A well-formedness theory for C source inclusion

Abstract

One of the features of C that makes it so popular among developers and at the same time unpopular among researchers, is that C does not impose any constraints on the way resources (procedures, variables etc.) are distributed among source files. Many resource references are left to the linker to resolve, something that makes the task of software maintenance much more difficult.

We present a well-formedness theory for source inclusion in C that, if imposed, can improve the maintainability of C systems. We then use, SoFi, a re-engineering tool, to examine various publicly available systems and see whether and why developers follow these rules or not.

1 Introduction

Software development in C has many features peculiar to the language. One of them is the freedom it provides developers to distribute resources in source files arbitrarily. While this approach can prove to be very convenient during the development stage, it can be a big hindrance for maintenance.

In this paper, we present a well-formedness theory for source inclusion in C systems that, if imposed, can improve the system's maintainability. We also present diagrams of the source inclusion "anomalies" found in real-life shareware systems, along with our observations on why developers follow or ignore the rules of this theory in practice.

These diagrams were produced by SoFi, an in-house re-engineering tool, that automatically extracts dependency information from the source code and visualizes it in the form of a diagram as these found in the appendix.

The structure of the rest of the paper is as follows. Section 2 presents the rules of the well-formedness theory along with the reasoning behind them. Section 3 presents various examples of real-life systems, as well as our observations on the way they are structured. Finally, section 4 concludes the paper by presenting our overall conclusions.

2 Well-formedness theory

Let us define the following sets:

$$\begin{gathered} R = \text{the set of all resources (procedures,} \\ \text{variables etc.) of a system} \\ C = \text{the set of all .c files of the system} \\ H = \text{the set of all .h files of the system} \\ F = C \cup H \end{gathered}$$

The following relations between files and resources must also be defined:

- File f *declares* resource r, iff file f contains a valid C declaration for the symbol r.
- File f *references* resource r, iff file f contains a valid C reference to the symbol r.

We can now define two of the most important relations between source files, namely *includes* (I) and *depends* (D), as follows:

$$\begin{gathered} I \subseteq F^2, \text{ where } f_1 I f_2 \Leftrightarrow f_1 \text{ contains the string} \\ \text{\#include "}f_2\text{"} \\ D \subseteq F^2, \text{ where } f_1 D f_2 \Leftrightarrow \exists r \in R : f_2 \textit{ declares} \\ r \ \& \ f_1 \textit{ references } r \end{gathered}$$

These two relations partition space F^2 into four sets as demonstrated by figure 1. Except for the uninteresting $F^2-(I \cup D)$, the remaining three sets are:

$$\begin{gathered} G = I - D \\ S = D - I \\ N = I \cap D \end{gathered}$$

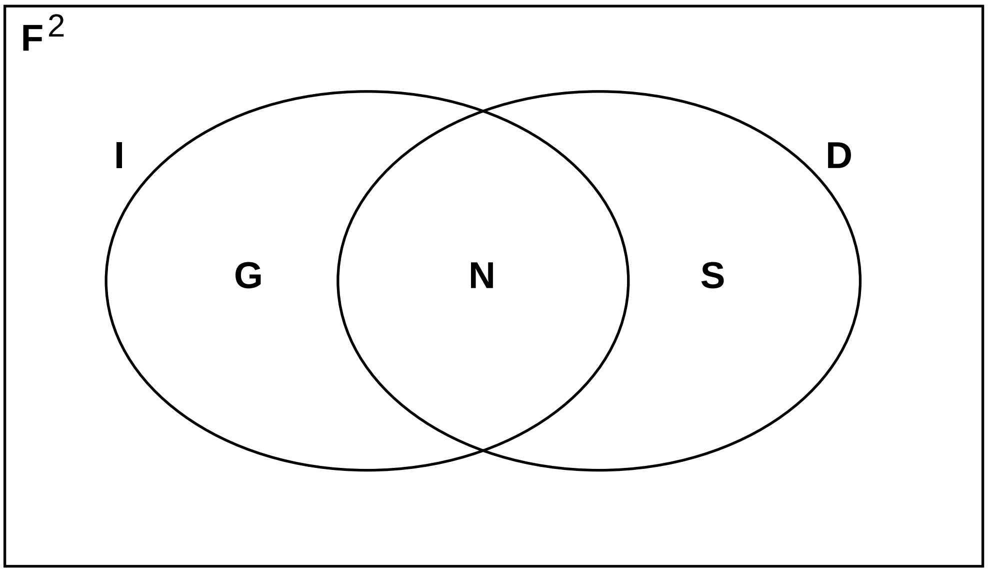

Figure 1: Partitioning of the F^2 space

We will call G the set of gratuitous dependencies, S the set of sneak dependencies and N the set of normal dependencies.

