames the interface to Timer0. Timer defines some events and commands, one of them is startPeriodic. The argument of startPeriodic is of type uint32_t which is a typedef and in reality an unsigned long int. The arguments name is dt.
When finding an error, a `String`-message, location and an optional `Insight` can be reported in `parser` (using the `AnalyzeStack`). These objects are wrapped into an `IMessage` and then forwarded to `core`. `Core` will create an `IMarker` which contains some of the information of the `IMessage`. Later `core` will ask for `quickfixes` for the `IMarker`. `Parser` will re-create the original `Insight` of the message, and call all the `quickfix-rules` in order to collect new `quickfixes`. `Rules` can use the `Insight` to create the fixes.

`Parser` offers the extension point `nesc12.parser.quickfixes` to add new rules. `Rules` can either be “single” (capable of handling one message only) or “multi” (capable of handling several messages at once).

`Core` offers the extension point `TinyOS.Quickfixer` to add new rules which are independent of the `parser`. `Rules` added here will not have access to vital information like the `AST`, and thus the use for this extension point is limited for anyone but parser plugins.

### 4.2.7 Code Completion

Code completion uses, like `hyperlinks` and `quickfixes`, `rules` to generate “proposals”. Other than `hyperlinks` and `quickfixes`, code completion cannot rely on a complete, consistent or even up to date `AST`. While the latest problem can be solved by just updating the list of proposals once a new `AST` becomes available, the others are more resilient. Some `rules` solve these problems by not looking at the location where the code will be inserted, but a few characters before. There the chances that the `AST` is not yet corrupted are much higher. Other rules just don’t use the `AST` at all. Most `rules` will use the information of the `AST` only as a hint, but actually analyze the source code. They read a few words or characters and try to guess what the `AST` would be if it were correct. `Rules` will use the `global index, bindings` or “ranged collections” to come up with proposals (`Ranged collections` are maps with visibility ranges as keys and names of fields, functions, etc. as values).

### 4.2.8 Error detection

Error detection is implemented as a recursive algorithm. Every node of the `AST` becomes the opportunity to check its content. Some problems can only be detected if many nodes work together, e.g. using the same name for two different functions can only be seen if the two function-nodes compare the names. Communication between nodes is handled either through a stack or by direct access. There is often a ”super“-node which sets up a testing environment on the stack and ”sub“-nodes then use this environment.
For example the ASTNode Module pushes a factory for ModuleFieldPusher onto AnalyzeStack. Every node which represents a field accesses or creates a FieldPusher associated with the name of the field. The node adds a binding Field to this FieldPusher. When all the children are finished, Module and the set of ModuleFieldPushers can check if the fields are valid in the context of a module.

4.3 Environments

The purpose of environments can be described by two words: "search files". An environment is responsible for everything that might result in searching files. That ranges from finding the example applications, finding the platforms and the associated directories, to finding a header file that is needed for parsing. They are also responsible for invoking external tools like ncc.

The environment plugins are small and do not contain sophisticated algorithms. Their only source of complexity are the sharp distinctions between different versions of TinyOS and operating systems.

The linux and windows environments are built upon an abstract environment, that allows maximal reuse of code. The linux environment is called TinyOsUnixEnvironmentWrapper2, the name of the windows environment is TinyOsWinXPEnvironmentWrapper and the abstract environment is called TinyOsAbstractEnvironmentWrapper.

4.4 Core

Core combines all the other plugins. While parser and environments were rewritten from scratch, core was taken from YETI and upgraded. Upgrading often included two steps. First a layer was introduced separating reusable from outdated code, then the outdated code got replaced. At some places the outdated code remains in core as backup, e.g. the old parser can be used if parser is not installed.

4.4.1 Project Organization

Each TinyOS-project is internally represented through a “project model”. The project model is a central hub for everything that has to do with parsing code. It can create new parsers, it can start the “build chain”, it manages all caches of the project. It also contains the global index of the project.

The project model is represented by the class ProjectModel.

4.4.2 AST-Model

Core needs access to the abstract syntax tree of a file in order to display views like the outline view. However it is impossible to specify how an AST has to look
like, when future language modifications or new parsers-plugins are possible. The AST model is a layer above the AST, creating a more general interface.

