Property-Part Diagrams: A Dependence Notation for Software Systems

Daniel Jackson and Eunsuk Kang
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

Some limitations of traditional dependence diagrams are explained, and a new notation that overcomes them is proposed. The key idea is to include in the diagram not only the *parts* of a system but also the *properties* that are assigned to them; dependences are shown as a relation not from parts to parts, but between properties and the parts (or other properties) that support them. The diagram can be used to evaluate modularization in a design, to assess how successfully critical properties are confined to a limited subset of parts, and to structure a dependability argument.

1 Introduction

A traditional dependence diagram consists of a node for each component, and an arc from A to B when component A depends on component B. Such a diagram has many uses – in reasoning, in guiding division of labor, in determining the impact of changes, in identifying reusable subsets, and so on. Every dependence is a potential liability, so reducing or eliminating dependences is a primary design goal, and the presence (or rather, the absence) of dependences is a key indicator of design quality. Our interest in dependence diagrams is primarily in their application to dependable systems, in determining which components are relied upon in the performance of critical functions.

In their standard form, however, dependence diagrams have two fundamental limitations that prevent them from being as widely applied as they deserve to be.

- Dependence is a boolean notion. A dependence between components is present or absent, and no account is taken of its extent or purpose. Consequently, a design move that replaces one major dependence with several minor ones – a frequent consequence of the use of design patterns – appears to be bad. Also, a component that uses other components only for relatively unimportant functions will appear to depend on them no less than on components used for critical functions. The dependence diagram will not show that the designer has successfully localized critical functions within a small part of the system.
- Dependence does not capture the notion of collaborating components. When two components work together to achieve an effect, their coupling cannot be shown except by making one dependent on the other, or making some other component dependent on both. As a result, dependence diagrams do not extend naturally to system-level interactions, involving a combination of software components and human operators or peripheral devices. At the root of this problem is the implicit assumption that any desired property of a system can be assigned to the interface of a single component that is responsible for ensuring it (with the help of the components on which it depends).

Many of the puzzles that arise when attempting to use dependence diagrams can be traced back to these limitations. For example, related to the first limitation:

- *Cycles in the dependence structure* – which are prevalent in object-oriented code, and the target of elimination in some approaches – are not well explained. If A depends on B, but B depends on A, does that mean that A depends indirectly on itself, and its correct functioning is based on a circular argument? An answer is found by qualifying the dependences. It is possible that A and B implement mutually recursive functions, in which case B will indeed depend on A for the very functionality that A offers in depending on B. But, more likely, A depends on B for one function, and provides a *different* function for the dependence of B.
- *Dynamic linking* is not easily accommodated. A hash table component, for example, will fail if the inserted key objects do not provide appropriate hashing and equality methods. It thus appears that a library component (the hash table) depends on an application-specific component (the one providing the key at runtime) suggesting, incorrectly, that the hash table is not reusable without the key. This conundrum is easily resolved by noting that the hash table depends only on very limited functionality of the key component – namely that it provide equality and hashing methods satisfying a standard contract.

And related to the second limitation:

- *Who depends on whom?* In a system that relies on a function being scheduled at a certain frequency, does the function depend on the scheduler or vice versa? Neither uses the other in a standard sense; the specification of the scheduler does not mention the effects of the tasks that are scheduled, not does the specification of the function mention who often it should be executed. The solution to this dilemma is simply to regard the function and scheduler as working together.

2 A New Notation

A *property-part diagram* is a graph of nodes and arcs, much like a traditional dependence diagram. The nodes, however, comprise two separate categories: *parts* and *properties*.

A part may be a software component or 'module', a physical component (such as a peripheral device), or a human operator or user. A property is a claim about observable behaviour, for example that some physical phenomenon occurs or does not occur, or that execution of some function has some given effect. We use the term 'property' rather than specification because a property may describe behaviour only partially, and may not be associated with one part alone.

An arc may point to a part or to another property, but always originates at a property. In Alloy:

```
sig Property {
   support: set (Property + Part)
}
sig Part {}
```

The properties and parts that a property p is directly connected to by outgoing arcs (p.support in Alloy) constitute the *support* of p: together, they are sufficient to establish it. This means that if the properties in the support hold, and the parts behave according to their descriptions (in the case of software modules, their code), then p will hold also. When the support of p includes a part or property q, p is said to *depend* on q.

