TRANSCLUSION IN SOFTWARE FOR THE COMPLEX, THE CHANGING, AND THE INDETERMINATE

PHILIP TCHERNAVSKIJ
STUDENT ID: 20118057

Master's Thesis
Advisors: Susanne Bødker & Clemens Nylandsted Klokmose
Department of Computer Science
Aarhus University

May 2016

Philip Tchernavskij, studienummer 20118057:
Transklusion i Software til det komplekse, det foranderlige, og det ubestemmelige,
Kandidatspeciale
VEJLEDERE:
Susanne Bødker og Clemens Nylandsted Klokmose
Institut for Datalogi
Aarhus Universitet
maj 2016

ABSTRACT

I investigate the unexamined phenomenon of software transclusion. My study is a theoretical inquiry into the potentials, challenges, trade-offs, and consequences of software transclusion as an architectural composition mechanism and user interaction. I take up three research themes collecting challenges for user software: *reconfigurable software*, *asymmetrical collaboraiton*, and *ecology of software artifacts*. I develop a model for transclusive software, which formulates software systems as networks of information substrates applying transclusion as a fundamental composition mechanism and user interaction. I apply this model in a reformulation of a system taking up my research themes, and analyze the resulting design sketch to reflect on the generative power and limitations of my model. I conclude that software transclusion is a novel mechanism for my themes and has productive synergy with the models instrumental interaction and information substrates, that my model of transclusive software implies several future design and research challenges, especially for reliability and security models, and for common software artifacts.

ACKNOWLEDGEMENTS

I am grateful for the help of some outstanding scholars and friends in producing this thesis. Thanks to my advisors, Susanne Bødker and Clemens Klokmose, for discussions, critiques, and patience. Thanks to Niels Olof Bouvin, Henrik Korsgaard, and Ted Nelson for volunteering their time to clarify their work in personal communications. Thanks to Casper Færgemand in particular, for feeding me, and for being an insightful and encouraging critic.

In my time at Aarhus University, I have had the fortune of interacting with many scholars who have held me to a high standard of scientific practice. Thank you for expanding my mind and always reminding me to put in the work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In 2016, personal computing has diversified to an ecology of devices including laptops, desktops, phones, tablets, and many special-purpose devices for, e.g., navigation or entertainment. User software running on these devices is generally expected to be reliable, simple to install, and aesthetically pleasing. While the infrastructure of software is being reconfigured significantly as always-online computing has become feasible, the interaction paradigm of especially older device types has not changed radically in the last three decades. On newer devices, market leaders are competing to produce the smoothest user experience, refining touch-based interfaces iteratively with every release.

One could be forgiven for assuming that user software is so stable because it has established itself as an effective, efficient, and painless mediator of human activity. Under critical examination, however, it becomes clear that as personal computing has matured, assumptions about the limitations of interactive systems have been embedded in our software.

Studies of artifact use in collaborative and creative practices show that there are many "real-world" interactions which are fundamentally ill-supported by user software, characterized generally by complexity, changeability, and ambiguity (Klokmose and Zander, 2010; C. P. Lee, 2007; Sørgaard, 1988; Star and Griesemer, 1989). Some of these interactions seem natural, in the sense that we may expect them to be available based on experience with non-computer-mediated activity. Others have been envisioned as the promise of the software revolution decades ago, and are theoretically possible, but are impossible to implement well within the current infrastructure of software. For example, user software rarely facilitates

- freely changing interaction method/device format,
- modifying or circumventing internal models of domain objects,
- expanding the capabilities of our tools,
- applying data from one working context in another as we see fit, or
- applying our skills in others' work contexts.

These specific problems all transcend assumed boundaries, between individual software applications, devices, and use contexts.

The practical result of this situation is that complex use situations, e.g., ones requiring collaboration around shared digital artifacts, lead

to breakdowns where software is frustratingly inflexible and opaque. These breakdowns are noticeable across several studies of complex software-mediated activity systems, from laboratories to digital agencies.

I take up three research themes for user software, outlining challenges for software use, design, and architecture. These are:

ASYMMETRICAL COLLABORATION Computer-mediated cooperative work situations involving shared artifacts and unequal collaborators (Having different tools, needs, skills etc.) are challenging domains for user software. An alternative software must define and support interactions that resolve the challenges of these use situations. Asymmetrical collaboration is a common phenomenon in "real" life but presents many problems for traditional user software, which is typically built from a one-size-fits-all philosophy. This is especially the case when interaction involves professionals who have complex and idiosyncratic practices that cannot be easily readapted to a standardized software ecology.

RECONFIGURABLE SOFTWARE Software architectures currently predominant in user software assume a single user and a single device, operating on well-defined data through an unchanging interface. In practice, users may need to work with heterogeneous and volatile data, while expecting a uniform experience of interaction. They may also need to distribute activities across several devices. Finally, users may need to add, remove, modify or combine features to best support their specific practices.

