Títol: Static Analysis on Imperative Languages

Volum: 1/1

Alumne: Aleix Pol i Gonzàlez

Director/Ponent: José Miguel Rivero Almeida

Departament: Llenguatges i Sistemes Informàtics

Data: November 7, 2011

DADES DEL PROJECTE

Títol del projecte:	Static Analysis on Imperative Languages
Nom de l'estudiant:	Aleix Pol i Gonzàlez
Titulació:	Enginyeria Informàtica
Crèdits:	37,5
Director/Ponente:	José Miguel Rivero Almeida
Departament:	Llenguatges i Sistemes Informàtics

MEMBRES DEL TRIBUNAL *(nombre y firma)*

President:	Juan Luis Esteban Ángele
Vocal:	Natalia Sadovskaia Nurimanova
Secretari:	José Miguel Rivero Almeida

QUALIFICACIÓ

Qualificació numèrica:

Qualificació descriptiva:

Data:

Contents

Chapter 1

Introduction

This project's central idea is to create a framework that will make it possible to analyze source code and report relevant information the best possible way. To do so, we split the project into 3 main parts: the data recollection, the check development, and a number of ways to present the conclusions to the user. Likewise this document has been prepared in the same way.

This project, even if it could be adapted to many languages, is focuses C++ analyzing because: On the one hand, it's an important and widespread language that could make good use of the project right away. On the other hand, we have a working implementation of it in KDevelop[2], which narrows down our work to extending the analysis process to actually do what we're looking for, saving us the non-trivial task of designing and developing a C++ parser.

1.1 Why Static Analysis?

When I was thinking about choosing a subject for this project I knew I wanted to work on programming languages. Moreover I wanted to create something that would help developers create better software. My KDevelop background was helpful when I decided to start creating tools that would end up in this framework creation.

First of all I wanted to start a project that would make good use of the amazing KDev-Platform infrastructure to create checks. There are (only) few static analysis tools in the Free Software world and my idea was that we could make a difference there.

This was why I decided to create a tool that will find certain problems in random software projects. To do so we started to create the checking infrastructure and I began to work on creating checks that point out the most simplistic problems that a compiler generally won't.

Using this project we want to achieve simplicity with regard to the check development. It should be relatively easy to create new checks which make sure that trivial problems don't arise and support some of the not-so-trivial ones.

1.2 KDevelop as a Starting Point

As we have mentioned before, this project is largely built on top of the KDevelop project, which was already working when the current project started. In KDevelop we could find some abstractions already created, like the plugin infrastructure or facilities to ask the platform to parse a project and then tell the language support to parse all its files.

This project changes the KDevelop project the following ways:

- We modified the KDevPlatform to understand Check plugins by specifying the interfaces they need and the introduced data types.
- We modified the KDevelop's C++ parser to extract the data that we need and fill the data structures to be fed later to the plugin.

Then the rest of the project was to create satellite projects that can be integrated in the KDevelop project (and will be soon).

- We created infrastructure to run checks on what the user wants and needs. That is, select what plugins to run, run them and provide the results to the user in different ways.
- We created infrastructure to easily develop checks: by providing a checks pack that can be distributed easily and adapting an external programming language that can be used to distribute the checks easily.

All in all this project is thought to be the creation of a framework for creating checks. Here the KDevelop project is both the support we need and a very good candidate to use the project results to get it in the users hands in the end.

1.3 Why Free Software?

Given the nature of this project, we believe that having the most freedom on the code base is the best approach.

We have created a project that should keep alive by two means: all projects interested should be able to benefit from it and the developers interested in contributing to the project, should be able to do so, either by creating new checks or by extending the current infrastructure.

The software produced by this project is going to be released under the LGPL license, together with the rest of KDevPlatform and KDevelop or in a separate package.

In the end of this document, you'll find an appendix explaining some of the benefits for using Free Software

1.4 Goals

Here the project's main objectives are summarized:

- To abstract the language support from the check creation. The check developer doesn't have to be responsible for the modification of AST[1] visitors to check the information, check developers will get the semantics they need and will be able to compute it to achieve their goals.
- To provide a set of checks as minimal check set as a proof of concept. In order to make sure that the gathered data is useful, some checks are being developed.
- To provide a set of tools to display the checks findings in a flexible and user friendly way. Being able to check source code is not enough, we need ways to present it to the user. We tried to identify the most useful use cases for our static analysis framework and I implemented them on top of architecture.
- To provide a set of tools to create checks. Being able to run checks is not enough, sometimes we are developing checks and we need to know what's going on. In that regard, we adapted the tools already discussed to make it easier to the developer to find where are the problems, if any.

1.5 State of the Art

Code analysis tools have been around since long time, I won't say since the first day but as soon as developers realized that the input might not be correct we started to produce errors.

[1]Stands for Abstract Syntax Tree, it's the code representation provided by the parser.

At the beginning it was just syntactic errors in an assembler, but analysis evolved together with language complexity and computing power. When it comes to analyzing high level languages, the analysis weight has increased a lot in time, also. Modern compilers have far more error messages and warnings that they had some time ago, also some language specifications have driven the languages to places where it's easier to know when code can lead to uncertain states by providing solid type.

In this project we will be centering ourselves in improving error checking in imperative languages by using C++ as a departure point. C++ is specially interesting when it comes to static analysis because together with C it's one of the most wide-spread languages but also it's one of the hardest to parse and analyze. The main reason for that is that, first of all, we have the pre-processor that adds some complexity when it comes to telling what comes from where. The C/C++ pre-processor[2] is a language that is run before even parsing our code that puts together all the information we need to compile a program unit in a consecutive byte chunk and it's not after doing this step that we can start figuring out what does the code is telling. Also it can depend on the pre-processor arguments that we pass, poorly coded C/C++ can be very hard to analyze successfully.

When taking a look at what people is doing when it comes to C++ static analysis there's a rather misleading panorama. There are some tools that claim they will improve your code but most of those they just mean adapting the code to some specific code styles (such as indentation and spacing) but it doesn't have a lot to do with the logic which is what we're interested the most. Also there are quite some tools that just support C or that just support a C++ subset. We are interested in C++ so we won't consider those. Also there are a lot of closed-source programs that claim to do C++ static analysis but that are never very specific on what do they offer further than saying they are the best. In the next paragraphs I'll summarize what I found the most interesting to display what's the current status.

My first experience with static analysis was with the **krazy**[3] project. This one is a set of Perl scripts that we have in the KDE[1] project to make sure API's are being used correctly. It's not very interesting from this project point of view, but I think it's important to note that it's useful to tell the user when she's doing things wrong. It's not always that the developer doesn't want to, maybe she doesn't know or she knows in the future. In this case it had a very good impact in my getting on track with proficient Qt/KDE development.

One of the recent developments I'm most interested in in the field of compilers, is the

[2]http://en.wikipedia.org/wiki/C_preprocessor

LLVM[4] project and in this specific case, the clang[5] project. Not only they are aiming at having a fully-featured C-like-languages compiler but they are providing the full API for the parser and some code models so that we can build tools on top. One of the tools they are providing is the **clang analyzer**[6] that is doing more or less what I'm intending to do in this project. I didn't use the clang project as a base for my project because the parser wasn't ready when I started working in this project but it probably would have been also a good base for my work. They claim to check dead writes (writes that are never read), de-referenced pointer returning, dividing by zero, etc. Certainly a technology to keep an eye on.

Cppcheck[3] was a fortunate surprise I had when I started to work on this project, while investigating similar projects. It has different kind of checks: pointer dereferencing, makes sure all the class attributes are initialized in the class constructor, etc. They have a very strong focus on having a small amount of false positives.

Inside the Mozilla project there have been a couple of projects that try to provide some features regarding C++ and static code checking. Those projects are **Pork** and **Dehydra** to create some static analysis and code refactoring tools. The tools are targeting mostly internal code constructions in the Mozilla project, but it's specially interesting how they managed to work it out, also considering how big of a code base it is. Also it's interesting to see that they also decided to create the checks in a scripted language, JavaScript in this case, probably because of its flexibility and because they know a little about JavaScript in Mozilla. Apparently they have given up with the project, though. Most of the Pork work is being deprecated in favor of clang, and Dehydra is in maintenance mode where new checks (hydras) can be contributed but no new architecture development is going on.

There are other tools, most of them are C-only, so I don't think they can be compared properly. There are also some closed source solutions that technically are comparable but the fact that they are closed makes it difficult to adapt them to your project (like Mozilla did, for instance), so they usually target more abstract aspects of the code. Probably one of the most important players in this market is the Coverity[8] company, although it's quite hard to figure out what are the actual features from the marketing speech.

1.6 Summary

This document is distributed in the following chunks:

[3]http://cppcheck.sourceforge.net

1. The changes that we have had to do to the KDevPlatform and KDevelop to make sure everything can be done.

   Chapter 2. Architecture and Data Collection Researches into the essence of what we're aiming for to provide the common interfaces needed to make this project work. Describes how this was implemented.

   Chapter 3. Checking infrastructure Describes the abstractions created to implement checks that can be easily integrated to the platform

2. The additional plugins and tools that we created to make it possible to develop checks and to run them.

   Chapter 4. Check Execution Describes the tools that have been created to make it possible to run checks and easily review the results.

3. An explanation regarding how are checks developed, how do they work and a summary of some checks that have been created already.

   Chapter 5. Creating Checks Describes the process of developing checks and provides ideas on how this could be improved. Suggests alternatives to C++ checks.

   Chapter 6. Functional Checks Describes the chosen development platform for check development and describes what was needed to make it happen.

   Chapter 7. Checks Lists the checks that are provided by this project and describes them thoroughly.

4. There are some chapters oriented to describe and summarize this project in order to ease its comprehension and to show the outcome of it.

   Chapter 1. Introduction This chapter. Explains what have I done in this project, why did I do it and I introduce the related technologies.

   Chapter 8. Planning Describes how did the project evolve over the time and compares the first project idea we had with the final result.

   Chapter 9. Conclusions Wraps up the project by providing thoughts on the overall experience and some ideas for the future of this project.

5. The document also provides two appendices that tell us how to try the results of this project on your system and another one explaining what tools have been used to create all this and why.

Appendix A. Code Repositories Lists where was the code developed and stored, in case it has to be reviewed or used.

Appendix B. Free Software Relations Lists the other free software projects that took part in this project and explains where did they influence.

Chapter 2

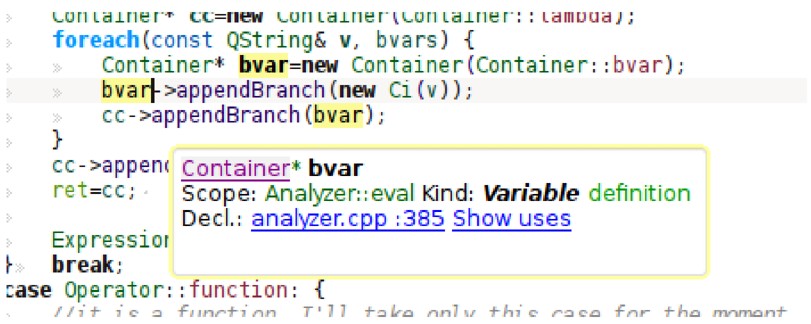
Architecture and Data Collection

First of all, to be able to come to conclusions regarding some code, we will extract the data we need and we will put it in abstract interfaces. An alternative method would be to provide the raw AST, but I believe that this way we will be able to get more powerful checks with less effort. For the moment, we will be providing 3 main data sources: the DUChain, a Control Flow Graph, and information about the data access for every variable in the code. I believe that, based on this information, a wide range of problems can already be addressed.

Regarding the DUChain, even though it's not provided by this project (it already was in KDevPlatform) I will try to explain it briefly so that the reader will understand why is it useful. DUChain is short for Declaration-Use Chain and its main purpose is to create a structured database with all the code symbols and their relevant information. Originally the idea behind it was to provide a solid base to create code completion and code navigation on top of the raw parser information. However, it has proved to be a very useful information repository that can be useful in different cases. In the DUChain we will store any variable or type declaration together with their type and uses. This makes it possible to analyze it and extract very useful information.

In this picture we can see some features from the DUChain that are being used in the KDevelop editor. When hovering a variable use, we're highlighting the declaration and all the uses for the symbol and also we're showing a tool tip window with information relevant to the variable declaration. As we said, we have everything indexed and we can navigate it to find whatever we need.

More information about DUChain can be found in KDevPlatform's DUChain Documentation [7].



```
    Container* cc=new Container(Container::lambda);
    foreach(const QString& v, bvars) {
        Container* bvar=new Container(Container::bvar);
        bvar->appendBranch(new Ci(v));
        cc->appendBranch(bvar);
    }
    cc->appen
    ret=cc;

    Expressior
}   break;
case Operator::function: {
    //it is a function. I'll take only this case for the moment
```

```
Container* bvar
Scope: Analyzer::eval Kind: Variable definition
Decl.: analyzer.cpp :385 Show uses
```

Figure 2.1: KDevelop's graphical representation of the DUChain.