Each model contains “nodes” and “connections”. Nodes represent the same things as nodes in an AST, for example a function. Connections represent the relations between nodes. A connection can either be a child-parent relation, or a reference (e.g. when a component provides an interface, then there might be a reference from the component to that interface).

The AST model is represented by the interface IASTModel, nodes by IASTModelNode, connections by IASTModelNodeConnection.

Nodes never know each other directly. They only know connections to other nodes. The connection then only knows the identifier of the node it points to. That allows to replace parts of the model, or load parts of the model lazily, without having to worry about dangling references.

Each node and connection can be marked with “tags”. Tags are a simple way to describe how a node or connection should be treated, what it is good for or what icon to use when shown in a view. While tags seem to be only a nice detail, they are used massively by core and parser. The separation of core and parser would not be possible without them.

Tags can roughly be put together in these groups:

**Type** What kind of element a node represents. A component, a typedef, the specification block of a module, ...

**Modifiers** A more precise specification of the type. A command function instead of just a function, a generic module instead of just a module.

**Usage** For reference-connections only, how the reference is used. For example whether a referenced interface is “used” or “provided” by a configuration.

**View** How connections and nodes should be treated in views. For example should the icon depend on the “tag-set” of the connection or the node? Should a node initially be expanded or collapsed in the outline view?

Tags are represented by the class Tag, sets of tags by TagSet. Parsers are free to specify new tags.

4.4.3 Build Chain

In order to have accurate error messages, a file needs to be “built”. Editing the file will “unbuild” it. Also editing an (explicitly or implicit) included file can unbuild. The discovery of a new, missing resource will unbuild a file as well. The build chain ensures that all unbuilt files will be built again.
TinyOSProjectBuilder2 is called when resources changed. It forwards the changes to TinyOSBuilder. Each project has one such builder. Afterwards TinyOSBuilder calls the buildInit and buildUpdate methods of ProjectModel to "rebuild" the project.

Building a set of files always starts with indexing (called the "initialize phase"). All unbuilt files are given to the initializer, which will extract interfaces and components. These elements are visible through the whole project and without them many false error messages "missing xyz" would appear.

The initializer is represented through an INesCInitializer.

In a second sweep the main parser is used to analyze single files. If a file has implicit includes then the "recursive collector" can be used to find them. It may happen that a file cannot be analyzed without another file built first. If such a case is found, and the other file is not yet built, then a "simplified build" is made without active error detection. This simplified build will not trigger further builds, so there is no danger of infinite build-loops.

The recursive collector is stored in ProjectModel. It is represented by the interface IProjectDefinitionCollector. Currently there is only one implementation: LocalProjectDefinitionCollector. It is the only component which uses the "wire cache", it accesses the cache through WireCache. It further uses an INesCDefinitionCollector to collect the declarations of a single file.

As a side effect the "dependencies" of the built file become known. The dependencies tell for each file, from which other files it depends. This information is later used to unbuild a file if one of its dependencies changes.

4.4.4 Caches

In order to speed up the build process, several caches are used. These caches store all their entries on hard drive and only maintain a copy of a subset of them in memory. Thus a restart of Eclipse will not delete any information that is created by the build chain. The list of caches has 6 entries:

**Init Cache** Stores the global declarations found by the initializer.

**Inclusion Cache** Stores the declarations visible only when a file gets included, which is also the output of the collector (the recursive collector just combines the output of several runs of the collector, its output has not to be stored).

**Dependency Cache** Stores for each project file on which other project file it depends. This includes any dependencies, even if they are indirect. This
information is needed to unbuild a file when one of its dependencies has changed.

**Wire Cache** Stores for each file on which other files it depends. This only includes the top level dependencies, the files which are included directly. This information can be used by the collector to prevent parsing a file when only the declarations of its included files are searched.

**Missing Cache** Stores for each project file which resources were not found while building the file. If such a resource is found later, then the file gets unbuilt.

**AST Model Cache** Stores the *AST model* of files.

All caches can be accessed through IFFileModel. The ProjectModel offers access to one of these “file models”.