THE ECOLOGY OF SOFTWARE ARTIFACTS It is necessary to challenge the ontology of software, of documents, files, applications etc. How we represent and act on shared digital artifacts, groups of devices, and users within software constrains what kinds of interactive systems we can create. Alternative nouns and verbs for software can be the basis for radically alternative software practices, where collaboration and reconfiguration is the norm, rather than outlier cases for specialist software. This is the case on both the architectural level (i.e. objects and methods, MVC) and on the level of user interaction (i.e. files and applications).

The themes provide alternative ideals for user software, and a critical mirror for concrete design proposals.

Transclusion is the inclusion of part or all of an electronic document within another document by reference. The term was originated by Ted Nelson, who describes transclusion as "reuse with original context available" and "the same content knowably in more than one place" (Nelson, 1995, 2012).

Klokmose, Eagan, et al. (2015) have illustrated that it is possible to expand the notion of transclusion from hypertext documents to interactive software. I hypothesize that transclusion can be applied as an novel and useful mechanism for software composition and user interaction, allowing software artifacts to be decoupled from individual users, and from individual application contexts.

In this thesis, I examine the potentials, virtues, trade-offs, and challenges for transclusion in user software. My working hypotheses are that

- Software transclusion is a mechanism for software composition and user interaction which supports novel architectures and interactions in user software.
- Software transclusion as a design principle can be applied to support reconfigurable software and asymmetrical collaboration.
- Software transclusion and information substrates provide a basis for a model of software as a shared material which can be applied for description, evaluation and generation of software constructs and systems that take up my research themes.
- Klokmose, Eagan, et al.'s implementation of software transclusion has shortcomings limiting the mechanism from its full expressive potential.

STRUCTURE OF THIS THESIS

Chapter 2	I review related work on common artifacts, transclusion in hypertext, and precedents in software design research.
Chapter 3	I define software transclusion and my themes with reference to related work.
Chapter 4	I develop a model of transclusive software, which speculatively unfolds the concept of software extensively applying information substrates and software transclusion. I describe software architectural virtues, useful constructs, and potential user interactions in transclusive software based on related and previous work. I evaluate Webstrates critically as an implementation of transclusive software.
Chapter 5	I apply the transclusive software model in a reformulation of the co-located active learning system InPlenary. I analyze this design sketch in a reflection on the generative power and limitations of my model.
Chapter 6	I discuss methodological limitations and the research potential for an in-depth common artifact perspective on transclusive software.
Chapter 7	I review my findings and conclude the thesis.

2 RELATED WORK

EVERYTHING IS DEEPLY INTERTWINGLED.
—Ted Nelson, *Dream Machines (1974)*

In Section 2.1, I review some theoretical perspectives and empirical studies on collaborative practices and common artifacts. This work gives a broad understanding of the dynamics of artifacts mediating collaborative interactions, and introduces the genre of situations in which issues under my research themes occur.

My hypothesis is that transclusion is a concept with unexplored potential for interactive software. To work usefully with transclusion, I must first establish a precise understanding of what it is. I take a historical approach inspired by (Wardrip-Fruin, 2004), in which the concept of hypertext itself is recovered from its many reinterpretations through a review of the intentions and context of its original and codifying descriptions. I apply this approach in reviewing, in some detail, the original context of transclusion, Ted Nelson's visions of hypertext.

In Section 2.2.2, I review recent research applying transclusion.

In Section 2.3, I summarize work in hypertext and software which emphasizes collaboration and flexible interconnection. Finally, in Section 2.4, I review recent research in post-WIMP and multi-user software centering around the interaction model *instrumental interaction*. The concepts and design principles established in this work is the theoretical context into which I place software transclusion in Chapter 4

2.1 ARTIFACTS IN COOPERATIVE PRACTICES

2.1.1 *Shared material*

In (Sørgaard, 1987, 1988), in the then-emerging field of Computer-Supported Cooperative Work, Sørgaard concerns himself with the exact nature of cooperative work, and how we can develop computing technology which supports it.

Sørgaard defines cooperative[1] work as a prototypical kind of work with particular, desirable properties. His criteria for cooperative work are:

1 Sørgaard distinguishes between cooperation and collaboration. He states that collaboration denotes more generally working together or with someone else, whereas cooperation means working or acting together for a shared purpose. I do not distinguish between these terms in this thesis.

> (1) People work together due to the nature of the task, (2) they share goals and do not compete, (3) the work is done in an informal, normally flat organisation, and (4) the work is relatively autonomous. *(Sørgaard, 1987)*

These criteria outline a strict definition, which real cooperative work does not necessarily conform to. Instead, it describes the prototype of cooperative work, and its main characteristics.