Figure 2 illustrates the aforementioned kinds of dependencies in a more intuitive way.

We propose the following three rules that we believe if imposed would make the design more modular, the source file structure cleaner and would facilitate the development and maintenance process.

1. $I \subseteq (C \times H) \cup H^2$. This rule forbids inclusion relations between .c files. This way the system's design becomes more modular since every part of the code must now have an implementation (.c) and an interface (.h) part.

2. $G = \emptyset$. This rules forbids the existence of gratuitous dependencies. This rule will improve the compile-time perfomance of the system and will clarify the source file structure by removing unnecessary dependencies from it.

3. $S = \emptyset$. This rules forbids the existence of sneak dependencies. The maintenance process should benefit from this rule. Any symbol that a file references will be declared in it or in one of the files it directly includes.

This set of rules of course is not the only possible one. For example, we could have been more flexible in the definition of the set of sneak dependencies by defining it as $S = D - I^+$. This way, depending on a symbol declared in a file that is transitively included would be "legal".

Another, even weaker variation of the third rule, could be to allow some sneak dependencies but insist that there exist no cyclic sneak dependencies.

For the rest of the paper we examine various C systems in order to see whether and why these rules are followed or not. For this purpose we use SoFi, an in-house software re-engineering tool, that can produce visualizations of the source file structure of C systems.

In all the diagrams presented, gratuitous dependencies are presented as thin edges, while sneak dependencies as dashed thick edges. Normal include dependencies have been elided from the diagrams to make them more readable.

3 Example systems

Table 1 presents the characteristics of all the systems examined in this section.

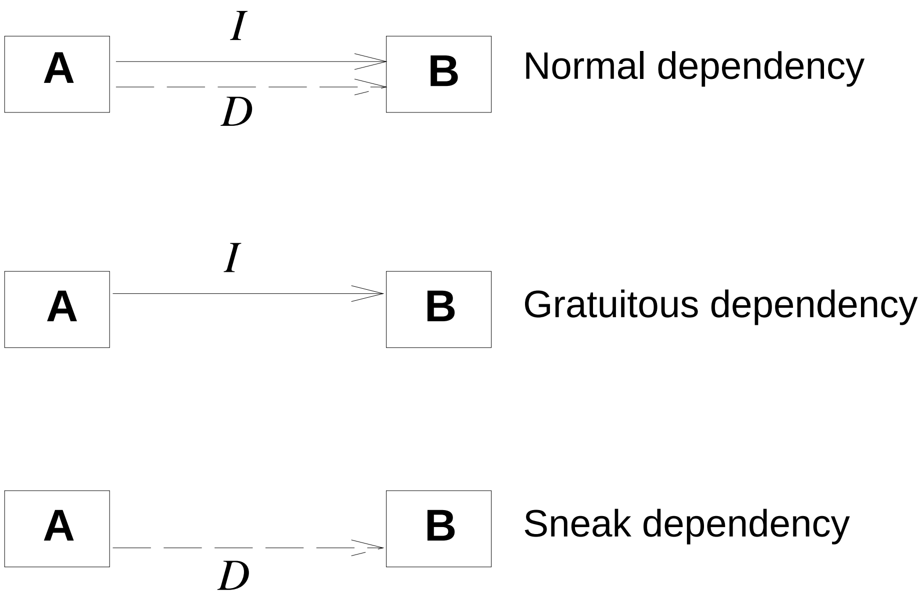

Figure 2: Possible kinds of dependencies between two source files A and B.

system name	figure	lines of code	number of source files	# of gratuitous dependencies	# of sneak dependencies
BRIDGE	3	2145	16	2	3
BACKGAMMON	4	4717	22	1	2
ZIP	5	8163	21	10	1
MPEG	6	13762	28	22	3
OOT	7	12037	25	16	1
XFIG	8	37550	102	58	35

Table 1: Systems presented in this chapter.