Before diving into the interfaces, I'd like to put in words why we are making such a separation between the code back-end and the checks. It's arguable that we need to provide such abstractions and it wouldn't be fair to you, the reader, that I didn't comment on it. Code analysis is not a simple process, there are different steps involved and we will be most likely require all of them in some of the checks we are using. In this project we are going for creating an infrastructure that eases the use of a lot of small checks. For the moment, I'm not interested in being able to build a big piece of code that does everything, we want to provide tools that fit any check developers' needs. If we are missing something, then the missing part should contributed again to a common place, where all checks can access. We want to unleash the possible check contributors' creativity. I believe this is reachable by providing diverse, useful and well-defined resources, more than leaving space for implementation freedom, such as the AST exposition, because it could lead to similar implementations on every check. We want to reuse code, we want every tool we make to add some value to the next.

All check interfaces are language independent theoretically, but in practice checks might not be cross language. This is fine, because the important thing is that just by knowing how the language works the knowledge transfer, so does the documentation, so when implementing checks for a new language we will just need to provide the abstractions and make sure what translates and whatnot.

2.1 Interfaces

In the following section I will explain the interfaces that have been created for both, accessing and constructing the information.

During this project an important amount of time was dedicated to the interface designing. The power of our future checks depend on how flexible those can be so, first of all, let's define them carefully.

2.2 Data Access

This interface will tell us for every variable if it's being read, modified, or both. By variable here, I mean an identifier representing some data. Therefore, we will be able to extract the information where the data comes from by accessing the position's Declaration or Use.

2.2.1 Implementation

There are two main objects that we are going to use to access and store the Data Access information: the DataAccess and the DataAccessRepository.

```
class DataAccess
{
    public:
        enum DataAccessFlag { None=0, Read=1, Write=2, Call=4 };

        bool isRead()  const;
        bool isWrite() const;

        RangeInRevision valueRange() const;
        CursorInRevision pos() const;
        DataAccessFlags flags() const;
};
```

The class DataAccess is used to identify every one access to any identifier provided in the code. All of those contain a *CursorInRevision* instance that can be used to relate it to its position in the code (and also to its corresponding Declaration instance), also a valueRange that, in case it's a Write and it's possible, will tell us which expression is responsible for the variable's write, and finally also the flags that will tell us what type of data access it is, either a Read, Write or Call.

It should be noted that when we are talking about cursor and ranges, we are talking about code positions. The cursor and range abstractions are provided by the KDevPlatform through RangeInRevision and CursorInRevision. A cursor will be, mainly, a line in a column that, together with the URL, will specify a position in some code. A range is like a constrained pair of two cursors: one pointing to the start and the other pointing to the end of the range.

```
class DataAccessRepository
{
public:
  void addModification(CursorInRevision cursor,
                       DataAccess::DataAccessFlags flags);

  void clear();
  QList<DataAccess*> modifications() const;
  DataAccess* accessAt(CursorInRevision cursor) const;
  QList<DataAccess*> accessesInRange(RangeInRevision range) const;
};
```

The DataAccessRepository is the code structure we will use to retrieve data access information in a source file. We are providing 3 methods to retrieve this data:

modifications This method will return a list of all data accesses found in the repository's file in the order that the language implementation provided it.

accessAt It will return the data access that we have at the *cursor* position. If there's none, a pointer with 0 value will be returned.

accessesInRange It will return all the accesses in the repository that are contained by the *range* argument, just as if filtered the modifications method above with cursor's method *contains*.

To see a little better how it works, I'll provide an example with its expected results:

```
int a, b;
int g(int&, int);

int f()
{
  return g(a, b);
}
```

In this example there are 3 declarations: 2 variables and a g function that has two arguments, a reference integer and an integer. Finally there's the f function that just calls g and uses the global variables to call the arguments.

In this code we will find 3 data accesses:

DataAccess::Write—DataAccess::Read A read/write access for the first function argument, because the function declaration is saying that it's a reference, so this function could be modifying the argument (otherwise it would be `const`).

DataAccess::Read A read for the second function argument.

DataAccess::Read—DataAccess::Call A read/call for the function argument name.

With this we should be able to know what kind of data is being accessed and why, more or less. This data will be specially useful together with the Control Flow Graph and the DUChain.

2.2.2 Use cases

The use case for the data access data structure is rather simple. Especially in the case of imperative languages, the ones treated in this project, it's very useful to understand where we are accessing data and where we are modifying it. It's a very common problem to add code that changes the value semantics in unnecessary ways. We need to be able to know where values are coming from and what is being changed. Together with the Control Flow Graph we can predict how the program will evolve in the future.

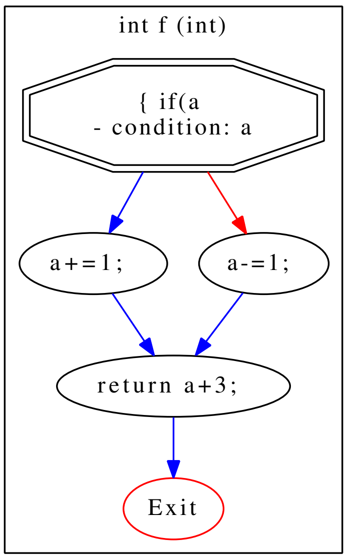
Another interesting use case for the Data Access structure is the constant conditional case where we have to be able to track where the data is coming from if we want to be able to tell if some data is constant or not along the program.

2.3 Control Flow Graph

The other data interface that we have introduced is the Control Flow Graph. It is used to create a representation of how the code is going to behave considering a current state. This is achieved by specifying what the different code paths are and which parts tell who decide what's the alternative path to take.

This data structure can be abstracted like a graph where every node is a sequential chunk of code that will be run sequentially and every edge is the places where there's some run-time decision to be taken that will decide where does the execution follow.

```
int f(int a)
{
  if(a)
    a+=1;
  else
    a-=1;
  return a+3;
}
```



Here we can see the output[1] of the code placed on the left hand side in form of a control flow graph in the figure on the right hand side. We can see that instead of having the code in a form of text array the relationships regarding the code flow are visualized.

2.3.1 Implementation

We have 2 important classes to know about regarding the Control Flow infrastructure:

```
class ControlFlowNode
{
  public:
    enum Type { Conditional, Sequential, Exit };

    Type type() const;

    void setStartCursor(CursorInRevision cursor);
    void setEndCursor(CursorInRevision cursor);

    RangeInRevision nodeRange() const;
    RangeInRevision conditionRange() const;
```

[1]As will be discussed later, this graph is automatically generated by the unit tests

```
    ControlFlowNode* next() const;
    ControlFlowNode* alternative() const;
};
```

The ControlFlowNode class will be used to specify one code chunk that will always be executed sequentially as a block. Here we're providing different information:

type tells if the node has 1, 2 or no following nodes.

- **Sequential** If it is sequential node, it means that the node just points to 1 node; it will inevitably move on to the next one.
- **Conditional** If it is a conditional node, it means it has two following nodes. In this case the `conditionRange` will be specified which will be responsible for where to go next.
- **Exit** If it's an Exit it has no following nodes, which means that it's an end of the current code path.

nodeRange tells the range that defines the node, from its start to its end.

conditionRange returns the range that will specify which range will directly dictate which of the 2 paths to take, in case it's a conditional node, otherwise it will return an invalid node.

next and alternative properties will provide us with the different nodes that will follow the current node, depending on the current state.

```
class ControlFlowGraph
{
public:
  void addEntry(ControlFlowNode* n);
  void addEntry(Declaration* d,
                ControlFlowNode* n);
  void addDeadNode(ControlFlowNode* n);
  void clear();

  QList<Declaration*> declarations() const;
  ControlFlowNode* nodePerDeclaration(Declaration* d);

  QList<ControlFlowNode*> graphNodes() const;
  QVector<ControlFlowNode*> deadNodes() const;
};
```

The ControlFlowGraph is, like the DataAccessRepository, the class that will store every file's gathered information and will provide easy ways to access it from the checks or from wherever it will be needed.

In this class we will be providing 3 methods to feed nodes into it: addEntry will let us add a random node found in the code, addEntry with a symbol Declaration (we mostly expect functions and variables definitions here), that will add it and additionally will link the Declaration to the node, so that we can know what it is doing and when it is going to be called. Finally we also have the addDeadNode method that will add the node just like the plain addEntry but will save it as a dead node. These dead nodes are the ones that will never be run.

Before going on with the use cases, let me define a little what is a dead node. Dead nodes are those that will never be called given the language syntax. It doesn't mean that there's no logic that can lead there, but that there's no path that leads there, but still there's the code. Let's see some C++ example:

```
int f()
{
  int x=2+sin(pi)/4;
  return x;
  awesome_function();
}
```

In this example, there's not even the possibility that `awesome_function` is ever called, then we will generate a dead node from right after the return until the end of the function.

```
int f()
{
  int x=2+sin(pi)/4;
  for(;;) {}
  awesome_function();
}
```

In this example there won't be a dead node because it's not a syntactical reason that tells us that `awesome_function` won't be called, therefore we will have to figure it out in some other way.

2.3.2 Use cases

If we want to be able to analyze some code base we need to know what the code is going to react to and where it will lead to. The Control Flow Graph tackles that by providing us with the network of states that could happen in it.

A typical use case example would be the constant condition problem, where we get to statically tell that a condition is always going to jump to the same place, which would mean that it's an unneeded condition that can be removed.

2.4 Data Structures' Implementation for C++

To implement both of these interfaces in the C++ support we added two new AST walkers that understand the code structure and provide the data we are looking for. This way, as we exposed before, we will not need to link (or even understand) the language implementation.

First of all the Data Access interface was implemented. At first glance it appeared to be easier to implement than the Control Flow Graph. Eventually this was not so, although the difficulties encountered arose due to some initial design errors. The implemented algorithm is quite simple. As we said, it's a typical AST visitor. Here we want to track to what type the current variable is being fed. In case it's a reference the variable will be read and written. Otherwise the reference will be read only. With this idea in mind we keep tracking what argument we're in and in the cases the type is built-in by the language we also generate our own parameter types. For the write-only types like definition and assign this will be done right away so that we can properly provide a valueRange.

The other implemented interface is the Control Flow Graph. The resulting implementation was quite forward – like the previous one this is an AST visitor. However in this case we:

- Wait to find a code context (either a declaration or a function/method definition)

- There we create the first node in the graph. We walk into the code node recursively and finally we feed it back to the graph together with its declaration if possible.
- While walking the expression node we will find different conditional structures such as: `if`, `switch`, `while` or `for`. When we want to create a new node we will proceed as follows:
  - we set the ending cursor of the node.
  - we create a new node with the appropriate start cursor.
  - we link it to the previous node and proceed the visiting.

All this has to be done in a sufficiently reliable way. That's why we created some unit tests for both interfaces. This way we can ensure that everything works as expected.

The data access interface has a test that provides some code and a list of DataAccess::Flags values. The unit test parses and analyzes the code, then compares the results to what was expected, like in the Control Flow Graph's implementation, but this time we just pass the tested code and the number of nodes. This is already quite a significant check if there have been any regressions. Given that by using this approach we're not taking into account the ranges, when we run the code a graphviz file [10] is generated for every test case so that we can review its consistency ourselves.

Chapter 3

Checking Infrastructure

Once we have all the data we have to start thinking about what a check is and about how far we want to take them. The KDevPlatform semantics are rich enough regarding problem generation but we want to define what we're aiming for, so that we can act accordingly in the future. Furthermore, we don't want to exclude any possibilities. We tried to make a flexible interface that fits our needs but we don't make sure it's used entirely the way we expect it to.

Given the scope of this project, we can consider a check as a program that given some data: adds problem items to the project, tying them to its file. In this chapter we will explore what limitations and possibilities this approach offers.

3.1 KDevPlatform Abstractions

As discussed before in this document, one of the most important reasons to base our project on top of KDevelop was the abstractions that the KDevPlatform provides. That's important because it helped us, to some extent, to keep our project modular and with its very specific abstractions defined.

Before rushing into explaining the interfaces that we created for the project, I'll explain a little why are they needed and how do we take advantage of those.

Our plugin infrastructure is based both in the KDE's services look up infrastructure and C++/Qt's introspection capabilities for the run-time identification.

Let's see how it works: KDE has a database with all its plugins listed, called sycoca (SYstem COnfiguration CAche). This database will read all the desktop files in the system and cache them. When we request a plugin, this cache will be queried about plugins with the service type KDevelop/IPlugin that provide the interfaces we're looking for. The information about these will be returned in a *KPluginInfo* instance that will be providing us information like a human-friendly name, the actual binary we have to open or the supported interfaces. The KDevPlatform abstracts all this by providing a PluginController that can receive any kind of query. For example, at start-up, we can ask for the plugins marked to be always opened or when we open a project we ask for project plugins that can import a project like the one we are opening.