Cooperative work occurs in specific situations, e.g. where the immediate goal is not mass production, technology is rapidly changing, and there is high task uncertainty.

Sørgaard argues that classical system development methods for computerization of work practices erase or disrupt naturally occurring cooperative work.

In computer support for cooperation, he distinguishes between computing applied as a medium for explicit communication, and as shared material through which implicit coordination can be manipulated. Tools are individual, and are used to manipulate shared material and access media. Coordination through explicit communication and through the material manipulated in the process.

> Shared material is a metaphor in the same sense as the tool metaphor. When we apply the tool metaphor to a text processor, we address its ability to let the user have his/her primary attention directed to the document, not to the text processor. Similarly the shared material metaphor addresses the implemented material's capability to reflect that it is used and manipulated by several people.
> *(Sørgaard, 1987)*

The shared material metaphor addresses issues like

- Visibility of users in the shared work, and communicating their actions to facilitate compensating, completing, or resulting actions by other users.
- Permissions and dependencies between users.
- Controlling access, i.e. implicit coordination of work by constraining who and when can access specific parts of the shared material.

All of these mediating qualities are often implicit and subtle in non-computer-mediated cooperative work: "in general it is hard to know how a traditional material supports cooperation, its properties in this respect may be hidden to the observer because they only work due to the idiosyncracies of the work process in question." When designing systems for computer-supported cooperative work, we must explicitly take up these issues and design shared material which mediates

work according to the needs of the situation. Digital shared material also allows us to go beyond physical metaphors, and create shared material with unique properties:

> More importantly we can construct entirely new sorts of materials with properties desirable for cooperative work. Hypertext systems can, for example, allow for writing comments and annotations in documents without painting it with red ink, this could be used to collect comments from many people in one hypertext document. The people giving comments might be allowed to see each others comments, thus relieving them from pointing at issues already commented upon. *(Sørgaard, 1987)*

> In system development environments it is not enough to reimplement or copy the properties of existing materials like documents or paper-based files. One has to address the question of what, for example, a program is: a text-file, an abstract syntax tree, a prescription for an information process, etc. [...] The chosen representation has to reflect the way programs are used cooperatively.
>
> *(Sørgaard, 1987)*

In (1988), Sørgaard specifically considers object-oriented design and programming for implementing shared material for cooperation digitally.

He argues that the flexible use patterns observed in non-computer-mediated cooperative work is best supported by emphasizing modeling shared material, not procedures. The norm of computerization as rationalization tends towards software systems which calcify specific procedures as the only options available to workers, with little or no room for flexibility. This approach heavily favors a view of computer-supported practice as strictly formalized and factory-like.

Sørgaard is critical towards procedure-oriented software models of work. He finds that the goal of creating an immutable model of the distribution of work within an organization at a particular point in time is counterproductive in several respects. Computerization of procedures ignores learning, creates a work practice which is difficult and expensive to change, and tends to ignore or underemphasize cooperative aspects of work.

Sørgaard recommends Implementing shared material which models the relevant data, and provides access to domain-specific primitive operations, which are expected to remain somewhat static. Of course, in my theme of asymmetrical collaboration, even the assumption of a static model of shared material with a set of primitive operations does not necessarily hold. Sørgaard does note that in real life, models of work are liable to change, and that models are by their nature always

limited descriptions of reality. From this, He argues that a system consisting of long-lived objects may need to support modification of underlying models at runtime.

Finally, Sørgaard describes a short list of technical requirements for implementing shared material as software objects: Shared material objects need to be permanent and persistent across individual program executions. Persistence is, in fact, the essential requirement for digital shared material. Objects need a well-defined model of sharing, depending on the domain, e.g. rotating exclusive access or concurrent modification. Objects must also tolerate distribution across several devices.

2.1.2 *Boundary objects*[2]

Boundary objects are a model of shared material mediating asymmetrical cooperation.

As developed in (Star, 1989; Star and Griesemer, 1989), they are artifacts which allow information to travel between different communities of practice. The empirical origin of the concept of boundary objects lies in the observation that it is common to see situations of interdisciplinary collaborative work in which there is no pre-established consensus guiding the common practice (Star, 2010).

Boundary objects are characterized by interpretive flexibility. Since their purpose is to enable collaboration, they are structured to serve informational and processual needs of an interdisciplinary work situation, as well as those within the intersecting practices. Boundary objects continuously transition between these two states, being ill-structured in the common context, while becoming more tailored and strongly structured when used locally (2010).

Both 1989 articles are based on a historical study of the complex web of collaborative communities of practice active in the development of Berkeley's museum of vertebrate zoology.

