Managing consistency between dependent objects

Jeroen Bouwmans

MSc Thesis
INF/SCR-09-100

August 25, 2010

Center for Software Technology
Department of Information and Computing Sciences
Utrecht University

Supervisor:
prof. dr. S.D. Swierstra

Abstract

During the development of software many products are produced. In addition to the source code of the software, all kinds of by-products arise, by formulating requirements, creating models and diagrams, setting up tests, generating output by compiling, interpreting and running code, writing different types of documentation, etc. To ensure the quality of these objects, it is necessary to keep them consistent. When this is not the case anymore, inconsistencies between the objects occur. Preventing and resolving inconsistencies is a time consuming and cumbersome process. We therefore research how users can manage the consistency between their dependent objects.

We have investigated how dependencies between objects can be detected or can be created manually by users, and how inconsistencies between objects can be detected and resolved. For this, we have given clear definitions of the concepts that are involved in managing consistency between dependent objects. We have focused on software development products, but have kept the definitions and properties as abstract as possible, to be able to apply them to other fields as well.

We have implementd a prototype of a tool—the Consistency Management Assistant—in the programming language Haskell, using our definitions and properties, to support users as much as possible in keeping their products consistent. The tool can be extended to let users use their own types of objects, and is used completely separated from the existing tools users currently use, to be as unintrusive as possible.

Contents

1 Introduction

1.1 Let's start with an example

Imagine you wrote a Haskell program that prints the text "Hello World!" to the command line:

> **module** *Main* **where**
> *main* = *putStrLn* `"Hello World!"`

It works great, so you decide to share it with other people. To help them understand and run your program, you add a piece of documentation:

> **hello**
> Description:
> Says hello to the world via the command line.
> Usage:
> *./hello*

After a while a user asks you to extend the program. Instead of saying hello to the world, he wants to say hello to the entire universe. A nice improvement:

> *main* = *putStrLn* `"Hello Universe!"`

You publish version 2.0 of your program, together with the documentation, which seems to still be fine. But then one day a new user of your program sends in a complaint, because he expected the program to say hello to the world, while it said hello to the entire universe instead. It turns out that you didn't notice that your documentation still states that your program says hello to the world.

One might think that putting in a little extra effort would prevent developers from creating these kinds of inconsistencies. But actually situations like this example abound. And things would have gotten much worse if the program had consisted of tons of lines of code and the documentation of hundreds of pages...

1.2 Problem description

During the development of software many products are produced. In addition to the source code of the software, all kinds of by-products arise, by formulating requirements, creating models and diagrams, setting up tests, generating output by compiling, interpreting and running code, writing different types of documentation, etc.

These products rarely live completely on their own. A product is almost always related to other products. For example, an architecture implementation relies on its model, source code files are being transformed into executable code, and documentation describes and includes source code. To ensure the quality of the products, it is necessary to keep them consistent. When this is not the case anymore, inconsistencies between the products occur. It can even be the case that a product is inconsistent on its own, when parts within one product are inconsistent with each other.

Like demonstrated in the example, preventing and resolving inconsistencies is a time consuming and cumbersome process. In the case of keeping documentation consistent with the source code, it is necessary to check the documentation text and the source code for inconsistencies whenever changes to the code have been made. Doing this manually takes a lot of time and is also not very reliable.

Other types of products suffer from the same problems. For example, when the requirements of a software product change, the source code must be changed too to fulfil them. When the architecture of a system is modified, the models must be adapted to match the new situation. And when changes to functions have been made, the corresponding tests must be updated to test the new behaviour of these functions.

The main research question of this thesis project is: How can we support users in managing the consistency between their dependent objects?

1.3 Criteria

We have formulated a number of criteria for this thesis project. During the project, we take these criteria into account. In chapter 9 we determine if they have been met.

1.3.1 Definitions and properties

We give clear definitions of the concepts that are involved in managing consistency between dependent objects, like objects and dependencies and their properties. We focus on software development products, but the definitions and properties should be applicable to other fields as well. Therefore we keep them as abstract as possible.

1.3.2 Tool prototype

During this project we develop a prototype of a tool—called the Consistency Management Assistant—to support users as much as possible in keeping their products consistent. Users must be able to manage the dependencies between their products in our tool. The tool must be able to detect as many dependencies and inconsistencies, and resolve as many inconsistencies as possible. The Consistency Management Assistant initially supports only a few types of products and dependencies, so it has to be made extensible, letting users create functionality to use the tool for their own types of products and the dependencies that arise between them. The tool has to be as little intrusive as possible. This means, for example, that we want to minimize the impact on the products, and let users keep using their existing tools.

1.3.3 Using Haskell

We want to investigate the advantages and disadvantages of using the functional programming language Haskell for our tool, and for this kind of problems in general.

1.4 Previous work

There has been quite some research on how to keep software development products—also called *artefacts*—consistent. At multiple places in this thesis we refer to this work.

Donald Knuth's concept of literate programming [Knuth, 1984] can be seen as an early approach to keep products consistent with their documentation. By tying together source code and documentation and structuring programs as an interconnected web, inconsistencies between the code and the documentation are directly visible and can easily be resolved. Aguiar and David [2005] have used ideas of literate programming to create a wiki that weaves different kinds of products into a single document, preserving its consistency. However, because of the fundamental differences with current widely-used programming languages, literate programming has until now not become popular. We therefore don't use this approach.

Many recent studies focus on creating and maintaining traceability links between products, using textual analysis. For example, some use natural text analysis [Fantechi and Spinicci, 2005, Antoniol et al., 2002] to find similarities between free text and source code, or use Latent Semantic Indexing (LSI) [Marcus and Maletic, 2003, Poshyvanyk and Marcus, 2007, Wang et al., 2009], a more sophisticated technique that identifies the meaning of parts of free text and the relationships between them. Grechanik et al. [2007] and Grundy et al. [1998] make more use of the structure of the text. McMillan et al. [2009] combines both textual and structural analysis to find traceability links. The methods using textual and structural analysis can be used in the Consistency Management Assistant to detect dependencies and resolve inconsistencies.

Other research is about creating constraints about the products written in a domain-specific language, and checking them for validity whenever changes to the products have been made. Nentwich et al. [2001] for instance use constraints to check specification models. Sunetnanta and Finkelstein [2001] and Goedicke et al. [1998] use graphs to validate specification and architecture models, and the approach of Denney and Fischer [2009] take textbook formulas and algorithms as input to identify and verify mathematical concepts in the code. Egyed [2010] claims that using his approach any language can be used to express the constraints. Nuseibeh et al. [2001] combine tools that check for consistency by applying rules, each for a different phase of software development. These methods can be used in the Consistency Management Assistant to detect inconsistencies.

Lastly there have been studies about synchronizing changes in products that rely on each other, called *round-trip engineering*. Foster et al. [2007] propose an approach to synchronize changes in abstract and concrete views of tree-structured data. [Van Paesschen et al., 2005] do the same, but focus on data models and object-oriented implementations. In section 2.2 we describe how round-trip engineering can be used in our tool.

1.5 Outline of this thesis

In chapter 2 we give definitions for objects and dependencies, and define properties that can be used to define common causes of dependencies, and to specify and implement our tool. We describe a number of causes in chapter 3. In chapter 4 we describe how dependencies can be detected automatically and created manually, and how to manage them. In chapter 5 we describe how changes and inconsistencies in objects can be detected, and how inconsistencies can be resolved. For the implementation of the Consistency Management Assistant, we first specify the functionality in chapter 6. In chapter 7 we give details about the implementation of the tool and in chapter 8 we evaluate the tool. In chapter 9 we give a conclusion and make suggestions for future research.

2 Objects and dependencies

Many different kinds of software development products are being used in institutions and companies. There exist also many kinds of relations between these products. We would like the Consistency Management Assistant to support as many kinds of products and relations as possible. However, it is obviously unfeasible to identify all products that are being used and all relations that occur in institutions and companies. We therefore create a generic model that can be used to express many kinds of products and relations, using their properties. Users are then able to use our tool for their own kinds of products.

2.1 Collecting examples

As a starting point for our model, we have created an overview of products commonly used at the Department of Computing Sciences of Utrecht University, and the relations between these products. This overview can be seen in figure 1. The overview covers many similar situations at other institutions and companies where different kinds of products and relations are being used. We describe the products and relations that are displayed in the figure.

2.1.1 Specifications

Before implementing code, often specifications are created. These specifications can for example be in the form of written text, but also graphical models can be created. There is a relation between the code and the specifications, because the code must implement the functionality that is described in the specifications.

2.1.2 Transformations

During the development of software, all kinds of transformations are being executed. In most cases this involves the compilation of source code into an executable program using a compiler, like the Glasgow Haskell Compiler (GHC) [GHC Team, 2010a] or the Utrecht Haskell Compiler (UHC) [Dijkstra et al., 2008], or into binary code, like when using pdfTEX [Thành, 1998] to generate PDF files from TEX source code. Also preprocessors perform transformations on source code before it can be used by other transformation programs. Examples are Shuffle [Dijkstra, 2009], a tool for manipulating source fragments, and lhs2TEX [Hinze and Löh, 2009], which generates LATEX code from literate Haskell sources. Other transformation programs are Haddock [Marlow, 2010], for the creation of API documentation from inline annotations in Haskell code, and Pandoc [MacFarlane, 2010], for conversion between markup formats.

2.1.3 Data sources

Many software applications use data from external sources, like files and directories on the file system, or data from databases. When this is the case, there is a relation between the application and the data source.

