Robust Content Creation with Form-Oriented User Interfaces

Dirk Draheim
Institute of Computer Science
Freie Universität Berlin
Takustr. 9
14195 Berlin, Germany
draheim@acm.org

Christof Lutteroth
Department of Computer Science
The University of Auckland
38 Princes Street
Auckland 1020, New Zealand

Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street
Auckland 1020, New Zealand

{lutteroth,g.weber}@cs.auckland.ac.nz

ABSTRACT

In this paper we describe how content can be created in a way that ensures its integrity at all times, and how the user interface for such a content editing program can be modeled using the methodology of form-oriented analysis. The paper discusses aspects concerning the data that is being created, as well as aspects of the content editor itself. We show that technological features like typing, opaque identities and user transactions can facilitate the process of content creation as experienced by the user significantly, and that these features can be effectively incorporated when using the form-oriented analysis model.

Categories and Subject Descriptors

H.4 [**Information Interfaces and Presentation**]: User Interfaces—*Theory and methods*

Keywords

form-oriented user interfaces, robustness, content creation, configuration

1. INTRODUCTION

For many people computers are tools that, ideally, help its users to express and manage the products of their creativity. Their significance is not their functionality in itself, but the way it can be utilized by the user who wants to take advantage of the manifold possibilities of digital content. As more and more computer systems find their place in our everyday life until people can hardly avoid them any more, it becomes especially important to make their benefits easily accessible and shield the end user [13] from the shortcomings of computer technology. What we need are *user interfaces*, which deliver functionality in a way suitable to the user, in contrast to *system interfaces*, which deliver functionality in a way suitable to the system.

CHINZ '05 July 6-8, 2005 Auckland, NZ

One of those shortcomings is that in many systems, it is easily possible for the user to bring the system into a corrupted state. When the data a program operates on can be corrupted easily, the user frequently finds himself or herself at a dead end. In those cases the programs themselves are usually not capable of providing the user sufficient help, and even though most of those problems can eventually be overcome quite easily, it is very common that they take up quite a lot of valuable time. Of course, we could expect the user to learn how a program should be used, but the fact is that most people are not *power users* but *casual users*, who use most programs rather occasionally. Spending time on learning how to use a program is an investment that not everyone is willing to make, and not every program is worth the time.

This paper considers systems for the creation of digital content, how to model them and how to give them a property which we call *robustness*. Our approach is to design systems in a way that guarantees a form of data consistency at any time. Rather than dealing with error detection and error handling, these systems focus on *error avoidance*. The ability of a system to avoid any interaction that might cause inconsistency is what makes the system be robust.

In order to be able to make meaningful statements about usability of content creation systems, we are in need of a *theory* of such systems, and fortunately we already have such a theory at hand: the form-oriented analysis methodology [5]. While it is not limited to the specification of content creation systems, one of the contributions of this paper is to show why it suits this purpose very well and how it can be applied to the modeling of content editing systems with various forms of interaction.

We will frequently refer to the creation of web content in order to exemplify different points discussed in this paper. Web content is by its very nature active content in contrast to mere text documents. Many users use the computer to a large extend for browsing, and consequently web pages are the human computer interface they use most. However, web technology has not succeeded enabling end users to develop these interfaces for themselves.

2. THE FORM-ORIENTED USER INTERFACE MODEL

Form-oriented analysis [5] is a methodology for specifying submit/response-style information systems. This means

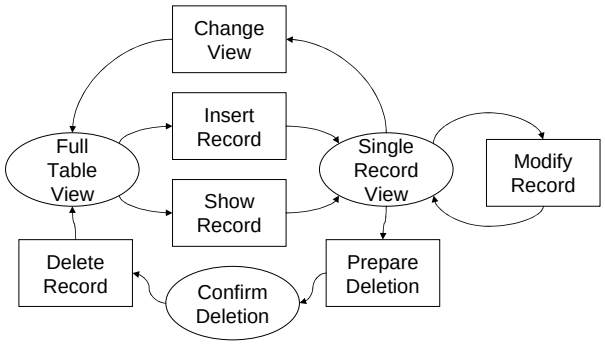

Figure 1: Formchart model for simple data editor.

that you can specify systems in which information is received and displayed to a user, in which the user can submit information to the system and thereby get a response that is made visible and opens up new ways of interaction. The form-oriented model can be used for many different application types, like web applications[4, 8], but it is important to observe that this model can be of particular value in the specification of content editing systems.

