Usable-by-Construction

Steve Reeves

Department of Computer Science, University of Waikato

Abstract. We propose here to look at how abstract a model of a usable system can be, but still say something useful and interesting.
We take the view that when we claim to be designing a usable system we have, at the very least, to give assurances about its usability properties. This is a very abstract notion, but provides the basis for future work, and shows, even at this level that there are things to say about the (very concrete) business of designing and building usable, interactive systems.
In this note, we introduce the idea of usable-by-construction, which adopts and applies the ideas of correct-by-construction to (very abstractly) thinking about usable systems.
We outline a set of construction rules or tactics to develop designs of usable systems by picking a few, and we also formalize them into a state suitable for, for example, a proof assistant to check claims made for the system as designed.
In the future, these tactics would allow us to create systems that have the required usability properties and thus provide a basis to a usable-by-construction system. Also, we should then go on to show that the tactics preserve properties by using an example system with industrial strength requirements. And we might also consider future research directions.

Keywords: usability, usable-by-construction, correct-by-construction

1 Introduction

The position taken in this note is that more abstraction in the methods for modelling and designing interactive systems would be a good thing.

In previous work over the last decade or so [1] we have taken a fairly abstract view of interactive systems (which itself has been criticised in some quarters for being *too* abstract) and shown how systems can be modelled using that view, and also how we can move towards more concrete models from that abstract view. Here we try to go to the abstract extreme (i.e. even more abstract that what has already been criticised!) just to see what is possible.

The aim of this note is to introduce the idea of *usable-by-construction*. This, in essence, takes the ideas of *correct-by-construction* that formed much of the work of Dijkstra [3], Gries [4] and many others, and applies them to the problem of usable systems. In particular we want to see how we can develop a set of construction rules or tactics which allow us to build designs of usable systems without having to perform, say, *post hoc* verification on the constructed system. That is, we want tactics that can build *only* usable systems: any system built with the tactics *will necessarily* have the required usabilty properties simply due to the nature of the construction tactics themselves.

Since we are trying to hit the spot between maximum abstraction and maximum simplicity, we leave the question "what exactly do you mean by usable?" unanswered. If pressed for an answer we would say that our abstraction allows *any* answer to that question that you personally are happy to accept. Here we are totally agnostic about what usable means. Think of the definition of usable as being a parameter of our rules.

The aims of these techniques can be summarised by saying that we are trying to bring some good engineering principles to bear, namely:

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2503/paper1_3.pdf

- model a system before going to the expense of building it;
- use maths to check that the system is fit for purpose;
- for a good design, don't get concrete too early or we'll lock ourselves into design choices too soon;
- build a system, not by adding features at will and on the fly—but in a controlled, structured way as this keeps complexity under control.

These points are usually summarised as—

- modelling
- maths
- abstraction
- compositionality (i.e. start with a small set of very simple, basic actions and then have a few simple rules which given already constructed pieces allows us to compose them into new, larger systems).

2 Basic definitions

We assume that each system is made up of components (which might be people, computers, software systems and so on right down to simple widgets) and connections between them. A connection represents a use of one component by the other. That is almost all we say, so we are going for maximum generality here.

In order to define this precisely, we make the following definitions.

Definition 1. *A* system *is* $\langle C, N\rangle$*, where* C *is a component set* $\{c_1, c_2, ..., c_n\}$*, and* N *is a connection set,* $\{n_1, n_2, ..., n_m\}$*, where each* $n_j = \langle c_i, c_k\rangle$ *for some* $c_i, c_k \in C$*.*

This definition is a starting point, but it clearly too general to be very interesting. With our problem in mind, namely designing usable systems, we introduce two further ideas. Firstly, we have a subset of C, called I, which is a set of components which can be interacted with. Secondly, each interactive component, $i \in I$, is associated with its own set of components that are allowed to interact with it, A_i, which we refer to as *an interactive component's allowed set.*

For a component to be allowed access to an interactive component, that component needs to be added into the allowed set of that interactive component. These sets of "allowed" components can be thought of as expressing propositions about the system—that, assuming that allowing the components access to the interactive components is sensible or allowable (a decision of the designers, perhaps based on experimental or past design knowledge or experience, *etc.*), the system as whole is accepted as usable. The sets can be thought of sets of permissions too: if a component is in an interactive component's allowed set then that component has permission to use that interactive component while keeping the whole system ajudged as usable.

