L. Feijs
Philips Research Laboratories,
Eindhoven University of Technology
R. van Ommering
Philips Research Laboratories

Architecture Visualisation and Analysis: Motivation and Example

Abstract

This report explains some ideas on software architecture visualisation and analysis. Techniques for software architecture visualisation and analysis can be used to perform a kind of architecture verification, meaning that high-level views and low-level views of a complex system are kept consistent with each other.

The report contains an example, which however is a true toy-example, in order to keep this report as short as possible. Some of the verification steps shown in this report can easily be automated in a piece of software which we will call a "relational calculator" the example has been devised as a mock-up demo of the relational calculator, just assuming its existence. The example is followed through several steps of a simplified development process, summarised as follows:

- the software architect defines an architecture as a set of graph-like structural rules,
- then the programming team goes and starts creating code,
- architecture-conformance is checked regularly by extracting structure from code and comparing this actual structure to the architect's rules; this calls for the 'calculator' as a data/graph/set manipulation engine.

The development project benefits from this architecture verification in several ways:

- supporting a common understanding of the intended architecture,
- ensuring consistency between code and 'official' architecture.

This report is based on an idea that an architecture A is a specification of the intended structure of a large design, and a concrete design D is a structure 'as realised'. There is evidence that technically it will become possible to have a vocabulary to express A and the technology to verify whether a given D does satisfy A.

Keywords: *software structuring, graph algorithms, reverse engineering, formal specification, relation algebra, software architecture*.

1 Introduction and Motivation

Whereas for programming in the small there exist well-established concepts of specification and implementation, the subject of 'programming in the large' (or software architecture, as we sometimes call it), has hardly any established terminology. In this report we elaborate the idea that an architecture A is a specification of the intended structure of a large design, and a concrete design D is a structure 'as realised'. We present some evidence that it will become possible to have a vocabulary to express A and the technology to verify whether a given D does satisfy A.

The software complexity of many Philips products increases, amongst other reasons because many features are to be realised in software instead of in hardware. Often there is a need to deal with product families instead of just products. Not only the processor performance grows exponentially, but most often the software size grows exponentially too. Other complicating factors are that closed boxes become part of open systems, that media become part of multi-media, and that in many business groups there is a growing interest in optimising software re-use.

In today's software engineering practice, products go through a concept phase in which the architecture is well-visualised by means of diagrams, while the team is still growing, and while a good architect is present. But in the realisation phase the architecture diagrams may have become obsolete, and although low-level coding standards and analysis tools exist, it is usually not possible to check the real software against the high-level architecture. In the realisation phase, there can be serious problems in managing the software complexity. Yet there is hope: some visualisation techniques exist already and for example the tool TEDDY turned out to be already very useful for analysing evolving architectures; several other developments point into the same direction, for example QAC and Graphical Designer.

Some of the ideas reported in the present report were developed in the context of the project 'design engineering methods'. We list some key ideas of this report:

- many relevant structures in a software architecture are nothing but binary relations,
- manipulating relations is a way of obtaining views on intended or concrete architectures,
- alternative views on concrete software architectures can be visualised,
- a modest amount of automated support could be of great help.

It is a goal of this report to explain the idea of software architecture verification. We believe that there are technical options which will help in understanding the evolving Philips architectures. It would be beneficial to have a kind of continuity in the architecture evolution in the sense that in all project phases there are explicit architecture rules, and up-to-date high level views which are kept consistent with the real software.

The main body of this report consists of a story about a fictitious software team which develops a fictitious product and applies some verification techniques to its evolving architecture. In order to keep this report as short as possible, the fictitious product is just a toy example, but nevertheless the example will convey the idea of software architecture verification. Sections 2 to 6 present this story. Please note that the data structures used to define the architecture as well as the structure extracted from the code are just examples.

Related work: The idea of architecture verification is also presented in [1] and [2]. The TEDDY tool is described until now only in Philips internal reports. There are also Philips internal reports containing a detailed study of the theory of relations for purposes of softwarearchitecting. For more information about the mathematical theory of relations we refer to [3].