Star enumerates these properties of cooperative work among scientists:

1. cooperate without having good models of each other's work;
2. successfully work together while employing different units of analysis, methods of aggregating data, and different abstractions of data;
3. cooperate while having different goals, time horizons, and audiences to satisfy. *(Star, 1989)*

In (1989), four forms of boundary objects are documented, based on different kinds of cooperative activity.

2 This section and Section 2.1.3 are largely reused from (Tchernavskij, 2015)

Repositories are ordered collections of objects indexed in a standardized fashion. They allow heterogeneous objects to be accessed and used for different purposes by people from different practices.

Ideal types is an object describing a thing in a non-domain-specific and non-localized manner. For example, a species is no particular animal, but a generalized description of a category of animals. Ideal types are adaptable to various domains because they describe a common work object at a high level of abstraction.

Coincident boundaries are objects that share boundaries but not content. For example, a map may outline a physical space in which collaborative work takes place, even if different kinds of maps of the same territory hold information used for different purposes.

Standardized forms are objects which enable common communication schemes across boundaries of practice. A literal paper form, for example, can be used to constrain people of various backgrounds to present information in a standardized format which allows for statistical analysis.

2.1.3 *Boundary negotiating artifacts*

In (2007), C. P. Lee challenges the ubiquity of boundary objects as an analytical framework in CSCW studies. From an ethnographic study of a multidisciplinary group collaboratively designing a museum exhibition, she argues that artifacts can serve to establish and destabilize protocols, and to push boundaries. Emphasizing this transformation of boundaries, she introduces the notion of boundary negotiating artifacts.

Whereas Star saw that collaboration took place in absence of consent, Lee argues that collaboration may also take place without a clear understanding of practices and the boundaries between them. A great deal of boundary work, she says, is concerned with discovering, testing, and pushing boundaries. For example, a group may attempt to modify the division of labor or organization structure within a collaborative work situation. In this view of collaboration, artifacts can be both coordinative and disruptive.

Taking Star's cue, Lee describes five types of boundary negotiating artifacts.

Self-explanation artifacts are artifacts that are created by one person to learn, record, organize, remember, or reflect. They are characterized by being used in tight groups, rather than crossing boundaries between communications of practice. They move out of this private context indirectly, through the creation of inclusion or compilation artifacts.

Inclusion artifacts are used to propose new concepts and forms. They are created from self-explanation artifacts and represent a new idea originating from one community of practice to another. They

naturally become symbols at the center of the informal gatekeeping process where the transformation of boundaries represented by the inclusion artifact is negotiated.

Compilation artifacts are used to bring two or more communities of practice into alignment temporarily, to create a shared understanding of a problem, and to pass along crucial information between communities. When knowledge is distributed across different groups of specialists, this kind of alignment is continuously necessary to facilitate the development of a collaborative product, since interdependent elements of the product must evolve in parallel.

Structuring artifacts, like compilation artifacts, are used to coordinate work and understanding, but also to establish ordering principles, and direct the activity of others. In Lee's empirical study, the hierarchical structure of the collaborators was flat enough that different communities could argue about, e.g., the purpose of the exhibition. Hence, different structuring artifacts would compete for primacy in dictating the common agenda. Structuring artifacts may be used to push and negotiate boundaries themselves.

Borrowed artifacts are artifacts that are taken from one community of practice and used in an unanticipated manner by members of another community of practice. Communities in close proximity may borrow artifacts from each other to augment their understanding of a problem.

Star explicitly sees boundary objects as the material byproduct of a cyclical process of developing standardized systems. Over time, a system produces residual categories, things which are not accounted for in the current models and processes, causing the creation of new boundary objects (Star, 2010). Lee illustrates a collaborative work situation where standardization is not a long-term goal. As opposed to the effortless traversal across boundaries that boundary objects represent, the negotiation processes that Lee describes are effortful. Within the kind of complex, non-routine project she studies, boundary objects are not a useful model for artifact usage. She argues that this points towards a reevaluation of continua of both collaborative work situations and collaborative artifacts.

2.1.4 *Syntonic seeds*

(Klokmose and Zander, 2010) take up laboratory notebooks as an example of a type of complex common artifact which challenges the design of user software. The authors apply *syntonic seeds* as an analytical model to describe and critically evaluate both analog and digital artifacts. Syntonic seeds have three main properties:

- They are persistently representing contradictions, and are inscribed to sublate (negate and preserve) them.

- They are simultaneously or sequentially a mediator in one or many activities and objects of one or more activities.
- They can at any given moment oscillate between being mediators and objects of activity. *(Klokmose and Zander, 2010)*

Klokmose and Zander show that digital tools aimed at replacing laboratory notebooks fail to handle some of the contradictions that paper notebooks sublate, and are as a rule developed to function in a limited subset of the web of activities that lab notebooks are used in.