### 4.4.5 Editor

The editor is just a standard Eclipse-text-editor, configured to call the methods of core when a task, like syntax highlighting, is at hand.

The editor is implemented by the class NesCEditor. The nesc specific configurations are set up by a NesCSourceViewerConfiguration.

### 4.4.6 Outline View

The outline view relies on the *AST model*. It connects itself with the editor, instructs parser to create an *AST model* when reconciling the editor, and then shows this model. It searches for any node which has a tag outline but not a tag included, and uses these nodes as root of the tree. As long as possible the outline view takes nodes from the model provided by the editor. If some node cannot be found in that model, then the outline view uses the global *AST model* provided by the project.

The NesCOutlinePage implements an INesCEditorParserClient and registers the client at the NesCEditor in order to be informed when parsing starts or stops. It further uses a NodeContentProvider to show the tree.

### 4.4.7 Graph View

Similar to the outline view, the graph view uses the *AST Model* to build up its content. Any node of the *AST model* can have a factory for “figures”, a figure can be shown in the graph view. While nodes are absolutely free in deciding how their representation might look, a small set of default figures is provided.
The factory for figures is called IASTFigureContent. This interface has one method which will create an IASTFigure. This one method receives an IASTFigureFactory which can help to transform AST model-nodes or connections into figures.

Figures work like a tree: each figure is a node which can either be "expanded" or "collapsed". If collapsed only an icon and a title are visible, when expanded child-figures are shown. Some figures load their children lazily, the graph view supports this by expanding each figure in its own job. Thus the view remains responsive even while a complex figure is creating its children.

The method expandAST of IASTFigure receives an IExpandCallback. If it has to do a lot of work, it can start a new thread and later inform the caller whether the method succeeded or was canceled.

The graph view is connected to the outline view, whenever a node is selected in the outline view the corresponding figure is marked as well.

Only figures which implement IRepresentation can be marked through the outline. An IRepresentation can have some IASTModelPaths telling the view which nodes are represented by the figure. A figure can only be marked when its paths and the paths of its parent build the same chain of nodes as the selection-path in the outline view has. If a node has more than one figure in the graph, then the figures which are not selected are highlighted in another color.

5 Future Work

There are many ideas of what could and what needs to be done in the future.

• Instead of having one parser for the whole plugin, each project could have its own. That would allow to support many different versions of nesC at the same time.

• Environments could be written in a way that more than just one TinyOS installation is supported.

• More platforms and sensorboards: currently only the platforms and sensorboards within the tos-tree are recognized. It should also be possible to recognize platforms and sensors which are included though an -I directive.

• Support TinyOS 1.x. It is still in use and not everyone has the possibility to upgrade.

• Rewrite option-system: The make-option management is less than optimal. It is inflexible for changes, and hard to access (for users and developers). At this point a partial or complete rewrite seems like a good
solution. The rewrite might include several new ideas. Instead of working with String-keys real references could be used. Or at least clients should not be bothered with the String keys. The option view could be removed, and instead an xml file for each make option could be used. This would work the same way as ANT's build.xml. The make option dialog could be replaced by a form, like it is done for plugin.xml of an Eclipse plugin.

- Writing a new parser which also works correctly when a typedef is missing.

- There are a number of header files which are just included everywhere. The plugin does not find all of them, especially those belonging to the hardware are missing.

- Refactoring (e.g. renaming of variables) could be implemented as well.

6 Conclusion

To our knowledge YETI 2 is currently the most advanced Eclipse plugin for TinyOS 2.x. The standard approach to writing TinyOS applications without an IDE is to write some code, call the compiler and be happy if the compiler does not report errors. The compiler ncc just starts at one main-component and checks files only if they are included, applying macros and typedefs already found in other files. Yeti 2 on the other hand analyzes each file as if it were the main-component. This is much more restricting since every file has now to behave as if it were a correct application. We think however this is a better way to look at a project. It encourages developers to write correct code right from the beginning, making reuse and reorganization of code in later project phases easier.

There are still flaws in YETI 2. Not all errors are found, speed is an issue, code completion could do much more, refactoring is not supported at all. It is our hope that more developers will enhance YETI 2 in the future.

References