The most common patterns are:

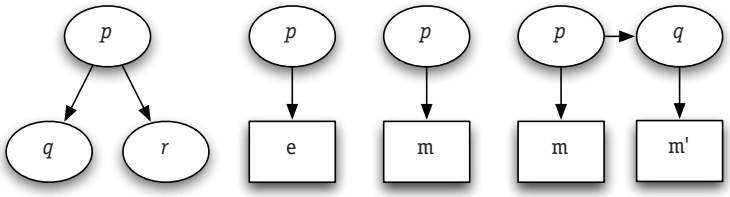

Figure 1: Standard patterns. From left to right: (a) property decomposition, (b) domain assumption, (c) satisfaction, (d) contingent use

- *Property decomposition* (Figure 1a). A property p depends on properties q and r, making the claim that p is implied by the conjunction of q and r.
- *Domain assumption* (Figure 1b). A property p depends on a single part e representing an aspect of the environment in which the software operates, making the claim that the environment has this property. The property is an *assumption* that should follow from the description denoted by the domain part. In practice, the description may be omitted, and the properties are then not formally justifiable, but are instead confirmed by experiment or by the judgment of domain experts.
- *Satisfaction* (Figure 1c). A property p depends on a single part m representing a module, making the claim that the part m satisfies the property p. One property may be established by more than one module, in which case they are claimed to establish it in combination, and a module may satisfy more than one property.

Note that the domain assumption and satisfaction patterns are drawn in the same way, even though their validation is very different, because the analysis required in both cases is logically the same: determining that some property of a part follows from its formal description.

- *Contingent use* (Figure 1d). A property p depends on a part m and another property q, where q depends on another part m′ used by m. In this case, p is claimed to follow from the combination of m and the property q. Unlike a property decomposition, this claim will not generally be established by showing an implication, but rather by using the property q about m′ in an argument that m satisfies p. This pattern may be counterintuitive at first; readers familiar with traditional dependence diagrams might expect the dependence edge to originate in the part m rather than the property p. Drawn this way, however, the diagram would fail to show that the use of m′ is specific to property p. The property-part diagram shows not only which property of m calls for a use of m′, but additionally which property of m′ is required in that use.

From the basic model, some auxiliary notions can be defined:

- The *exposure* of a property p is the set of parts reachable from p in one or more steps (in Alloy, p.^support & Part). These are the parts that are responsible for establishing the property, and whose breakage (or failure to satisfy properties) might undermine p.
- The *argument* for a property p is the subgraph rooted at p (in Alloy, p.*support <: support), namely that containing all parts and properties that p depends on directly or indirectly.
- The *impact* of a part m is the set of properties that a breakage or error in m might compromise (in Alloy, ^support.m & Property).

```
1   public class QuoteApp {
2       public static void main(String[] args) throws Exception {
3           Timer timer = new Timer();
4           for (String ticker: args)
5               timer.schedule (new Tracker (ticker), 0, 10000);
6       }
7   }
8   public class Tracker extends TimerTask {
9       String ticker;
10      int hi = 0; int lo = Integer.MAX_VALUE;
11      int MOVE = 10;
12      public Tracker (String t) {ticker = t;}
13      public void run () {
14          int q = Quoter.getQuote(ticker);
15          hi = Math.max(hi, q);
16          lo = Math.min(lo, q);
17          if (hi - lo > MOVE) {
18              System.out.println (ticker + ": now " + q + " hi: " + hi + ", lo: " + lo);
19              hi = lo = q;
20          }
21      }
22  }
23  public class Quoter {
24    static String BASE_URL = "http://finance.yahoo.com/d/quotes.csv?s=";
25    public static int getQuote (String ticker) {
26          URL url  = new URL(BASE_URL + ticker + "&f=l1");
27          String p = new BufferedReader(new InputStreamReader(url.openStream())).readLine();
28          return (int) (Float.valueOf (p) * 100);
29    }
30  }
```

Figure 2: Stock quote tracker (import statements omitted)

A primary aim of design is to reduce the exposure of critical properties to a small set of parts that are highly reliable (that is, satisfying their properties with high probability). The argument for a critical property should be small too – not only in the size of the graph, but also in the size of its parts and properties – since the size of the argument is likely to be strongly correlated to the cost of assuring adherence to the critical property. Parts that are unreliable (for example, human operators or complex physical peripherals) should preferably not have critical impacts.

3 Example: Tracking Stock Quotes

The program of Figure 2 implements a complete stock quote tracker that takes a list of ticker symbols and displays a message when the stock associated with a symbol moves more than some predefined amount. It works as follows:

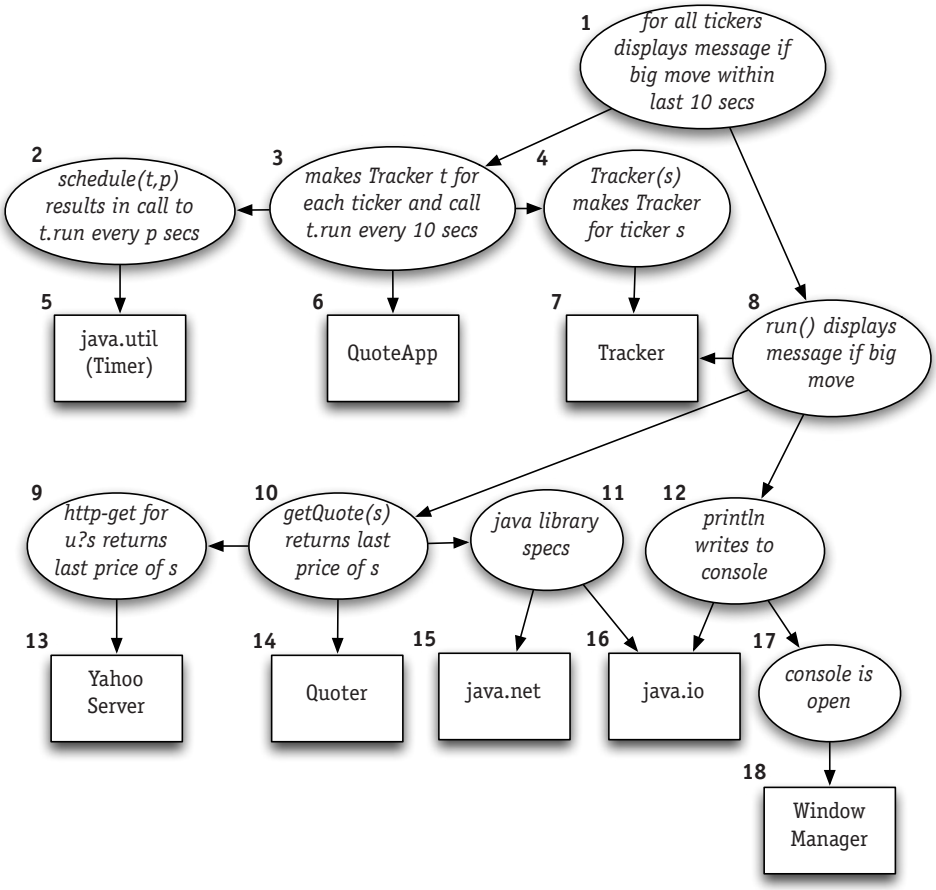

Figure 3: Property part diagram for stock tracker example (Figure 2)

- `QuoteApp` class. A Java timer object is created (line 3) for scheduling the downloading of quotes. For each ticker symbol presented as a command line argument (line 4), a tracker object is created and registed as a timer task with the timer for invocation every 10,000 milliseconds (line 5).
- `Tracker` class. A tracker object maintains as state high and low watermarks (line 10) on the value of the stock corresponding to the ticker symbol, and declares a constant (line 11) that defines how large a move in the stock spurs an alarm. Every 10,000 milliseconds, the Java timer calls the run method of the tracker, which causes the value of the stock to be obtained (line 14) and the watermarks to be updated. If the gap between the high and low watermarks exceeds the preset constant, a message is displayed on the console (line 18) and the watermarks are brought together so that a subsequent message will be generated only if another such move occurs.
- `Quoter` class. Stock quotes are obtained using the Yahoo quote server. A URL is constructed that includes the ticker symbol and a formatting string indicating what kind of quote is desired (line 26). Using Java's networking and I/O libraries, an HTTP get is then performed and