We refer to systems with these two additional sets as *acceptable systems.*

Definition 2. *An* acceptable system $\langle C, N, I, A\rangle$ *extends a system* $\langle C, N\rangle$*, where* $I \subseteq C$ *is a set of components that are interactive and* A *is a family of sets of components* $\bigcup_i A_i$*, where for* $i \in I$*,* $A_i \subseteq C$*.* A_i *is a set of components allowed (for agreed reasons) to use the interactive component* i*.*

Now we need to think of how we can combine such systems, via *tactics*, preserving usability—so we are going for a compositional approach here. These tactics are connect, disconnect, create, and delete. Given the space restrictions we look at just two.

In what follows we make the (surely benign) assumption that every component has access to itself.

3 Tactics

We introduce various rules (which we call tactics) for building systems from smaller systems, and ultimately from single components, with the aim that the result of each tactic, assuming we start with usable systems, will necessarily be usable systems. Note that this system is very liberal, in the sense that connections between systems containing interactive components are generally allowed, but we also without exception keep track of what components have access to what interactive components. So, usability here comes about because we are completely open and honest about who has access to what.

Recall that our overall plan is to have simple rules to start with, see how far we can get, then introduce further rules carefully to get us closer to our aim, without disturbing (as far as possible) the simplicity of the modelling.

To help the presentation we introduce the notion of a path:

Definition 3. *Given a system $\Sigma = \langle C, N, I, A\rangle$, a path exists between components c and c' in C, which we write as $c \rightsquigarrow_\Sigma c'$, iff either:*

1. *c is connected to c', i.e. $\langle c, c'\rangle \in N$, or*
2. *there is some $d \in C$ such that $c \rightsquigarrow_\Sigma d$ and $\langle d, c'\rangle \in N$*

We drop the subscript where the context allows, and we iterate the notion of path, so that, for example, $c \rightsquigarrow d \rightsquigarrow e$ abbreviates $c \rightsquigarrow d \wedge d \rightsquigarrow e$. Also, by $\forall\, c \in a \rightsquigarrow b.P$ we mean that all components on the path between a and b satisfy the predicate P.

3.1 Connect Tactic

The connect tactic adds a connection between two components of a system under certain conditions. (We can also see this rule as allowing us to join two such systems together via the joined components. In the full treatment we have rules to deal with this eliding of meaning.) This is done by creating an edge between two components.

Definition 4. *Connects to*
If $\Sigma = \langle C, N, I, A\rangle$ then making a new connection between c_a and c_b in C means creating $\Sigma' = \langle C, N \cup \{\langle c_a, c_b\rangle\}, I, A\rangle$, and $\langle c_a, c_b\rangle \notin N$ with the condition that:

$$\forall\, i \in I.\forall\, c, d' \in C.\forall\, d \in c_b \rightsquigarrow d'.\forall\, c' \in c \rightsquigarrow c_a.d \in A_i \Rightarrow c' \in A_i$$

The condition here simply says that any elements on any path $c \rightsquigarrow c_a$ that gets joined to a path $c_b \rightsquigarrow d$, where this path contains elements allowed access to any interactive component, must already be allowed access to those same interactive components. This is, of course, a very general condition, i.e. it means that almost every component in any system might have to be allowed access to almost all interactive elements in that system. For the moment, though, we are concerned with giving rules which preserve usability.

Some simple results follow directly from this definition. For example, given some system containing c and i, if $c \rightsquigarrow i$ for i an interactive component, then $c \in A_i$, which is itself a special case of the more general result that if $c \rightsquigarrow d$ and $d \in A_i$ for some interactive component i, then $c \in A_i$.