Acknowledgements: The author would like to thank René Krikhaar for the cooperation on the subject of this report. In fact, this report is only an extremely simplified case study of things which the second author and René are already putting into practice in real-life systems. The author would like to thank Peter van den Hamer for the discussions and for his feedback on an earlier version of this report. Special thanks go to Henk Obbink, project leader of the 'design engineering methods' project, which turned out to be a fruitful place for discussions and for the exchange of ideas.

2 The initial software architecture

In the beginning of the system's conception there is a software architect whose task it is to define the system's software architecture. The software architect will listen to all the marketeers, customer's representatives and hardware specialists. Of course there is a project document listing goals like flexibility, modularity, future-proofness as well as a list of performance requirements and cost constraints.

Performance requirements and cost constraints strongly influence the number and types of microprocessors, the programming language (assembly, C-like, C, C++, SDL, and so on) and the kind of operating system or RTK that is affordable. But the modularity goals are covered by a software architecture diagram. For a real-life 512 Kbyte system, say, this diagram may in the initial phase take the form of an A4-sized drawing with some 40 to 70 named boxes, organised in layers.

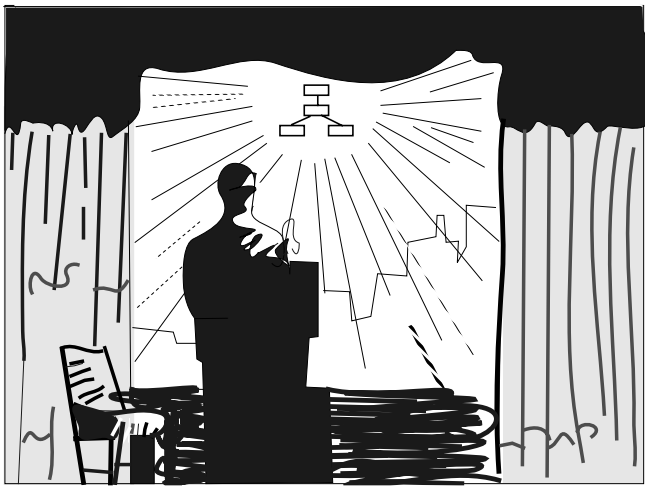

Fig. 1. The software architecture is presented.

In this paper we shall only follow a mock-up development process of a toy architecture, so instead of 40 boxes (software components) we look at a diagram of just 4. But we shall be explicit about the 'uses' relations between these software components.

So let us assume that after some time the software architect presents the key software components and a diagram of the designed component-level 'use' relation. He presents this to the designers and programmer(s) who are going to further detail this design and who will eventually make the C programs. Let us assume that amongst other things, the architect says:

> "Dear friends, Figure 2 is our software architecture: there are four software components, which I will explain now. `RSRC_MNGR` is the Resource Manager, which will contain the main procedures of all our processes and these will be scheduled by the HW and SW of our platform. `SYS_FUNC` contains the System Functions, and this is the heart of our system. This will provide the data transformations our customers are waiting for. `HW_ABSTR` is the minimal Abstraction of the special Hardware of our platform. `ERR_DRVR` is the Error Driver which provides for error printing and contains a driver for the special error LED. The lines in Figure 2, directed from top to bottom, show the 'use' relations foreseen. So for example from `SYS_FUNC` you are allowed to call the functions of `HW_ABSTR`."

Although here Figure 2 has been made by hand, it could also have been made with TEDDY, a browser which proposes an initial diagram layout after reading the file with the essential information of the ‘uses’ relation, and which allows the user (the architect) to modify the layout interactively.

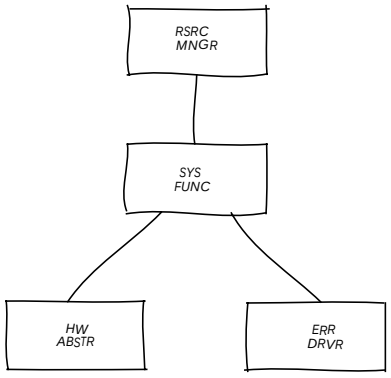

Fig. 2. Architected component-level ‘use’ relation.