A basic notion in this model is the concept of *actions* and *pages*. Pages are parts of the system that report information and offer possibilities of interaction to the user. We call these possibilities of interaction *forms* since they usually allow the user to provide more bits of information, the parameters. Once the user submits the information entered into a form, this information is sent to another part of the system which is an action. An action processes the information in some way and might also access different kinds of data stored in the system, and eventually sends a result back to a page.

This alternating structure of pages and actions can be formally described as a bipartite typed state machine, and a simple way of visualizing it is a *formchart*. Figure 1 shows a simple formchart that describes a simple editor for structured data. The bubbles represent pages that display information to the user, while the boxes represent actions that can be invoked from pages. In this figure the captions describe the content that is displayed or the functionality that was activated, respectively. The transitions between pages and actions describe the possibilities of user interaction.

3. CONTENT MODELING

As we will see, the way content is modeled can have an immediate impact on the way a user interface can provide access to it. A suitable content model does not only make it easier to implement a good user interface, but can be of significant help to the user when interacting with an application.

We will also discuss a suitable implementation strategy for our content model. For this purpose we will use the relational data model [1]. This data model was invented in the 70's and is the one most frequently used in industrial environments. It provides the basis for most database systems, and there is a standardized and well-established language for defining and manipulating data on such systems, *SQL* [10].

Using this industry standard for data management is not just a matter of implementation, but it also solves a problem which can be of direct concern to the end user: data ownership. Of course, usually the user has the legal rights on his or her own creations, but that does not change the fact that many commercial programs for content creation hinder the end user to use the data the way he or she likes to. Commercial programs usually store data in their own proprietary data format, which can make it difficult to use that data in other programs. In the extreme case we are bound to a single program and, consequently, to a single user interface. In contrast to that, the way data is stored in a relational database is always transparent to the user, and the user will never be bound to a single interface for working with his or her content.

The next three Sects. 3.1, 3.2 and 3.3 discuss aspects of content modeling that can directly provide added value to a user interface. It makes clear that the way we handle content as end user can directly benefit from the way we think about content when modeling its structure.

3.1 Typing

Data types play a significant role in many application domains and have to be represented in the user interface: dates and currencies are classical examples, email addresses are a further example. The use of types is the essential technique to impose structure on content. We divide the universe of possible values into subsets, which may be disjoint or not, depending on the kind of type system we use, and each of these subsets is called a type. A value that is part of such a subset is said to be an instance of the corresponding type.

With the relational data model we are able to give every bit of data in our system an explicit type, which means that the type of data will be immediately evident in our system. A type is not only used to express structural properties of its instances, e.g., the fact that they are numbers. At least as important as the structural properties are the semantic properties of data that can be expressed by giving them types. A number, for example, can be defined to express a postal code, the size of a salary or the size of a person's shoes.

SQL offers us a number of atomic types, i.e., types that cannot be divided any further, like types for representing numbers and strings. It also offers a way for creating composite types, so called relations, that are made up of basic ones. Each atomic part of a relation is called an attribute, and each attribute has a label that identifies it uniquely within the relation it is defined in.

The properties of content that we make explicit by distinguishing types are not just important for the internal logic of our system. In the area of end user development the support for typing gets a different motivation. The types are not mere helpers for the learned software engineer, and the reporting of type errors is not the right presentation of type concepts any more: in end user development the system itself must manage typing issues once the user has declared his/her typing preferences. The purpose of typing here is therefore to free the user from the possible mistakes that are understood as typing errors. As a consequence, the user interface does not offer the wrongly typed choices in the first place. The type system is therefore facilitating development of content that is type-sound.

Let us consider user interface development with screen diagrams – a user interface diagram type which is, in contrast to formcharts, not backed by a typed data model. Figure 2 shows a screen diagram for a simple book inventory

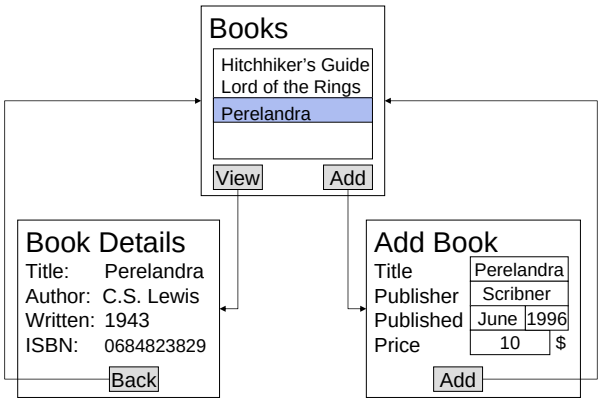

Figure 2: Screen diagram of simple book inventory.