The PluginController returns IPlugin instances, this class is derived from QObject and we can use it to extract the actual interface by casting to the Plugin interface we need by using `IPlugin::extension <Interface*>()` or `qobject_cast <Interface*>()`.

In our case, we'll define interfaces for checks and whenever we need to run checks, someone will have to ask the PluginController for ILanguageCheck, it will load and return all the available instances, and then those will implement whatever we ask them to by specifying the methods as abstract (C++ reminder, it's `virtual type function()=0`), from here it's regular C++ development.

3.2 Check Interfaces

When we start thinking about implementing a check, the first thing we will do is to consider how we achieve that our code is triggered at the correct time, so that we can start thinking about what approach to take. We don't want check developers to wonder about all these problems so we will try to solve it once in the platform and let check developers work on their own issues.

To do so, firstly, we will define the interface which the check developer will implement. It's called ILanguageCheck interface and looks like the following:

```
class ILanguageCheck
{
public:
  virtual QString name() const = 0;
  virtual void runCheck(CheckData data) = 0;
};
```

Here we are requiring to implement two things: to provide a name, in order to have some identification for the check for UI usability and debugging purposes, and also the important method, the runCheck. This is the one that will be called whenever we need the check to start analyzing the data and, in case it's needed, to provide the problems the check finds.

In case we want to provide different checks from a plugin we will also have the ILanguageCheckProvider:

```
class ILanguageCheckProvider
{
public:
  virtual QList<ILanguageCheck*> providedChecks() = 0;
};
```

This one will provide different ILanguageCheck instances just like those explained above which will be implemented in the same way.

```
class CheckData
{
public:
  KUrl url() const;
  TopDUContext* top() const;
  ControlFlowGraph* flow() const;
  DataAccessRepository* access() const;
};
```

Whenever we need to run the code, we will receive a CheckData class that will provide the required information. For the moment we will be getting the file location in a KUrl form, the file's TopDUContext to access the DUChain data, the file's ControlFlowGraph to know about its code paths and the file's DataAccess's information to know how the program is dealing with the data. We hope it will be possible to create a great number of checks with this.

Chapter 4

Check Execution

Now that we have reviewed our check architecture we need different pieces to make it possible to make full use of the provided checks. In this chapter we will explain the tools that have been created in order to do so.

4.1 Running Checks

Besides being a less technology related subject, I think this is a very important part of the project. A project like this, without users, has not much to offer, and if the users don't have the tools they need, they will be just frustrated.

While developing all the project, one of the main goals has been to provide a meaningful solution to make it possible to have a day to day solution for static analysis and I understand that for users it's very important to have the needed set of tools available. Here we will present at least two different ways to integrate the check results into our daily work.

4.1.1 Editor integration

The first chosen approach was to create a KDevelop plugin that we called *KDevChecksRunner*, mainly for the sake of simplicity. In KDevelop we already have most of the environment logic prepared: we are importing the projects, we are parsing the project's files, we have places in the GUI where to show the found problems. In this case what we did was to prepare a plugin that hooks to the KDevelop GUI and there performs the checking when it's due.

This means we are getting automatically the checks being run while developing KDevelop. It's not our final use case but I think it's a very interesting feature.

4.1.2 Project checking

Having our project checked while coding is an interesting feature but it has some problems. First of all, it seems like we can't force everyone to use KDevelop as his IDE, also we want to check code that has been written already or to see new interactions after adding code somewhere else. We want to have a tool that opens a project all together, parses it with all the data we need and provides reports of the project all at once.

We created a tool called `kdevchecker` that does that. These are the different arguments we can provide:

–project $\langle path \rangle$ will open a `kdevchecker` instance and parse the specified project

–export $\langle file.json \rangle$ In case we want to output the results in a file to be consumed by another application, this argument lets you specify where to put the information. This information will be given in a JSON[1] file that can be read from any application, although it's specially thought to be consumed by web services, to post check results on the open to be fixed by any developer.

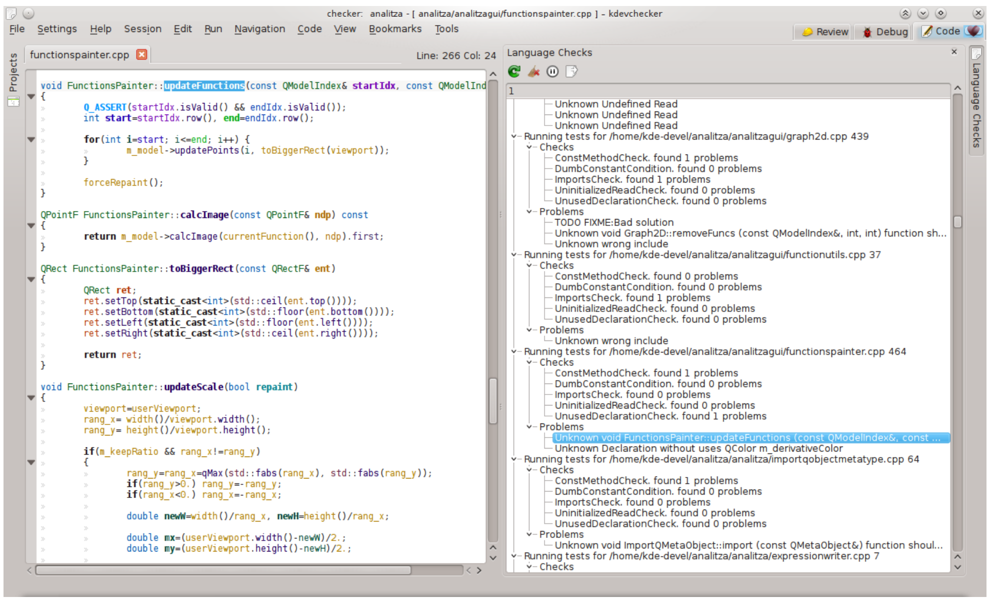
This feature could be interesting also for being able to unit test checks against test projects. It's quite easy to compare two files and figure out what changed, so it's an interesting new feature to be developed in the future.

–check $\langle name \rangle$ will tell the program what check do we want to run. It's important mainly for development purposes. When you are working on a check, you don't want others to slow down the process and clutter the report UI, it's very useful to use this argument in this case.

–verbose By default, all the program output is inhibited because there was too much output irrelevant to the user. With this argument it's enabled back, it's also mainly useful when developing checks or even the architecture.

The *export* option is specially interesting, we are analyzing the code once, but we probably want to share the results with different people. I wanted to come up with an idea to be able to do so in a way that's easy to export it through the network but also that is possible to integrate it easily with other software. My first thought was to export HTML reports that anyone could read on their browser, it was easy and clean but it had the problem of being almost impossible to integrate to any editor or to inspect is data with other tools, that's why

[1] JSON. JavaScript Object Notation, is a lightweight data-interchange format.


checker: analitza - [analitza/analitzagui/functionspainter.cpp] - kdevchecker
File Settings Help Session Edit Run Navigation Code View Bookmarks Tools
Review Debug Code
Projects
functionspainter.cpp
Line: 266 Col: 24
Language Checks

```
void FunctionsPainter::updateFunctions(const QModelIndex& startIdx, const QModelInd
{
        Q_ASSERT(startIdx.isValid() && endIdx.isValid());
        int start=startIdx.row(), end=endIdx.row();

        for(int i=start; i<=end; i++) {
                m_model->updatePoints(i, toBiggerRect(viewport));
        }

        forceRepaint();
}

QPointF FunctionsPainter::calcImage(const QPointF& ndp) const
{
        return m_model->calcImage(currentFunction(), ndp).first;
}

QRect FunctionsPainter::toBiggerRect(const QRectF& ent)
{
        QRect ret;
        ret.setTop(static_cast<int>(std::ceil(ent.top())));
        ret.setBottom(static_cast<int>(std::floor(ent.bottom())));
        ret.setLeft(static_cast<int>(std::floor(ent.left())));
        ret.setRight(static_cast<int>(std::ceil(ent.right())));

        return ret;
}

void FunctionsPainter::updateScale(bool repaint)
{
        viewport=userViewport;
        rang_x= width()/viewport.width();
        rang_y= height()/viewport.height();

        if(m_keepRatio && rang_x!=rang_y)
        {
                rang_y=rang_x=qMax(std::fabs(rang_x), std::fabs(rang_y));
                if(rang_y>0.) rang_y=-rang_y;
                if(rang_x<0.) rang_x=-rang_x;

                double newW=width()/rang_x, newH=height()/rang_x;

                double mx=(userViewport.width()-newW)/2.;
                double my=(userViewport.height()-newH)/2.;
```

```
1
      Unknown Undefined Read
      Unknown Undefined Read
      Unknown Undefined Read
Running tests for /home/kde-devel/analitza/analitzagui/graph2d.cpp 439
  Checks
      ConstMethodCheck. found 1 problems
      DumbConstantCondition. found 0 problems
      ImportsCheck. found 1 problems
      UninitializedReadCheck. found 0 problems
      UnusedDeclarationCheck. found 0 problems
  Problems
      TODO FIXME:Bad solution
      Unknown void Graph2D::removeFuncs (const QModelIndex&, int, int) function sh...
      Unknown wrong include
Running tests for /home/kde-devel/analitza/analitzagui/functionutils.cpp 37
  Checks
      ConstMethodCheck. found 0 problems
      DumbConstantCondition. found 0 problems
      ImportsCheck. found 1 problems
      UninitializedReadCheck. found 0 problems
      UnusedDeclarationCheck. found 0 problems
  Problems
      Unknown wrong include
Running tests for /home/kde-devel/analitza/analitzagui/functionspainter.cpp 464
  Checks
      ConstMethodCheck. found 1 problems
      DumbConstantCondition. found 0 problems
      ImportsCheck. found 0 problems
      UninitializedReadCheck. found 0 problems
      UnusedDeclarationCheck. found 1 problems
  Problems
      Unknown void FunctionsPainter::updateFunctions (const QModelIndex&, const ...
      Unknown Declaration without uses QColor m_derivativeColor
Running tests for /home/kde-devel/analitza/analitza/importqobjectmetatype.cpp 64
  Checks
      ConstMethodCheck. found 1 problems
      DumbConstantCondition. found 0 problems
      ImportsCheck. found 0 problems
      UninitializedReadCheck. found 0 problems
      UnusedDeclarationCheck. found 0 problems
  Problems
      Unknown void ImportQMetaObject::import (const QMetaObject&) function shoul...
Running tests for /home/kde-devel/analitza/analitza/expressionwriter.cpp 7
  Checks
```

Figure 4.1: Screenshot of the kdevchecker application.

I decided to export JSON files. It will be printing a JSON file with records for all files, listing all the problems found together with its location. Let's see an example:

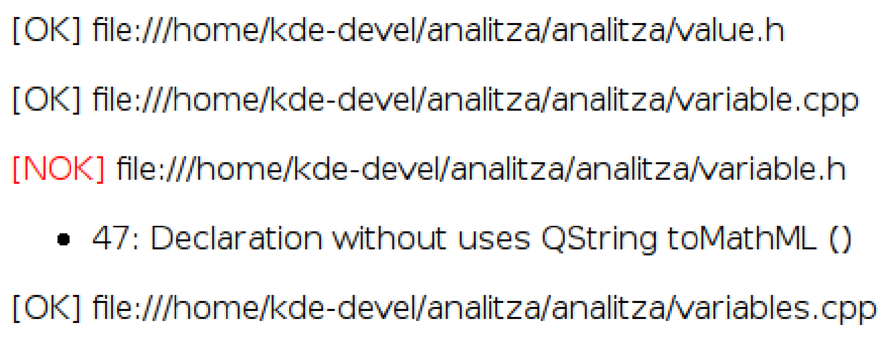
```
{
  "files": [
    {
      "name": "kalgebra/analitza/abstractexpressiontransformer.cpp",
      "problems": []
    },
//...
    {
      "name": "kalgebra/analitza/expression.cpp",
      "problems": [
        { "description":
            "Object* Expression::tree () should be const",
          "line": 704
        },
        { "description":
            "bool Expression::check (Apply*) should be const",
          "line": 146
        },
        { "description":
            "bool Expression::check (Container*) should be const",
          "line": 185
        }
      ]
    },
//...
  ],
  stats: {
    "ConstMethodCheck/Elapsed": "2123",
    "ConstMethodCheck/ProblemsFound": "5",
    "kalgebra/analitza/expression.cpp/Elapsed": 32
  }
}
```

As we can see, we have a list of files that contains its name and specific problems, so that we can have all the needed information to produce good reports. Additionally we have a statistics record that will provide us some information useful to analyze how has the execution evolved and will give us some hints about what to improve in the future and what is giving best results.

The considered use case for this is to have a kdevchecker call in cron that will put the results in a common place for the developers to consume. To make it a little easier to believe, the KDevChecksRunner is providing also a Python[11] program that will output an HTML document that can be used to upload easily for the users to read.