3.1 Small systems

Figures 3 and 4 present the gratuitous and sneak dependencies discovered in two small C programs implementing two games (backgammon and bridge). As can be seen by these figures, even small systems contain gratuitous and sneak dependencies. Gratuitous dependencies can be attributed to the fact that small systems are usually composed of several .c files implementing different parts of the system and just a few .h files containing mainly global declarations, exactly as is the case with these two systems. Developers tend to include these .h files even if they do not really need them.

Sneak dependencies are due to the fact that small systems are usually implemented without having modularity in mind. That is why some .c files implicitly depend on other .c files instead of including their interface.

Of course, small systems can be easily kept under control regardless of the number of gratuitous and sneak dependencies they contain, due to their size. However, it is interesting to point out that deviations to our rules exist even for systems where they should be easily applicable.

3.2 Medium systems

As can be seen in figures 5 and 6, the number of gratuitous dependencies in the video sequence decoder system and the compression utility is large. The number of sneak dependencies is slightly reduced, perhaps due to more careful design.

An attempt to utilize these diagrams in order to improve compile-time efficiency was made for the video sequence decoder system. The three gratuitous dependencies originating at the file *24bit.c* (leftmost in the third row of figure 6) were removed from the source code. The system still "made" successfully and the part of the compilation corresponding to the *24bit.c* file now took only 4 seconds, while it initially required 6 seconds.

This result shows that automatic discovery of gratuitous dependencies and their subsequent elimination from the source code can improve compile-time efficiency significantly.

Another interesting observation is that in these diagrams the main .c source file does not always appear in the upper layer of the layout as shown in figure 6 (main.c appears in the third layer). That is because other source files may implicitly depend on symbols declared in it.

For the same reason, diagrams depicting sneak dependencies can contain cycles, which is not possible for diagrams of include relations. As mentioned before, the disallowance of cyclic sneak dependencies could be a rule for a more elastic well-formedness theory than the one we present in this paper, since they can really complicate the maintenance process.

Figure 7 presents the diagram produced by SoFi when it was applied to the main directory of the OOT source code. As can be seen by the diagram, due to the modular design of the source code there exists only one sneak dependency in the system. The number of gratuitous dependencies however still remains high.

When this diagram was presented to the developers, they were able to remove most of the gratuitous dependencies from the source code without affecting its correctness. They still chose to keep some of them though for two reasons:

1. They prefer to include global definitions even if they do not need them, so that they do not have to worry about it in the future.

2. Compatibility with other versions of the code being developed in parallel.

As it turned out, most of the gratuitous dependencies that were removed from the source code, were there for historical reasons, meaning that there once existed the need for this dependency, but not anymore.

3.3 Large systems

Figure 8 presents the output SoFi produced when it was applied to the source code of the *xfig* utility, a large drawing utility comprising over 100 files.

As is obvious from the picture, this system contains a large number of gratuitous and sneak dependencies. This serves to demonstrate that large systems are not necessarily designed in a modular way that can prevent sneak dependencies.

Such designs may speed up the development process because they allow developers to use resources without worrying where they are declared, but will also slow down maintenance because the effects of a change in some resource are not immediately obvious.

4 Summary

Experimenting with SoFi's dependency analysis facilities has provided us with interesting conclusions on the way C systems of various sizes are built. These conclusions are:

1. Almost all C systems contain a number of gratuitous and sneak dependencies. This was demonstrated by a large number of systems that was examined but not presented in this paper.

2. Gratuitous dependencies are usually due to inclusion of source files containing global declarations even if they are unnecessary. Developers seem not to worry about the compile-time efficiency of their systems.

3. Another cause for the existence of gratuitous dependencies are historical reasons. The need for the inclusion of a file may cease but the *#include* statement will not be removed from the source code.

4. Sneak dependencies exist because they are convenient during the development process but can become problematic during maintenance.

A Figures

This appendix contains all the figures mentioned in section 3.