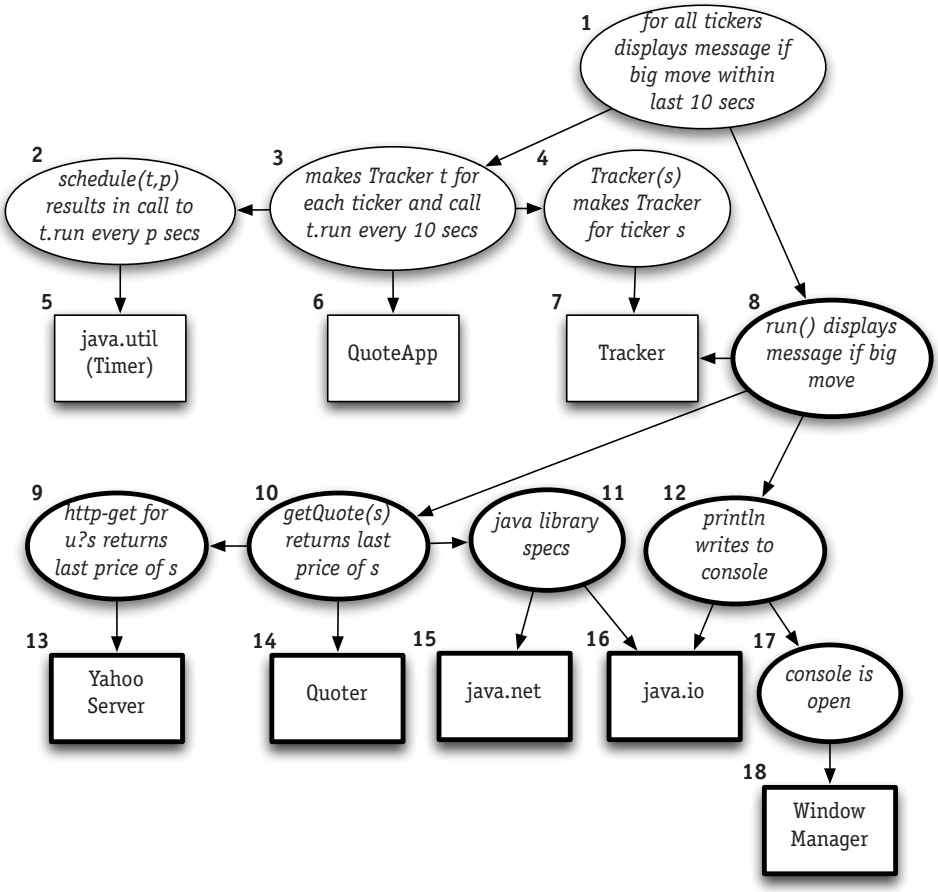

Figure 4: Exposure of property (8)

the returned page, which contains only the quote, is stored as a string (line 27). This string is parsed as a floating-point number, multiplied by 100 (to convert from dollars to cents), and then returned as an integer (line 28).

Although small and crude, this program exhibits three key features that are of interest for dependence analysis: use of libraries, a dynamic call-back mechanism; and reliance on an external service.

The property-part diagram (Figure 3) has eight parts: one for each of the three user-defined modules (6, 7, 14), three for Java libraries – for networking (15), for I/O (16), and for Timer and the classes it uses (5) – one for the Yahoo server (13), and one for the window manager of the local machine (18), whose role will be explained shortly.

The system's requirement (1) is shown as a property at the top of the diagram: that a message will indeed be displayed for any ticker included on the command line if the stock moves by the preset amount in the last 10 seconds. Since our focus is on the structure of the diagram and the relationship between properties and parts, we have not carefully formalized the properties themselves. For a critical system, this would be essential, to make sure the properties are clear and well-defined, and to enable mechanical reasoning. Formalizing the requirement would force

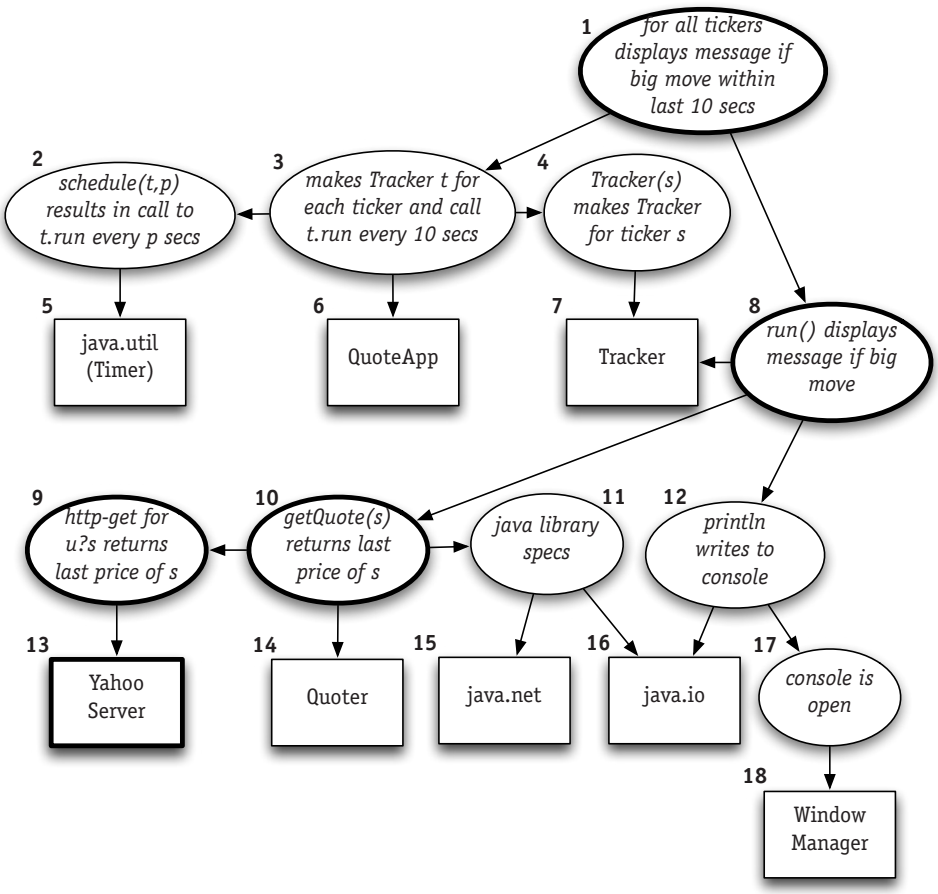

Figure 5: Impact of part (13)

us to decide exactly what is meant by a 'move in the last 10 seconds'; our implementation obtains only the current value from the Yahoo server, and thus would fail to catch large fluctuations occurring between checks.