4.2 Check Distribution

When starting to develop checks, the first thing that occurred to me in order to start developing checks, was to start the checks pack. Here we have a plugin that is responsible for the infrastructure by providing every check inside the pack, which will be in a separate .cpp file that statically registers itself to the plugin to be initialized when it's required. It works great but it has all the advantages and disadvantages of C++ checks that are being described in the



```
[OK] file:///home/kde-devel/analitza/analitza/value.h
[OK] file:///home/kde-devel/analitza/analitza/variable.cpp
[NOK] file:///home/kde-devel/analitza/analitza/variable.h
    • 47: Declaration without uses QString toMathML ()
[OK] file:///home/kde-devel/analitza/analitza/variables.cpp
```

Figure 4.2: Example of some output converted to HTML.

appropriate chapter. In short, regarding distribution, it will mean that we will need to have a KDevPlatform development environment set up to be able to compile and run the checks. It's not ideal, although it can be done very efficiently by using the traditional distribution channels provided by GNU/Linux distributions.

There's also another player, the functional checks (described in length in the chapter with that name). In that case, checks are also a couple of text files in the correct directory, this opens a wide range of possibilities: they can be distributed just like C++ checks, they can be easily distributed from a web service, they can be provided by a private vendor, etc. The limit is the imagination.

I'm not saying those possibilities are not possible with compiled checks, it's just harder.

Chapter 5

Creating Checks

When we were talking about the project's goals, we said that one of the important parts is to make it easy to create further checks. In this chapter we will explain (what) the proposed way to create checks (is), what we expect the developer to do, and what the developer will have to worry about. In this document we have specified how to create a wide range of opportunities for the check developers, here we will try to narrow these possibilities down in favor of flexibility.

5.1 C++ checks

Developing checks in C++ can be a very good option if we want to be aware of how the check code will be executed when it's working on a computer, but that is a very specific case and it isn't always what we are looking for. Sometimes we will want to favor ease of distribution or code simplicity instead of run-time efficiency. That's why we have created this section, where we will discuss the advantages and problems of every option.

5.2 Scripted Checks

C++ checks are great, lets us to use the KDevPlatform library as it was meant. We can also be very specific about what we want the check to do so that it's efficient. And we can do whatever we feel like with it, it's a KDevelop plugin and our only limitation is our creativity. On the other hand we found some limitations that we consider worth addressing.

First of all in the C++ code we're defining how to check something but, in a way, we're going for what to check. This means that sometimes writing those checks can be a little hard in C++. In addition there's the distribution problem, a C++ check has to be compiled before installing and, given the different places where it can be run, it's probably a good idea to be able to have an alternative.

5.2.1 Choosing a language, Alternatives

I will try to list what requirements I wanted, once we decided to look for an alternative language to define checks, before I will turn to the actual decision.

As we have said before, the problem we're dealing with here is pretty well defined: we have a bunch of data and we want to turn it into a set of problems. We need a language that is great when dealing with various data structures. Neither IO, nor threading, nor UI, etc. This definition works very well if we think about the functional paradigm. Moreover, we want to remove the compilation requirement, so scripted sounds good, too.

- Prolog was the first language we considered. It makes complete sense because it's aimed at data analysis and problem solving. The issue there was that it looked quite hard because most C++ bindings I found required us to feed all the data to Prolog. Our conclusion was that, given that there's a lot of information already fetched in C++ data types, it might not be the best idea to use it. If that was a wrong premise, this could lead to a very interesting research project in the future.

- Haskell seemed eligible, too. Its pure functional point of view would have been great for our needs, the main problem here was that its code requires compilation. Also there's some problems that we wouldn't be able to solve with the default *c2hk* tool being mostly targeting C instead of C++, so we would need to create bindings for those anyway.

- Python and Ruby were considered as well. Since their paradigms are quite similar I will treat them together. I quite liked both because of their scripting nature; they both are very scriptable even if they are not really intended to serve as plugin languages, but as full applications bindings. Both require quite some learning and effort to create the bindings. The main problem with them is that there's no such concept for type checking which would be very helpful for solving this kind of problem (turning a lot of different data into problems).

  The lack of type checking was considered very bad in this case, as we want to make sure the checks are consistent. In a project like this one it makes sense to use languages that can be statically checked.

- JavaScript was a very good candidate as well. It's very easy to integrate with the KDevPlatform because Qt [1] already provides ways to support JavaScript scripts from our applications (not so) easily. While it's possible to feed classes ,they (also) have to be QObject based. This is fine, yet not great because it would require the creation of bindings. In addition there's the problem that there's no type checking whatsoever.

- KAlgebra's Analitza module was considered mainly because it's being developed by me. The language was initially created to support a calculator. Over the course of time it has got some advanced features like static type checking with type inference and some speed improvements that made it worth considering. It is scripted, we can run its scripts by simply linking against a library and feeding the source and bindings there, it felt good.

  It has some of the advantages and disadvantages of most languages listed here, but having developed it for several years also helps to make sure that we're getting the most out of the language, even if it required many additions to the KAlgebra language.

5.3 Check development tools

Developing checks is not a simple task. It requires different steps and organization to work. Just like when it comes to writing applications, we'll need a set of tools to assist as much as possible. The development procedure itself raised the needs that are required to have enough flexibility when developing.

When developing a check, first of all we'll create small programs with specific chunks of code that we will use to check our code has been analyzed correctly. We won't care about the problems that the test projects have at all, we will focus on our test.

When I started to work on checks I developed the KDevChecksRunner plugin. This enabled me to start checking if they worked by having the files being parsed when coding: it was good but not very usable. Then I started working on the *KDevChecker* sub-project that eased it a little because I just had to start it and see what was working and what wasn't. The next step further was letting the *KDevChecker* to run just one check so that I didn't get distracted with extra information from other stuff that could be not working properly and got in the way.

Furthermore, when I started working on the functional checks some other tools where created to work on it directly, but this will be addressed after describing its development process.

[1] The Qt Project is the library KDevelop and KDevPlatform are based on.

Chapter 6

Functional Checks

As we discussed on the previous section, we decided to use the KAlgebra project. Here we are going to see what parts are really being used, what parts can be reused, what changes had to be made and what tools were created in order to make it smooth enough.

6.1 What is the KAlgebra project?

The KAlgebra project was initially meant to be just a calculator with plotting capabilities to be used in the KDE Edu[9] context.

KAlgebra's language is based on MathML, this means that when we're compiling the code instead of providing a regular AST, an XML-based code is provided to be parsed. This has a reason to be, we are not basing our language features to our syntax but to a specification, which eased the initial development. The served as a base to create a fully featured language and started to introduce some functional ideas, like lambda, into the project.

The Analitza module is the part of the KAlgebra project that is able to calculate mathematical expressions. It started as a library to be able to properly define unit tests that make sure that all is working. Soon a lot of useful alternative uses appeared, that's when the console version and the Plasmoid [1] appeared. Furthermore, Cantor back-end arrived some years later providing a very different and interesting way to use a calculator by integrating it in a styled sheet, like other programs like Maple do. Nowadays there are some other projects using the Analitza module that are work in progress: the KAlgebra Mobile to take some KAlgebra features to mobile devices of different form factors, a new 2D and 3D plotting application yet

[1] Plasma is KDE's desktop environment. A Plasmoid is an applet that can live integrated in the desktop environment.

to be named and an Analitza back-end for Rocs, a KDE Edu application to teach graph theory algorithms.

All in all we can consider Analitza as the library that calculates the things and KAlgebra the global project name as well as the application that started it and that is living in the KDE Edu project.

6.2 History

The KAlgebra project started as a calculator application back in the year 2005, because since in my last years of mathematics studies I wanted to have it. I finally decided to start working with it, when I wanted to investigate a little about source code parsing towards different features like 2D/3D function plotting. I maintained this project since then, with a strong focus on having a feature-full, not-too-flexible language. I still was learning a lot about C++ and software development. It was not the best quality in the beginning but it already started to attract some education-based distributions to distribute it and letting me receive its feedback.

I used that project to put in practice a lot of the skills I acquired in software development, it has been rewritten many times over the years and I have tried to have it as polished as possible so that it can be used as good as possible. It has to be noted that myself, I'm not a KAlgebra user anymore, I don't really need to do this kind of calculations in my day to day life, so my main interest in it was both to adapt it to its actual users needs and to adapt any *cool* feature I could think of, based on my initial premises.

That lead us to the inclusion of the inference type checker, a slow but steady language enhancing and quite some run-time optimization, because we all like fast applications.

6.3 Language adaptations

As I said, even if it made sense to adapt the KAlgebra language, it was missing some very basic features like string support, built-in methods (or bindings if you prefer) and abstracting a little the way we have to import.

Type Checker Stabilization When I started this project the type checker was already in place but it lacked some features regarding the lambda types inference that were needed right when I started to implement high order functions like map and filter, that initially were written in the Analitza language.

String Support Until this project there was no need to have String support on the language because it was only used in calculators. The implementation was straight forward. A string is considered a list of characters, so we didn't have to add new semantics. We added the Character primitive and we adapted the parser to understand quoted inputs and we were good to go.

The fact that they are lists meant that we didn't have to add functions to deal with them, because we could already use the `union` and `selector` functions, for instance.

Built-in Methods When we decided to go for KAlgebra to get our checks run, the first important thing to do was to implement some way to call code that is not defined inside the scripts, so I created this built-in infrastructure that lets us define code in C++ that can be called from within KAlgebra scripts without needing to put them in the Analitza code. This feature will be described further in the next section.

Custom Types One of the most important reasons to use Analitza for the check running was its type checker, but this made evident the requirement to add a new and flexible way to define types. To do so I made it possible to define custom types with a name that can be any string, then the type checker can perform a simple string compare to see if it's the same type or not.

File Execution Until now KAlgebra code was mostly run expression by expression from a Graphical User Interface. It was possible to import external files but it was not very optimized for it. Some work was added in order to make it possible to properly parse files with multiple expressions properly.

Comments Like in the last case, such a typical feature was not present before in the KAlgebra language. A new token '//' was added that defines where starts and ends the comment.

New KAlgebra functions With the new type of coding happening, we got new use cases. Because of the nature of the checking that we're doing, the needs for further boolean functions raised, that's why *exists* and *forall* were added. The high order functions *filter* and *map* were added because its implementation in KAlgebra code were not performing good enough so it was decided to build them inside the language, because they are generic enough.

Here we can see a detailed list with the changes.

exists `exists(x : x@list {true, false, false})`, tells if there is any true statement in the bounded value.

forall `forall(x : x@list {true, true, true})`, tells if there is all statements in the bounded value are true.

filter `filter(x->x>3, list {1,2,3,4})`, will return the elements list that matched the first argument condition.

map `map(x->x+3, list {1,2,3,4})`, will apply to all the elements the first argument function and return a list with all the results.

6.4 Using the Analitza module

6.4.1 Getting started

First of all to read our expression into something that can be computed, we pass the code into an Expression instance that will create a rich AST, to be fed to the Analyzer (or whatever that might want to access the tree, we'll discuss about it later). The Analyzer knows about the variables that we have defined and the different builtin features we might have. Whenever an Expression is tied to it, the type checker will be run so that we can start executing it, which is what we're going for, in the end.

At this point we will have different options, most importantly:

calculate In case there's an unresolved variable it will error, otherwise it will provide a result as fast as possible:

simplify Without changing the semantics of the operation, it will modify the Analyzer's expression into an optimized version.

evaluate It substitutes variables and simplifies. It's like calculate but instead of calculating, it simplifies to be flexible with the undefined variables.

calculateLambda In case it's a lambda function what we have in the expression, it lets us pass the needed arguments with the "setStack" method and executes the lambda in function of those. It's useful to calculate functions to be plotted or, like we do in this project, to call a check with the different relevant data.

In the following code snippet we can see how would we invoke some calculation:

```
#include <analitza/analyzer.h>
#include <analitza/expression.h>
#include <iostream>

int main()
{
  Analitza::Expression exp("2+sin(pi/2)");
  Analitza::Analyzer a;
  a.setExpression(exp);

  Analitza::Expression ret = a.calculate();

  if(a.isCorrect())
    std::cout << "result: "
      << exp.toString().toStdString()
      << " = "
      << ret.toString().toStdString()
      << std::endl;
  else
    std::cout << "errors:\n - "
      << a.errors().join("\n - ").toStdString
      << std::endl;

  return 0;
}
```

The execution of this code is as simple as expected. This code could be extended to more complex cases, but that's a different issue.

```
$ g++ main.cpp -I /usr/include/QtCore/ -l analitza
$ ./a.out
result: 2+sin(pi/2) = 3
```

6.4.2 Built-in functions

One of the most important features that we will need in this project is the Built-in Methods. The whole point of using a bindable language is to be able to navigate through some data repository that we can access through some C++ bindings.

To do so, we created some simple infrastructure that lets us create functions given a random name, type and an object that will define what to execute and return. Please note that here it's up to the developer to make sure that the types requested are the ones we're looking for.