Of course, instead of manual drawing or using TEDDY, other visualisation software could be used. The only requirement is that a diagram and an explicit binary relation are easily kept consistent. The best way of keeping two things consistent with each other is to generate one of them automatically from the other. It is important to note that the essential information of Figure 2 is an intended ‘use’ relations on components, which is as follows:

```
RSRC_MNGR SYS_FUNC
SYS_FUNC  HW_ABSTR
SYS_FUNC  ERR_DRVR
```

In many real projects, the processor is more powerful than the processor of the previous generation of products and there is two or four times as much ROM available, so most often the team is optimistic that that the software architecture is feasible. Also in our case, the programmers agree with the software architecture and happily they start filling in the details.

3 Details of the real system

After some work, the programmers will come up with C code and for example the error driver ERR_DRV could consist of two functions, `err_pr()` which calls `led_33()` and `led_33()` which calls `err_pr()`.

```
err_pr() { led_33(); }
led_33() { err_pr(); }
```

Since in this paper we are only following a mock-up development of a toy architecture, we shall not discuss the complex algorithmics of real-life software, nor the specification techniques necessary for that, but we only show some extremely simple C functions, calling each other, but with no meaningful functionality whatsoever. Note also that we do not stick to the usual layout conventions, in order to save space. After all these disclaimers, we give the C code of all software components.

```
/***************************************************
 *    Component: ERR_DRVR                          *
 ***************************************************/

err_pr() { led_33(); }
led_33() { err_pr(); }

/***************************************************
 *    Component: HW_ABSTR                          *
 ***************************************************/

#include "ERR_DRVR.h"

power() { err_pr(); i2c(); }
i2c()   {                  }

/***************************************************
 *    Component: RSRC_MNGR                         *
 ***************************************************/

#include "SYS_FUNC.h"
#include "HW_ABSTR.h"
#include "ERR_DRVR.h"

init()   { e(); led_33();            }
reboot() { power(); init(); power(); }
step()   { while (1+1==2) a();       }

/***************************************************
 *    Component: SYS_FUNC                          *
 ***************************************************/

#include "HW_ABSTR.h"
#include "ERR_DRVR.h"

a() { b(); c(); }
b() { power();  }
c() { d(); g(); }
d() { i2c();    }
e() { f();      }
f() { g(); err_pr();  led_33(); }
g() { h();                      }
h() { err_pr();                 }
```

4 Extracting relations from the real system

For a 512 Kbyte system, it may take a year from conception to completed code, and then the code is not as easily surveyed as the one page of C code of our toy example. After this year, there is no more focus on architecture, because everybody is busy with testing, debugging and adding shortcuts and tricks for meeting the performance requirements. New people joined the project and maybe the architect has already left.

If this were a real-life project and it would continue for yet another year, the project could find itself in a reverse engineering phase. There is even a danger that the project finds itself in the middle of a spaghetti: nobody understands all of the code and nobody understands the system's modular structure and the associated 'uses' relations.

But stop, we shouldn't wait until the spaghetti-phase. As soon as the initial code is available, it can be checked against the architecture diagram. Of course in a real-life project, there could be several levels of hierarchy, and there may be even more kinds of 'uses' relations than just the 'calls' relation on C functions, but the essential idea remains the same. It is possible to extract the real 'uses' relation, as opposed to the architected 'uses' relation from the code. All the information is on-line available; the only problem is that there is too much information. The obvious solution is to use automated support for extracting the essential information from the code. Again this information can be cast into the form of tables. There are technical options to perform this extraction, for example QAC, although of course they depend on the programming language in use.

For the present example the the essential 'use' information is easily extracted and stored in a file called `uses`.

```
err_pr  led_33
led_33  err_pr
power   err_pr
power   i2c
init    e
init    led_33
reboot  power
reboot  init
reboot  power
step    a
a       b
a       c
b       power
c       d
c       g
d       i2c
e       f
f       g
f       err_pr
f       led_33
g       h
h       err_pr
```

But it is not obvious that this satisfies the initial architecture of Figure 2. As a first step, it is already helpful to visualise this 'uses' relation, see Figure 3. This is made with a structure browser (TEDDY). Of course this browser is not only useful for analysing a design once finished, it could in principle also be used for drawing diagrams of relations which do not exist yet, but which will be implemented next. Note the thick line between `err_pr` and `led_33` which reveals the mutual usage (a thick line is a 'uses' arrow going upward, and since there is a cycle, there *must* be at least one thick arrow).