3.2 Disconnect Tactic

The disconnect tactic removes a connection between a source component and a target component. This is done by removing a connection between two components in the connection graph. Structurally, a use of the target component is revoked from the source component. (It may be that this gives us two completely disconnected sets of components.)

The disconnect tactic requires two parameters, i.e. the source component and the target component.

Definition 5. *Disconnects from*
If $\Sigma = \langle C, N, I, A\rangle$, *then disconnecting* $c_a \in C$ *from* $c_b \in C$ *means deleting a connection,* $\langle c_a, c_b\rangle \in N$, *and creating* $\Sigma' = \langle C, N', I, A\rangle$, *where* $N' = N \setminus \langle c_a, c_b\rangle$

3.3 Grant Tactic

The grant tactic allows a component to have access to an interactive component. This is done by adding a component to the set of components that are allowed to have access to the interactive one.

Definition 6. *Add to Granted set*
If $\Sigma = \langle C, N, I, A\rangle$, *then allowing* c_i *access to* c_j *means creating* $\Sigma' = \langle C, N, I, A'\rangle$, *where* $A' = A \oplus \{c_j \mapsto A_{c_j} \cup \{c_i\}\}$ *iff* $c_j \in I$. *Otherwise,* $A' = A$.

The grant tactic loosens the restriction of the connect tactic. Given this loosening, we have to be careful about what we claim for a system constructed with our tactics. In particular, we have to ensure that the *assumption* that the family of sets of components A have been allowed access to interactive components is made explicit in any *guarantees* we give about the system constructed. So, if we have constructed the system $\langle C, N, I, A\rangle$ then we have to say that:

> *assuming* that the family of sets of components A have correctly been allowed to access certain interactive components, *then we guarantee* that the constructed system is usable

which we might write formally as[1]:

$$A \vDash < C, N, I >$$

So, when we “hand” a system to a client, we hand them something that, as long as they use it in the right context, i.e. a context in which the assumption is satisfied, i.e. a context where it is permitted to allow the interactions that have been allowed to the components that they have been granted to, then we guarantee that the system is usable. Stated alternatively (as we mentioned above in a previous section) we can think of A as recording the permissions for accessing interactive components, so $A \vDash < C, N, I >$ is saying that assuming that we are happy to allow the permissions as given in A, then the system as described by $< C, N, I >$ is usable.

It is useful to think of this notation as stating a *contract* between modeller and client. It makes plain exactly what is being assumed (A), and exactly what may then be taken to be a usable system ($< C, N, I >$) *under those assumptions.*

[1] The symbol $\vDash$ is borrowed from formal logic, and there it is usually called a turnstile. These are conventionally used to seperate assumptions from conclusions, hence our use of the symbol here

4 The rules

In this section we re-state the rules above in a more formal setting. This does two things: it makes clear exactly what is being assumed and what is being concluded; and it allows us to move towards a *logic* for usable systems, which itself (via a proof assistant, theorem-prover or other programmed form of the rules coupled with some search strategy) leads to algorithmic construction of usable systems. We expand on these points as follows:

Rules are good because:

- they allow goal-directed construction (examples below), because a rule read backwards (or upwards) tells us what we must show in order to have the conclusion we desire;
- they are good for design in general since such rules—
  - provide some guidance (the shape of the desired system determines, to some extent, the rules that must be used to build it);
  - suggest a pattern to look for (the use of a restricted set of rules soon gives rise to repeated patterns of development, which then gives rise in turn to derived rules which usefully encode recurring, common patterns);
  - ease explanation (the structure suggest the form and content of answers to the question: how was this system constructed, and why is it usable?);
  - promote understanding (see: all the above);
- although we trade away complete flexibility (*i.e.* on the fly, *ad hoc* design), we gain better understanding, structure, robustness *etc.*
- they take us towards a method for checking and building systems: the rules, being formal, can easily be read as algorithms.

The rules will (following standard methods) follow from the definitions of the tactics that we have given earlier in the paper.