The requirement (1) depends (*property decomposition* pattern) on two properties: that QuoteApp creates a tracker object for each of the ticker symbols whose run method gets called every 10 seconds (3), and that calling run displays a message if a move has occurred since the last time it was called (8). The first property (3) depends (*contingent use* pattern) on the code of QuoteApp (6) and on the properties that the parts it uses meeting their specifications (2, 4). These properties depend only on the parts they describe (*satisfaction* pattern), although in fact, as we shall see later, this is erroneous: the property that registering a timer task with schedule causes its run method to be called at the specified interval (2) actually depends on more than the code of Timer and its associated classes (5).

The property that calling run has the desired effect (8) is decomposed into the properties that the getQuote method of Quoter works (10) and that a call to println causes a message to be displayed on the console (12). The property that getQuote works (10) depends on the code of Quoter (14), on the Java libraries' meeting their specifications (11), and on the property that an HTTP get with an appropriately formed URL containing a ticker symbol will return the value of the cor-

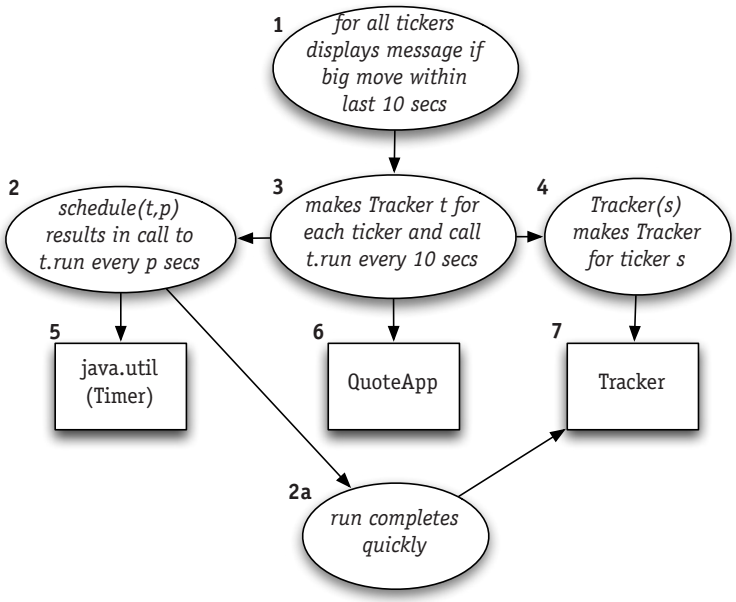

Figure 6: Corrected diagram showing assumption about timer tasks

responding stock (9). This last property depends, of course, on the Yahoo server (13) operating as advertised and being reachable in the network (not shown).

The println property (12) is more subtle than one might expect. It depends not only on the code of the relevant Java library (16), but also on a console window's actually being open (17). The println method writes to the standard output stream. Whether a write to this stream is displayed depends on the state of the window manager; if the console is not showing, the write will not be visible.

Exposure, argument and impact are easily read off the diagram by simply following paths. Thus the argument for property (8) – that run has the desired effect – is obtained by selecting all nodes reachable from it (Figure 4). The impact of a failure in the Yahoo server (13) is obtained by selecting all property nodes reachable backwards (Figure 5); not surprisingly, these include the requirement (1). Because this particular system is so simple and performs only a single function, these reductions are less useful than they would be in a larger system.

Constructing the dependence diagram and carefully reviewing each property and its dependences revealed, in addition to the issue regarding println mentioned above, a problem with the Timer class. Its official Java documentation warns

> *Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.*

In short, our property (2) does not depend on the code of Timer alone (5). In addition, it depends on a property we failed to state (shown as 2a in Figure 6): that the run method of the timer task completes quickly.