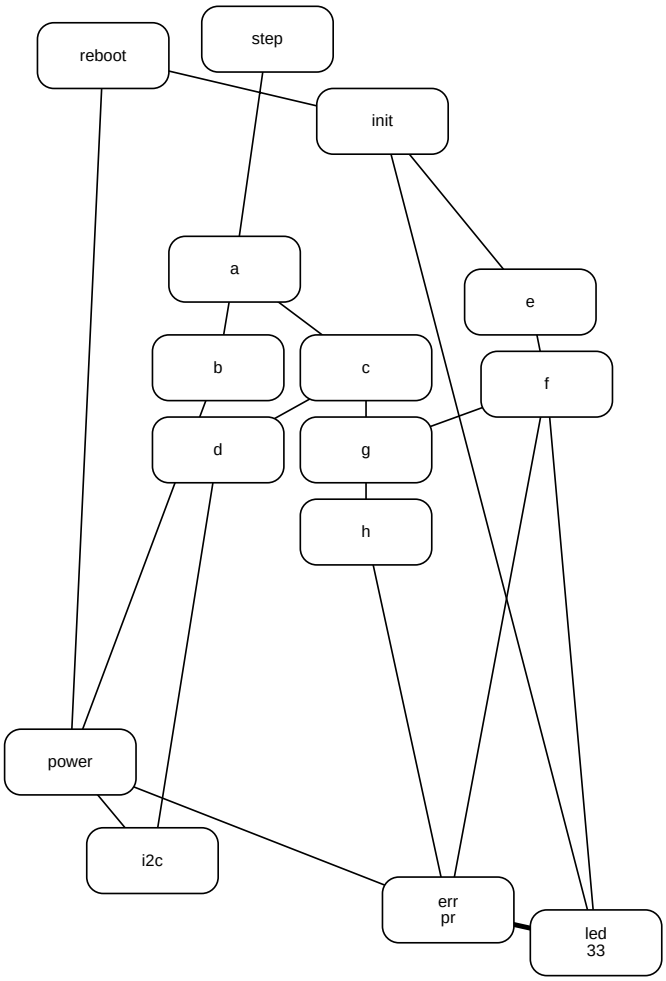

Fig. 3. The use relation at the C function level.

In order to apply a suitable process of abstraction of the ‘use’ relation we need the ‘part-of’ relation too. In this case, the ‘part of’ relation (which tells for each C function its component) is as given by the TEDDY diagram of Figure 4. This diagram shows the functions as boxes with rounded corners and the components (files) as rectangular boxes. Again, the essential information behind the ‘part of’ relation is just a binary relation; it can be stored as a file containing pairs of identifiers.