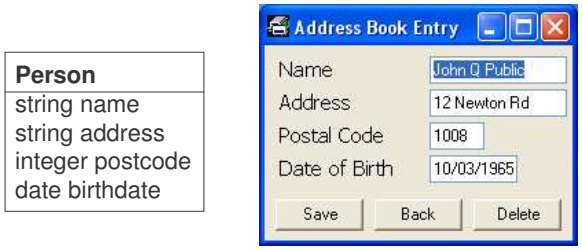

Figure 3: Record type and "Single Record View" user interface.

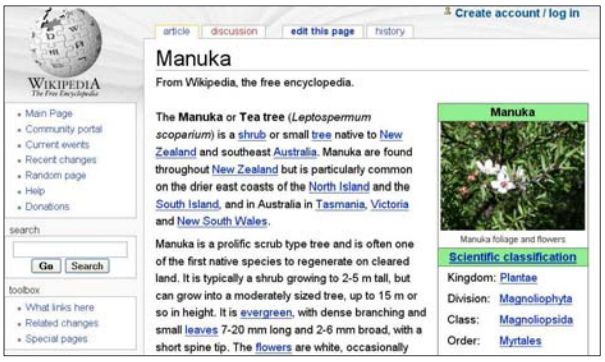

Figure 4: Article view of the Wikipedia.

management application. Screen diagrams are probably the most frequently used user interface diagram type because they correspond directly to what the user will see on the screen, which makes them very intuitive. Arrows starting at control widgets and leading to other screens indicate how the user interface changes when certain interactions are performed. However, as Fig. 2 demonstrates, the lack of a data model makes user interface development with screen diagrams prone to type errors: it can easily happen that representations of the same content on different screens do not match properly, e.g., items might be missing or the format or level of detail of certain parts of the content might be different. With a typed data model the content is well-defined, and such mistakes can be effectively avoided.

To illustrate the relationship between the type system and the user interface, let us consider the page "Single Record View" of the user interface model in Fig. 1 with the assumption that the system we are modeling is an address management program. In the real system the page will probably look similar to that in Fig. 3. On the left side of this figure we see a visualization of the type that represents an address book entry; on the right side we see the user interface for editing data of that type. It is plausible that each editable field in the GUI corresponds to an attribute in the data type, and furthermore, in a good user interface the type of the attribute influences the way in which the user can interact with the different GUI components of the page (about this kind of interaction within a page see also Sect. 4.1). In our example, the user should probably be able to enter arbitrary strings into the editable fields for name and address; the fields for postal code and date of birth, however, should only allow the user to enter a number and a valid date smaller than the current date, respectively. This context sensitive restriction of user input is an important principle for the creation of robust user interfaces, i.e., user interfaces that do only permit interaction that leads to a feasible system state.

The HTML/HTTP-based world wide web has from its first days a strong tradition of ASCII-based development. Many authors still use simply an ASCII editor to create their HTML pages. This approach is ambivalent - it has its clear limits, it nevertheless works as well. If we look at today's wiki systems [2], editing web pages is only one click away from reading them, as we can see, for example, in the example web page in Fig. 4. It becomes clear that editing the web page is part of the *same* user interface as reading it, therefore web development can be seen as a task that is also intended for the end user. Nevertheless, in the case of wikis the paradigm is a *dualistic*, not a unified one: the user interface of the editor is visually altogether different from that of the viewer, as we can see when comparing Fig. 4 with the corresponding Fig. 5.

The problem with text based editing views of web pages or web applications is that they create an inherent point of instability. The user can break the system by entering incorrect input into the edit panel. This can be overcome by systems that use an edit pane that is similar to the content view [7]. By using form-oriented techniques one can deliver a robust interface for application development that overcomes this problem. We are proposing an interface technology that guarantees that user input keeps the system in a working state. The approach will depart from the text based editing metaphor, which is essentially untyped, and proceed to a typed editing approach that is - as a further advantage - much more consistent with the look and feel of the web and creates much more similarity between editing mode and viewing mode. It also fits to the usage of many wiki systems [6].