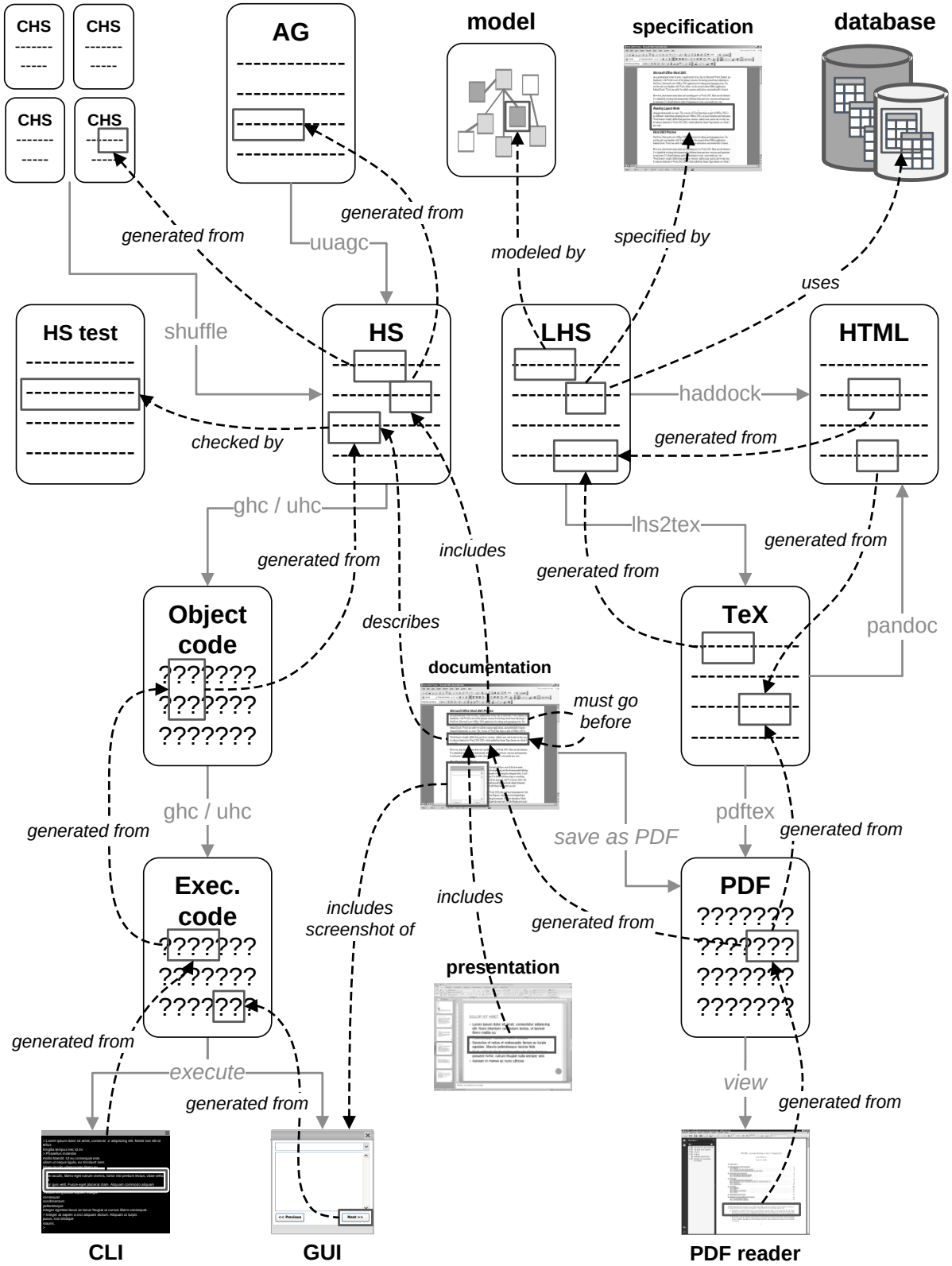


Figure 1: Products of software development commonly used at the Department of Computing Sciences of Utrecht University, and their relations, denoted by striped arrows.

2.1.4 Validation and verification

There are relations between source code and validation or verification methods. An example included in the figure is a QuickCheck [Claessen and Hughes, 2000] test, which checks Haskell source code for validity.

2.1.5 Documentation

Documentation of software can be created in several ways. End-user documentation, for example, is usually put in separate documents using a rich text processor, whereas technical documentation is often inlined in the source code. Other forms of documents that can be seen as documentation are presentations, beamer slides and lecture notes. Next to relations between the software and the documentation, there are also relations between documentation documents, for example when a presentation uses parts of a documentation document.

2.1.6 Executing code and viewing files

When executing programs, output is generated. When the program is executed on a command line interface (CLI), the output is plain text. In other cases, a graphical user interface (GUI) is generated. The text or the graphical interface is related to the binary code that represents the text or the interface (which is in turn related to the source code). Similar relations are involved when viewing binary files, like PDF files. The image that is displayed on the screen has relations with the binary code of the PDF file, from which the image is generated.

We use these examples to give generic definitions and describe properties that are relevant for keeping software development products consistent.

2.2 Definitions

The term *product*, that we have used until now, is rather vague. A product might be defined as a *larger entity*, but then there remains a grey area for which it is not sure if it is a product. For example, is one Haskell file a product? It depends: if the file represents a complete program, then we might say it is, but if it is part of a library consisting of many Haskell files, we would say it isn't (in this case we could call the library the product). Because of the problems with this definition we will not use the term *product* anymore. Instead we will use the term *object*, which covers both larger and smaller entities and therefore eliminates the grey area.

Definition 1: Object, Format
An *object* is a part of a product, or a product in its entirety. Objects are encoded in a certain *format*.

Examples of objects are the complete source code of a project, a directory, a source file, a Haskell function, a specification, a documentation document, an image in a document, a database, and a database table. Examples of object formats are Haskell, TEX, PDF and JPEG. Also directories and files are object formats (independent of their contents).

For all the relations in figure 1 there is the possibility for inconsistencies to occur when changes in the objects are made. We can therefore say that these relations are actually not just ordinary relations, but more specifically *dependencies* between the objects.

Before we can give a definition of dependencies, we must determine how many objects are involved in a dependency. In figure 1 all dependencies are created between two objects. It can, however, be the case that more than two objects are involved. This makes it much harder to formulate the definition and properties of dependencies. We therefore define a dependency to be from one object on another object. Creating dependencies between more than two objects can be expressed using multiple dependencies.

It is also necessary to define what happens when two objects become inconsistent. When both objects become inconsistent, it is hard to determine which object should be changed to resolve the inconsistenty. We therefore require that only one of the objects "speaks the truth", which means that in that case only the other object can become inconsistent.

Definition 2: Dependency, Cause, Dependant, Authority
Given two distinct objects D and A, a *dependency* from D on A arises from a certain *cause* when D might become inconsistent with A as the result of changing, creating or deleting either D or A. D is called the *dependant* and A is called the *authority*.

An example of a dependency is the dependency between TEX source code and the result of compiling the code, for example a PDF file. After changing the source code, there is a high probability that the previous compilation result doesn't match the described behaviour of the code anymore. The compilation result then becomes inconsistent with the code. In this case, the TEX source code is the authority and the PDF compilation result is the dependant.

A dependency is by definition unidirectional, because the relation goes in one direction: the dependant is dependent on the authority. It can be the case, however, that both objects are equally authoritative, such that the authority can also become inconsistent when changes to the dependant have been made. The relation between the objects is then a bidirectional (circular) dependency. A bidirectional dependency can be expressed using two dependencies: one from the first object to the second object, and one from the second object to the first object. An example of bidirectionality is when two documentation documents contain the same text fragment, and both documents must be updated when the other document changes. Methods for round-trip engineering, like described in the previous work in chapter 1, can be used to resolve inconsistencies in objects with a bidirectional dependency. Special care has to be taken to prevent our tool from getting in a loop when resolving these inconsistencies.

Dependencies can be composed when the dependant of one dependency is the authority of another dependency, forming an *indirect dependency* from the dependant of the first dependency on the authority of the second dependency. An example is shown in figure 2. Indirect dependencies exist only virtually, i.e. they are not stored as a real dependency.

In the remainder of this chapter we describe the properties of objects and dependencies. Later we use these properties to define common causes of dependencies and to specify and implement the Consistency Management Assistant.

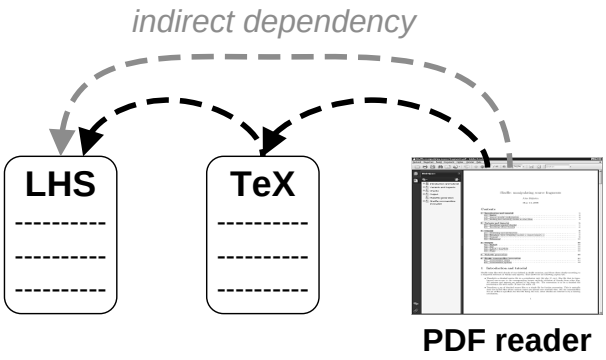


Figure 2: When a dependency from a PDF file on a TEX file, and a dependency from a TEX file on a LHS file exist, an indirect dependency from the PDF file on the LHS file can be formed by composing the dependencies.

2.3 Object properties

The following properties can be defined for the formats of objects. *When a property is defined for an object format, all objects of this format have this property.*

2.3.1 Precision types

Object formats can have multiple types of precision. The precision is a way to point at a specific location in an object. This is useful for situations in which a dependency is defined on a more specific part of an object than can be expressed as a path in the object tree. For example, if a dependency on a TEX paragraph is not on the complete paragraph, but on just one or more specific lines, the precision is a line range. The start and the end of the range can be expressed as line numbers relative to the start of the paragraph. Another type of precision is a coordinate, which can for instance be used to point at a specific location on a page in a PDF file.

In the Consistency Management Assistant, when types of precision have been specified for object formats, users can for example point at specific locations in these objects by clicking on a location.

2.3.2 Parseability

Objects of a certain format are parseable when it is possible to analyse the structure of the object. A parser can then create a hierarchy of this structure, in the form of a tree of objects. For example, a directory is parseable because a tree of files and directories contained by the directory can be created, and a Haskell file is parseable because a tree of functions can be created. The objects in the hierarchy do not necessarily have to be of the same format as the object that has been parsed. For instance, an object tree created from a TEX file can contain not only TEX objects, but also Haskell code fragments. For parseable objects a parser has to be available that can turn the source code (usually in the form of a string) into an object of the regarding object format.

2.3.3 File extensions

For each object format a number of file extensions can be defined. These extensions are only applicable to objects that can be encoded in files, and can be used to determine the format of the object in a file that has been opened in the Consistency Management Assistant. For example, for the TEX format the file extension `.tex` is defined, to indicate that files with this extension are encoded in the TEX format.

2.3.4 Locatability

The user has to be able to open objects in the Consistency Management Assistant to manage their dependencies and inconsistencies. For the formats of objects that can be opened, it must be specified how it can be located. This in contrast to objects that can only be part of other objects, which don't have to be opened directly in our tool. A directory or file, for instance, can be located by typing in a file path or by browsing through the file system. A database can be located by connecting to a server with a username and a password.

It could be specified that the location of objects is always in the form of an URL. Objects on the file system, like directories and files, could then be located by a URL starting with `file://`, and databases for example through a Java Database Connectivity (JDBC) URL, like `jdbc:mysql://host/database`, or a Database Source Name (DSN), like `mysql:host=localhost;dbname=test`. However, because of the possibility that the location of some object formats cannot be expressed as an URL, we keep the concept of locatability more general.

2.3.5 Viewability

An object of a certain format is viewable when it can be viewed in the tool. TEX and PDF files are for instance viewable, because the contents can be displayed to the user. Viewable objects can be viewed internally in the Consistency Management Assistant.

2.3.6 Markability

Objects of a certain format are markable when it is possible to display the object in the Consistency Management Assistant, highlighting a specific location in it using one of the precision types that have been specified for the object. PDF objects are for example markable, because it is possible to mark a coordinate on a page in a PDF file.

2.3.7 Pointability

Pointable objects are objects in which a specific location can be pointed by the user in the Consistency Management Assistant using one of the precision types that have been specified for the object. For instance, TEX objects are pointable because the user can point at a specific line or line range in the object.

2.3.8 Editability

Objects of a certain format are editable when there is an internal editor for the Consistency Management Assistant available to make changes to the object. Most ASCII files are editable, while object code and executable code generated by a compiler is usually not editable.

2.4 Dependency properties

The following properties can be defined for pairs of object formats. *When a property is defined for two object formats, all dependencies with a dependant of the first format and an authority of the second format have this property.*

2.4.1 Creation order

The creation order denotes which of the objects is created first: the dependant or the authority. This indicates which of the objects causes the dependency to come into existence, namely the other object. In most of the cases, the authority is created first, and then the dependant becomes dependent on the authority when it is created. This is for example the case when transforming a TEX file into a PDF file. An example situation in which the dependency is created first, is when a QuickCheck test for a Haskell function is created. The test, which is in most cases created after the implementation of the function, becomes a sort of specification for the function and therefore becomes the authority.