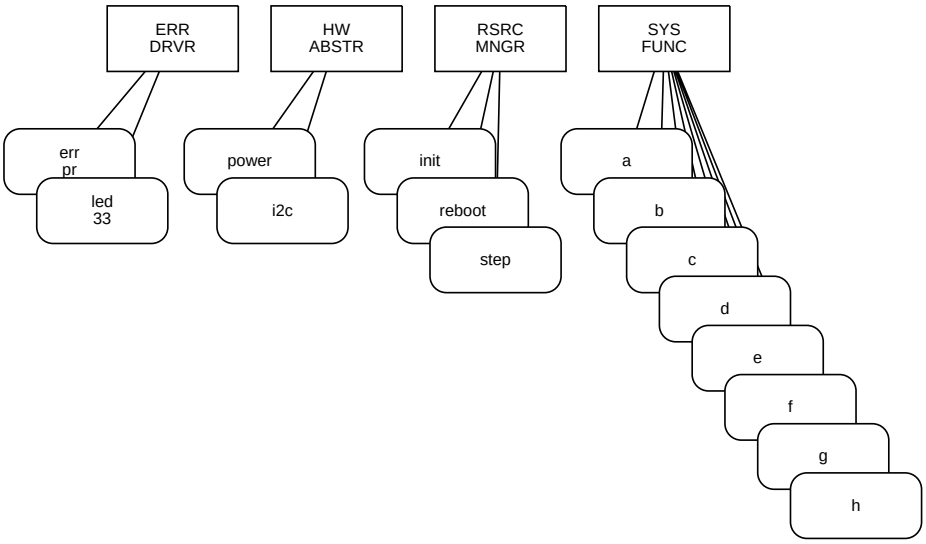

Fig. 4. 'Part of' relation between functions and components.

In particular, `err_pr` is part of `ERR_DRVR`. `led_33` is part of `ERR_DRVR`. `power` and `i2c` are part of `HW_ABSTR`. The function `init` is part of `RSRC_MNGR` and so are `reboot` and `step`. Finally, `a` to `h` are part of `SYS_FUNC`.

Now we have everything needed in order to compare Figure 2 and Figure 3. This is done by transforming the use relation from the C function level to a relation amongst software components. We call this transformation *lifting*: we move the relation from the level of the small objects (the C functions) to the level of the big objects (the software components). The key to this lifting is of course the 'part-of' relation of Figure 4.

In detail, the process of lifting goes as follows: from file `uses`, note that `err_pr` uses `led_33`. From the 'part of' relation (Figure 4), `err_pr` is in `ERR_DRVR` and `led_33` is in `ERR_DRVR` so `ERR_DRVR` uses itself, which is not so interesting. Conversely `led_33` uses `err_pr`, which adds no new information. Next, `power` uses `err_pr` and since `power` is in `HW_ABSTR` and `err_pr` is in `ERR_DRVR` we may conclude that `HW_ABSTR` uses `ERR_DRVR`. In the same way we find that `RSRC_MNGR` uses `SYS_FUNC`. When carrying along, the following relation is obtained; Assume that it is stored in a file called `Uses`.

```
HW_ABSTR  ERR_DRVR
RSRC_MNGR SYS_FUNC
RSRC_MNGR HW_ABSTR
RSRC_MNGR ERR_DRVR
SYS_FUNC  HW_ABSTR
SYS_FUNC  ERR_DRVR
```

This transformation of lifting, that is transforming a 'uses' relation to get a relation at a higher level, is a key concept for software architecture verification. It combines abstraction and advanced cross-referencing. The outcome is visualised in Figure 5.

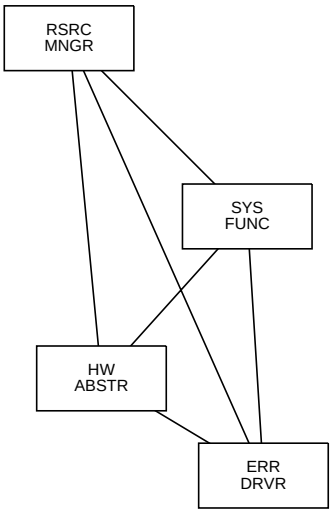

Fig. 5. The C-level use relation viewed at the component level.

There is also another route to arrive at the same information, namely by looking at the `#include` lines in the C code. When using a UNIX system, it suffices to type `grep include *.c` and after some trvial post-processing we have a component-level 'uses' relation. Fortunately it is the same relation; if the `grep`ed relation would have had additional lines, this would indicate that there are superfluous `#include` lines (which happens not to be the case). The converse cannot happen at all, because if each component is compiled separately, the compiler will check that all C functions used are listed in one of the included header (`.h`) files; here we assume that the header files themselves are correct, in the sense that they list all headers of functions of their components, and nothing else.