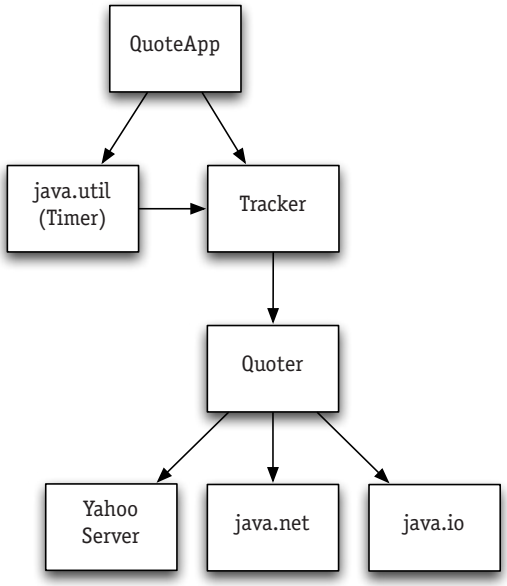

Figure 7: Traditional dependence diagram for stock tracker

For comparison, a traditional dependence diagram is shown in Figure 7. To construct such a diagram there needs to be at least an implicit specification of each component (so that A can be said to depend on B when the adherence of B to its specification is required for A's adherence). We therefore assigned each property to a part; the requirement was assigned to QuoteApp. This diagram is appealingly small, but it conveys very little information. The arrow from QuoteApp to Timer is due not only to the explicit calls to its methods, but just as importantly to the fact that QuoteApp uses Timer to make calls to the run method of Tracker. The arrow from Timer to Tracker is not a necessary consequence of Timer calling the run method of Tracker; it would be absent were it not for the fact that Timer relies on Tracker to return quickly in order to meet its specification.

This example reveals a subtlety of traditional dependence diagrams that is usually not recognized. The specification of Timer promises that calls to the timer task will occur with the given period *contingent* on them completing quickly. If we changed the specification to say that the frequency of execution is the given period *minus* the completion time, we would effectively shift the burden onto QuoteApp, and the dependence arrow from Timer to Tracker would no longer be shown! Dependences, in other words, are *property specific*: whether A depends on B cannot be determined unless we know what property A is expected to meet, and what properties other modules might be providing to A.

4 Related Work

Notions of dependence in compilation and parallelization have been widely studied, but notions of dependence in program and system design have received less attention from researchers.

4.1 Parnas's Uses Relation

The dependence diagram appears to have been invented by David Parnas. He defines the *uses* relation as follows [14]:

> *A uses B if there exist situations in which the correct functioning of A depends on the availability of a correct implementation of B*

This definition reveals how Parnas must have grappled with the complexities of dependences. Note that the correct functioning of `A` need not always depend on `B`; more can therefore be inferred from lack of dependence than from its presence. The significance of the word 'availability' is unclear; perhaps it was intended to allow `B` to be replaced by an equivalent component, or perhaps it emphasizes the need for not merely the existence of `B` but its availability in the context of use. Either way, the definition seems to imply that `B` is a specification but that `A` is an implementation. 'Correct functioning' of `A` is presumably with respect to its specification.

Parnas recognized that dependences do not necessarily follow procedure calls: that some calls result in no dependences, and that some dependences are present in the absence of explicit calls.

Despite their enormous value, dependence diagrams appear not to have been widely adopted, and are rarely taught at universities. At MIT, dependence diagrams have been emphasized in software engineering classes for 25 years, due to the efforts of Barbara Liskov and John Guttag who advocated them as a fundamental means for expressing and evaluating designs [11].

Extending dependence diagrams to object-oriented code is not straightforward, for the reason explained in the introduction (with the hash table example).

4.2 Class Diagrams

A *class diagram* mixes elements of an object model (how fields of one class point to another, and how classes implement interfaces or subclass other classes) with elements of a dependence diagram (how methods of one class call another). Design patterns are often depicted using class diagrams [5].

Class diagrams are, unlike dependence diagrams, easy to construct (both by programmers and tools), since they capture purely syntactic properties. But this limits their utility. When a field points to a generic class or interface, the class diagram will not show what class it is bound to at runtime. And when a class makes a call to another class through an interface, the actual class called will not be shown. With some amount of fudging, these problems can be overcome. One can replicate interfaces and abstract classes for their different contexts of use, and show, for each context, which concrete classes they are instantiated with. This enrichment of the class diagram makes their extraction from code far more challenging, however, and although tools [6, 13] have been designed to produce diagrams in this form, they may not scale well and even defining correct output turns out to be surprisingly tricky.

4.3 Design Structure Matrices

The design structure matrix (DSM) is a dependence graph represented as an adjacency matrix. It was introduced by Steward in the 1960s [18]. Various algorithms have been develop for automatically discovering structure in DSMs, for example, by topologically sorting the graph to assign modules to layers, and clustering modules into equivalence classes to eliminate cycles. Until recently, DSMs had been used primarily for streamlining manufacturing processes [4], but there has been increasing interest in using DSMs to capture modularity in design [1]. Lattix has developed a tool [16] (now imitated by offerings from other companies) that can extract a DSM from a large codebase, and help identify dependences that violate the intended architectural structure.