2.4.2 Transformability

The transformability indicates if an authority of a certain format can be transformed into a dependant of another format. Transformable dependencies are for example those that are written down in `Makefile`s [Feldman, 1979]. Often transformable dependencies arise by the compilation of source code, in which case the source code is the authority and the compilation result the dependant. An example is the transformation of TEX into PDF. Another situation in which a transformable dependency arises, is when transformation is applied at the inclusion of an object in another object. When for example including source code in a documentation document, formatting or syntax highlighting of the code can be applied.

2.4.3 Splittability

A dependency is splittable if it is generally possible to split it into one or more dependencies on a lower level in the Consistency Management Assistant. When for example a dependency exists from one Haskell module on another module, it can be split into multiple dependencies from functions in the first module on functions in the second module (see figure 3). A dependency can only be splittable if either the dependant or the authority is pointable, because otherwise there is no lower level to create dependencies on. Of course dependencies on the lowest level can never be split.

The opposite of splittability, joinability, is not a property. This is because if two dependencies on the same lower level exist, it is always possible to create a dependency

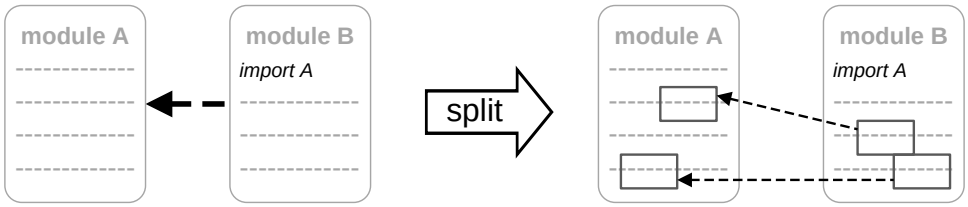


Figure 3: A dependency from Haskell module B on module A is splittable, because it can be split into multiple dependencies from the functions of module B on the functions of module A.

on a higher level, independent of the formats of the dependant and the authority. In the most extreme case it is always possible to create a dependency from the product containing the dependant to the product containing the authority.

2.4.4 Orderability

If there are cases in which an order between the objects of a dependency can be defined, the dependency is orderable. An example is two TEX objects in the same document, in which case one of the objects is before the other. For the same objects formats there are most likely many cases in which no order can be defined, for example if two TEX objects are part of different files.

2.4.5 Previewability

A dependency can be previewed in the Consistency Management Assistant when the dependency is transformable and the dependant of the dependency is viewable. Looking at a preview is a quick way to see the result of the transformation. Previewing can often be done faster than applying the transformation to the complete authority. For instance, rendering a preview of an image can be made faster by lowering the quality, and rendering a preview of a PDF file can be sped up by just creating the page that is currently being worked on.

2.4.6 Detectability

A dependency is detectable when it is possible to detect the dependency programmatically in the Consistency Management Assistant. This is be possible when the dependencies are stored somewhere, or when the dependant and the authority contain similarities. More about the detection of dependencies can be read in chapter 4.

2.4.7 Checkability

A dependency is checkable when it is possible to check for inconsistencies programmatically in the Consistency Management Assistant, for example by comparing objects on contents, modification date or by applying an even smarter technique. Chapter 5 elaborates

on this. When it is not possible to automatically detect inconsistencies, a dependency is not checkable.

2.4.8 Resolvability

Inconsistencies arisen in dependencies that are resolvable can be resolved programmatically in the Consistency Management Assistant. Sometimes it is possible to resolve an inconsistency completely without user intervention, for example by reapplying a transformation, in other cases the resolving can be supported programmatically but some user intervention is necessary. A dependency has no resolvability if it is not possible to resolve an inconsistency programmatically at all.

Resolvability can be different for inconsistencies resulting from changes to the authority and inconsistencies resulting from changes to the dependant. For instance, while inconsistencies between source code and the compilation result when the source code has changed can be easily resolved by reapplying the transformation, it can be very difficult to resolve the inconsistency when the compilation result has changed while the source code has not been changed.

2.4.9 Recoverability

When objects cannot be found anymore, for example because they have been moved or have been deleted, it is necessary to recover them. The recoverability denotes if this can be done programmatically, without or with some user intervention.

3 Common dependency causes

Dependencies can arise from different causes. Using the definitions and the properties from the previous chapter, we give definitions for four common dependency causes: transforming an object, describing an object, using an object in another object, and creating objects in a certain order. These causes can be used in most situations, but it is likely that there are more causes leading to dependencies. Other causes can be easily defined in a similar way.

3.1 Transformation

When code is transformed from one programming language into another, the cause of the arising dependency is a *transformation*. More in general, a transformation dependency comes into existence when a source object (the input) is transformed into a target object (the output). The target object is dependent on the source object, so the target object is the dependant of the dependency, and the source object the authority. A transformation dependency can, obviously, only arise between two objects when the source objects is transformable into the target object.

Examples of transformation dependencies are the dependency from the resulting code on an attribute grammar after preprocessing, the dependency from a Haskell program on the source code, and the dependency from a PDF file on the corresponding TEX file. A transformation dependency does not only come into existence when transforming code, but for instance also when transforming images.

Transformation dependencies are often splittable, because next to the dependency between the objects, in many cases also multiple dependencies between parts of the objects on a lower level arise. For example, when transforming a TEX file into a PDF file, not only a dependency from the PDF file on the TEX file arise, but also multiple dependencies from coordinates on pages in the PDF file on the corresponding lines in the TEX file (see figure 4).

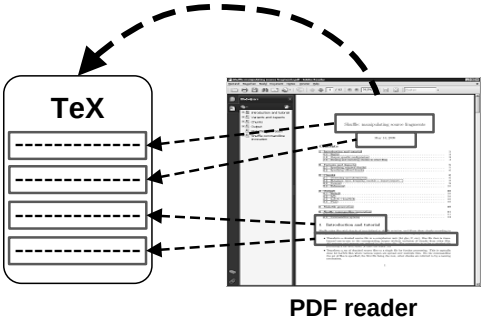


Figure 4: A transformation dependency from a PDF file on a TEX file, and multiple transformation dependencies on a lower level from coordinates on pages on the corresponding lines.

3.2 Description

Many of the by-products that are created during software development describe one or more other products. An example is the end-user manual for a software product, explaining how to use all the features of the software. When an object is created that describes other objects, a *description* dependency comes into existence. The object that is being described does not necessarily have to exist—sometimes a description of a product is already there, but the described product has yet to be created. We make a distinction between these two kinds of description dependencies.

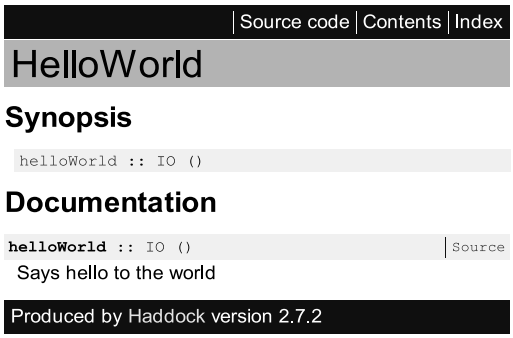
3.2.1 Documentation

A *documentation* dependency is a description dependency that arises when an object describes another object that already exists. More precise, the object that describes the other product is dependent on it and therefore is the dependant. The product that is being described is the authority.

Examples of documentation dependencies are technical documentation, which describes source code, algorithms, interfaces and APIs, and end-user documentation, describing the output of executing code, like command-line interfaces and graphical user interfaces. The documentation can be inlined in the source code, for example when creating Haddock documentation for Haskell code (see figure 5).

```
module HelloWorld (helloWorld) where
  -- | Says hello to the world
helloWorld :: IO ()
helloWorld = putStrLn "Hello World!"
```

(a) A Haddock annotation for a Haskell function



(b) Generated documentation for the function

Figure 5: A documentation dependency relation from an inline Haddock annotation on a Haskell function is made by putting the annotation directly above the function declaration.

3.2.2 Specification

The opposite of the documentation dependency is the *specification* dependency, in which the object being described is usually created after the object that describes it. The important difference with the documentation dependency is that in a specification dependency, the product that is being described is the dependant, and the object describing the dependant is the authority.

Examples of specification dependencies are requirements documents, specifications and architecture or design models, which all describe software before it is implemented.

Like stated before, the dependant of a specification dependency is most of the times created after the authority. An exception is the creation of a test that checks the validity of source code. In this case the creation order is usually opposite. Although a test is created after the source code has been implemented, the test actually becomes a sort of specification for the source code: if the behaviour of the source code does not satisfy the test, the code should be changed, and not the test. In this case, the source code is dependent on the test, so the test becomes the authority and the source code becomes the dependant. An example of this kind of specification dependency is the creation of QuickCheck tests for Haskell functions.

3.3 Usage

When an object makes use of another object, the cause of the dependency is *usage*. The object that uses the other object is the dependant, because it is dependent on the object being used, which is the authority. The authority can be part of project of the dependant, but can also come from an external resource. Examples of usage dependencies are importing other Haskell modules in a module, using an external library in a project, and communicating with a database.

3.3.1 Inclusion

A special kind of the usage dependency is the *inclusion* dependency. In this case, the object that is being used is literally included. Examples are including a TEX file in another TEX file and including a screenshot of a graphical user interface in a documentation document. Sometimes an included file is being transformed before being included. An example is formatting or syntax highlighting an included code fragment in documentation.

3.4 Ordering

An *ordering* dependency arises when an object is created that is dependent on the position of another object. The object that should go after another object is the dependant, the object that should go first is the authority. An ordering dependency can only be created between objects if the two formats are orderable.

An example of an ordering dependency is an ordering between two fragments in a documentation document that comes into existence if it is necessary to read the first fragment before the second fragment can be understood.

4 Dependency detection and managing

In this chapter we describe how dependencies can be detected automatically or be created manually, and how to update and delete dependencies.

4.1 Proactive vs. reactive

The way the Consistency Management Assistant interacts with the objects has a high impact on the detection and managing of dependencies. Therefore we have to investigate this at this point and make a decision. There are roughly two possibilities: to track changes while the objects are edited (a proactive approach), or to scan the objects for changes periodically (a reactive approach). Using a proactive approach, every single change in an object is tracked by the tool. This allows for determining the consequences for the dependencies very precisely, and makes it possible to warn users when changes to objects lead to inconsistencies before they save the changes. Using a reactive approach, the tool does not know about the edit actions that are performed by the user, but can only calculate the differences between the previous object contents and the current contents during the periodic scan. These differences might become large, which makes determining the consequences for the dependencies less precise. Also changes to objects will only be detected after they have been saved, which makes it impossible to warn users for inconsistencies beforehand.

An example demonstrating the differences between a proactive and a reactive aproach is when a user rearranges text fragments in a file. A tool taking a proactive approach tracks every cut and paste action, and checks or updates the dependencies accordingly. A tool taking a reactive approach is only able to detect the changes after the file has been saved, and finds fragments that are completely different from the fragments found during the previous scan. The tool is then not able to check and update the dependencies.