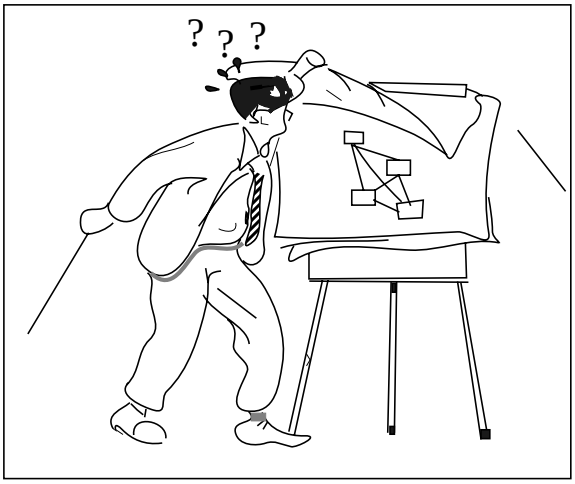

Fig. 6. The architect discovers the real system.

Now it is time for a comparison. The team calls the architect and visualises the contents of the `Uses` relation, as in Figure 5. But look, Figures 2 and 5 are not the same. What has happened?

5 Discussion

Of course everybody wants to know what went wrong and why Figures 2 and 5 are not the same. Maybe the architect will say that the programmers have turned his clean architecture into a mess and maybe the programmers will say that the architect has not enough knowledge about *real* software.

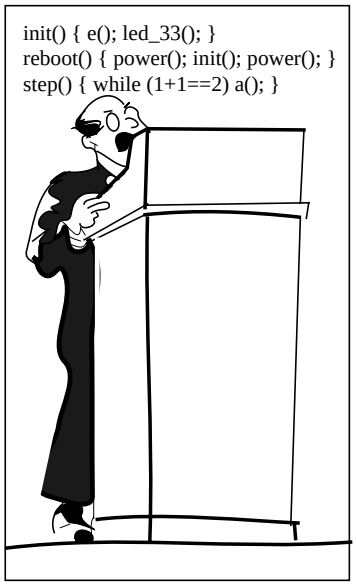

Fig. 7. The programmer explains why RSRC_MNGR must use ERR_DRVR.

But then the team realises that maybe there are no stupid mistakes at all and that maybe the problem is more subtle. One of the programmers explains an important observation first:

> The resource manager component RSRC_MNGR has three C functions, `init`, `reboot`, `step`, each of which can be viewed as an independent main program. Of these, `init` and `reboot` are tied to the hardware reset interrupt and the software interrupt (trap), whereas `step` is supposed to be called in an eternal loop. The architected component-level 'use' relation of Figure 2 has been made with the `step` function in mind. But everybody knows that for initialisation and rebooting one has to do some low level tricks every now and then. For example `reboot` has to call `power` and indeed, this causes a direct 'uses' line from RSRC_MNGR to HW_ABSTR. This explains why Figure 5 has more lines than Figure 2. And if you look at it this way, we have in fact respected the original architecture.

This seems a plausible explanation, but how can one be sure if this is really true? This demands a further analysis. There is a technique for analysing the code, namely by means of lifting. If one could (for the sake of the analysis) remove all 'uses' lines and all C functions not relevant to `step`, then one gets a thinned version of Figure 3. And then lifting could yield a thinned version of Figure 5.

It is clear that for small examples one can perform the lifting transformation manually, but for large systems automated support could be of great help. Let us assume that we have a piece of software which can do lifting of 'uses' relations and a few more related transformations. Let us call this piece of software a *relational calculator*. One of the main purposes of the present report is to explain the idea of a relational calculator as a technical option; we will not discuss any make-or-buy decisions, but we just assume we that we have it (also without the calculator many of its calculations are easily done by shellscripts or other ad-hoc programs).

A possible user-interface of the calculator is shown in Figure 8. The calculator works with binary relations, and just like a normal pocket-calculator it is important to show the outcomes of the calculations on some kind of display. Maybe one wants to choose between two kinds of display: a text-file format and a graphical format. The text-file display is trivial and the graphical display is already existing: use TEDDY or similar visualisation software.