The authors outline visions for alternative laboratory software tools generated by laboratory workers in a modified future workshop. These visions are generated by the technology-literate subjects of the workshop, whom the authors present with *ubiquitous instrumental interaction*, a conceptual springboard for novel digital tools (Section 2.4.1).

2.1.5 *Previous work - Case study*

In (Tchernavskij, 2015), I conduct a two-part project about software for asymmetrical collaboration scenarios. The first is an empirical study of a web agency, carried out as a collaboration with Martin Thorhauge. The second is an experimental prototyping study, which I review in Section 2.4.5.

The subject of the case study was the Aarhus branch of the website development agency Creuna. We investigated cross-domain collaboration in the website design and development process at Creuna.

Practitioners at Creuna are "T-shaped", having broad skills in and adjacent to their particular specialist role in the development process. They are technical experts, who have a large degree of control over their own tools and workflow, and are supported in changing their personal specialization over time.

Project work at Creuna is structured as a linear relay process, where a system designer first sketches a wireframe of the complete website, then distributes the work on individual modules to frontend designers, who produce detailed mockups that are passed along to frontend developers, who implement the mockups as interactive websites, with backend work happening in parallel. Development teams on the same project are disjoint.

The interesting problem in the case study is that this group of collaborating specialists in practice often experience breakdowns which force them to work outside the formal plan of their project and their tools.

We found tensions and challenges for software originating in the distributed work process. At some point in the design process, a breakdown occurs, where the design specified in one stage of the process turns out to be unworkable with the material of a latter stage, e.g.,

the UI mockup does not map to a well-functioning interactive webpage. Creuna workers respond by subverting their rigid process and tool ecology, and establishing ad-hoc interdisciplinary collaborations to resolve the design issue. These collaborations are typically mediated by pen and paper, rather than specialist tools, because the latter have no awareness of adjacent practices, and cannot cross-integrate with different software.

Both the digital tools and artifacts at hand as well as the overtly linear designed process essentially ignore collaborative design activities. Much like the digital laboratory notebooks in (Klokmose and Zander, 2010), design and programming tools applied at Creuna are oriented towards authorship of a specific type of digital artifact, and do not make allowances for integration with mediating artifacts. When the product of one part of the design process is passed between team members, it changes from a dynamic artifact to an immutable product.

Knowledge sharing and retention are at the root of a few recurring issues. Information work to support interdisciplinary cooperation is ill-supported by software which limits sharing of work artifacts. A work situation in which community boundaries are unclear and continuously developing provides an additional challenge for informational work. There is a challenge in facilitating continuous dialogue between practitioners as an aspect of iterative software development. There is a tension in aligning communities of practice towards a common goal, between maintaining the big picture and allowing individuals to separate themselves from the constraints of other domains.

We reapplied the future workshop method from (Klokmose and Zander, 2010) to leverage the expert knowledge of the web developers in rethinking their tools and distribution of work. Because of the documented issues, the critical theme of the workshop was broadly tools and artifacts which enabled cooperative activities and effectively mediated the disjoint design process. As the stimulating component, we presented and demonstrated Webstrates (Section 2.4.4) between two vision generation phases.

Some of the envisioned solutions to challenges for software at Creuna were found to relate to similar challenges covered in Klokmose and Zander's study, including:

- A work situation that emphasizes the need for tools that support idiosyncratic usage patterns rather than monolithic applications.
- A user need for commonly accessible work artifacts, useful in multiple activities, retaining modifications.
- Digital artifacts that oscillate between being objects of activity and acting as mediators.

- Version control as a property of non-code artifacts, allowing users to rewind, branch, and recall versions of, e.g., images or lab setups. *(Tchernavskij, 2015)*

Summary

These studies outline a spectrum of collaborative practices, with different organizations, practical needs, and asymmetries. Table 1 summarizes the different types of collaborations and artifact perspective of each study. Common artifacts are used to mediate the normal and natural troubles of collaboration, but software as a whole has considered interpersonal meditation exclusively an issue of explicit communication, rather than an opportunity to apply unique material properties to common digital artifacts.

Software typically produces naive and inflexible common artifacts (Klokmose and Zander, 2010; Sørgaard, 1987, 1988; Tchernavskij, 2015). My working hypothesis is that this problem is not inherent to software, but instead is caused by the rigid assumptions that have been embedded in it over time. These empirical studies illustrate, categorize, and criticize how artifacts are used to handle complexity, instability, and ambiguity. In the rest of this chapter, I uncover how software might be differently conceptualized to effectively mediate these aspects of human activity.

2.2 TRANSCLUSION IN HYPERTEXT

I give a short overview of the early history of hypertext technology, an in-depth account of transclusion in the work of Ted Nelson, a review of interesting first- and second-generation hypertext systems, and another review of recent research applying transclusion.