Unfortunately, also proactive approaches have a big disadvantage: the editors that are being used to change the objects, must be able to track every single change. One possibility to take care of this, is to write a script for tracking these changes (sometimes called an *edit script*) for each editor that users currently use. However, this requires a lot of effort, and in some editors it will not be possible to integrate such a script at all. Another possibility is to force users to use a specialized editor, like an integrated development environment (IDE), offering support for all the formats of the objects they use. However, it takes too much effort to implement the features of existing editors before users are willing to switch. Reactive approaches, on the other hand, don't have this disadvantage, because users are still able to use their own editors, and there is no need for creating scripts to track changes.

We expect that the disadvantages of a reactive approach can more easily be overcome than the disadvantage of a proactive approach. For this reason we have decided to take a reactive approach for the Consistency Management Assistant.

4.2 Automatic detection

To be able to detect dependencies between objects automatically, it is necessary to determine the format of these objects. For example, when searching for dependencies involving

a certain file, the file extension can be examined or the format can be determined by analysing the contents of the file.

According to the object format, certain types of dependencies can be searched for. For instance, when the format of an object is TEX, it can be checked if there are dependencies from a PDF object on that TEX object, by searching for a file that contains relations between the objects, like files generated by the LATEX packages *pdfsync* [Laurens, 2004a] and *SyncTeX* [Laurens, 2004b]. Another example is searching text documents for the usage of fragments of the TEX object.

When the object is parseable, the corresponding object tree that is created by the parser can be traversed to search for more dependencies on a lower level.

4.2.1 Using stored relations

Sometimes the relations between objects are already stored somewhere. This is mostly the case for objects that have been transformed from other objects. It is relatively easy to translate these relations into dependencies.

In some cases relations between objects are stored in one of the objects. Examples are results of transformations performed by the Utrecht University Attribute Grammar Compiler (UUAGC) [Baars et al., 2009] and the preprocessor lhs2TeX. When transforming attribute grammars or literate Haskell code into Haskell code, references to corresponding line numbers in the original source code can be added to the output. For this so-called `LINE` pragmas [GHC Team, 2010b] are being used, compiler instructions in Haskell especially made for this purpose (see figure 6a).

In other cases the relations are stored separately, for example in a file or a database. This is for instance the case when using *pdfsync* or *SyncTeX* to generate the relations between lines in the TEX file that is being transformed to coordinates on pages in the PDF file that is being generated. During the transformation these relations are stored a file with the same filename as the source file, but with a different extension, namely `.pdfsync` or `.synctex` (see figure 6b). This file can easily be parsed to generate dependencies between the lines and the coordinates.

Also in other cases the file extension can be used to detect dependencies between files. For instance, when both a file with the extension `.tex` and a file with the extension `.pdf` exist that have exactly the same name, it is most likely that the PDF file is a transformation of the TEX file. Another example is the relation between files with the extensions `.hs` and `.hs-boot`.

4.2.2 Interpreting source code

A common construct in programming languages is the usage or inclusion of other source files in the code. When a programming language supports this, the source code can be (partly) interpreted to detect these dependencies. For instance, in a TEX file, the inclusion of other TEX files with the `\input` or `\include` statements might be found. In a Haskell file the same can be done for the importing of other modules with `import` declarations. When interpreting source code more detailed, dependencies on a lower level can be found, like the usage of Haskell functions by other functions.

```
{-# LINE 2 "./DeclBlocks.ag" #-}
import Code (Decl, Expr)
 {-# LINE 51 "../src-derived/GenerateCode.hs" #-}
 {-# LINE 101 "GenerateCode.ag" #-}
  -- remove possible @v references in the types of a data type.
cleanupArg :: String → String
cleanupArg s
   = case idEvalType (SimpleType s) of
     SimpleType s' → s'
 {-# LINE 59 "../src-derived/GenerateCode.hs" #-}
 {-# LINE 115 "GenerateCode.ag" #-}
```

(a) A fragment of Haskell code generated by the UUAGC, with inline references to lines in the original attribute grammar.

```
thesis
version 1
l 0 296
l 1 296
l 2 296
⋮
s 1
⋮
p 1 8360485 43954993
p 7 15293543 42775345
p 9 10905346 40874800
```

(b) Fragments of a file generated by *pdfsync*, with relations between lines in the source code and locations in the PDF file.

Figure 6: Examples of relations that are stored inline and separately

Another example of the detection of dependencies by interpreting source code, is the detection of dependencies between source code and inline documentation. When the programmer adds a description of a function to the source code, a dependency from the description on the surrounding code arises. Documentation generators, like Haddock for Haskell and Javadoc [Sun Microsystems, Inc., 2002] for Java, are already able to detect such dependencies and generate API documentation. An example is shown in figure 5.

4.2.3 Textual and structural analysis

When no stored relations between objects exist and it is also not possible to interpret source code, the last possibility to detect dependencies automatically is to perform textual or structural analysis on the objects. In chapter 1 previous work about methods for textual and structural analysis is described. These methods can be used to compare the text and structure of objects to find similarities, which can indicate dependencies between the objects. Of course, these methods are less reliable, because there is a chance that dependencies cannot be detected and, maybe worse, false positives can arise.

For instance, when source code fragments have been copied and pasted into documentation documents, there might be a chance that they can be found when scanning the documentation document and the source code for similarities, but as there might have been made small changes to the source code when it was included in the documentation document, like adding newlines, syntax highlighting or removing irrelevant parts, it might also very well be that it cannot be found.

For some specific cases, textual analysis might actually work very well. An example is the detection of dependencies between the input and output of Shuffle, which combines chunks of text into larger units. Because the input files are literally included in the output, it is certain that the contents of input files can be found when scanning the output to create the dependencies between them.

4.3 Creating dependencies manually

In several cases, dependencies cannot be detected automatically. For instance, when relations between objects are not stored and the objects have little in common, it is not possible to detect the dependencies automatically. It might also be that automatic detection has not been implemented for the regarding object formats. In these cases it is possible to create dependencies manually. The user has to pick the dependent objects, indicating what the cause of the dependency is and which of the objects is the dependant and which is the authority.

To support users in remembering why they have created their manually created dependencies, and to allow them to record other details about them, we make it possible to add annotations to dependencies.

The dependencies should be stored somewhere, so the user doesn't have to create them again each time he uses our tool.

4.4 Updating dependencies

For both manually created as automatically detected dependencies, the cause, the dependant and the authority can be changed by hand. It is also possible to delete them and to add, edit and delete annotations.

Automatically detected dependencies can also be updated by re-executing the detection process. To prevent automatically detected dependencies that have been changed or deleted by hand from being restored during a re-execution, the original values of the dependencies will be saved in the background, to be able to ignore them during the re-execution.

During the re-execution, automatically detected dependencies that no longer exist will be deleted.

4.5 Broken dependencies

Because of the reactive way the Consistency Management Assistant interacts with the objects, it is impossible to prevent users from making changes to the objects of a dependency that lead to a situation in which the objects cannot be found anymore. For example, when the user deletes an object or changes it heavily, the tool might not be able to locate it anymore. When this happens, the dependency becomes *broken*. This concept is similar to broken links (also called *dead* links) on the world wide web.

4.5.1 Prevention is better than cure

There are a couple of measures that can be taken to prevent dependencies from becoming broken. Firstly, users can be advised in creating dependencies. For example, they can be discouraged to create dependencies on a very low level. This way, details of objects can be changed safely without the dependency becoming broken. For example, if a fragment in a documentation document refers to a fragment in another document, the dependency could be created between the sections of the fragments, instead of the specific fragments themselves.

Secondly, technical measures can be taken to support our tool in locating objects. A possibility is to include markers in the objects themselves, to indicate the start and the end of objects. When the format of the objects supports adding comments or annotations, like many programming languages do, these can be used to add markers without actually affecting the objects. This can be worked out in the same way as the `LINE` pragmas described in section 4.2.1. When users make changes to the objects, the markers automatically move along. An additional advantage is that users can also take the markers into account when making changes, so that broken relations can be prevented and changes can be included or excluded from the object explicitly.

4.5.2 Recovering broken dependencies

When, despite of the efforts of the user and the Consistency Management Assistant, dependencies have become broken, they need to be recovered. For dependencies that have been detected automatically, the detection can simply be executed again, but for dependencies that have been created manually, much more effort is needed. The only way to recover these broken dependencies automatically, is to search near the original location for an object similar to the original object. When no object can be found for which it is pretty certain that this is the replacement of the original object, the only way to recover the dependency is to let the user create the locations of the objects manually, or to let the user delete the dependency.

When replacing a large product, like a library, with a new version, many dependencies might become broken. The same approach then has to be taken for all these dependencies.

5 Inconsistency checking and resolution

When dependencies between objects have been created, everything will be fine until users make changes to the objects. When changes have been made, inconsistencies occur when the dependant of a dependency is not consistent with the authority anymore. The Consistency Management Assistant must be able to detect changes in objects, and determine if these changes lead to inconsistencies. If possible, our tool must be able to resolve the inconsistencies automatically.

5.1 Detecting changes

Because of the reactive approach we have chosen for our tool, which doesn't allow for tracking changes in objects while they are being edited it is necessary to scan the objects of the dependencies for changes at a regular interval. During this scan, all the objects have to be inspected. When an object is being scanned, it has to be compared with the previous version.

One possibility is to compare objects by size, but this is not very reliable as the contents might have been changed, but the length might have not. For some object formats, for example those contained by a file or directory, the modification dates can be compared. This is reasonably reliable, but can lead to problems, for example when both objects are in the same file and therefore share their modification date.

Other objects, like database tables, don't have a modification date. A common technique that can be used to compare the contents of these objects, is the comparison of so-called checksums, which are short strings that can be computed from the object contents. Comparing by checksums is very reliable as long as a sophisticated algorithm is being used. An example can be seen in figure 7. Also objects contained by files or directories can be compared using checksums to get a higher reliability for detecting changes.

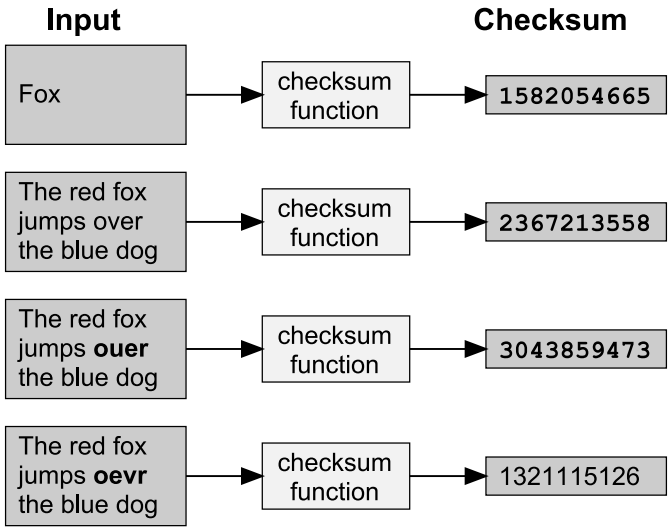


Figure 7: Example by Stolfi [2008] of checksums computed by the Unix `cksum` utility. Note that small changes in the text lead to completely different checksums.