4.1 Disconnect rule

Consider the disconnect tactic, and recall its definition:

> *Disconnects from*
> *If $\Sigma = \langle C, N, I, A\rangle$, then disconnecting $c_a \in C$ from $c_b \in C$ means deleting a connection, $\langle c_a, c_b\rangle \in N$, and creating $\Sigma' = \langle C, N', I, A\rangle$, where $N' = N \setminus \langle c_a, c_b\rangle$*

This gives us two rules: one "introduction" rule for moving from a system with a certain connection to one where a disconnection (*i.e.* removal of that certain connection) has happened:

$$\frac{A \vDash \langle C, N, I\rangle \qquad \langle c_a, c_b\rangle \in N}{A \vDash \langle C, N \setminus \langle c_a, c_b\rangle, I\rangle}\ \textit{disconnect}^+$$

and an "elimination" rule which, given a system that has had a disconnection performed on it, can "reverse" this (somewhat artificially perhaps, but it is a rule we gain nonetheless):

$$\frac{A \vDash \langle C, N, I\rangle \qquad c_a, c_b \in C}{A \vDash \langle C, N \cup \{\langle c_a, c_b\rangle\}, I\rangle}\ \textit{disconnect}^-$$

4.2 Connect rule

$$\frac{A \vDash \langle C, N, I\rangle \quad c_a \in C \quad c_b \in C \quad \begin{array}{c}\forall\, i \in I. \forall\, c, d' \in C. \forall\, d \in c_b \leadsto d'. \forall\, c' \in c \leadsto c_a. d \in A_i \Rightarrow c' \in A_i \\ \vdots \\ d \in A_i \Rightarrow c' \in A_i\end{array}}{A \vDash \langle C, I \cup \langle c_a, c_b\rangle, I\rangle}\ \mathit{connect}^+$$

Note that we may have to use *granted* before these rules in order that we can connect to an interactive component.

4.3 Granted

The point of the granted set for some interactive component is that it records our actions in granting access since this forms part of our contract. Recall that $A \vDash \langle C, N, I\rangle$ simply means that assuming we have correctly (acceptably) granted access as recorded in A, then the system is usable.

$$\frac{A \vDash \langle C, N, I\rangle \qquad i \in I \qquad c \in C}{A \oplus \{i \mapsto A_i \cup \{c\}\} \vDash \langle C, N, I\rangle}\ \mathit{granted}^+$$

5 Tiny examples

5.1 $\mathit{connect}^+$ and $\mathit{disconnect}^+$ are inverses

If we make a new connection and then disconnect immediately afterwards we get back to the same system that we started with.

As we can see from this tiny proof in Figure 1, under the assumptions that we start with the system $A \vDash \langle C, N, I\rangle$ where $c_a \in C$ and $c_b \in C$ and assuming all the components involved respect the accessability requirements of i as summarised in A_i (which is what the fourth—big–premise is saying), which allow a connection to happen, then undoing the connection results in the system $A \vDash \langle C, N, I\rangle$ we started with.

This is about the simplest general property we would expect to be provable if the rules have correctly captured the intended meaning of such systems and their properties. Showing that such proofs are possible is part of the usual validation process for any formalisation.

We also give the proof required to construct a small part of a system, where we prove that the fragment where G and H connect with interactive component A, as in Figure 5, is constructable.

6 Usage

Apart from giving us a logic for reasoning about systems, these rules can help guide us in the construction of systems. For example, say we had to construct a system of the form $A \vDash \langle C, \{\langle d_a, d_b\rangle, \langle c_a, c_b\rangle\}, I\rangle$, then $\mathit{disconnect}^-$, read "upwards" tells us that we must show how to construct a system of the form $A \vDash \langle C, \{\langle d_a, d_b\rangle\}, I\rangle$ and also show that $c_a, c_b \in C$. So, starting with a desired system, and using the rules upwards, we get some guidance, via pattern matching the system we want with the conclusions of the rules, as to how to build it. If we continue this process along all branches of the proof tree that we thus construct until we reach its tips, which require no further proof (for example the system $\emptyset \vDash \langle\emptyset, \emptyset, \emptyset\rangle$ is trivially constructible and usable; it is like the *zero* of usable systems) then by reading the proof tree "forwards" we see both how to construct our desired system and have a proof that it is usable.