In (2013), Barnet recounts the early history of hypertext through its innovators and seminal ideas and implementations, focusing on some landmark systems as "visions of potentiality", that were all eventually eclipsed by the world wide web. Barnet's book is structured around interviews with those innovators, recalling how a new field of technology emerged and was codified, as well as the revolutionary spirit of the hypertext field in its early experimental phase.

Ted Nelson coined the term in the early 1960's, following his early exposure to computing technology, settling at some point on the popular definition: branching and responding text, best read at a computer screen. (Nelson, 1987, p. 0/2)[3] Barnet refines Nelson's definition of hypertext somewhat, to "Written or pictorial material interconnected in an associative fashion, consisting of units of informations retrieved by automated links, best read at a screen." (Barnet, 2013)

3 *Literary Machines* numbers pages within chapters, so "0/2" means "Chapter 0, page 2".

Study	Nature of collaborative practice
(Sørgaard, 1987, 1988)	• People work together due to the nature of the task, • they share goals and do not compete, • the work is done in an informal, normally flat organisation, and • the work is relatively autonomous. *(Sørgaard, 1987)* • Cooperative work is coordinated by explicit communication and through the manipulation of shared material.
(Star, 1989) (Star and Griesemer, 1989)	• People cooperate without having good models of each other's work; • they successfully work together while employing different units of analysis, methods of aggregating data, and different abstractions of data; • they cooperate while having different goals, time horizons, and audiences to satisfy. *(Star, 1989)* • Shared objects reside between communities of practice, where they are ill-structured. • When necessary, the object is worked on by local groups who maintain its vaguer identity as a common object, while making it more specific, more tailored to local use, and therefore useful for intradisciplinary work. • Groups that are cooperating without consensus tack back-and-forth between both forms of objects. *(Star, 2010)*
(C. P. Lee, 2007)	• A multi-disciplinary group of specialists collaborate on a complex and non-routine project. • A stable organizational context is never established. • Consensus and direct collaboration between groups of practice is temporarily established for situational problem solving. • The collaborators use shared artifacts both to exchange information across boundaries of practice, and to shift the boundaries themselves.
(Klokmose and Zander, 2010)	• Single-discipline collaborators use an inscription object, laboratory notebooks, in a complex web of activities. • Laboratory notebooks *sublates contradictions* in scientific activities. • The notebooks support idiosyncratic organization and elegantly handle heterogeneous data. • The notebooks are an intermediate externalization of scientific work and knowledge, which integrate smoothly with adjacent activities. They oscillate between being objects and mediators of activities.
(Tchernavskij, 2015)	• People collaborate to handle breakdowns in a linearly structured design and development process. • Collaboration temporarily disrupts the formalized distribution of work, hierarchy, spatial and temporal separation between workers. • Collaborators have overlapping and changing practices. • An underlying goal is to minimize documentation and knowledge work. Learning resulting from collaborations is typically ephemeral.

Table 1: A summary of the studies of collaborative practices that are the basis for my research themes.

Note particularly that I will not differentiate between the terms hypertext and hypermedia, as (Wardrip-Fruin, 2004) shows that the concept of hypertext has always been agnostic towards the media content of texts.

The first generation of hypertext systems were originally mainframe-based, focused primarily on text nodes, and used display technologies with little or no graphical capabilities. All of these first generation systems, however, included at least some support for medium to large teams of workers sharing a common hypermedia network. (Halasz et al., 1988) With workstation technology supplanting mainframes, second-generation hypertext systems could support advanced graphical user interfaces, with animations and formatted text. These workstation based systems were generally targeted at single users and small groups. (Halasz et al., 1988)

Then, in the early 90's, the World Wide Web arrived publicly, and eventually exploded, to a large extent codifying hypertext into a settled technology:

> What the World Wide Web did was two things. One is that it compromised, as it were, on the 'vision' of hypertext. It said 'this is the kind of linkage it's always going to be, it's always going to work in this way', [but] more importantly it said that the really interesting things happen when your links can cross from one computer to another [...] So global hypertext – which is what the Web is – turned out to be the way that you could really engage, well, ultimately hundreds of millions of users. (Bolter 2011) *(Barnet, 2013, p. xxi)*

2.2.1 *The origins of transclusion*

In Nelson's mind, the idea of hypertext at some point germinated into the grand design of Xanadu, a comprehensive system for reading, editing, and publishing documents. Xanadu has never been built, though aspects of it have been demonstrated in print and demonstrations.The project is described at length in *Literary Machines* (1987). In that book, Nelson argues for a complete technological solution to the problems of human memory and creativity. Originally released in 1980, it has been significantly revised more than once since. While I cite it extensively in this historical account, the word transclusion does not actually appear in the 1987 version of the book. In published material, transclusion is introduced later, and used at length in (1995; 2012). Instead, Nelson describes the same concept as *virtual inclusion*. This is the same concept as transclusion, and indeed the same idea has reoccurred in Nelson's work since at least 1965, as in Figure 1 (Nelson, 2016, Personal communication). What it is is a hypertext mechanism for "Reuse with original context available,