Here I wrote a not-so-simple example that shows us how to use this feature. We created a C++ function that given a list of strings and a string returns a string with the list of strings joined by the second argument. The example will join with a coma ", " the strings "joan", "pere", "maria".

```
#include <analitza/analyzer.h>
#include <analitza/expression.h>
#include <iostream>
#include <QtCore/QList>

using namespace Analitza;

class JoinStrings : public FunctionDefinition
{
  virtual Expression operator()(const QList<Expression>& args)
  {
    //we extract the first argument as a list of expression
    QList<Expression> exps = args.first().toExpressionList();
    QStringList strings;

    //We turn every expression into a string
    foreach(const Expression& exp, exps)
      strings += exp.stringValue();

    //We extract the second argument
    QString joint = args.last().stringValue();
    return Expression::constructString(strings.join(joint));
  }
};

int main()
{
  Analyzer a;

  //we define the type, in haskell terms:
  //[[char]] -> [char] -> [char]
  ExpressionType joinType(ExpressionType::Lambda);
  joinType.addParameter(
    ExpressionType(ExpressionType::List,
             ExpressionType(ExpressionType::List,
                    ExpressionType(ExpressionType::Char))));
  joinType.addParameter(
    ExpressionType(ExpressionType::List,
             ExpressionType(ExpressionType::Char)));
  //return value
  joinType.addParameter(
    ExpressionType(ExpressionType::List,
             ExpressionType(ExpressionType::Char)));

  //we insert the function by passing:
  //the name, the type and the functional object
  a.builtinMethods()->insertFunction("join", joinType,
                                    new JoinStrings);

  //join( list{ "john", "peter", "maria" }, ", " )
  Expression exp("join( list{ \"joan\", \"pere\", \"maria\" },"
                 " \", \" )");
  a.setExpression(exp);
  /* we proceed just like in the last example */
}
```

The output again, is what we would have expected. A whole string with the list's contents separed with coma's.

```
$ g++ main.cpp -I /usr/include/QtCore/ -l analitza
$ ./a.out
result: join(list {"joan", "pere", "maria"}, ", ") = "joan, pere, maria"
```

This feature doesn't go much further but essentially this, together with the custom types, have made it possible to use the Analitza module to be used in this project all together. This kind of support could be enhanced in many different ways, but for the moment it's been good already.

6.5 KAlgebra-based check development

To understand how does it all fit in our project, let's schematize how does it all work. We have different parts:

C++ support First of all, we have KDevelop's C++ support that, given a C++ file it will generate the different internal data structure we've been describing.

KDevChecksRunner There's also the Checks Runner that will figure out when do we need to check a file, ask for the needed data to KDevPlatform's Language support and then it will invoke every found check to be run.

KDevAnalitzaChecks This plugin will have is responsible for listing the installed Analitza-based plugins, compile them (while assuming all went well, it won't report errors) and execute them for every file. To see if a check is correct we'll have the `kdevanalitza checkcompiler` tool that will print us if there's any error and will provide some data in case we want to debug the application.

6.5.1 Binding to the language

To create the KDevPlatform bindings we needed an automatic way to create a bunch of functions that communicate our scripts to the KDevPlatform, without spending too much time on the task since it could get very repetitive.

Bindings development

To do this we created an application that automatically generates the bindings. Firstly, we needed a way to extract information from the C++ class specifications, in the header files, generally. For the parsing we decided to use cpptoxml. It was very convenient because it gets a C++ file and it automatically outputs an XML document. It's really easy to read XML, there's a lot of technology already created for that, and it would help us concentrate on the transformation from an interface to the actual bindings instead of working on reading the C++ header files.

With this, we created a Python [11] application that: given a header file and which classes we want to extract from it, it extracts the classes definitions in the said XML to be analyzed. From here we will extract all the methods and create an Analitza built-in function from each method. We will also read every enumeration and will create variables for enumeration value we find there.

It should be noted that for every method we will have to check all the arguments and in case it's something that KAlgebra understands we have to translate it to its corresponding KAlgebra type (like in numbers, lists, strings, etc.). If it doesn't exist we can create a KAlgebra custom object value that KAlgebra treats as if it were a black box, which can't be read, but passed around only. Also we will only generate bindings for the methods that don't change the object state, that is `const` methods and static functions.

With all this, we'll generate a `bindings.cpp` file that will contain the bindings for all the files. This file will provide a function that takes an `Analitza::Variables*` and a `Analitza::BuiltinMethods*` instance and it will fill those with the said bindings.

Bindings generation example

After explaining a little how does it all work, let's take a look at what it looks like. First of all, we'll create a simple class header as an example:

```
#include <QtCore/QString>

class Parent {};

class SomeClass : public Parent
{
public:
```

```
double coolMethod(int a) const;

QString getName() const;

//Won't be exported because it's not const
void setName(const QString& name);

//some attributes
int z;
};
```

In this example we can see two classes, an empty Parent one and SomeClass. In this example, SomeClass will be exported to the Analitza facilities to be used from scripts. As discussed before, it's going to be exported by using the `cpptoxml` tool. Before rushing into the final results, let's see what we're getting here.

```
<File >
  <Class name="SomeClass" bases="Parent;" class_type="class" >
    <Function name="coolMethod" constant="1" access="public"
      type_name="double">
      <Argument name="a" type_name="int" />
    </Function>

    <Function name="setName" access="public" type_name="void" >
      <Argument name="name" type_name="QString const&amp;"
        type_constant="1" type_reference="1" />
    </Function>

    <Function name="getName" constant="1" access="public"
      type_name="QString" />

    <Variable name="z" access="public" type_name="int" />
  </Class>

  <Class name="Parent" access="public" class_type="class" />
```

```
</File>
```

As you can see, it's a typical XML structure. To improve the readability of the file, some information was removed for a better understanding by the reader. If the In this file it can be easily seen how it's structured with the classes and the relations specified, now let's see what kind of code do we produce from this.

Here we can see how the arguments are converted from C++ types to KAlgebra specific types by constructing Cn on runtime and Expression::Value as a type.

This first example will be commented for better reading, the following examples will work likewise.

```
struct SomeClass_coolMethod : public Analitza::FunctionDefinition
{
  //This function operation will be called when we need
  //this function to be called.
  virtual Expression operator()(const QList<Expression>& args)
  {
    //The first argument is the *this pointer
    SomeClass* thisElement=args[0].customObjectValue()
                                .value<SomeClass*>();

    //We assert to make sure it's a valid object instance
    Q_ASSERT(thisElement && "calling SomeClass::coolMethod");

    //We extact the actual method arguments
    int arg1=args[1].toReal().intValue();

    //We make the call
    double theRet(thisElement->coolMethod(arg1));

    //We return the value in an expression.
    return Expression(Cn(theRet));
  }

  //Here we specify the type for the whole function
  static ExpressionType type() {
     //It's a function
     return ExpressionType(ExpressionType::Lambda)
              //This is the *this argument
              .addParameter(ExpressionType("SomeClass*"))
              //The integer argument
              .addParameter(ExpressionType::Value)
              //The return value
              .addParameter(ExpressionType::Value);
  }
};
```

Here we have a simple get method that returns a string. There are two main things to note here: first of all, we can see that we'll still have the `this` pointer as the first argument, it's maybe evident but worth noting. Also we are returning a string, and we want to be able to read it in our code, so the bindings generator instead of constructing a QVariant[2] with a QString inside, we'll create a string expression specially constructed by the Analitza module.

```
struct SomeClass_getName : public Analitza::FunctionDefinition
{
  virtual Expression operator()(const QList<Expression>& args)
  {
    SomeClass* thisElement=args[0].customObjectValue()
                                   .value<SomeClass*>();
    Q_ASSERT(thisElement && "calling SomeClass::getName");
    QString theRet(thisElement->getName());
    return Expression::constructString(theRet);
  }

  static ExpressionType type() {
    return ExpressionType(ExpressionType::Lambda)
      .addParameter(ExpressionType("SomeClass*"))
      .addParameter(ExpressionType(ExpressionType::List,
                              ExpressionType(ExpressionType::Char)));
```

[2]QVariant is a Qt class that works like a C union that can contain almost-any type or class.

```
  }
};
```

Here we are accessing a variable attribute. The are just implementing it like a get method, since we're not letting the checks to modify the code model state. Here it's a number type again, so it will have a `ExpressionType::Value` type again.

```
struct SomeClass_z : public Analitza::FunctionDefinition
{
  virtual Expression operator()(const QList<Expression>& args)
  {
    SomeClass* thisElement=args[0].customObjectValue()
                                  .value<SomeClass*>();
    Q_ASSERT(thisElement && "calling SomeClass::z");
    return Expression(Cn(thisElement->z));
  }

  static ExpressionType type() {
    return ExpressionType(ExpressionType::Lambda)
    .addParameter(ExpressionType("SomeClass*"))
    .addParameter(ExpressionType::Value);
  }
};
```

As you have seen, there are just the `const` methods exported, that way we make sure that the state of the DUChain has not changed. Also it should be noted the presence of the `Q_ASSERT` call on every function call. It's useful mostly for debugging situations where you can immediately have an accurate hint about your problem without running the application through a debugger.

```
//we generate casts first
builtin->insertFunction("SomeClassToParent", /* the downcast */);
builtin->insertFunction("ParentToSomeClass", /* the upcast */);
builtin->insertFunction("ParentIsSomeClass", /* the cast check */);
```

```
//we generate class methods
builtin->insertFunction("SomeClass_coolMethod",
                        SomeClass_coolMethod::type(),
                        new SomeClass_coolMethod);
```

Here we can see how the methods have been generated and are passed to the `BuiltinMethods` instance. I have just put one example of the methods because it's the same every time. First of all we have the casts that we will be able to do to with the type to and from its parent, then we have a list of the functions to be added that first passes the name of the function, then the type and finally an instance of the object that will be called when the function is triggered.

Problems with the binding generator

Having a tool that generates the code you need automatically is good, but making sure that it all works properly can be tricky sometimes. During the development of this tool we have had to make sure all the typing is coherent in every case: is `RangeInRevision` different to `KDevel::RangeInRevision`? Our language doesn't have name-space abstractions so we have to make sure we use the same nomenclature everywhere.

In addition, `cpptoxml` tool is not perfect, there are some problems that are being workaround in our code. It's not ideal but sometimes we need just to deliver. Another problem was that some include headers are not prepared to be used with compilers other than `gcc` so we must try to create an environment that is the closest to it, if we don't want to fall in some *pre-processor traps* like the one that was calling `#define const` in some `#ifndef` clauses. It can get complex to debug those issues.

6.5.2 Developing an Analitza-based check

The check development definition is quite simple: we receive some input data that we want to pass to every check to deal with and in exchange we return a list of problems. When starting to work on checks, I found out that a pattern emerged quite automatically, that's why I defined this function:

```
abstractCheck := (problemConstructor, isCorrect, elements) ->
                  map(problemConstructor,
                      filter(inc->not(isCorrect inc), elements))
```

This function defines the steps that we will do in most checks:

1. Extract a list of elements to be checked. These elements can be of any type, depends on what we are checking.
2. Define a function that given one of those elements tells if it's correct or not.
3. Given an incorrect element, construct a problem out of it.

The logic of the `abstractCheck` function states that the first step will provide the elements, those elements will be filtered with a negated call to the second step function and the resulting element list will be mapped to the problem constructor to be given back to KDevelop, that will register it into the code model. It should be noted here that the check is not modifying the state of the checks data, just reading it and providing results out of it.

To give a better taste of how do these functional checks work, I'll show a stripped-down version of a simple one.

```
r_declarationsIn := (ctx,top) -> union(
    KDevelop_DUContext_localDeclarations(ctx,top),
    foldr((x,y)->union(r_declarationsIn(x,top), y),
                       KDevelop_DUContext_childContexts ctx)
)

//Recursively find all the declarations in a file
declarationsIn := top->r_declarationsIn(TopDUContextToDUContext top, top)

//It didn't have any use, so we create a problem stating so
createProblem := decl -> Problem(
    DeclarationToDUChainBase(decl) | KDevelop_DUChainBase_range,
    "Declaration without uses"
)

//We check that either it has uses or it's a function definition
isCorrect := decl->or(
  and(KDevelop_Declaration_isFunctionDeclaration decl,
      KDevelop_Declaration_isDefinition decl),
  KDevelop_Declaration_hasUses decl
)

//This check function is the one that will be called
//data contains the relevant information for the current file to be checked
check := data -> abstractCheck(
    createProblem,
    isCorrect,
    KDevelop_CheckData_top data | importtopctx | declarationsIn
)
```

This check will look up all the declarations in a file and figure out if there's any use of it. If there isn't, a new problem will be reported. This check can be improved in many ways but it's useful because it's compact enough. This example only relies on the DUChain to provide any results, but all the data structures are available to the language and are used for other checks.

6.5.3 The Analitza Check Compiler tool

As discussed earlier, the possibility to know if the check is well built is very important and we want to make full use of it. If we think about it, given the check's code and all the bindings' definitions we will be able to know whether its check has coherent typification. A simple (yet powerful) tool was created for that.