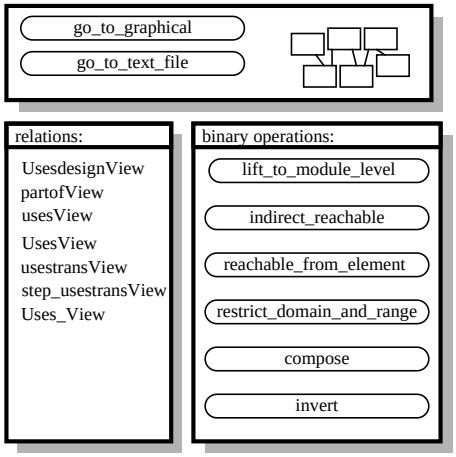

Fig. 8. The relational calculator.

The lower part of the user-interface consists of two halves. The left-hand side part allows for entering file-names for the storage and retrieval of relations. For example `Usesdesignview` is the name of Figure 2, `partofview` is the name of Figure 4, `usesview` is Figure 3, `Usesview` is Figure 5, and the remaining names belong to relation files which will arise soon if we continue the development of the toy architecture.

The right-hand side part allows has a number of buttons, one for each calculation which can be performed. For example in order to lift `Usesdesignview` of Figure 2 with the `partofView` of Figure 4, these two files must be selected and then the 'lift_to_module_level' button must be pressed.

6 Calculating a revised ‘Uses’ relation

Next the real analysis is performed, which begins with the removal of all ‘uses’ lines and all C functions not relevant to `step`. The first question is: ‘which functions are used by `step`?’ Somewhat more precise one wants to have all functions used by `step` as well as all functions used by such functions and so on. Therefore, one proceeds as follows: first calculate the transitive closure of the uses relation. This is exactly what the second button, labeled ‘indirectreachable’ is meant for. The outcome is visualised in Figure 9.

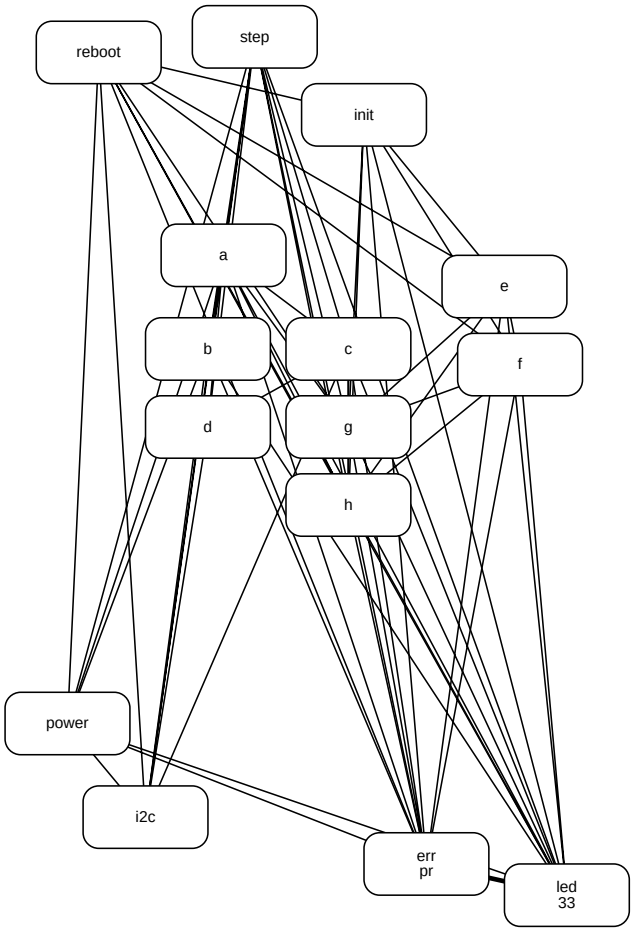

Fig. 9. Transitive closure of the ‘use’ relation on functions.

Next, this relation is restricted to those ‘uses’ pairs which begin with `step`, by means of the button ‘reachable_from_element’. The outcome is visualised in Figure 10.