5.2 Checking for inconsistencies

Checkable object pairs can be checked for inconsistencies. For each pair, different functions for checking for inconsistencies can be implemented. The functions can be custom created, but also an existing method or an external tool can be implemented or used.

The most basic checking function only checks if the authority has been changed. If this is the case, it might be that an inconsistency arises. Of course it can be the case that the contents has changed but no inconsistencies have arisen. But since we don't know this for sure, we always assume that the objects have become inconsistent. The number of false positives will therefore be high, but we stay on the safe side.

Some assumptions can be made to lower the number of false positives. For instance, when also the dependant of the dependency has been changed, it might be assumed that the user has already updated the dependant to let it agree with the changed version of the authority. Another example is to assume that objects that are literally next to each other, like inline code documentation and the code it documents, are automatically checked for inconsistencies by the user when changing the objects. Obviously, when making these assumptions, the chance that false negatives arise becomes bigger.

The detection of inconsistencies also depends on the cause of the dependency. For ordering dependencies, not only the authority has to be checked for changes, but also the dependant. Also it must be checked if the dependant still comes after the authority. For the other dependency causes, more sophisticated checking functions can be used to determine if the authority and the dependant have become inconsistent. Examples are methods that are able to interpret code, natural language or models by using textual or structural analysis or artificial intelligence, and methods for checking constraints on the objects. Both types of methods are described in the previous work in chapter 1. Because these methods can be more precise, the number of false positives will be lower, but care must be taken to prevent false negatives.

5.3 Resolving inconsistencies

For each resolvable dependency and for each dependency cause, a function can be implemented which resolves inconsistencies between objects of the formats of the dependency with that cause.

Inconsistencies in transformation dependencies, that have come into existence when the input has been changed, can be resolved by reapplying the transformation. After that, it can be assumed that all the inconsistencies have been resolved. Most of the transformation programs only allow reapplying the transformation on the complete input, for example a whole file, but sometimes is it also possible to only reapply the transformation to the objects that have been changed. Only the inconsistencies in those objects will then be resolved.

For inconsistencies in ordering dependencies and dependencies for which more sophisticated checking functions are available, it can be automatically detected if the inconsistencies have been resolved by checking if the inconsistencies still exists. For ordering dependencies this can for instance be done by checking the positions of the objects. When the dependant comes after the authority, the inconsistency is resolved.

The user must be warned about inconsistencies in dependencies that are not resolvable. He then has to resolve the inconsistency manually and explicitly mark it as resolved.

6 Functionality

This chapter specifies the functionality of the Consistency Management Assistant, i.e. the actions that a user should be able to perform. The implementation of the functionality is described in chapter 7.

6.1 Managing dependencies

Users must be able to manage dependencies between a wide variety of objects.

6.1.1 Defining

Dependencies between these objects must be detected automatically or be created manually.

1. Manage objects for which dependencies are managed
   (a) Add an object
   (b) Remove an object 1
2. Detect dependencies between objects automatically
   (a) Perform the detection at a specified interval
   (b) Start the detection manually
   (c) View a list of dependencies that have been detected
3. Create dependencies between objects manually
   (a) Select two objects and specify the dependency cause
   (b) Set a precision
   (c) Create annotations

6.1.2 Viewing

Users must be able to view existing dependencies between objects, at both a high and at a low level.

4. Put the contents of two objects next to each other and show the dependencies between them
5. View indirect dependencies between objects
6. Preview a previewable dependency

6.1.3 Updating and deleting

It must be possible to select a dependency to update or delete it.

7. Update a dependency
8. Change the dependant and the authority
9. Change the dependency cause
10. Change the precision

11. Create, edit and delete annotations
12. Split a dependency
13. Edit editable objects using an internal or an external editor

6.1.4 Recovering

Broken dependencies can be detected and recovered.

14. Regularly check if broken dependencies have arisen
15. Recover these dependencies automatically, if possible
16. Notify the user in case automatic recovery is not possible

6.2 Dealing with inconsistencies

Inconsistencies that have been detected should be resolved automatically. If this is not possible, the user must be warned about them.

6.2.1 Checking

Our tool checks for inconsistencies between the dependent objects.

17. Check for inconsistencies at a specified interval
18. Start the check manually
19. View a list of detected inconsistencies

6.2.2 Resolving

Detected inconsistencies are resolved, where possible.

20. Resolved inconsistencies automatically, if this is possible
21. Notify the user when automatic resolving is not possible.
22. Resolve an inconsistency manually
    (a) Display both objects of the dependency
    (b) For editable objects, edit the objects directly
    (c) Mark the inconsistency as being resolved
    (d) Update the dependency
    (e) Delete the dependency

6.3 Persistence and version management

The user must be able to store dependencies and load them in the tool again at a later moment. It must also be possible to add the dependencies to version control and share the dependencies with other users.

6.3.1 Storing dependencies

To allow for the use of existing version management tools, like Subversion [Pilato et al., 2008] and Darcs [Roundy, 2005], the dependencies are stored on the file system.

23. Store the dependencies and detected inconsistencies between one or multiple objects in a file, called a *project*
24. Store relative paths to objects, so that they can be moved
25. Open a previously stored project

6.3.2 Applying version management

Version management can be applied using an external version management tool.

26. Put a project under version control
27. Commit changes to the project automatically at a specified interval
28. Commit changes to the project manually
29. Revert to a previous version of the project

6.3.3 Sharing projects

Users are able to share projects with other users. When the version control repository is reachable for other users, they can check out the project using an external version management tool.

30. Check out a project from a repository
31. Locate the objects of a project on the local machine

6.3.4 Merging projects

An issue is that different users have different compositions of the objects they use. This can result in multiple projects being created for the same objects. An example is when one user has created a project file for objects A, B and C, and another user has created a project file for objects B, C and D. When these users want to share their dependencies between objects B and C, they would have to manage two projects, resulting in double work. A solution for this issue is to allow people for opening multiple projects, merging the dependencies of objects that occur in more than a single project.

32. Merge a project into another project

7 Tool implementation

We have implemented a prototype of the Consistency Management Assistant in the programming language Haskell, taking the requirements of the functionality we described in chapter 6 into account. In this chapter we describe the implementation of the data structures, the functionality and the user interface.

In figure 8 the user interface of the Consistency Management Assistant is shown. In this interface, users can select objects and manage the dependencies and inconsistencies between them. We will use the user interface as a base to describe the ingredients that are used for implementing the tool. This means that we will describe the implementation by following a user through the tool. This way we will arrive at the different aspects of the Consistency Management Assistant and its implementation. We attend to the implementation of objects and dependencies, the detection of dependencies and checking for inconsistencies. The details of the user interface itself are described in section 7.3.

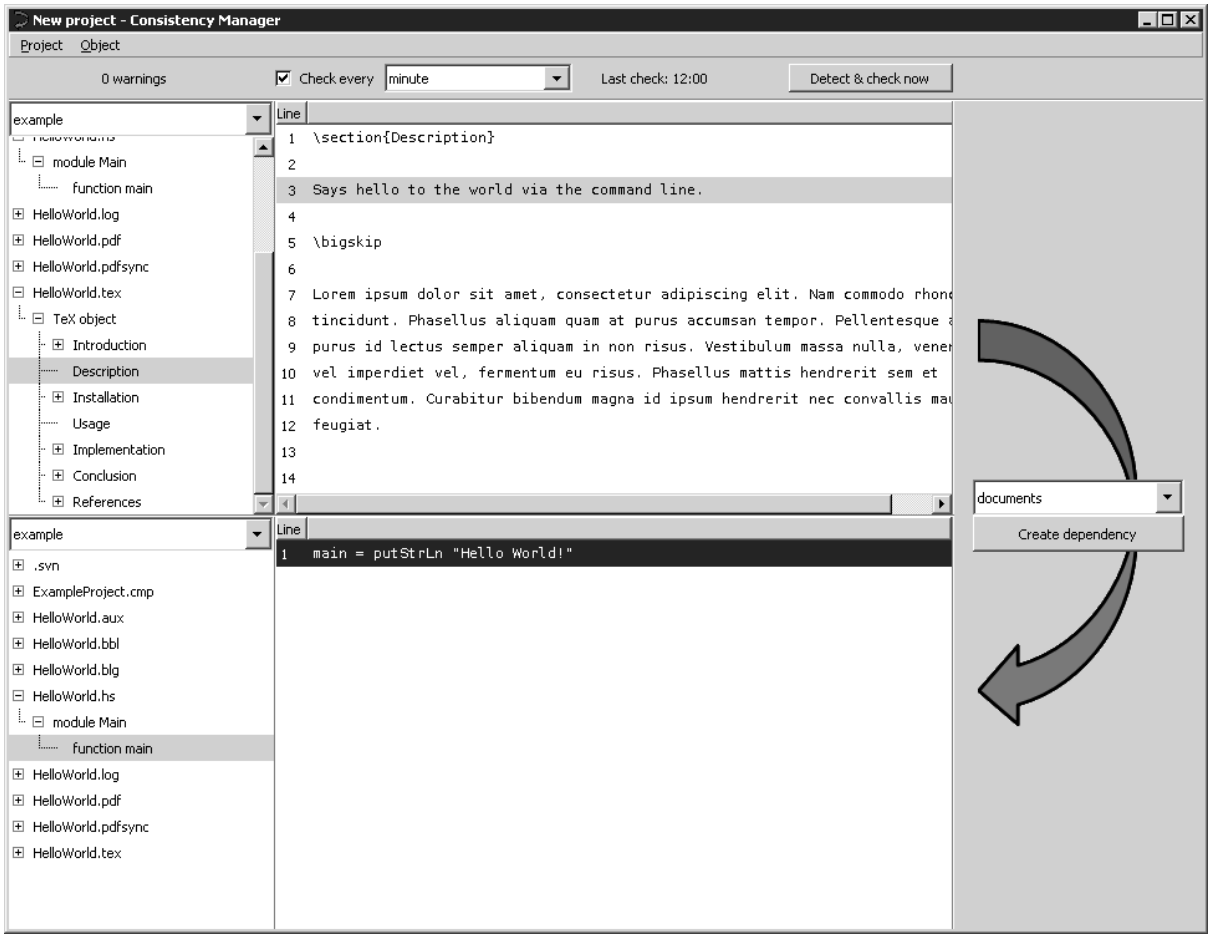


Figure 8: The user interface of the Consistency Management Assistant.

7.1 Data structures

Before following our imaginary user, something must be said on data types, classes and instances in Haskell, which are used to implement the definitions and properties given in this thesis. Data types in Haskell are used to specify a domain model. In our case, different data types define the structure of the object formats and dependencies. For instance, there is a data type for TEX objects, containing the name and the code of the object. For the dependencies, there is one data type that takes two of these object format data types, that are the dependant and the authority of the dependency. Values of these data types represent specific objects and dependencies.