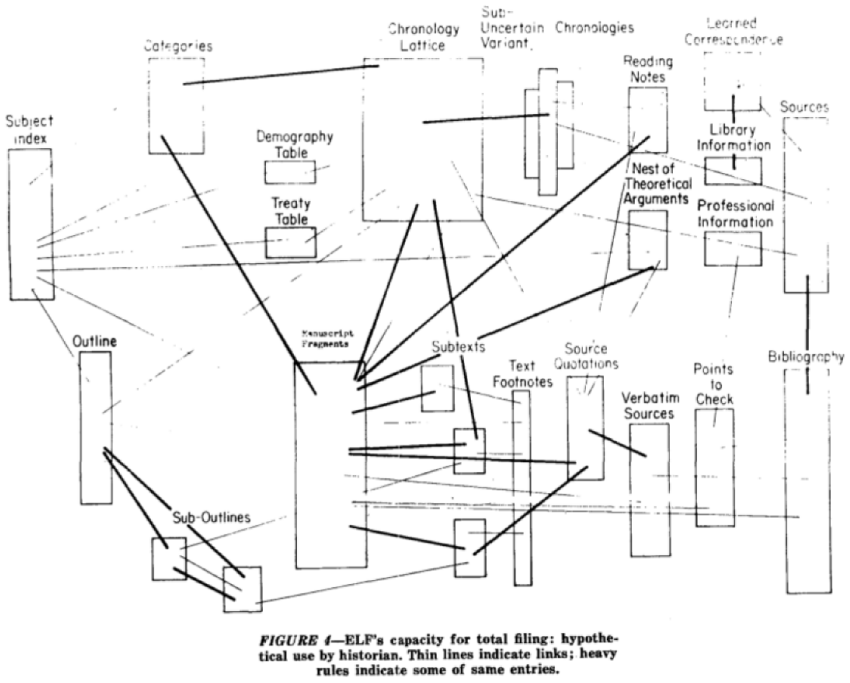

Figure 1: An early drawing of a hypertext. Note the bold lines representing "some of the same entries". Reprinted from (Nelson, 1965) with permission.

through embedded shared instancing" (1995), or "the same content knowably in more than one place" (2012).

Xanadu is motivated by what Ted Nelson sees as extant issues in the creation, sharing, consumption, and maintenance of texts, which computing technology has failed to deal with. Nelson argues that graphical word processing applications and their associated model of texts are mainly concerned with emulating paper (badly), rather than rethinking the medium of text in a digital context. For Nelson, the potential of computing is (In the spirit of Vannevar Bush's utopian technology essay *As We May Think* (1945), reproduced in an introductory chapter of *Literary Machines*), systems that interface naturally with human thought and creativity. One issue is that the nature of text has been muddled up. Nelson proposes that the true structure of texts can be decoupled completely from their representation:

> The structure a user sees should be the intrinsic structure of his or her material, and not (as in many "word processing" systems) some combination of the material itself with some set of obtrusive conventions under which it is stored. *(Nelson, 1987, p. 2/5)*

For Nelson, this structure is one of arbitrary media modeled virtually as simple streams of bytes which are uniquely addressable down

to the character. The virtual aspect is that a document does not necessarily consist of bytes stored in one place, in one order, originating with one author. Non-native byte-spans are transclusions. The underlying mechanism of transclusions ensures that virtual documents can be addressed as coherent, continuous byte-spans while transcluded bytes maintain a connection to their native context. Documents are robust to change, as they are fully versioned, and any past version can be retrieved by referencing the content that it had at that point in time.

Links in Xanadu are *meta-virtual* structures, in that they connect virtual documents. In fact, they connect arbitrary spans, meaning that a link may create a unique virtual document by linking to or from some new combination of byte-spans. (Also creating transclusions in the process). They are stored independently of documents, though they may be collected within documents (Nelson proposes directories as a user-led structuring mechanism, in absence of any hierarchical storage structure 1987). Links are created and owned by users, and maintained by the Xanadu backend. Like documents, links are resistant to change, and do not break as content is relocated or modified.

Virtual documents and links are the primary entities of a publishing and reading system where everything is available to reuse at arbitrary granularity, structured only by the whims of the reader or curator.

Hence, the assumptions underlying transclusion are that the mechanism is operating within a structure that is disjoint, anarchic, flexible and yet 'all-together', compatible, and robust.