The notion of dependence in a DSM, especially for software, is not precisely defined. Tools tend to rely on syntactic dependences. Sullivan and his collaborators, however, have revisited Parnas's work in the context of DSMs [19], and, using design decisions as the nodes of the dependence graph, have given a formal characterization of dependence in terms of logical constraints [2]. Extending these ideas to graphs in which the nodes are components (rather than design constraints) remains to be done.

4.4 Goal Notations

A variety of diagrammatic notations [20, 10, 3] have been invented for depicting goals and their relationships. The initial motivation was to capture the rationale for system requirements, prior to the articulation of the requirements themselves.

Of these, Goal Structuring Notation (GSN) [10] is closest to ours, since it aims to represent the structure of a dependability argument. A GSN 'goal structure' is superficially very similar to a property-part diagram: a key requirement of the system is decomposed progressively, and related to knowledge of the software and its environment. In addition to goals, however, which are similar to our properties, a goal structure includes 'strategies' that represent the activities performed (proof, testing, etc) to establish the goals; and the focus of the structure is not the relationship between the goals and the components, but rather between the goals and the strategies that justify them. Thus the structure of the dependability argument is based not on the structure of the *system*, as in our approach, but rather on the structure of the *process*, which need not be related either to the structure of the system, or to the structure of the argument for its safety. For example [10], a top level goal 'logic is fault free' may be decomposed into 'argument by satisfaction of all requirements' and 'argument by omission of all identified hazards'.

Peter Henderson is currently working on a dependence model that represents argument structure directly. Its elements are claims and evidence, with dependences of claims on the claims and evidence that support them. His purpose is to build tool support to make it easier to navigate and maintain large arguments.

KAOS [3] is a goal notation that supports both behavioural goals (similar to our properties) and soft goals (which are 'satisficed' in Herbert Simon's sense), although, in contrast to GSN, these are usually about the product and not the process. Unlike property-part diagrams, KAOS supports 'or' decompositions, which are useful for showing design alternatives in a single diagram. Whether 'or' decomposition is needed to describe systems that make use of redundancy is not clear. KAOS is backed by a temporal logic and a catalog of refinement patterns which can be used to formally validate a design down to a low level. It seems that KAOS naturally represents the property decomposition and satisfaction patterns, but perhaps not the contingent use pattern.

4.5 Enriched Module Dependence Diagrams

The first author made an earlier attempt at overcoming the shortcomings of traditional dependence diagrams [7]. Modules were viewed as 'specification transducers', mapping specifications they provided (to clients) to specifications they required (as clients of other modules). An additional relation, in the spirit of architectural connection [12], represented the binding of provided 'ports' to required ports.

For example, if a module `B` provided a service `Q` so that a module `A` could provide a service `P`, the module `A` would map `P` to `Q`, and `B`'s provision of `Q` would be bound to `A`'s requirement for `Q`. In contrast, a property-part diagram would show the property `P` depending on module `A` and property `Q`, with a subtle shift in the interpretation of the properties: `P` and `Q` are no longer

descriptions of anonymously provided services, but assert that modules A and B provide these services. This is what allows the contingent use relationship to be captured without any outgoing dependences from A.

Although this earlier model solved some of the problems, it still required properties to be bound to modules: every property was a specification of a single module. This makes it unsuitable for system-level description, supporting (in Michael Jackson's terminology [8]) only *specifications* and not *requirements*. Moreover, a ternary dependence relation (specification-module-specification) is hard to draw, so in practice the diagrams added specification labels to dependence arcs between modules, but did not show the internal dependences that are essential for fine-grained tracking.

4.6 Frame Concern Diagrams

The property-part notation was inspired by Michael Jackson's problem frame diagrams [9], which show (requirements) properties explicitly as nodes. Jackson's 'frame concern diagram' shows the archetypal form of an argument for a particular class of property following from properties of the constituent domains. Seater, in his doctoral thesis [17], extended the problem frame diagram to an *argument diagram* that makes explicit the properties of the individual domains, but does not link them together in a dependence structure. Property-part diagrams grew out of his work, and can be seen as an attempt to layer a dependence relation on top of the argument diagram. An early version of the property-part diagram was in fact much closer to the argument diagram, as it included shared-phenomenon links between parts, but these links were dropped as they did not seem to be necessary.

5 Conclusion

Dependences are not innate properties of the parts of a system, but arise from the particular way in which a designer chooses to assign responsibilities. A part's functionality does not determine its responsibility. Just because module A calls module B does not mean that A guarantees to its clients the properties of B; it might instead promise only to call B. This is why we have abandoned the idea of explicit dependences between parts, preferring instead to relate parts and properties. So rather than asking 'does the application depend on the database?', we would ask 'does *this service* provided by the application depend on the database?' If the service is merely to execute certain queries and updates in response to user actions, it will not depend on the database. But if the service is to provide persistent storage and retrieval of data, it surely will.