Note that the approach here is altogether different from the RDF approach [11] and the semantic web. The semantic web lives on a different layer. It is an implementation level concept, while our focus of interest is in the user experience. Secondly, the semantic web works on an ontological level. Our approach is more neutral with respect to the contents presented. It is just a framework allowing the user to express content.

3.2 Opaque Identities

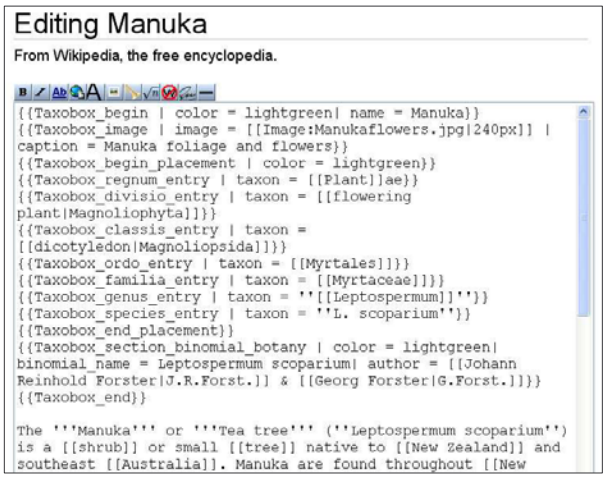

Figure 5: Article editor of the Wikipedia.

Opaque identities are the technical term for a an interface concept that can be realized in quite a number of ways. Opaque identities refer to the way how we identify, or name, things. If we want to name things, be it people, data fields or articles, the best strategy depends on way a name will be used. If, for example, we use the name in a discussion, or in a plain text, we necessarily have to have a plain name. But if we want, for example, to drag an article from one form of a user interface to another, we are not interested in a plain representation. We are rather interested in a unique identity, not prone to synonyms. Furthermore, we want it to be free of spelling problems and to have type consistency, so that an article can only be used as it should be.

The overall solution to this is the opaque identity. Instead of cutting and pasting, say, an article ID from one user interface from to another, we offer an abstract representation. In the most general form, this is a small icon that can be cut from, pasted and dragged into the input field of a form where it fits, similar to the user interface depicted in Fig. 7. But beside this very iconic representation, the concept of opaque identities refers to a very effective user interface solution that lets the user reference objects without manipulating their names or IDs directly.

Figure 6 shows an example where a user interface handles email addresses, which could be represented with opaque identities, in a user unfriendly way: in the edit window the two recipient email addresses are accessible in a table-like widget, however, it is not possible to mark and copy multiple cells of this table or all of it. The second window views the same email after it has been created, possibly on the sender or the receiver side, and this time the emails of the recipients are represented as running text. But again, it is not possible to mark and copy multiple email addresses here. The user interface uses different widgets to handle email addresses, and by their looks these widgets seem to offer certain standard functionality which they actually do not. The concept of opaque identities, in contrast, strives to unify the way such content is represented and handled.

Relational database systems support the concept of opaque identities by offering the mechanism of primary keys. Primary keys are values in data entries that identify each entry uniquely. Furthermore, a relational database system takes