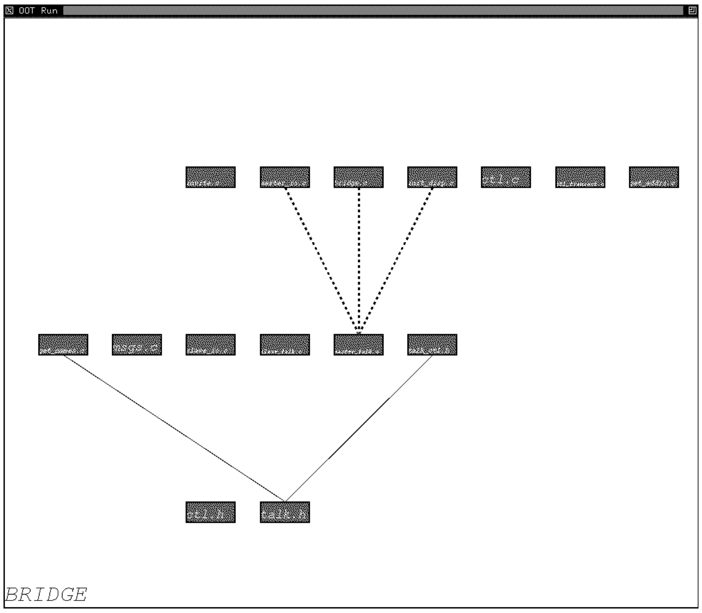

Figure 3: The bridge system

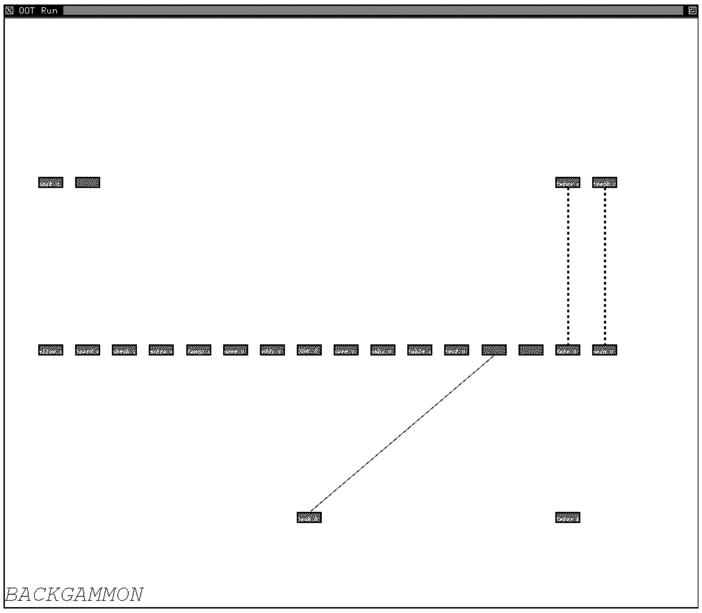

Figure 4: The backgammon system

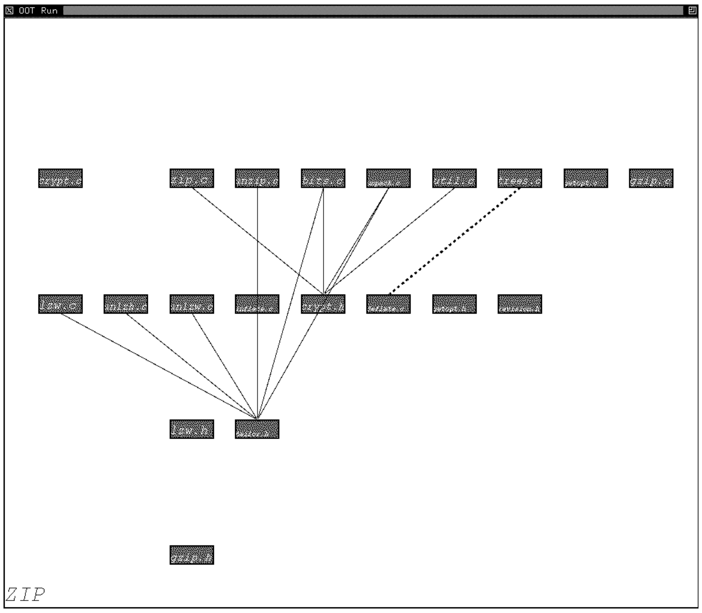

Figure 5: The compression utility