The main reason of the application is that we will run the application with some `*.kdevcheck` files as an argument and we'll have the errors being displayed (if any). If there are not we will be able to start trying them in the `kdevchecker` tool, otherwise there's no reason (actually it will lead the KDevAnalitzaChecks plugin to crash due to asserting about the check correction).

There are some other useful features:

- The `--doc <file.html>` argument lets us generate an HTML file with some list of facilities provided by the bindings.
- The `--variables` argument will list all the defined variables with its value and type.

It's very important to have this kind of tools, it lets us split our development process into what you should do to make the check to just work from sketching what you're looking for in the first place. As said previously in this document, in my case the type checking already solved me some problems regarding type coherence without even running the program. That saves up a lot of time debugging code that's just not correct.

Chapter 7

Checks

Like previously said, this project will provide a set of checks that, on the one hand, will show us how to make use of the facilities provided by this project and, on the other, will start to provide some features that we will be able to benefit from in our projects.

All these checks have been developed using the technologies created in this project, mainly the new data structures and the language that was adapted especially for this purpose, as evidence that it works fine and also that it's useful to create any kind of check.

These checks are not thought to be means to demonstrate that your project is going to work. They are targeting more at properly using the language's features, like proper `const`-ness and the removal of unused code, like in the imports, where we're telling the user that he's importing code that is not needed or the constant condition check that will tell the user that he wrote some code that won't be run.

7.1 Imports Check

This is the first check that was implemented using this infrastructure. It was decided to start with this one because it depends on the DUChain only, which is convenient as a first approach.

This one gathers all the Declarations used in the top context (that don't come from the same context) and checks if there's any import importing it. It's a good check because it helps to keep track of which imports are good and which ones aren't.

Note that in this context *import* would translate, in the C++ world, into an `#include` of another module.

In the last figure we can see more or less how has this check been organized. It's structure

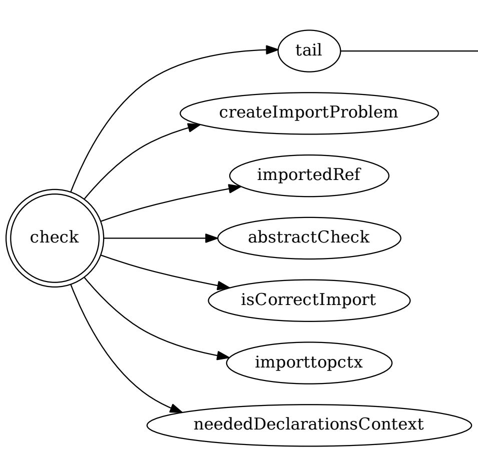


Figure 7.1: Imports Check's dependencies graph.

is very flat with the check procedure taking up most of the work.

Here what we do is, first of all, to extract a list of the files that are being imported. We'll provide those to the `isCorrectImport` call that, together with the results of `neededDeclarationsContext` that will tell us the non-local declarations used in the file, will check the import by seeing if there's any of those declarations provided by the import. If there isn't then it's not a correct import and it should be removed, so we create an error for it.

Let's see in which cases it would work and why:

```
#include <string>

int main()
{}
```

As discussed above, here we will be checking first of all, what imports it has: in this case just the one, afterwards the code will be scanned for all the relevant used declarations. Once we know them all, we will check if any of the declarations provided by the import is being used in the file. In this case there isn't any declaration using the string import, therefore it can be removed, so a problem will be registered.

```
#include <string>
```

```
int main()
{
  std::string s;
}
```

In this case, it will work the same way. The big difference here is that we have a `std::string` use. When we are checking the `std::string` declaration against the string import, it will tell us that the import is being used, meaning that it will be a valid import and no problem will be created.

I'd like to note here that the use of `std::` or `using namespace std;` is not relevant. These checks are run after the semantic analysis, so checks don't really worry about it.

7.2 Const Method Check

This check is relying on DUChain and Data Access. It tells us if any class method is not modifying any class attribute and in this case it adds an error telling that the `const` is missing.

I think it's an important check. If we want people to follow good coding practices such as the `const` method decoration, if developers use it safer code will be produced by making sure not to have side-effects. What a better way to enforce its use, than by pointing out what methods are missing it automatically, so that they can be changed?

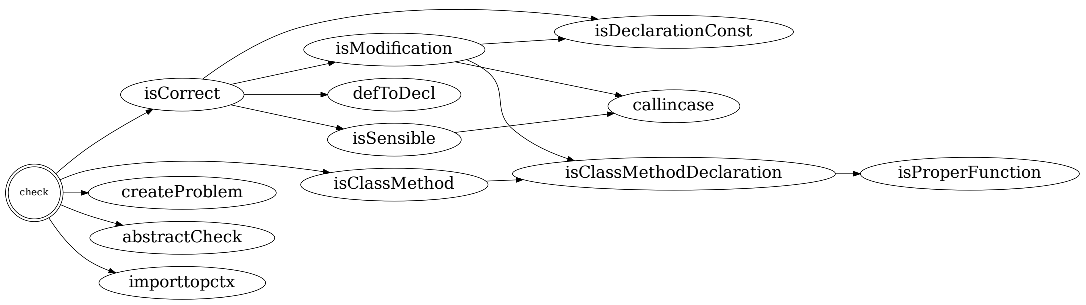


Figure 7.2: Const Method Check's dependencies graph.

This check is a little more complicated to analyze, let's go step by step. We want to know if any method declarations should be `const`. First of all, we extract all the function declarations and we filter the class methods from the rest (functions and static methods don't have attribute members so there's no need).

Among those methods, we first check if it's already `const`, if it is, then it's ok, if it isn't we check all the variable accesses; to see if there's member being modified and the function calls to see if they are non-`const` class methods being called. If there are class writes or non-`const` calls, we create a problem for it.

Let's see how it would behave.

```
class A {
  int b;

public:
  int f() { return b; }
};
```

In this case, the f method should be `const`, because we are not modifying any class field or calling a non-`const` method. First of all, the check will find all the methods, it will filter the ones that are already `const` and then it will analyze the contents of the function. If there's nothing inside that would prevent the method to be `const`, then it should be. In this case we are just reading the variable *b* so it should be, therefore a problem will be created asking the user to add the keyword.

```
class A {
  int b;

public:
  void setB(int b) { b=3; }
};
```

This one is correct already. The function won't be filtered out because it's a method, so it will proceed by checking inside the function's body. There we will see that there is indeed a write to a class member. In this case a problem shall not be created.

```
class A {
  void f(int) const;

public:
  void g(int b) { f(b+3); }
};
```

Calling methods will work similar. In this case *g* should be `const` because it's just reading arguments and calling other constant methods, so it can be turned to `const` as well. In this case the code will find all the methods, filter the ones that are interesting to us because they can be turned, then we will find what's in the methods we're interested in. In this case just reading arguments and calling other `const` methods from the same class instance, therefore this is a problem to be notified.

7.3 Unused Declaration Check

This check will be using only the DUChain and it probably has the simplest logic among the provided checks, but not for being simple it's less useful. It will tell us if either a variable, a class, a function or whatever we have indexed as a declaration doesn't have a use. This one will help us to clean up our code from useless declarations.

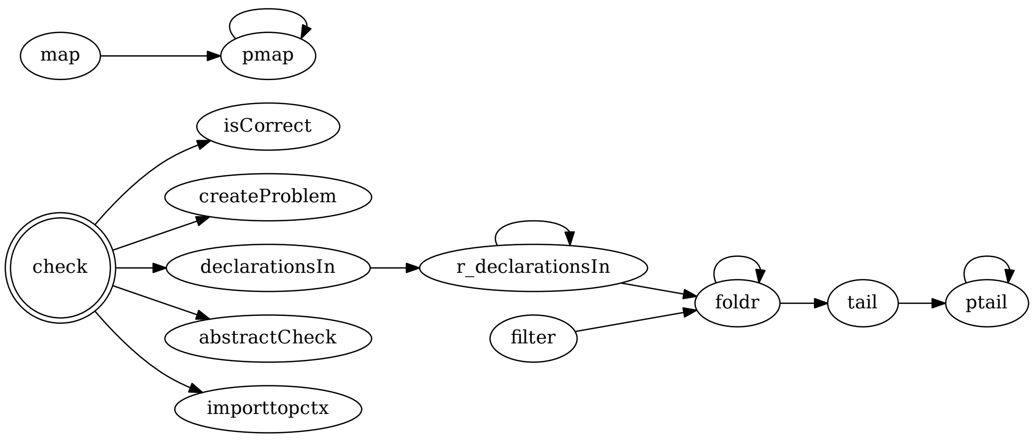


Figure 7.3: Unused Declaration Check's dependencies graph.

In this check, first of all, we'll extract all the declarations in the file. For every declaration we'll ask the DUChain whether it has any use, if it doesn't a problem will be created telling us not to declare variables just because.

```
int f() {
  int a=0;
  return 3;
}
```

In this case, there are 2 declarations: *f* and *a*. None of them has any use at all, so two problems will be created, one for each symbol, displaying that nobody is using them and maybe they can be removed.

```
int f() {
  int a=0;
  return a;
}
```

In this case, the variable *a* will have a use, so this one won't generate an error. In this case the check will only generate a problem for the *f* function.

```
class C {};
```

In this case, we will look up for uses of the C declaration. The strategy here is to parse all the .cpp files before we parse the header files, this means that we will have all the possible uses in the DUChain already and we will be able to provide accurate checks so that when we check it, we'll get all the uses in the project that use the class *C*.

7.4 Constant Condition Check

This check will make a semantic analysis of the conditions in the code by using all the DUChain, the Data Access and the Control Flow Graph. It will check if there's a case where we can tell that the conditional value is always constant, if this is happening a problem will be reported. It's interesting to have this check because it lets you remove code that otherwise could have been cluttering our project, but that has no sense or meaning at all but the increased complexity.

When this check is being run, first of all we'll extract all the conditional nodes in the code and from those we will extract their ranges. From these ranges it will check where does the data come from: if it's a function call, then it's not constant[1], if it's coming from variables we will keep checking where is the value coming from and repeat the same procedure until we know if it's just a constant. To know where the value is coming from, we will consider all the

[1]we're not taking into account constant functions for the moment, the standard c++11 is too recent.

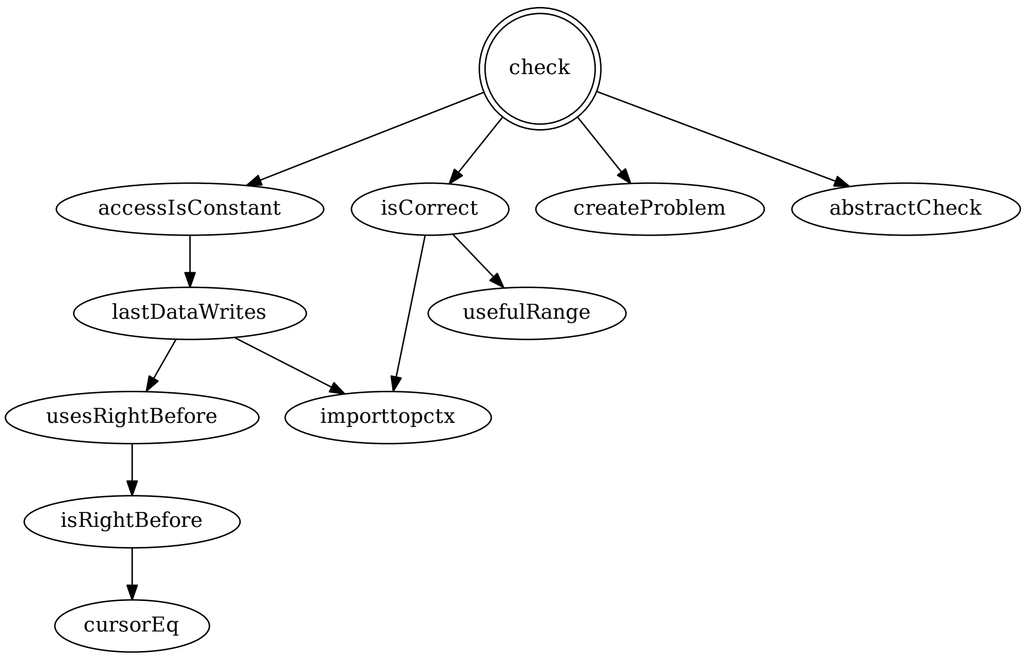


Figure 7.4: Constant Condition Check's dependencies graph.

different uses of the variable and make sure there's a path from the use to the place we are checking.

We will create a problem in case there's just one value option for every the value in the conditional.

```
bool a=false;

if(a) {
  //Very complicated code
}
```

In this code, first of all we obtain the range for the `if` condition, then we analyze the uses in its range. These uses can be, either a function or variable use. In case it's a function we won't continue the search, but given that it's a variable we'll go check the variable *a*'s value. In this case, when we analyze the value definition, we won't find any function call or different kinds of value that tells us it's not a constant, therefore this will mean it's a constant value.

```
bool a=false;

if(a) {
  //Very complicated code
}

a=true;
```

In this case we will also have two uses of *a*: one at the definition and the other after the conditional. In this case we will check the same cases as the first because there's no path from the use after the condition to the condition, so a problem will be reported again.

```
bool a=false;

while(a) {
  //Very complicated code

  a=true;
}
```

In this case we will proceed like in the example above. We'll check the condition, which will find a use of a. In this case we'll walk all the a writes again and we'll find out that there are two places where the variable is set that have a path to the condition, which will render evident that it's not a problematic case, because they can have different values.