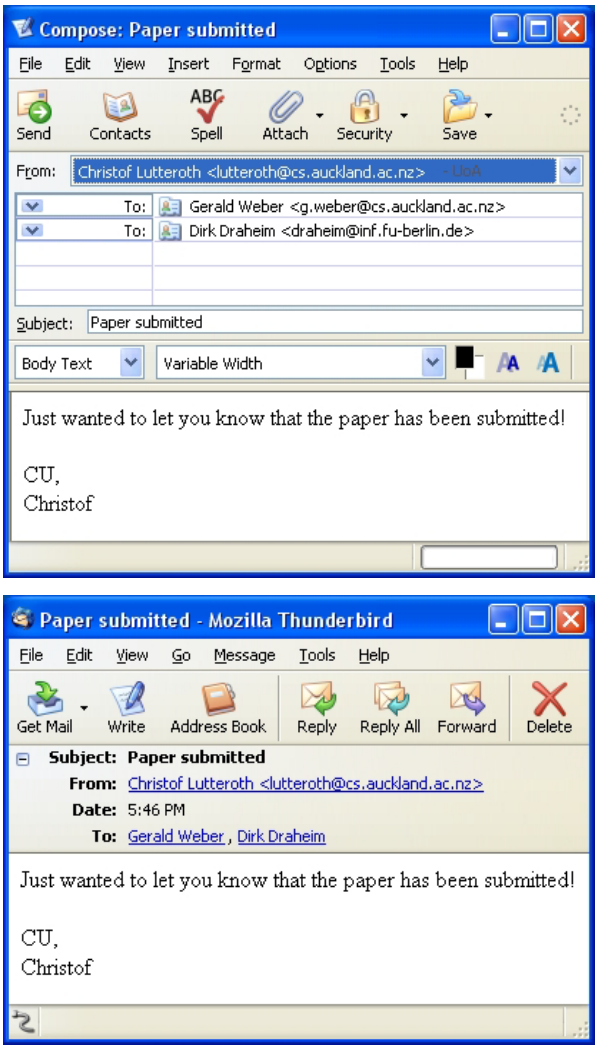

Figure 6: Edit window and view window of of the Mozilla Thunderbird email client showing the same email.

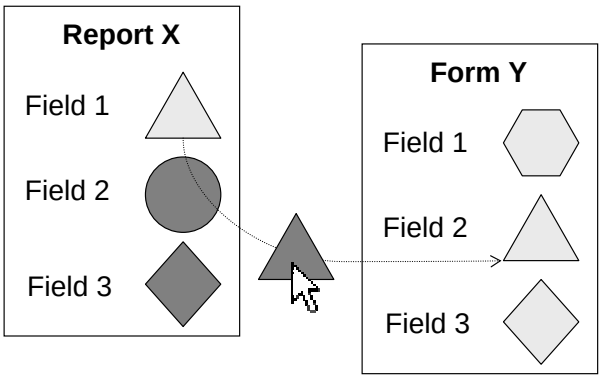

Figure 7: Handling opaque identities.

care that there can never be two data entries in a relation with the same primary key. This makes a relational database particularly useful for the implementation of robust systems.

3.3 Referential Integrity

In most systems we have different kinds of data that are related to each other. In our address book example, we might, for example, not just keep track of the people in our address book, but also of the companies they might be affiliated with. Consequently, we would not only have to store entries for people and companies, but also information about which person belongs to which company. Once we have such kind of referential information, there are a number of things that can go wrong.

A person can, of course, only be affiliated with a company that exists in our database. So when setting a persons affiliation, the user should only be allowed to choose from the ones for which data entries are available. It is also convenient when the user can immediately create a new company entry when he needs it, at the place of the user interface where the affiliation of a person is set. Another important question is what happens when a company entry is deleted, although it is referred to by person entries. Whatever strategy is chosen to deal with that case, it is crucial that after the deletion there are no remains left in the data that might raise the impression that the entry still exists. There must not be references that refer to no valid data at all and this has to be reflected in the user interface. The property of data that every reference refers to a valid data entry is called *referential integrity*.

One distinctive robustness issue with HTML concerning referential integrity is the avoidance of broken hyperlinks. Not only text based editing of HTML is prone to broken links; the absence of a high level infrastructure for links makes this a persistent problem. On the other hand, broken hyperlinks are still a relatively easy to fix class of errors. The frequency of updates that could create broken links can be considered low, and at any time the link is broken, it can be fixed by looking up the correct place. There exist alternatives to HTML that solve the problem of broken links [3, 12]. If we want to move on from a simple hypertext to a web application, robustness becomes a much more complicated issue. Looking at today's implementation technologies, like PHP, web applications have by no means simple semantics. The task of developing robust web applications is much more complicated that developing correctly linked websites.

Fortunately today's relational databases are able to ensure referential integrity automatically. It is simply not possible to refer to a non-existent data entry, and if a referenced entry is deleted, the database system takes a predefined action. A user interface that bases on such a database system is robust to referential violation.

4. TWO-STAGE INTERACTION

In form-oriented systems we distinguish between two kinds of user interaction. On the one hand, a page usually offers the user ways of interaction that alter or extend the content seen on the respective page, like editable fields to enter data and other kinds of GUI controls, which we subsume under the term *fine-grained interaction*. On the other hand, most pages also offer ways of submitting information in forms to the system which cause not necessarily the page content but the overall system state to change and possibly lead to other pages. For this we use the term *coarse-grained interaction*. Usually the end user will first interact with a page on the fine-grained level, providing input to and changing the content of a form, before submitting the data in a form and inducing a data change on the coarse-grained level. Therefore the overall notion is also called two-stage interaction.