For example, for managing general directories and files and the contents of files consisting of TEX code, we have to declare the following Haskell data types:

```
data Directory  = Directory  { dirName :: String }
data File       = File       { fileName :: String }
data TeXObject = TeXObject { teXIdent :: String
                           , teXCode  :: String }
```

Example values of these data types are:

```
Directory  { dirName = "thesis"             }
File       { fileName = "introduction.tex"  }
TeXObject { teXIdent = "Abstract"
          , teXCode  = "In this thesis..." }
```

Data types that have certain properties in common, can be grouped into classes. For example, the data type for TEX objects (*TeXObject*) and the data type for PDF objects (*PDFObject*) both belong to the class *Object o*. We can make a TEX object for example a member of the class *Object o* with the following instance declaration:

```
instance Object TeXObject
```

For dependencies, a class *Dependency dep auth* exists, that can be instantiated to create that a dependency from the dependant (*dep*) to the authority (*auth*) can exist. An example is the instance *Dependency PDFObject TeXObject*, defining that PDF objects can be dependent on TEX objects.

7.1.1 Object

Now we start following our imaginary user in its journey through the Consistency Management Assistant. The first thing the user will do is selecting objects for which the dependencies are managed. As the user can choose from a variety of different object formats, each data type of these formats is made an instance of the class *Object*. The class

```
class Object o where
  objectDescription :: o → String
  objectIdent       :: o → String
```

defines the minimal information needed by our tool to be able to manage the object. The *objectDescription* method returns a short description of the format. This description is the same for every value of that object format. The *objectIdent* method returns the identification of an object, which is in most cases unique. Examples are the name of a directory or file or the name of a TEX section. For example, the declaration of the instance *Object TeXObject* is:

```
instance Object TeXObject where
  objectDescription _ = "TeX object"
  objectIdent         = teXIdent
```

7.1.2 Dependency

After selecting objects, the user must be able to create dependencies between them. In contrast with the definition of a dependency given in this thesis, which states that a dependency is between two objects, in our tool the dependencies are actually created between two paths leading to objects. The structure of paths is defined by the data type *Pth o*, which is parameterised by the format of the object (*o*) at the end of the path. This is just an implementation detail, to make sure that the objects can be found by our tool. A dependency is created between two *Pth* data types, using the class *Dependency dep auth*. For example, the instance for dependencies from objects in the PDF format to objects in the TEX format is:

```
instance Dependency (Pth PDFObject) (Pth TeXObject)
```

The *Dep dep auth cause* data type is used to create values of dependencies. This data type is parameterised by the formats of the dependant and the authority, and the cause of the dependency. The definition of the data type is:

```
data (Dependency dep auth) =>
  Dep dep auth cause = Dep { dependencyDependant          :: dep
                           , dependencyAuthority          :: auth
                           , dependencyDependencyCause    :: cause
                           , dependencyAnnotations        :: [String]
                           }
```

The class constraint *Dependency dep auth* in the definition above forces that the formats of the dependant and the authority are an instance of the class *Dependency dep auth*.

7.1.3 Cause

It is necessary to know the cause of a dependency to determine which functionality is available for that dependency. For each dependency cause a data type is defined and is made an instance of the class *Cause c*. This class contains a method *causeDescription*, returning a short description of the cause. For the transformation cause, for instance, the data type *Transformation* is defined. The method *causeDescription* of the instance

Cause Transformation returns "transformation of", because the dependant of a transformation dependency is a transformation of the authority:

```
data Transformation = Transformation
instance Cause Transformation where
  causeDescription _ = "transformation of"
```

When creating a dependency between two objects, the user must indicate to the Consistency Management Assistant what the associated cause of this dependency is. The possible causes are the types which are made an instance of the class *Cause*. Which causes are available for a dependency from one object format on another format, is defined by instances of the *DependencyCause* class:

```
class (Dependency d a, Cause c) ⇒ DependencyCause d a c
```

Example instances are:

```
instance DependencyCause (Pth PDFObject) (Pth TeXObject) Transformation
instance DependencyCause (Pth TeXObject) (Pth TeXObject) Specification
```

7.1.4 Heterogeneous lists and trees

To keep track of the objects and dependencies in the Consistency Management Assistant, it is necessary to create lists of them. In Haskell, lists are homogeneous. This means that all the elements of a list have to be of the same data type. However, in our tool we have used different data types for the domain model, so it is impossible to create lists of them:

```
data TeXObject  = TeXObject String String
data PDFObject = PDFObject
  -- This is not Haskell!
objs = [ TeXObject "Introduction" "Chapter text comes here"
       , PDFObject
       ]
```

The usage of different data types cannot be circumvented, because the class instances we need, have to be defined on complete data types, and can for example not be defined on constructors of data types, like:

```
data Object = TeXObject String String
            | PDFObject
            | ...
  -- Again not Haskell!
instance DependencyCause PDFObject TeXObject Transformation where
  ...
```

Trees of objects, that are used in the implementation of the Consistency Management Assistant, suffer from the same problem. We therefore needed a way to create heterogeneous lists and trees, containing values of different data types.

One way to do this in Haskell is by using *existential types* (using the `forall` keyword). Existentials allow for using type variables on the right-hand side of data types, without specifying them as data type parameters (on the left-hand side):

```
data List = forall o ∘ Cons o List
          | Nil
```

This makes it possible to create lists with values of different data types:

```
exampleList = Cons (TeXObject "Name" "Code")
            $ Cons  PDFObject
            $ Nil
```

However, as existentials hide the type variables, it becomes impossible to look into the values. Because of this, it is not possible to read objects and dependencies into our tool that have previously been stored, since the read function does not know what types to construct when recreating the list. As we really need to do this, we had to conclude that we cannot use existentials. For the same reason, universal types (using the `Dynamic` data type [Cheney and Hinze, 2002, Abadi et al., 1989]) can't be used.

There are other possibilities to solve this problem, but most of them are very complicated. We chose the simplest solution, that uses algebraic data types to encompass data types which are an instance of the same class. One algebraic data type, which we call a *container data type*, has a separate constructor for each data type that is an instance of a single class. Using this data type, lists and trees containing values of different data types can be created. This is comparable to using embedding-projection pairs in generic programming [Hinze et al., 2007]. An example (note the underscores):

```
data Obj_ = Directory_ Directory
          | File_      File
          | ...
exampleList = [Directory_ (Directory "Name")
              ,File_      (File      "Name")
              ]
```

An important disadvantage of this approach is that switching types by pattern matching or by using **case of** expressions is necessary. Also this algebraic data type can only be created after the instances have been given, so if a user wants to extend the Consistency Management Assistant, he has to extend the algebraic data type as well, and also put his constructor on all places where type switching is applied.

7.2 Properties and functionality

All the properties of the objects and the dependencies are implemented using classes and instances. In this section we describe the implementation of the properties *precision*

types, *parseability*, *file extensions* and *locatability*, and how they are used in the Consistency Manger. The other properties that we introduced in section 2 have a similar implementation. Also we describe solutions for instance checking and getting a table of instances.

7.2.1 Precision types

The types of precision can be used to point at specific locations in objects. In the Consistency Management Assistant, this means that the user can select an object and point at a specific location in it. The precision types are implemented using data types. These data types are instances of the class *Precision p*. By default, a data type *NoPrecision* is available, which specifies that no precision has been set. Other common precision types are *LineRange*, which contains two integers to specify a range from one line number to another, and *Coordinate*, which can be used to specify a specific point, like a coordinate on a page in a PDF file:

data *LineRange* = *LineRange Int Int*
instance *Precision LineRange*
data *Coordinate* = *Coordinate Int Int*
instance *Precision Coordinate*

We have defined an *ObjectPrecision o p* class, that is used to link types of precision (p) to object formats (o). An example instance of this class is:

instance *ObjectPrecision TeXObject LineRange*

7.2.2 Parseability

For each of the object formats that are parseable, an instance of the class *Parseable o* has to be created, to get the tree of objects of an object in the format o. This class contains two parameters, one for the object format for which the instance is defined, and the other one for the way the object can be located (an instance of the class *Location l*):

class (*Object o*, *Location l*) ⇒ *Parseable o l* **where**
  parse :: $l \rightarrow o \rightarrow$ *IO* (*Tree Obj*_)

The *parse* method returns a tree of the object that has been parsed. For example, for a directory this is the directory tree, and for a TEX file this is the sectioning structure. The o parameter of the *parse* method is only used to select the right *Parseable* instance and should not be used by the method. For example, the value $\perp$ :: *TeXObject* could be passed as parameter o.

The most basic instance of the class *Location l* we have created is *Location FilePath*. Using this location, *Parseable* instances for files and directories can be created:

instance *Parseable Directory FilePath*
instance *Parseable File FilePath*

For locating objects that are not on a file system, but for example in a database, custom data types can be created, like:

```
data DatabaseServer = DatabaseServer String String String
instance Location DatabaseServer
```

7.2.3 File extensions

The file extensions are used to determine the object format of a file. These extensions are instances of the class *Extension e*. For each instance, the method *extension* is implemented, specifying the extension string:

```
data TeXExtension = TeXExtension
instance Extension TeXExtension where
  extension _ = "tex"
data PDFExtension = PDFExtension
instance Extension PDFExtension where
  extension _ = "pdf"
```

We use the class *Parseable o l* defined above to specify which object formats have which file extensions. For this we use a typed file path:

```
data TFilePath t = TFilePath FilePath
```

When defining an instance of the class *Parseable o l* for parsing object formats on a file path with a certain extension, the typed file path is parameterised with this extension, and used as location parameter *l*. For example, this is the instance for parsing TEX objects with the `.tex` extension:

```
instance Parseable TeXObject (TFilePath TeXExtension) where
  parse (TFilePath fp) _ = ...
```

7.2.4 Locatability

To be able to load into our tool, they have to be located. The way of locating objects of a certain format is specified by the class *Locatable o*. The *locate* method takes some information about the user interface, which can for example be used to create a dialog for opening a file or connecting to a database. If successful, the method returns the object that has been located and its location. The $Obj\_$ and $Loc\_$ data types are container types, like explained in section 7.1.4.

```
class (Object o) ⇒ Locatable o where
  locate :: o → Gtk.Window → IO (Maybe (Obj_, Loc_))
```

7.2.5 Detectability

For object formats for which dependencies can be detected, at a regular interval or when the user performs the detection manually, the class *Detectable d a* is implemented:

```
class (Dependency d a) ⇒ Detectable d a where
  detect :: d → a → IO [Dep_]
```

All detectable object format paris have to be made an instance of this class, like:

```
instance Detectable (Pth PDFObject) (Pth TeXObject) where
  detect d a = ...
```

The method *detect* takes two paths to objects and returns a list of dependencies that have been detected between the two objects.

7.2.6 Checkability

For dependencies for which can be checked if inconsistencies have arisen, the class *Checkable d a* is implemented:

```
class (Dependency d a) ⇒ Checkable d a where
  check :: (DependencyCause d a c) ⇒ Dep d a c → IO Bool
```

For these dependencies an instance has to be defined between the two object formats of the dependencies. The method *check* takes a dependency and returns if it has become inconsistent.

7.2.7 Instance checking

In our tool we need to check if an instance is defined for a data type. For example, when creating a dependency between two paths, we need to know which precision types are available for the objects at the end of the paths. Unfortunately, Haskell does not offer support for a construct like the `instanceof` construct in Java, which returns a boolean indicating if a class is a subclass of another class, or if it implements a certain interface. Of course, it is impossible for Haskell to support this, because of type safety. But as we really want to do this, since we want to know if we can create certain dependencies given two objects, we had to find a workaround.