We should note here that the part that decides is the possible paths counting, not the value itself. We just know if it's a constant value or if it comes from somewhere else.

7.5 Undefined Read Check

One of the most important limitations of the imperative programming languages is all the network of undefined states we can get to. One of the most important cases of those is the undefined reads. They can make a program not to work just because the algorithm is using some value with undefined value. Left aside the discussion regarding if the language should allow it or not, here we're proposing a check that will point out all these problems.

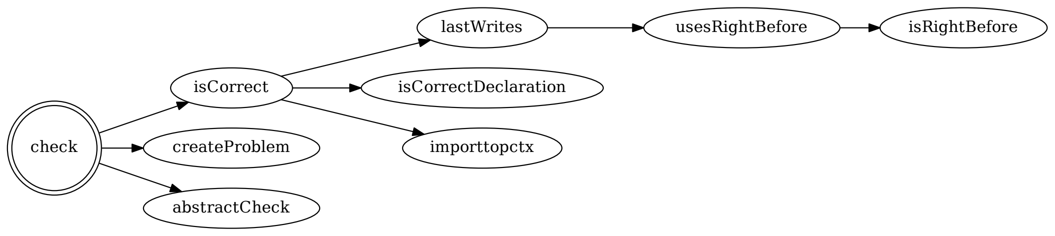


Figure 7.5: Undefined Read Check's dependencies graph.

To find undefined reads, what we will do is to start by extracting all data accesses and filter the reads. Once we have all the reads in the program we will extract the declaration of the accesses to figure out if it could be a problem, for instance classes have constructors, hence they don't need to be initialized explicitly, also non-local variables won't be taken into account either. When we have made sure it's a correct option to consider, it will start looking for the last write for every read. If there's no write in any of the paths, we'll mark it as an undefined read.

```
int f()
{
  int a;
  return a;
}
```

In this first example, we would find just 1 data access: A read on *a*. What would happen here is that we would get that data access, we would make sure it's a local variable and an integral data type, so it could be considered. There's no write data access in the only path from the declaration to the function return.

```
int f()
{
  int a=1;
  return a;
}
```

This example is similar to the previous one, but here the *a* declaration is being defined. This means that we will have a write data access in addition to the return's. First of all we will discard the non-read data accesses, so we will just analyze the return one and there we will find out there's a write data access on a before the read, so we won't trigger the create problem process.

```
int f(int x)
{
  return x;
}
```

In this case there is a read data access that is being return. If we analyzed it we might find out it's not being defined, this won't happen because it will be filtered out for not being defined inside the function, so no problem will be added.

7.6 Checks execution

Before finishing this chapter I'd like to wrap it up a little by discussing what's the outcome of all of this. There is many data that we can contrast to decide if those checks are useful and which ones should be created further. Also there's the false positives, there's the usefulness of the checks, etc. There's a lot of data that is being processed all the time, here I'll put some together and analyze it to have an overview about how did the checks behave and draw some conclusions.

In this section, will discuss the results of these checks on the Analitza code, which is a medium-to-small project, C++ mostly and with a lot of Qt constructions, which can be hard to analyze, specially when there are pre-processor macros being called.

First of all I wanted to check what is the ratio of the problems found on the Analitza code. I didn't remove the errors first, the reason for that is that I'm interested in analyzing code that is supposed to work but we're running the checks on top.

Here we can see the results we're having. As I said some Qt features can be problematic when running checks, in this case the QtTest module uses class run-time knowledge to call the tested methods, this increases already the problems count. Something similar happens with the `const` method check, where a lot of those test methods will be marked as errors while they aren't, also there are some other methods that should be `const` and they aren't. C++ Templates also trigger some false positives, but in this case we probably should adapt our checks better to these slightly-*weird* C++ constructions.

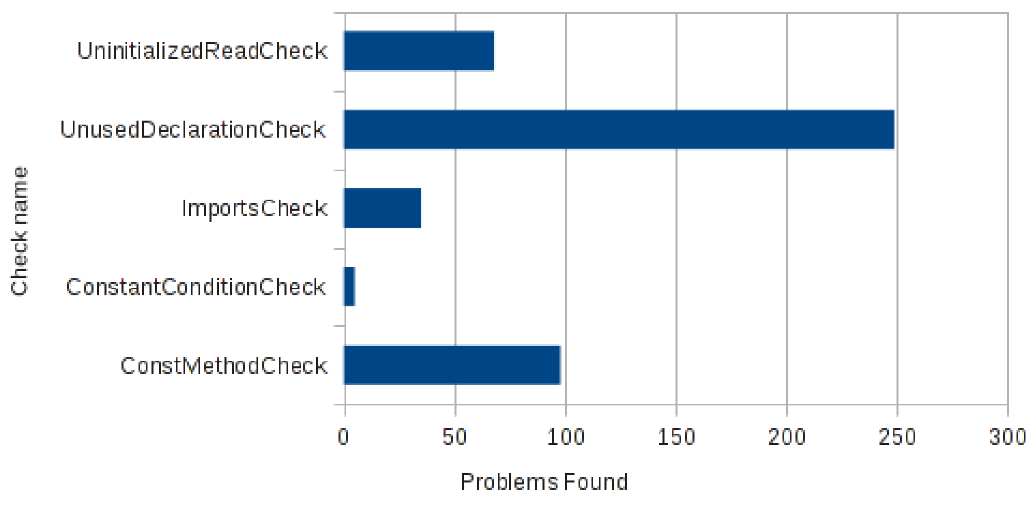


Figure 7.6: Problems per check

Furthermore, I was interested in how does the complexity vary depending on the different checks we have created. One of the concerns that came up when deciding to go for KAlgebra for our functional checks back-end was that it could not be as performing as other implementations. Of course here we don't have data to compare but we can share the results of what we have.

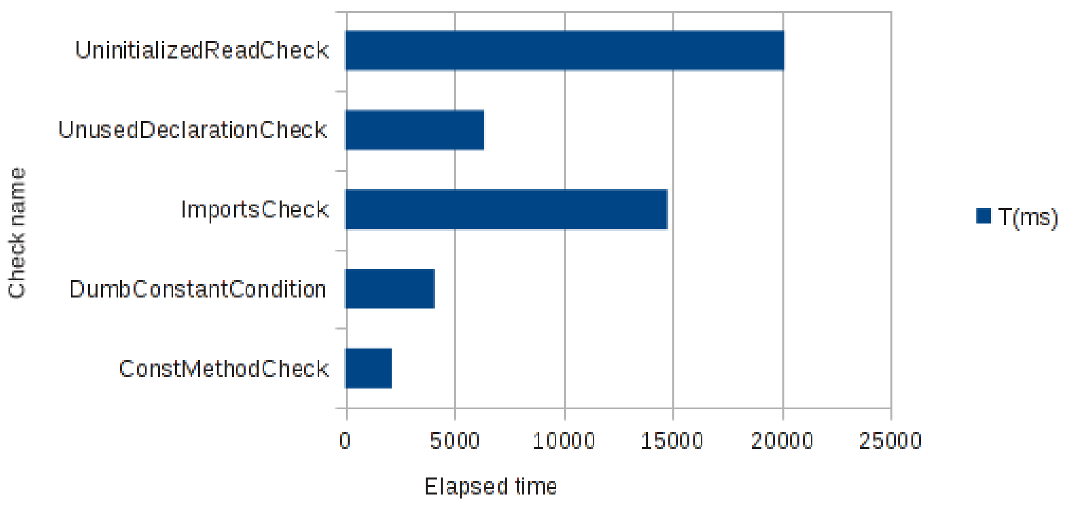


Figure 7.7: Elapsed time per check

In this graph first of all I'd like to note that there's no specific payload for running the checks on our functional infrastructure. Also I'd like to note that the less performing checks are the ones that rely on sophisticated calculations the most. We should note here that the KAlgebra run-time hasn't been optimized for the use case here, so there's some room for improvement, but that is out of scope here. Another thing I'd like to note is that there are some checks that take notably more time to run than others, so it would be probably a good idea to classify them by payload so, for example, some are just run when the project is batch-analyzed but not when the developer is coding.

Another thing I wanted to try is how does the source code size impact on the execution time. There would have been different ways to check that, I chose to count by file size over file execution time. The reason is that this way we are not skewed by any premise but for what we have, if we assume that all the code-base analyzed. One could argue that it would be more accurate to calculate it over tokens, variable definitions or source code lines, but I don't think it would display with enough fidelity what I'm looking for: how the run-time grows with source code size.

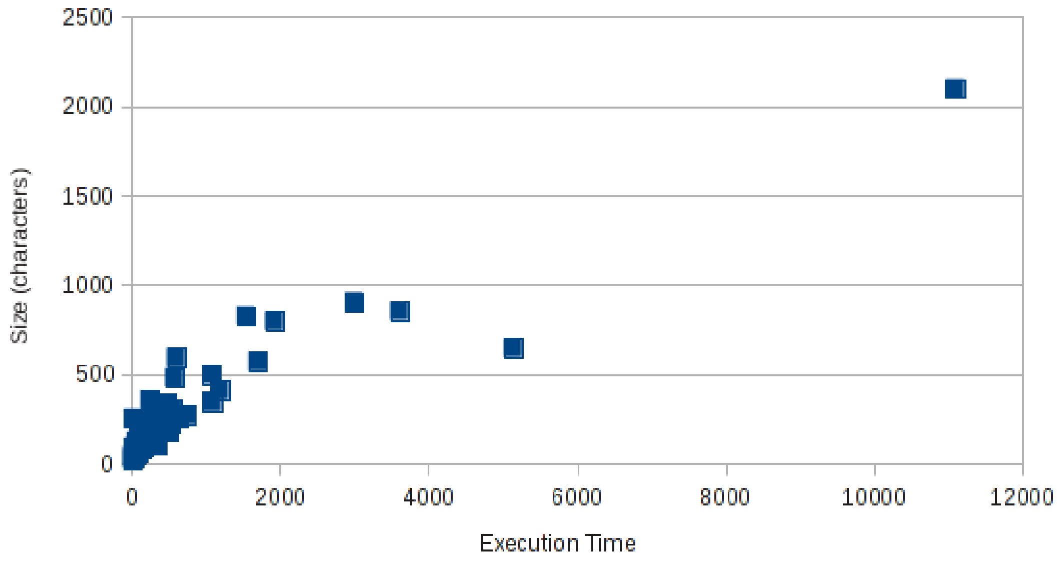


Figure 7.8: Run time on file size

At a first glance on the results I saw that most of the C++ header files had considerably less run-time than the implementation files. This makes sense, because the ConstantConditionCheck and the UninitializedReadCheck won't have any effect there, if the implementations are in the .cpp files, so the ConstMethodCheck will have less work there too. That said, in the graph we can see there's a quite linear relation. The more the file grows, the longer it takes.

The worst case is the analyzer.cpp file that takes 4.5s to run, but has around 5000 characters in it, so it's roughly at the same distribution line, if not a little lower.

All in all, my conclusion here is that it's not cheap to completely analyze a file, because it ranges from 500ms to 1500ms, but it's still feasible in a small amount of time which can also be diminished by reducing the amount of checks being run. Also results show us a too high amount of false positives that should be considered and taken into account properly when creating future checks.

7.6.1 Future

Ironically, whenever we try to leave something as done, we can't but think of what we can do to take it further. On one hand it would be really interesting to investigate on how can we make the run-time faster. There are many possibilities to explore, JIT code generation, making use of the type checking for the run-time, etc.

Also there's a lot of new checks to be written: double writes, API checks in STL or Qt, code style checks and a big list of items that we couldn't finish. Quality Assurance is an interesting field of research and we can use checks as a tool to find out what can we do to improve the overall development experience and help the code and projects in general to be maintainable.