Our new logic (for usable systems) inherits the internal consistency of the underlying logic since the new logic was produced via conservative extension, and so it is sound, which means that any system constructed via the rules is guaranteed to be usable.

$$\dfrac{\dfrac{A \vDash \langle C, N, I\rangle \quad c_a \in C \quad c_b \in C \quad \begin{array}{c}\forall\, i \in I.\, \forall\, c, d' \in C.\, \forall\, d \in c_b \rightsquigarrow d'.\, \forall\, c' \in c \rightsquigarrow c_a \\ \vdots \\ d \in A_i \Rightarrow c' \in A_i\end{array}}{A \vDash \langle C, N \cup \langle c_a, c_b\rangle, I\rangle}\ \mathit{connect}^{+} \qquad \dfrac{}{\langle c_a, c_b\rangle \in N \cup \langle c_a, c_b\rangle}\ \mathit{by\ set\ theory}}{A \vDash \langle C, N, I\rangle}\ \mathit{disconnect}^{+}$$

Fig. 1: Proof that connecting followed by disconnecting leaves a system unchanged

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vDash \langle \emptyset, \emptyset, \emptyset\rangle}\ \mathit{axiom} \quad \dfrac{}{A \notin \emptyset}\ ST}{\{A \mapsto \{A\}\} \vDash \langle \{A\}, \emptyset, \{A\}\rangle}\ \mathit{create}_2^{+} \quad \dfrac{}{G \notin \{A\}}\ ST}{\{A \mapsto \{A\}\} \vDash \langle \{A, G\}, \emptyset, \{A\}\rangle}\ \mathit{create}_1^{+} \quad \dfrac{}{H \notin \{A, G\}}\ ST}{\{A \mapsto \{A\}\} \vDash \langle \{A, G, H\}, \emptyset, \{A\}\rangle}\ \mathit{create}_1^{+} \quad \dfrac{}{A \in \{A\}}\ ST \quad \dfrac{}{G \in \{A, G, H\}}\ ST}{\{A \mapsto \{A, G\}\} \vDash \langle \{A, G, H\}, \emptyset, \{A\}\rangle}\ \mathit{granted}^{+}$$

Fig. 2: Fragment for proof in Figure 5

$$\dfrac{\dfrac{\mathit{Insert\ Figure}\ \mathbf{2} \quad \dfrac{}{A \in \{A\}}\ ST \quad \dfrac{}{H \in \{A, G, H\}}\ ST}{\{A \mapsto \{A, G, H\}\} \vDash \langle \{A, G, H\}, \emptyset, \{A\}\rangle}\ \mathit{granted}^{+} \quad \dfrac{}{A \in \{A, G, H\}}\ ST \quad \dfrac{}{G \in \{A, G, H\}}\ ST \quad \dfrac{}{A \in \{A, G, H\} \Rightarrow G \in \{A, G, H\}}\ ST}{\{A \mapsto \{A, G, H\}\} \vDash \langle \{A, G, H\}, \{\langle G, A\rangle\}, \{A\}\rangle}\ \mathit{connect}^{+}$$

Fig. 3: Part of construction of system using A, G and H in Figure 5

$$\dfrac{\mathit{Insert\ Figure}\ \mathbf{3} \quad \dfrac{}{A \in \{A, G, H\}}\ ST \quad \dfrac{}{H \in \{A, G, H\}}\ ST \quad \dfrac{}{A, G \in \{A, G, H\} \Rightarrow H \in \{A, G, H\}}\ ST}{\{A \mapsto \{A, G, H\}\} \vDash \langle \{A, G, H\}, \{\langle G, A\rangle, \langle H, G\rangle\}, \{A\}\rangle}\ \mathit{connect}^{+}$$

Fig. 4: Construction of system using A, G and H in Figure 5

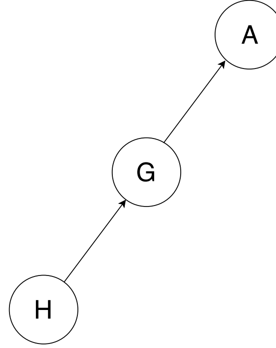

Fig. 5: A tiny system

7 Introducing usable components

Next we can introduce *usable* (not merely "interactive") components which some components in I might be, or that can play the role of, *wrappers* that *guard* the rest of the system against non-usability to make I components usable, or perhaps make C components usable (by wrapping and/or guarding).