In the following two Sects. 4.1 and 4.2 we will discuss fine-grained and coarse-grained interaction in more detail. We will illustrate how they can be modeled using the form-oriented paradigm and how their technological implementation relates to an added value for the user interface. In the case of coarse-grained interaction, this will lead us to a new notion, that of user transactions.

4.1 Modeling Fine-grained Interaction

The defining characteristic of fine-grained interaction is that the changes it causes to the system are only ephemeral. The persistent state of the system remains unchanged. Fine-grained interaction is used mainly for the following:

- **Input or modification of form data**
  This might be a textual modification of some sort of editable input field, like the ones we saw in Fig. 3. But it might as well be a modification of some other nature, like a change of a graphical object on the page.

- **Changes of data view**
  These do not change the data itself, but rather the way it is perceived by the end user. Typical examples for this are elision in tree-view-like widgets and sorting of table views, but it might as well be something like a shift from visual to auditive page representation.

- **Ephemeral side-effects**
  These are effects that neither change the representation of the page, nor persistent system state. Common examples of that category are a feature to copy data to a clipboard, or one to toggle between different modes of operation of an input device, like the overwrite and insert mode of the keyboard.

When entering or modifying form data on a page, the data in the form might temporarily become inconsistent. This is usually the case when the information in the form is still incomplete or currently being modified. Consider, for example, the editable field for the postal code in Fig. 3. It

would be possible and a good idea to make sure that only numbers are entered into the field during input and that the input does not exceed a certain length, but we would probably not want to set a lower bound for the length while input is in process. For a new address book entry the field would be initially empty, and naturally the length of the data would grow gradually. While the data in a form is inconsistent for one reason or another, it must not be possible to submit it to the system. Therefore, the "Save" button in our example could not be activated until the data in the form is complete and all right. The transient inconsistency of the data during fine-grained interaction does not compromise the integrity of the system because the general system state does not change until a coarse-grained submission of consistent data is done.

A question that arises is how fine grained interaction can be modeled with the form-oriented user interface model. The answer is simply that there is only one general way of modeling, which is the one described in Sect. 2 and illustrated in Fig. 1. Form-oriented models are flexible enough, though, to accommodate for both the needs of coarse-grained and fine-grained interaction.

What we have learned is that data can be submitted on a page, which means that it is sent to an action, which in turn sends back data for a new page. In the case of fine-grained interaction, the submitted data needs not be consistent because it is not integrated into the general persistent data model. This makes it possible to model input or modification of form data as submit/response cycle.

Furthermore, a submission may send data that is not perceptibly represented on the page. This data is usually either a superset or subset of the data displayed on the page, or additional information, e.g., about how the page is rendered to the output devices, e.g., the screen. In order to model changes of the data view, the page is given additional rendering information about the way the data should be represented, and possibly it is given more information than will actually be perceptible on the page. For instance, a table view might only display a subset of the columns in a table, depending on which ones it is configured to show, and a tree view will display only the root nodes of subtrees that have been collapsed by elision. Once the rendering parameters are changed, the data is sent to an action, where it is possibly transformed, e.g., sorted in a different way, and rendered on a page again, probably the same one it was rendered on before. Also in the case of ephemeral side-effects, additional hidden data is used to keep track of the ephemeral changes. This hidden data might, for example, be the content of a clipboard.

In order to illustrate the use of the form-oriented model for fine-grained interaction, let us consider how this kind of interaction could be modeled for a considerably complex user interface. Figure 8 shows the start page of an integrated development environment (IDE), a program for the creation of software. It contains mainly a menu bar and a tool bar at the top, a tab control with different tabs on the left, and a panel with a list of existent development projects on the right. On the currently open tab there is a tree view, which shows the local computer's file system hierarchy, above a table view, which lists the files in one of the file system's folders. Such a user interface could be modeled in arbitrary detail, but the overall structure of the model would look like Fig. 9. The different actions change the data sent back to the

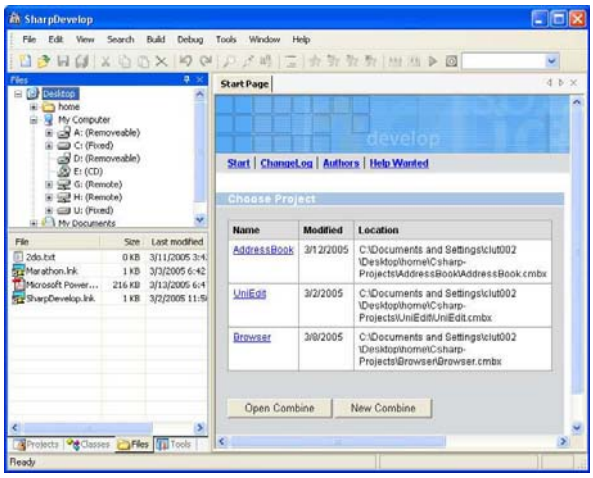

Figure 8: Start page of the SharpDevelop IDE.