The solution we have chosen is the use of overlapping instances, a type class extension of Haskell. This extension allows for creating instances that overlap, as long as one of the instances can be pointed at as the most specific one. We have added methods to our classes that check if the instance exists. For example, to the implementation of the *ObjectPrecision* class we have added a method *precision* that indicates if a precision type is available for a given object format:

```
class (Object o, Precision p) ⇒ ObjectPrecision o p where
  precision :: o → p → Bool
  precision _ _ = True
```

As can be seen, this method returns *True* for all *ObjectPrecision* instances by default. For applying the function *precision* to other data types, which are not an instance of the class *ObjectPrecision o p*, we have defined the following instance:

```
instance (Object o, Precision p) ⇒ ObjectPrecision o p where
  precision _ _ = False
```

When no more specific *ObjectPrecision* instance is defined for an object format and a precision, Haskell picks this general instance, resulting in *precision* evaluating to *False*. If a more specific instance is actually there, the default value is returned, which is `True`. This is exactly the behaviour we want to have.

For the other properties we have used the same approach.

7.2.8 Tables of instances

When our imaginary user wants to load an object into the tool, he has to select the format of the object. The list of possible formats from which the user can choose exists of the object formats that are locatable. For the implementation of this list, we need a table of instances of the class *Locatable l*, mapping the object format description to the corresponding *locate* method. Also in other cases we need such a table. In this section we will explain the trick that can be used to get this.

First we defined a nested cartesian product of data types of object formats:

```
type ObjectInstances = (Directory
                       , (File
                       , (TeXObject
                       , ...
                       , ())))
```

The next step was to create a class *LocatableObjectsTable* with a function called *locatableObjectsTable′* that walks through this nested cartesian product and generates a mapping table from object format descriptions to locate functions:

```
class LocatableObjectsTable ts where
  locatableObjectsTable′ :: ts → [(String, Gtk.Window → IO (Maybe (Obj_, Loc_)))]
instance LocatableObjectsTable () where
  locatableObjectsTable′∼() = []
instance (Locatable t, LocatableObjectsTable ts) ⇒ LocatableObjectsTable (t, ts)
  where locatableObjectsTable′∼(t, ts) =
    if locatable t
      then (objectDescription t, locate (⊥ :: t)) : locatableObjectsTable′ ts
      else                                          locatableObjectsTable′ ts
```

In the end we pass a ⊥ value with the type of the nested cartesian product to the *locatableObjectsTable′* function to get the mapping table:

```
locatableObjectsTable :: [(String, Gtk.Window → IO (Maybe (Obj_, Loc_)))]
locatableObjectsTable = locatableObjectsTable′ (⊥ :: ObjectInstances)
```

7.3 User interface

In this section we describe our choice for the library for creating the user interface and we describe the layout of the interface.

7.3.1 Library choice

For creating graphical user interfaces in Haskell, there are two libraries that are cross-platform compatible: wxHaskell [Leijen, 2004], which provides a Haskell interface to the wxWidgets toolkit, and Gtk2Hs [Hoste, 2008], which provides a Haskell interface to the GTK+ library. To choose one of the libraries, we first did some experiments. It turned out that Gtk2Hs has a slightly more extensive interface, allowing for richer interface elements and more advanced event handling. Both libraries are being maintained at the moment of writing, although the activity at the wxHaskell project is a bit lower than the Gtk2Hs project. At the moment of the implementation of the Consistency Management Assistant, Cabal [Jones, 2005] files were released for installing Gtk2Hs with `cabal-install`. This made the installation on different platforms much easier. wxHaskell already had Cabal packages, but getting it to work on different platforms was a bit harder. An advantage of the Gtk2Hs library is that the WYSIWYG editor Glade [Feldman, 2001] can be used to create user interfaces. Because of the advantages of Gtk2Hs over wxWidgets, we have chosen the Gtk2Hs route.

7.3.2 Layout

The complete user interface of the Consistency Management Assistant is shown in the beginning of this chapter (figure 8). The main part of the Consistency Management Assistant consists of two panes, which both contain a tree of objects and an area for the contents of objects. This way the contents of two objects can be viewed at the same time. When an object is selected from the selectbox, the corresponding tree is put into the tree view below it. By selecting an object from the tree, the contents is displayed in the content area to the right of the tree (see figure 9).

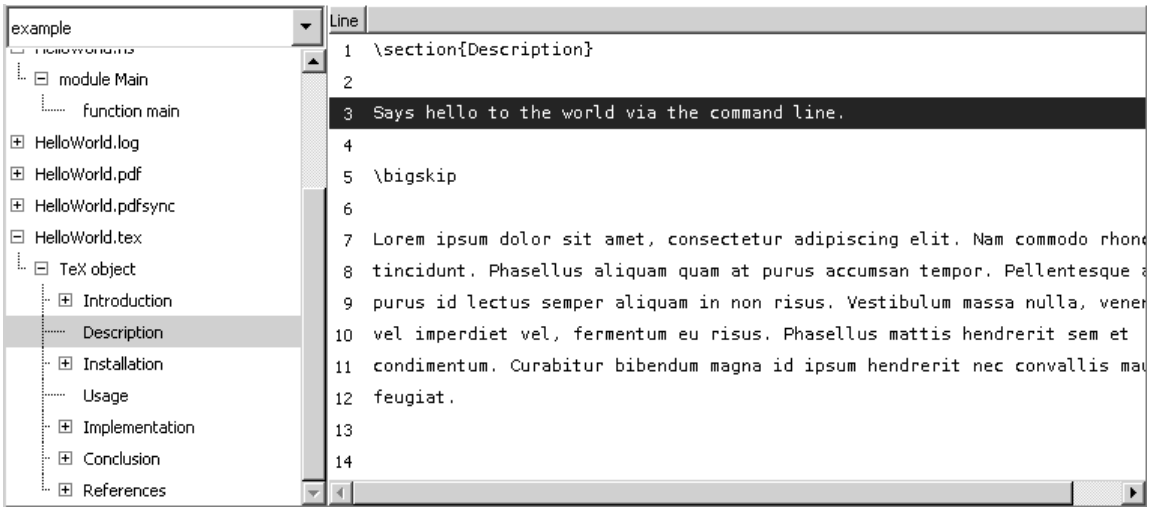


Figure 9: Displaying the contents of objects

When objects are markable, the locations in the contents of the objects that are involved in the existing dependencies, are marked. The locations are those indicated by the precision of the object at the end of the path. For pointable objects the same happens, but for those objects it is also possible to actually change the precision. For example, for ASCII files like TEX and Haskell objects, a line range can be selected. An examples of this can be seen in figure 9.

When creating or viewing dependencies, the dependant is displayed in the upper content area and the authority is displayed in the lower area. Next to the panes, an arrow is shown (see figure 10) directing from the upper pane to the lower pane, indicating that the object selected in the upper pane is dependent on the object in the lower pane. Via the selectbox a cause for the dependency can be selected. By pushing the button the dependency is created or modified.

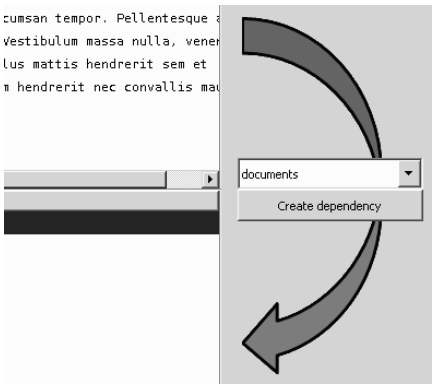


Figure 10: Creating dependencies with a certain cause

On top of the panes, there are some controls to set the interval for detecting dependencies and checking for inconsistencies, a button to detect and check for them manually, and some labels displaying relevant information (see figure 11). For example, when inconsistencies have been detected, a warning is shown containing the number of inconsistencies.

Via the menu bar, visible in the top left corner of figure 11, new projects can be created, projects can be opened and saved, and objects can be added to and removed from the current project.

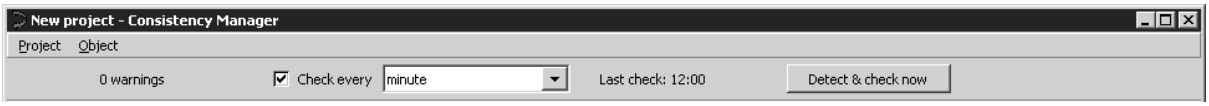


Figure 11: The controls for detecting dependencies and checking for inconsistencies, and the menu bar

8 Tool evaluation

In this chapter the Consistency Management Assistant is evaluated. We describe the functionality that is currently available in our tool, the implementation of our this functionality, the functioning of our tool and the usability.

8.1 Available functionality

We have only implented a prototype of the Consistency Management Assistant, which means that it has to be extended before it can be used in practice.

In the current implementation of our tool, directories and files can be added as objects to a project by locating it on the file system. The objects in the trees that are a result of parsing the directories and files can be selected and viewed. Currently only TEX, Haskell and (to a certain extent) PDF objects can be viewed. In TEX and in Haskell objects specific locations can be selected to indicate the precision of the object at the end of the path when creating dependencies.

The dependencies that can be created in the prototype are transformation dependencies from PDF on TEX, and specification, documentation, inclusion and usage dependencies between TEX objects, between Haskell objects, and from TEX objects on Haskell objects and vice versa. The precision types that can be used are line range precision (for TEX and Haskell objects) and page-coordinate precision (for PDF objects).

Most of the properties of objects and dependencies and the corresponding functionality have been implemented. The properties that are not yet implemented are editability, splittability, orderability and recoverability. Detecting dependencies has only been implemented for dependencies from PDF on TEX, using pdfsync files. Checking for inconsistencies has been implemented for dependencies from PDF on TEX and between TEX and Haskell objects. Automatic detection and checking has not been implemented, although the controls for changing their settings are already available.

8.2 Implementation

When starting the implementation of the Consistency Management Assistant, we first implemented the data types for the object formats and dependencies. Soon it turned out that dependencies needed to be created between paths leading to objects, instead of objects itself. Changing the data types wasn't a large alteration. Using the mechanism for classes and instances offered by Haskell turned out to be a good way to group data types, like the object formats into the class *Object o*.

During the implementation of the properties, it turned out that some of them had to be changed and it was necessary to add new properties. The implementation with classes and instances went well until we discovered that is was not easy to check if an instance is defined for a data type. Fortunately, the solution using overlapping instances was relatively easy to implement and worked very well. However, checking for instances at compile time became weaker as a result of the default instance that had been created.

We had already implemented a large part of the tool when we found out that our first approach for implementing the heterogeneous lists and trees, using existential data types, led to problems like converting string representations of these types back into actual data

type values. The solution using container types made it necessary to reimplement most of the data types, classes and instances. Also, using these container types resulted in a lot of overhead, because for each combination of data types separate constructors had to be created, and type switching needed to be added in a lot of places.

When testing the user interface, we noticed that during the execution of functions, the user interface was blocked. This became a problem when the execution of these functions took longer. We tried using threads for executing these functions in parallel, but this turned out to be impossible, because the threads were then blocked by the user interface. The cause of this is that in the interaction with the external Gtk+ library by the Gtk2Hs library no multi-threading is supported. Gtk2Hs offers a solution for this, which tells the Gtk+ loop to pause once in a while to give the Haskell threads a chance. This indeed resolved the problem of blocking either the user interface or the functions, but unfortunately the threads were executed very slowly. It seems like there is no good solution for this at the moment.