Then we have a condition that simplifies the structure by making the A_i smaller by shrinking the family of sets A: if some components have been proved to be usable, or been proved the protect the system against undesirable (and otherwise non-usable components—i.e. components that we might want in the system because of some very useful properties they have, but which are otherwise appallingly non-usable) by wrapping them up or filtering out or restricting their undesirable features, then we can take away some of the assumptions (which is what A is essentially giving us) and get a simpler design.

Here is a sketch of something we might introduce as a sort of healthiness condition which pares down large permission sets into smaller ones once the usable components have been introduced:

Lemma 1. *Healthiness due to usable components*
Consider a system Σ of the form $A \vDash< C, N, U, I >$. Assume one of the components u in C is designated as a usable component (as in the discussion above). Then we have an equivalently usable system Σ' of the form $A' \vDash< C, N, U \cup \{u\}, I >$ and A' is related to A as follows:

1. *for all interactive components $i \in I$, all components of A that are* **only** *on paths that are prefixes of paths from u to i are removed from A_i;*
2. *for all interactive components $i \in I$, all components of A on any paths that start at u and do not go through i are removed from A_i*

The set of components that that remains as A_i, due to the clauses above, form A'_i.

8 Conclusions

The work recounted here, which centres around a need to consider the ways of designing and constructing a usable system made of various components, has several properties:

– It allows an abstract characterisation of a usable system;

– It has logical rules that allow for checking and construction;
– Any complicated system may be built in terms of simpler ones using a small set of operations;
– We may use it as a basis for deriving further construction operations.

The rules here are, of course, tedious to use by hand (as the examples show), but we can express the rules very directly in the various proof assistants available (*e.g.* PVS [5]) or perhaps program them in a language that already deals with search, like Prolog. Then by giving the desired system as conjecture to the proof assistant or a goal to a Prolog program, it can be used to (mainly automatically, given the simplicity of our rules) then construct a proof that the system is usable. For realistically large systems, with hundreds of components, this would be an important feature.

Another way of seeing this utility is to acknowledge that one the problems with realistically large systems is keeping track of dependencies (like our "allowed access to interaction" idea) and the rules given here do that. The fact that a large system, once built, can automatically be checked for conformance to requirements of dependencies is obviously valuable (even if the idea of having a logic to construct such system does not appeal).

There remains the question of how a very abstract model, once we have one, can be used as the basis for an implementation. Our expectation would be to proceed via the existing and well-known and established techniques that are called *refinement* [2], which would take us from a design that provably has the required properties (*i.e.* built with usability as a constructed and provably existing property) to provably usable implementations, since the central point of refinement is that it allows us to move from abstract to concrete (design to implementation) while preserving meaning and properties. Taken together, then, we have rules that allow us to design and specify usable systems, and refinement rules that, preserving usability, take us to implementations or provably usable systems. As one of the reviewers said: "[it would be good] to apply the concepts on the ISO 9241 definition of usability and to a concrete small example (maybe a cash machine?) to see in practice the concepts developed" and I agree, a nice piece of work for the future.

Finally, this abstract system does not have the interpretation of components decided in any way, though the idea of "interactive" does begin to impose one. However, we have a set of rules here which simply allows construction of connected components where *some needed properties which make a system "proper"* can be kept track of—so this has general application.

References

1. J. Bowen and S. Reeves. Formal models for user interface design artefacts. *Innovations in Systems and Software Engineering*, 4(2):125–141, 2008.
2. John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Formal Approaches to Computing and Information Technology. Springer, second edition, 2014.
3. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
4. D. Gries. *The Science of Programming.* Monographs in Computer Science. Springer New York, 1989.
5. S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.