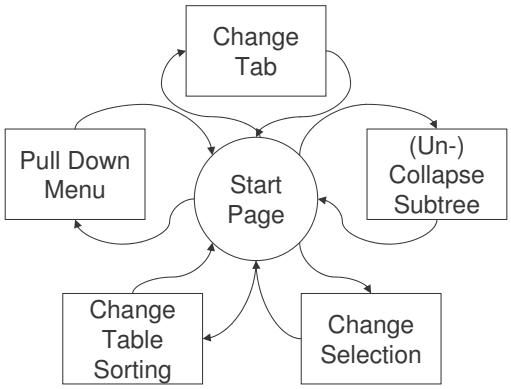

Figure 9: Fine-grained interaction formchart model.

page in a way that a pull-down menu is shown, a different tab is opened, a subtree is collapsed or unrolled, a different file selected, or the table sorting changed, respectively.

Many possibilities of fine-grained interaction are usually accessible on multiple pages of the system. In the case of the IDE, we have, for example, another page for editing source code that, like the start page, offers the possibilities to pull down a menu and choose between tabs. In many systems parts of the user interface are available at many different places and therefore crosscut the system's overall structure. Attaching the corresponding actions to every single page would not only mess up our model but fall short of the fact that these parts are always the same, which would compromise maintainability. Therefore, form-oriented analysis offers a model entity called a *state set* that allows the modeler to attach a model parts to a whole set of pages. Figure 10 shows how to model the common parts of the fine-grained interaction on the start and edit page.

Another question is how to separate the model for fine-grained interaction from the rest of the model, so that we are not unnecessarily confronted with too much details all the time. This problem can conveniently be solved by decomposition of form-oriented models. With this method different parts of the user interface can be modeled separately, with some parts possibly appearing in multiple submodels, and

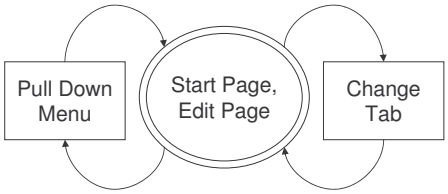

Figure 10: Modeling crosscutting concerns of the UI with state sets.

the different submodels can be arbitrarily merged.

4.2 Coarse-grained Interaction with User Transactions

The characteristic property of coarse-grained interaction is that it can cause the persistent data of the system to change. When dividing interaction into the fine-grained and coarse-grained categories, one of the deciding factors is whether the respective action should cause an ephemeral or a persistent change. In most cases of interaction, like the ones depicted in Fig. 1, the decision is a natural one.

Every action that relies on more than trivial user input and naturally involves a change in the data should be modeled as coarse-grained interaction because, as we will discuss now, this does not only prevent data loss but also offers other added value. When fine-grained interaction is interrupted by some system malfunction, e.g., a loss of power, all ephemeral data will be lost. However, it will be possible to restore the persistent state of the system, so that the overall loss is insignificant.

For coarse-grained interaction we use another mechanism to make our system fail-operational, i.e., capable of recovering and functioning after a failure. We give it the name *user transactions*. Every operation that is performed on the persistent data within a single submit/response-cycle is atomic. Naturally, in the implementation this will correspond to the concept of a database transaction, which means that the operations are guaranteed to adhere to the ACID principle [9]. Among other things, what will happen is that either all operations are successfully completed, or none, so that the persistent data will never be left in a half-unfinished corrupted state. When coarse-grained interaction is interrupted, the data of the form that triggered the corresponding action will be lost, the integrity of the persistent data, however, will be unaffected. All of today's relational database systems come with a built-in capability for transaction processing, so that this feature can be used with no additional effort.