8.3 Functioning

We tested our tool with a limited number of object formats, namely TEX, PDF and Haskell. We chose these formats because their properties differ, dependencies between objects of these formats can arise from many different causes, and we wanted to have both textual and binary object formats.

We have selected different TEX and Haskell files to generate object trees from them, which worked well. We were also able to point at specific locations in these files to indicate a precision, to create dependencies using the paths to the objects and the precision, and to view the dependencies we created.

The detection of dependencies from PDF files on TEX files was succesful. When both a PDF file and a TEX file with the same name existed, our tool looked for a corresponding pdfsync file. If such a file existed, the file was parsed and dependencies from the coordinates on pages in the PDF file on lines in the TEX file were created.

When detecting dependencies and checking for inconsistencies, it turned out that comparing all combinations of the objects in the available object trees led to efficiency problems when the trees consisted of a large number of objects. We have currently no solution for this.

8.4 Usability

During the implementation of the Consistency Management Assistant we have tried to create an interface that is as usable as possible. We limited the number of user interface elements, which resulted in an organized layout. The controls we have used are conventional, so users have no problem in understanding their use. Also the Gtk+ library nicely fits into different operating systems, adapting to their standards.

Next to the limited number of controls, we also limited the number of choices a user can make. For example, when creating dependencies, the user can only choose valid causes. And when creating a dependency is not possible, the button for creating a dependency is disabled.

The possibility to view two objects next to each other turned out to be practical. The arrow indicating the direction of the dependencies made using our tool more intuitive.

Because not all functionality that has been described has been built into the prototype yet, not all user interface interactions have been tested. For instance, the automatic detection of dependencies and checking for inconsistencies has not yet been implemented, so the usability of the notifications given to users when dependencies or inconsistencies have been detected has not been tested.

9 Conclusion

In this chapter we evaluate the criteria we have formulated in section 1.3 and we make suggestions for future research.

9.1 Criteria

9.1.1 Definitions and properties

We have given clear definitions of objects, dependencies and the properties they can have and were succesful to implement them into the tool. We have been able to use the definitions for a wide variety of object formats, dependency causes, precision types etc. so we have managed to make the definitions and properties generic.

9.1.2 Tool prototype

We have developed a prototype of the Consistency Management Assistant. Like described in chapter 7, we have managed to create a tool that can support users in keeping their products consistent. We have implemented properties to detect dependencies and inconsistencies automatically, but we have unfortunately not been able to implement instances of this for object formats. However, we expect that this will work well.

We have taken into account that users must be able to add their own types of products to the tool, and we have experienced that this is well possible. For example, we have implemented instances for TEX, PDF and Haskell objects including parsers and implementations of the properties rather quickly, and were able to manage the dependencies between objects of these formats.

Our implementation is completely separated from the tools that users already use, so we have managed to create an unintrusive tool that minimizes the impact on the products and the way the user works.

9.1.3 Using Haskell

In most cases we found it pleasant to use Haskell and its type system, because it allows for writing compact code that can easily be re-used and is checked for validity at compile time. In some cases, however, we would have liked some more flexibility, for example for creating heterogeneous lists, checking for class instances of data types and getting tables of data types that have a certain class instance. We have managed to find workarounds for these issues, but this comes at a price. For instance, because of default instances for classes, the class constraints are not checked in some cases, which might lead to crashes when the instances of data types are not checked properly manually. Also the use of container types requires writing a lot of extra code.

The support for user interfaces in Haskell is still somewhat limited. Although two rich user interface libraries, Gtk2Hs and wxHaskell, are available, problems occur because of the binding to external libraries like Gtk+. In the first place problems occur during the installation, because of missing C libraries. These problems can be hard to resolve. Although this problem is not really Haskell-related, and should be solved on another level, for example by using a more sophisticated deployment management system like

Nix [Dolstra et al., 2004], still Haskell users creating user interfaces are troubled with it. Next to these problems, also some features of Haskell cannot be used when binding to external libraries. The most important feature that is lacking is the support for multi-threading. Also not all code can be checked for correctness during the compilation, because the Haskell interface of the Gtk2Hs library does not completely correspond to the external functions of the Gtk+ library. Because of this, not all errors in Gtk+ are caught by Gtk2Hs, causing the Consistency Management Assistant to crash when such an error occurs.

We have gratefully made extensive use of existing Haskell libraries from HackageDB [Jones, 2005] by using the Cabal build and packaging system. The availability of these libraries and the possibility of sharing them is a large advantage of using Haskell.

9.2 Future research

9.2.1 Finish implementation

Not all functionality described in chapter 6 is currently implemented. The most important features that are missing are the automatic detection of dependencies and inconsistencies, the recovery of broken dependencies, updating and deleting dependencies and the merging of projects. Also not much attention has been paid to locating objects on the local machine of the user and the notification of detected dependencies and inconsistencies to users. These features need to be implemented before the Consistency Management Assistant can be used successfully.

9.2.2 Implementing more instances

At the moment, only a few object formats and a couple of properties of these formats are implemented. Before the Consistency Management Assistant can be used effectively by other users, more object formats and properties need to be added. Examples are other programming languages, document formats and binary files and corresponding parsers, functions to view and edit objects in those formats and functions to detect dependencies and check for inconsistencies. It can also be investigated whether other dependency causes exist.

9.2.3 Generating container code with Template Haskell

The solution we chose to create heterogeneous lists, by using container types, results in a lot of code. For every object format a constructor has to be added to the container type for objects. Also for every combination of object formats and precision types a constructor must be added to the container type for paths. And for every combination of object formats, precision types and dependency causes a constructor must be added to the container type for dependencies. Moreover, at many places these container types must be type switched, by pattern matching or using **case of** expressions. This is a lot of work. We imagine that it is possible to generate this container code automatically using Template Haskell Sheard and Jones [2002]. This can be investigated, and, if it turns out that it is possible, it can be implemented into the Consistency Management Assistant code.

9.2.4 Improving efficiency

The Consistency Management Assistant is currently not very efficient. For every two objects in the trees of objects in the current project, the tool checks if dependencies can be detected and if it is possible to check for inconsistencies. Additional research is necessary to see if this can be improved. Also, the paths leading to the objects of a dependency are stored inefficiently, because the complete objects are stored in the steps of the path, instead of pointers to the objects. This results in a lot of data when many dependencies are created, and makes loading and storing dependencies slow. It can be investigated if it is possible to improve this.

9.2.5 More extensive dependencies

At the moment dependencies are between two objects. It can be investigated if it is desirable to have dependencies between more than two objects, or to create dependencies between dependencies, instead of objects. An example situation is to fix the order between multiple objects. If a user wants to have two objects in the same order as two other objects on which they are dependent, like two exercises in lecture notes in the same order as the theory the questions are about, a dependency of ordering dependencies might be useful.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–227, 1989.

Ademar Aguiar and Gabriel David. WikiWiki weaving heterogeneous software artifacts. In *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*, pages 67–74, 2005.

Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

Arthur Baars, S. Doaitse Swierstra, and Andres Löh. Utrecht University Attribute Grammar Compiler. `http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem`, 2009.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104, 2002.

Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, page 268279, 2000.

Ewen Denney and Bernd Fischer. A verification-driven approach to traceability and documentation for auto-generated mathematical software. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 560–564, 2009.

Atze Dijkstra. *Shuffle: manipulating source fragments*, 2009.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The structure of the essential haskell compiler, or coping with compiler complexity. pages 57–74, 2008.

Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 79–92, 2004.

Alexander Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 99(1), 2010.

Alessandro Fantechi and Emilio Spinicci. A content analysis technique for inconsistency detection in software requirements documents. In *WER*, pages 245–256, 2005.

E. Feldman. Create user interfaces with glade. *Linux J.*, 2001(87):4, 2001.

Stuart I. Feldman. Make—a program for maintaining computer programs. *Softw., Pract. Exper.*, 9(4):255–65, 1979.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/6.12.2/html/users_guide/index.html`, 2010a.

The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html#line-pragma`, 2010b.

M. Goedicke, T. Meyer, and C. Piwetz. On detecting and handling inconsistencies in integrating software architecture design and performance evaluation. In *ASE '98: Proceedings of the 13th IEEE international conference on Automated software engineering*, page 188, 1998.

Mark Grechanik, Kathryn S. McKinley, and Dewayne E. Perry. Recovering and using use-case-diagram-to-source-code traceability links. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 95–104, 2007.

John Grundy, John Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Trans. Softw. Eng.*, 24(11): 960–981, 1998.

Ralf Hinze and Andres Löh. lhs2TeX. `http://people.cs.uu.nl/andres/lhs2tex/`, 2009.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in haskell. In *SSDGP'06: Proceedings of the 2006 international conference on Datatype-generic programming*, pages 72–149, 2007.

Kenneth Hoste. An introduction to gtk2hs, a haskell gui library, 2008.

Isaac Jones. The haskell cabal: A common architecture for building applications and libraries. In *6th Symposium on Trends in Functional Programming*, pages 340–354, 2005.

Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

Jerome Laurens. Official pdfsync page. `http://itexmac.sourceforge.net/pdfsync.html`, 2004a.

Jerome Laurens. Official Synctex page. `http://itexmac.sourceforge.net/SyncTeX.html`, 2004b.

Daan Leijen. wxhaskell: a portable and concise gui library for haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 57–68, 2004.

John MacFarlane. Pandoc user's guide. `http://johnmacfarlane.net/pandoc/README.html`, 2010.

Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.

Simon Marlow. Haddock: A Haskell Documentation Tool. `http://www.haskell.org/haddock/`, 2010.

Collin McMillan, Denys Poshyvanyk, and Meghan Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *TEFSE '09: Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48, 2009.

Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Static consistency checking for distributed specifications. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 115, 2001.

Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58:171–180, 2001.

C Pilato, Ben Collins-Sussman, and Brian Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., 2008. ISBN 0596510330, 9780596510336.

D. Poshyvanyk and A. Marcus. Using traceability links to assess and maintain the quality of software documentation. In *Proceedings of TEFSE '07*, pages 27–30, 2007.

David Roundy. Darcs: distributed version management in haskell. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, 2005.

Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002.

Jorge Stolfi. Checksum.svg. `http://en.wikipedia.org/wiki/File:Checksum.svg`, 2008.

Sun Microsystems, Inc. javadoc - The Java API Documentation Generator. `http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html`, 2002.

Thanwadee Sunetnanta and Anthony Finkelstein. Automated consistency checking for multiperspective software applications. In *In Proceedings of the International Conference on Software Engineering Workshop on Advanced Separation of Concerns*, pages 12–19, 2001.

Hàn Thế Thành. The pdfTEX program. In *Proceedings of EuroTeX 1998*, 1998.

Ellen Van Paesschen, Wolfgang De Meuter, and Maja D'Hondt. Selfsync: a dynamic round-trip engineering environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 146–147, 2005.

Xiaobo Wang, Guanhui Lai, and Chao Liu. Recovering relationships between documentation and source code based on the characteristics of software engineering. *Electron. Notes Theor. Comput. Sci.*, 243:121–137, 2009.