Chapter 8

Planning

It was not easy to plan how this project would evolve in the future, mostly because there was some research involved where the timeline was just an approximation. Here we have the initial schedule:

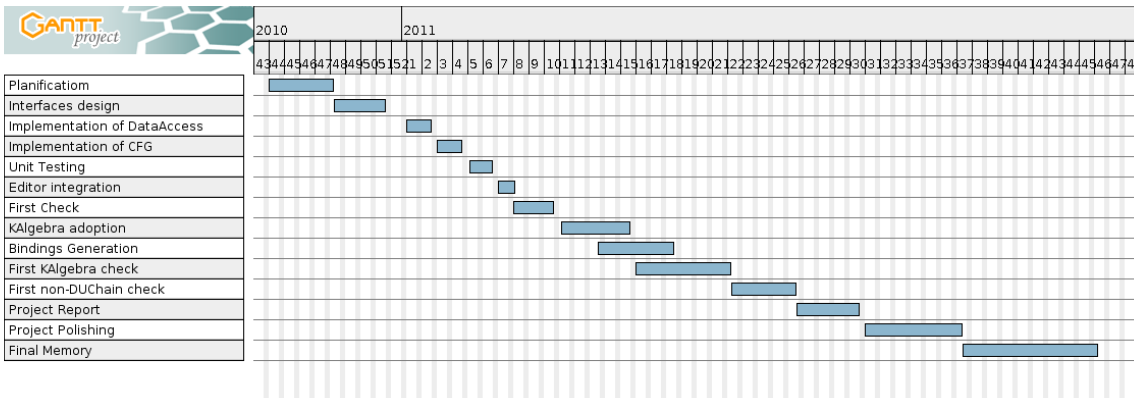


Figure 8.1: Prevision of the dedication

But of course reality is seldom like we expected it to be, here's some representation of what it turned out to be in the end.

As we can see in the graph, first steps followed the plan quite well while, although some steps took a different amount of time than what he had planned, sometimes more, sometimes a little less.

The first big difference that happened was when developing the C++ decorators, where the unit tests development ended up to be an iterative process together with the development itself more than some task that we just do before or after the coding part, which is interesting.

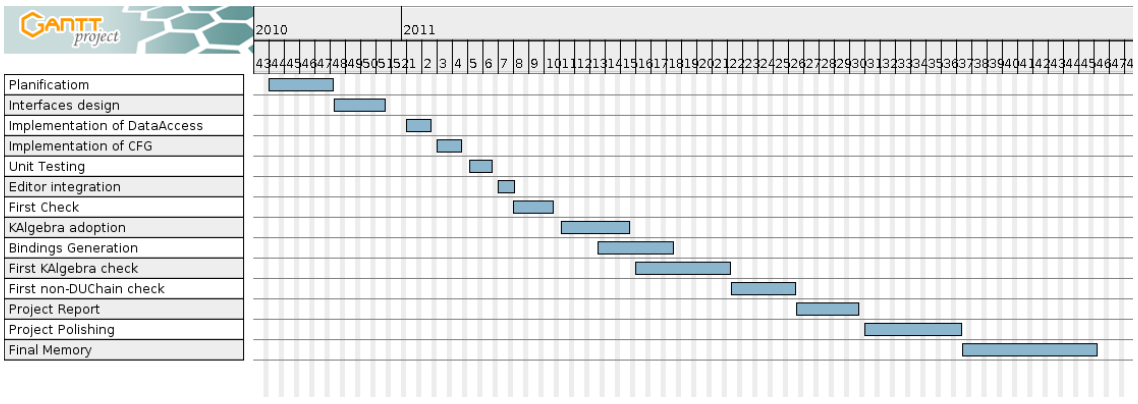


Figure 8.2: Actual dedication

The editor integration part was quite straightforward, we didn't need a lot of infrastructure for that and without any checks it was hard to put it in use, so an iterative approach was taken here too, where the requirements appeared by themselves and didn't need a lot of specific dedication.

Regarding the functional checks development part, here we had a very notable inflection, mostly because it was an area that needed more research than pragmatic work, so some backtracking was needed at some occasions. Also the bindings generation was not really straightforward and we had to go back to it to fix it in some occasions in order to keep working further with the checks.

The documentation part evolved as expected, maybe it had more time assigned that needed in the end, which was good anyway because it left some time for polishing.

The time dedicated to the project was irregular. In some stages of the project I got distracted by different external factors that reduced my dedication, the rest of the time I had the chance to devote my time to the project which was a good way to make sure everything ended as expected. Additionally, life is merciless and sometimes I couldn't dedicate as much time as I'd have wanted to the project for economical reasons, this explains some steps taking a little longer than they could have.

Quantifying the cost of the project wouldn't be very wise. The main goal of this project is to create new technology without any economic expectation or requirement. We could do the math and multiply the hours per hour price but that probably wouldn't be fair because of the research nature of the project.

Chapter 9

Conclusion

In this document, we have discussed in length about static analysis, about what tools we can create to assist the developers and what we have started to create a spoiled piece of software that points out to some places where our program might not work.

As engineers it's our duty not only to provide good solutions but to make sure those are reliable. Let's think about it for a moment, a lot of the code produced nowadays has very long lifetime, we expect it to work for very long time and in some cases there's not even the opportunity to put a fix in place in the future, so if we can make sure that there's some tool that has read *all* our project and told about the issues it knows about. It's not an insurance, but all help is good.

There's another important problem that we have to solve. Software changes in time, so does any project. If the tools we're relying on change, we want to adapt to it as fast as possible, we want to benefit from the tools that we're using as much as possible; either we pay for them or not, and we need tools that make sure that this process is happening like it should. Of course there's also the possibility of having someone dedicated to that end, but it doesn't sound fun either. In the end, in the software development world, if a job can be done by a computer, we should prefer the computer to do it and dedicate ourselves in making a difference in our project, which is a legitimate goal and it's hard enough, all alone.

I don't want to tell any project manager what goals to prosecute or any developer how to work, my main goal is to produce tools that let them focus on where they can be creative and efficient. From my point of view, the human mind is not good at focusing in small details all the time, usually we see the big picture first and then we break it into small pieces to make it possible to put our idea in practice. On that scenario I can see this project fitting really well, being because checks are being run as the developer types or because the developer will get notified by other means like version control system's hooks, bug reports or anything the project decides to work on.

In this project we have created an architecture to create checks. That's where I've put all the emphasis. That's important because, even though we are providing some checks in this project, I'm sure that widening the check community would take us all to a better place to develop and to better software to be produced. There's plenty of space to improve in such a community, wherever it is libraries who want to provide checks to make sure it's used properly or 3rd parties wanting to sell or give out checks for developers rejoice.

I see a bright future in development tooling. In the future, maybe it won't be as important, because better development paradigms might appear, but the present is clear: C is everywhere, so is C++, PHP, Python and Java. Most architectures written these days are using these technologies, even though better ideas appear in the future, we'll have to stick with the old projects around for a while.

Let's provide the best tools for today, let's improve our present, to have a better future.

9.1 Future

Even though all the project has been made public already, there's a lot of work to be done so that it can start to be used for a wide audience.

- Further research and development should be done towards providing checks and making sure the provided ones are useful to the user.
- It would be interesting to provide a window-less version, to make it possible to run it in a remote server without worrying about it.
- Problems solutions could be provided by the checks, there could be some that are easily fixable (if the user wants to).
- Integrating checks to libraries distribution would be interesting, this would make us aware of new checks just by downloading the libraries, easing the process which ensures that the code is correctly checked.
- It would be helpful to have easily integration rules with Version Control Systems where the developer wouldn't be able to commit if some checks produced problems.

- Port to different languages

All in all, there's three main ways this project could expand. From the checks development point of view, where we need people who wants either to investigate about how to produce more accurate and useful checks and also people who wants to adapt it to the different specific technologies where in addition to the C/C++ headers we can provide checks that make sure the library is used like it should.

Secondly, we can also improve the integration of the provided tools in development environments. It can be integrated in different IDE's other than KDevelop, we can get Web UI's, in short: we can provide new interfaces to the infrastructure to better adapt to the different use cases.

Additionally, we can bring support to new languages, that means to provide all the Control Flow Graph, Data Access and DUChain. The usefulness of this work depends largely on how semantically accurate the language gets to be, but experience shows that even on dynamically typed languages it's possible to create a DUChain with quite a lot of data. It's an area that falls out of the scope of this project, so it has not been discussed in depth in this document but it's indeed a very interesting subject to work on. Adding support for languages similar to C++, like Java, shouldn't be very hard in theory.

Appendices

Appendix A

Code Repositories

Directly related to the project repositories:

KDevPlatform clone with the modifications.

`http://gitweb.kde.org/?p=clones/kdevplatform/apol/kdevplatform-pfc.git`

KDevelop clone with the modifications.

`http://gitweb.kde.org/?p=clones/kdevelop/apol/kdevelop-pfc.git`

KDevChecksRunner The plugin responsible for the check invocation.

`http://gitweb.kde.org/?p=scratch/apol/kdevchecksrunner.git`

KDevChecksPack A pack for C++ checks.

`http://gitweb.kde.org/?p=scratch/apol/kdevcheckspack.git`

KDevAnalitzaChecks A plugin that provides the bindings to create Analitza checks

`http://gitweb.kde.org/?p=scratch/apol/kdevanalitzachecks.git`

KDevAnalitzaChecksPack The functional checks described in this document.

`http://gitweb.kde.org/?p=scratch/apol/kdevanalitzacheckspack.git`

A.1 Official Repositories

Used projects and their official repositories:

KDevPlatform :

`http://projects.kde.org/projects/extragear/kdevelop/kdevplatform`

KDevelop :

`http://projects.kde.org/projects/extragear/kdevelop/kdevelop`

KAlgebra :

`http://projects.kde.org/projects/kde/kdeedu/kalgebra`

Analitza :

`http://projects.kde.org/projects/kde/kdeedu/analitza`

In these URL's you'll find some websites that will tell you how to retrieve the code. In case you want to try it out, you'll have to compile it like regular KDevelop. Instructions can be found here[1], of course by using the repositories provided by this project.

[1] `http://techbase.kde.org/KDevelop4/HowToCompile`

Appendix B

Free Software Relations

Since the introduction I've put a lot of emphasis in reminding that, regardless the quality of the project itself, I wanted to make it free software. This project wouldn't have been possible without free software because:

- All the applications we used were free software; from the kernel, Operating System to the development tools like its compiler, the libraries used and assorted tools.
- Also it has been fundamental that the project was based on top of the KDevelop project that offered us the flexible base we needed and a good C++ parser to play with and modify at will. It's very hard to find these possibilities outside open source, and I think it made the project better.

Like in any engineering project, it's not just the work itself on the project that matters but all the projects it's based on. Otherwise I would have been thinking in front of a paper and some wires, which was not the case at all.

I'd also like to describe, as a reference as well as a tribute, the projects that were used in this project; direct or indirectly, in case further information about the development process is needed. These projects are not listed in any particular order.

CMake Used to generate the `Makefile` and make sure the compilation is going to be a simple process. Finds the dependencies and puts it. together to generate a build directory ready to have `make` executed.

Git Used to version our project and share my modifications with the rest of the community.

KDevelop among other things, it helped me while writing the code and easily navigate through the most complex code base.

GCC who turned all this mess of byte strings to different byte strings that happen to be understood by my system. Also, when my code was not correct was kind enough to guide me through dark paths by using the `gdb` tool.

Valgrind, KCachegrind, Massif-Visualizer who came to the rescue when something was wrong, or not, or it could be improved, maybe.

cpptoxml who converted the complex C++ header structures to kids play, or rather, python's play.

Python came to the rescue when I needed a tool to convert XML files into bindings.

GraphViz that helped me by providing graphical representations of some of the extracted data from the code for better understanding.

Qt What everything is built on top of, what shows the pretty pictures, what talks to the Operating System (whichever we're using) and tells what we're looking for.

KDE Provided the hosting for the project, helping in IRC channels, providing bug tracking systems and all the needed infrastructure to make this project something happening. In addition, they paid for my trip to Randa (Switzerland), where I gathered with the rest of the KDevelop team to discuss how to improve it.

Of course, there are many other projects that have been used indirectly. I'd like to use this occasion to thank all the developers and companies that have understood the benefits of developing in the open ecosystem so that we all could use them.

List of Figures

Bibliography

[1] The KDE Community is an international technology team dedicated to creating a free and user-friendly computing experience.
`http://kde.org`

[2] KDevelop is a free, open source, cross platform Integrated Development Environment by the KDE community.
`http://www.kdevelop.org/`.

[3] The English Breakfast Network is a collection of machines that do automated KDE source artifact quality checking. Basically, that means they have a SVN checkout of the entire KDE codebase (including documentation and whatnot) and they run checking tools on that.
`http://ebn.kde.org`

[4] The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.
`http://llvm.org`

[5] The goal of the Clang project is to create a new C, C++, Objective C and Objective C++ front-end for the LLVM compiler.
`http://clang.llvm.org`

[6] The Clang Static Analyzer is source code analysis tool that find bugs in C and Objective-C programs.
`http://clang-analyzer.llvm.org/`

[7] KDevPlatform DUChain Documentation, available at
`http://api.kde.org/extragear-api/kdevelop-apidocs/`.

[8] Coverity Static Analysis is a static code analysis tool for C, C++, C# and Java source code. It is a commercial product which originated as the Stanford Checker, which used

abstract interpretation to identify defects in source code.
`http://www.coverity.com/`

[9] The KDE Education project
`http://edu.kde.org`.

[10] Graphviz is open source graph visualization software.
`http://www.graphviz.org/`.

[11] Python is a programming language that lets you work more quickly and integrate your systems more effectively.
`http://www.python.org/`.