Another very desirable feature that can be implemented with transactions at some additional cost is version control. A common feature in modern content creation programs is an undo/redo function, which allows the end user to revoke actions that have been performed before. However, this feature is usually just implemented as ephemeral side-effect, and during a system malfunction this information is lost. Version control makes the history of coarse-grained changes in the system persistent, thereby allowing to search for and reuse any old version of the content that is edited. It also enables the user to manage different versions and keep track of content evolution, which is an absolute necessity in larger projects.

Relational database systems are usually not only capable of transactions, but also capable of concurrency control. In distributed systems this is necessary to avoid interference of different transactions that are executed concurrently, at which the transactions can be sent to the database from any place that is connected to the network. This means that if we model coarse-grained interaction using the paradigm of user transactions, our system will automatically be able to handle multiple remote users at the same time. At no additional cost we get a distributed multi-user interface system.

5. CONFIGURATION VS. CONSTRUCTION

When talking about the process of content creation, we distinguish two different views of what is actually done. Traditionally the process of content creation starts out with nothing or a small amount of data. This data is increasingly extended over time, but there may also be phases in which the amount of data decreases. However, the content may also, deliberately or accidentally, be perturbed, rendered invalid or become altogether corrupted during creation. For example, parts of data may be fitted together in a way that does not make up a complete model yet. We call such content and the way it is created a *construction*.

Using our terminology, constructions are not based on a robust user interface model. The fact that a state of inconsistency can be reached is opposed to our notion of robustness. Therefore we propose a different view on content creation, *configuration*. In the paradigm of configuration, every feasible content is a point in a multidimensional, possibly infinitely dimensional, space – the *configuration space*. Each dimension of that space is a possibly infinite semantic domain, containing values and a special value representing an undefined state. In the process of creation, we start out with the point that is undefined in all dimensions, and transform that point into another point that contains either more or the same amount of information as the point before. The only exception to this is when the end user deliberately deletes parts of the content, but even then, if we assumed that a deletion would create a new version of the content and that all prior versions are part of the configuration, the aforementioned property would still hold. Each single point, as well as the process itself, is called configuration. The sequence of points that the system passes through during configuration is called *configuration history* and can formally be understood as a chain in a complete partial order, starting at the bottom element. The fact that all points in the configuration space represent consistent content, together with the fact that we can never leave this space, makes sure that a user interface based on that paradigm must be inherently a robust one.

6. CONCLUSION

In this paper we introduced different ways and methods to make user interfaces of content creation systems more robust. We discussed issues of data modeling that are of immediate significance to the user interface, how the form-oriented methodology can be applied, and how it can be used to model fine-grained and coarse-grained interaction. Eventually, we proposed a new paradigm for the creation of content – configuration – that can serve as a formal basis for the creation of robust user interface models.

7. REFERENCES

[1] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[2] W. Cunningham and B. Leuf. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley, 2001.

[3] W. Dalitz and G. Heyer. *HyperWave : The New Generation Internet Information System*. Morgan Kaufmann, 1997.

[4] D. Draheim and G. Weber. Modeling Submit/Response Style Systems with Form Charts and Dialogue Constraints. In *Workshop on Human Computer Interface for Semantic Web and Web Applications (HCI-SWWA)*, LNCS 2889, pages 267–278. Springer, 2003.

[5] D. Draheim and G. Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, October 2004.

[6] D. Draheim and G. Weber. A Qualitative Analysis of Emerging Collaborative Web Structures. Technical Report UoA-SE-2005-5, Software Engineering Programme, The University of Auckland, 2005.

[7] D. Draheim and G. Weber. End-User Development of Web Applications. Technical Report UoA-SE-2005-4, Software Engineering Programme, The University of Auckland, 2005.

[8] D. Draheim and G. Weber. Modelling Form-Based Interfaces with Bipartite State Machines. volume 17, pages 207–228. Elsevier, 2005.

[9] International Organization for Standardization. 10026-1:1992. Information technology – Open Systems Interconnection – Distributed Transaction Processing, 1992.

[10] International Organization for Standardization. International Standard 1975:1999. Information Technology – Database Language SQL, 1999.

[11] O. Lassila and R. R. Swick. *Resource Description Framework (RDF) – Model and Syntax Specification*. World Wide Web Consortium, 1999.

[12] H. Maurer. *Hypermedia Systems and Applications: World Wide Web and Beyond*. Springer, 1997.

[13] A. Sutcliffe and N. Mehandjiev. End-User Development. *Communications of the ACM*, 47(9):31–32, 2004.