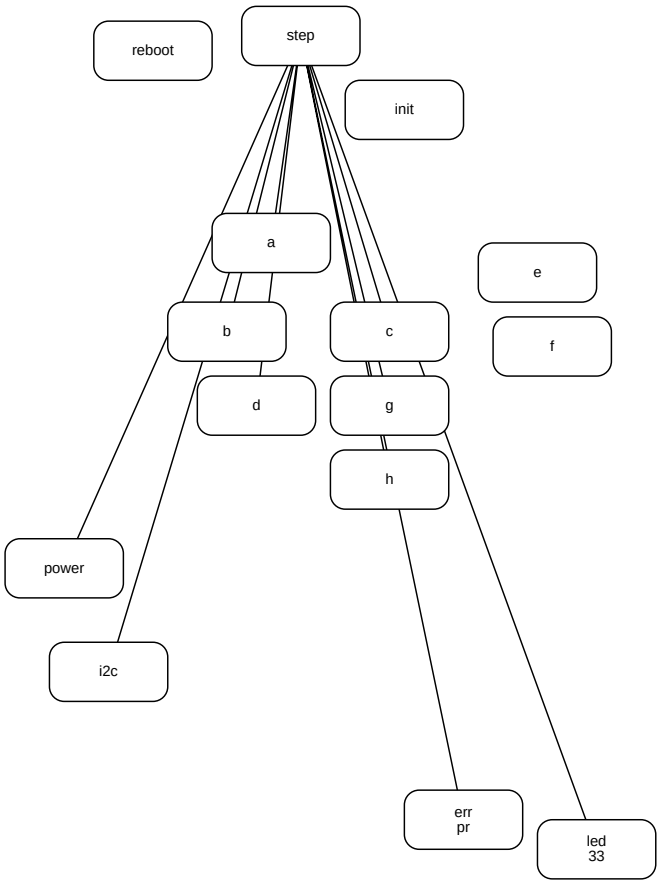

Fig. 10. Functions transitively connected to `step` function.

In fact this relation is not really interesting. The interesting things is the set of functions occurring in it. Let us assume that the calculator can work with sets instead of relations too (the buttons for that are not shown in Figure 8). Then the main result is the set:

```
a
b
c
d
err_pr
g
h
i2c
led_33
power
step
```

which means that `e`, `f`, `reboot` and `init` have been thrown out. This set must be used to restrict the ‘uses’ relation of Figure 3. Both the domain and the range of the latter ‘uses’ relation must be restricted. The button ‘restrict_domain_and_range’ makes the calculator perform this task. After that ‘lift_to_module_level’ must be applied to that result, which gives the main outcome of the analysis (Figure 11).

In this particular example, the same outcome would have been obtained by just removing `init` and `reboot` without removal of the functions which are not transitively connected to `step`, but in general, removal of such functions makes a difference, and therefore the latter part of the analysis may not be skipped. So now there is a main outcome, visualised in Figure 11.

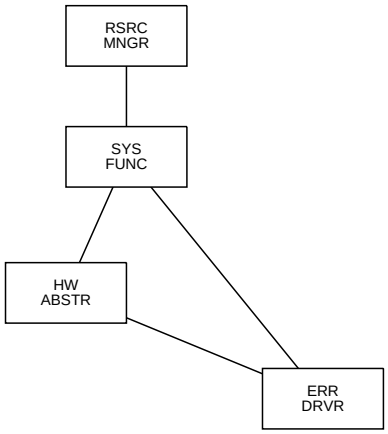

Fig. 11. Component level use relation with surpressed init and reboot calls.

Now it is clear that the programmer’s explanation is partially true, but not all of it. Indeed, most of the differences between Figures 2 and 5 are caused by `init` and `reboot`. But the ‘uses’ line from `HW_ABSTR` to `ERR_DRVR` is not explained that way. So the team must discuss this further. The team must arriving at one of two possible conclusions: either `HW_ABSTR` may not use `ERR_DRVR` and thus `power` should not invoke `err_pr` and should be modified; or it is really necessary that `power` invokes `err_pr` and thus the revised architecture of Figure 11 must be declared to be the official architecture. In this story, the latter alternative is chosen (see [4] for a nice dicussion of typical causes of such differences). By now, the team arrives at a common understanding of the architecture. The main lesson is that using some simple concepts about relations and a modest amount of automated support, several views on the system can be developed and compared.