Including properties in a dependence diagram is not optional; they were there all the time, albeit implicitly. Keeping them implicit had two disadvantages that property-part diagrams overcome: it obscured the rationale for the dependences, and it prevented a more fine-grained analysis that allows different properties to be traced independently. In an analysis of a proton therapy machine, the critical property that pressing the emergency button inserts a beam stop was found to have an exposure very much smaller than the entire system, but still larger than one would ideally want [15]; a traditional dependence analysis would have produced a graph that was close to fully connected.

Much work remains to be done to understand and refine the property-part diagram, to understand what kinds of inferences can be made from the diagram, and how it might be checked mechanically. The claim, for example, that a change to a part can only affect the properties within its impact clearly will not hold if a change to the 'alphabet' of the part is permitted, so that it

engages in entirely new phenomena. The property-part diagram makes it easy to see (unlike a traditional dependence diagram) what properties a replacement part should have, but less clear what other parts might be impacted by a replacement.

We are also interested in understanding when dependence of a property on multiple parts induces a coupling between them, and in reconsidering the (unjustified?) assumption that a module should be regarded as vulnerable to changes in the modules it uses but not to changes in the modules that use it. We plan also to investigate the notion of information hiding, to see how it might be accommodated, perhaps as properties over uninterpreted functions.

Acknowledgments

Derek Rayside made many suggestions about our ideas and their presentation. Axel van Lamsweerde helped us relate our work to KAOS, and has been generous in his encouragement, and tolerant of our slow recognition that dependences arise from goals.

The first author is grateful to Michael Jackson in whose honour this workshop was held, not just for his work on problem frames (whose influence on this work is pervasive), but for many years of wise advice and thought-provoking technical discussions. He continues to be an inspiring model (in the iconic rather than analogic sense) of what it means to be an engineer, a thinker, and a mensch. Ad me'ah ve'esrim shanah!

References

1. Carliss Baldwin and Kim Clark. *Design Rules: The Power of Modularity*. MIT Press, 2000.
2. Yuanfang Cai and Kevin Sullivan. Modularity Analysis of Logical Design Models. *Proceedings of 21th IEEE/ACM International Conference on Automated Software Engineering*. Tokyo, Japan, 2006.
3. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, 1993.
4. Steven D. Eppinger. Innovation at the Speed of Information. *Harvard Business Review*, Vol. 79, no. 1, pp. 149-158, January 2001.
5. Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.
6. Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering*, February 2001.
7. Daniel Jackson. Module Dependencies in Software Design. *Post-workshop Proceedings of the 2002 Monterey Workshop: Radical Innovations of Software and Systems Engineering in the Future*. Venice, Italy, 2002. Springer Verlag, 2003. Available at: http://sdg.csail.mit.edu/publications.html.
8. Michael Jackson. *Software Requirements and Specifications*, Addison-Wesley and ACM Press, 1996.
9. Michael Jackson. Problem Frames: Analysing and Structuring Software Development Problems, Addison-Wesley, Boston, Massachusetts, 2001.
10. Tim Kelly and Rob Weaver. The Goal Structuring Notation – A Safety Argument Notation. *Proceedings of the Dependable Systems and Networks Workshop on Assurance Cases*, 2004.
11. Barbara Liskov and John Guttag. Abstraction and Specification in Program Development. MIT Press, 1986.
12. Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, San Francisco, CA, 1996.
13. Robert O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD Thesis, Technical Report CMU-CS-01-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 2000.

14. David Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions On Software Engineering*, Vol. SE-5, No. 2, March 1979.
15. Andrew Rae, Daniel Jackson, Prasad Ramanan, Jay Flanz, and Didier Leyman. Critical feature analysis of a radiotherapy machine. *Reliability Engineering & System Safety*, Volume 89, Issue 1, Elsevier Science, July 2005, Pages 48–56.
16. Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems (OOPSLA 2005). (PDF).
17. Robert Morrison Seater. *Building Dependability Arguments for Software Intensive Systems*. PhD Thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA, February 2009.
18. Donald Steward. Design structure system: A method for managing the design of complex systems *IEEE Transactions on Engineering Management*, 28:33, 71–74, 1981.
19. K.J. Sullivan, W.G. Griswold, Y. Cai and B. Hallen. The Structure and Value of Modularity in Software Design. *Joint Proceedings of the European Software Engineering/ACM SIGSOFT Foundations of Software Engineering Conference (ESEC/FSE)*, Vienna, September 2001.
20. Eric Yu. Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering . *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering* , Washington, DC, pp. 226-235, 1997.