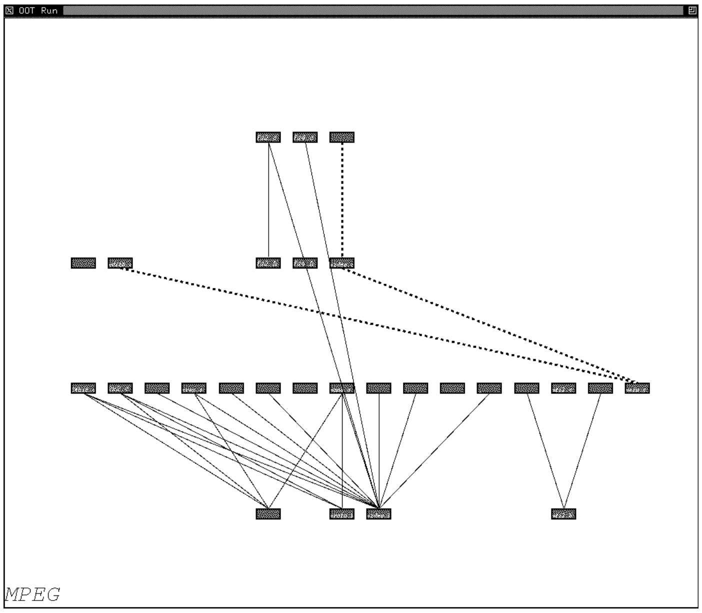

Figure 6: The video sequence decoder system

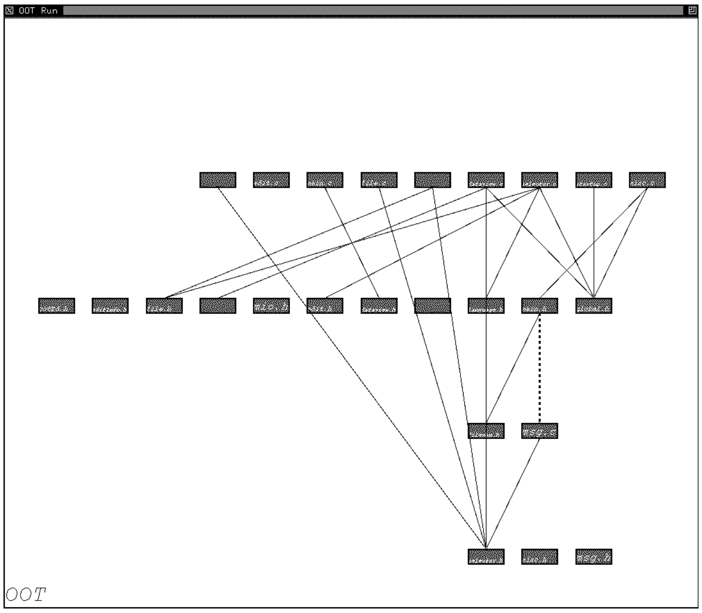

Figure 7: The OOT source code

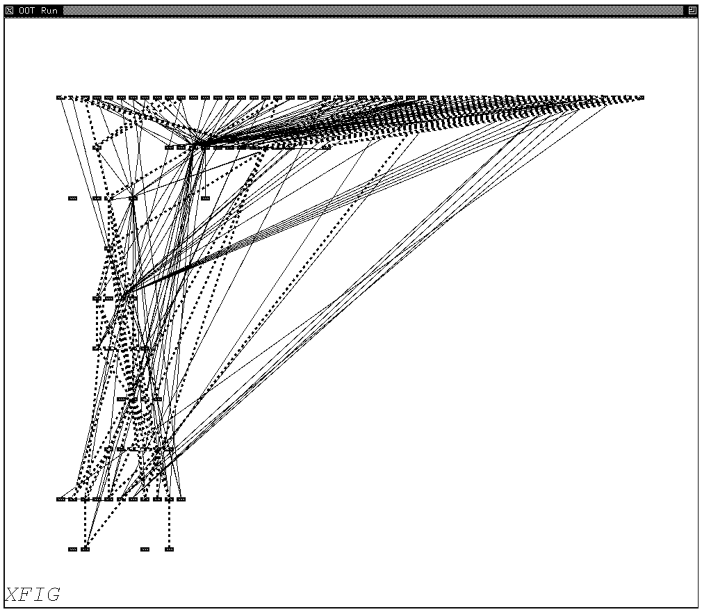

Figure 8: The xfig utility