The representation of this "docuverse" of content to users is not provided or constrained by Xanadu:

> (How you will look at this world when it is spreadeagled on your screen is your own business: you control it by your choice of screen hardware, by your choice of viewing program, by what you do as you watch. But the structure of that world– the system of interconnections of its stored materials– is the same from screen to screen, no matter how a given screen may show it.) *(Nelson, 1987, p. 2/7)*

In fact, Nelson envisions Xanadu as infrastructure, a public information utility, with private Xanadu repositories on personal devices coexisting with publicly accessible Xanadu terminals, as illustrated in Figure 2. In this infrastructural vein, Nelson also envisions a self-governing copyright system implemented in the Xanadu backend, with per-byte-royalties for viewing of copyrighted material. Importantly, this means that quotation and other transclusions preserve copyright, with royalties being distributed among authors of the material in compound documents proportionally to their share of the content. (1987)

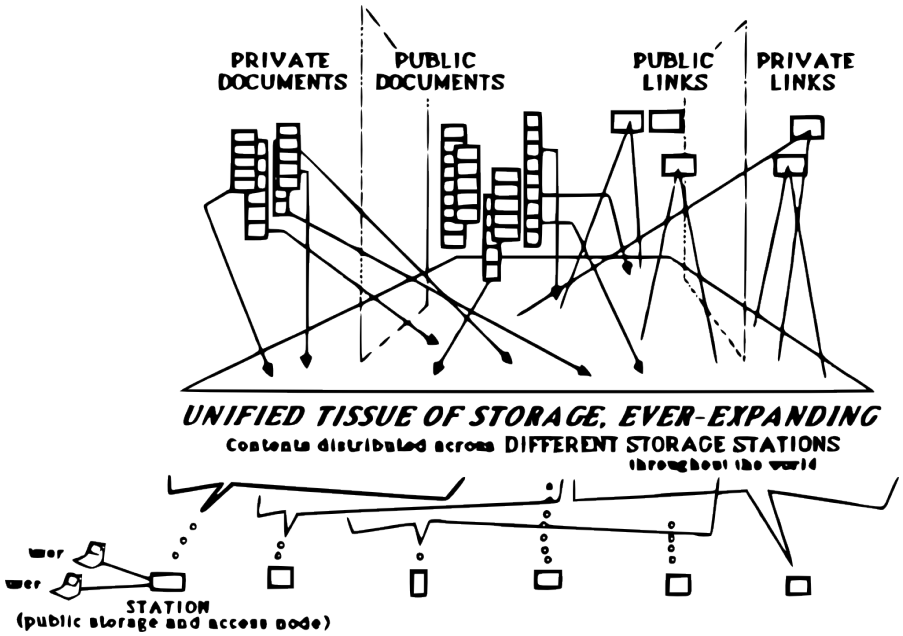

Figure 2: Xanadu as a massively distributed information utility, transclusively unifying documents and links in public and private storage. Reprinted from (Nelson, 1987, p. 0/10) with permission.

In (1995), Nelson presents transclusion and linking extracted from Xanadu as the essential interactive phenomena which realize his vision of hypertext. Here, he clarifies that the maintained connection to native data built into transclusion is not only a back-end concern, but also a core feature for users, providing the ability to traverse transclusions to their sources. This feature is demonstrated in Nelson and Juste's web demo illustrating text transclusions (2014).

Among the user scenarios Nelson presents sporadically are examples of many interesting hypertextual interactions, e.g.,

- Studying the history of a document by recalling versions. (Nelson, 1987, p. 2/15)
- Transcluding texts side-by-side to intercompare them, annotating via links. (2/24)
- Creating composite documents by transclusion. (2/32)
- Creating alternative versions of non-owned documents by transclusion. (2/38)
- Private documents integrating public byte-spans with transclusion. (2/42)
- User created directories of links for ad-hoc categorization and curation of the docuverse. (2/49)
- Experimenting with alternative organizational strategies for documents simultaneously. (Nelson, 1995)

individuals are free to do with material in the network as they please, down to how it is represented and recontextualized. this is because that is in fact a truer way for software to implement how people actually think and work (According to Nelson) Transclusion is greatly illustrative of this, because it allows (re)composition of others' material, while preserving the hypertext structure.

Nelson is adamantly focused on the idea of a system which maintains the structure of knowledge while allowing completely unconstrained interaction by users. Transclusion reifies on one hand the idea that information is inherently able to be in several places at once, and on the other the notion that anyone ought to be able to use their information the way they damn well please. Barnet notes the individualistic values inherent in Xanadu: "This aspect of Nelson's vision cannot be ignored. Xanadu was (and still is) personal in the most libertarian, 1960s Californian sense. Links, Nelson maintains, furnish the individual with choices, with the right to choose." (Barnet, 2013, p. 79)

2.2.2 *Recent work on transclusion*

I summarize themes and findings in recent research applying transclusion. In my analysis of software transclusion as a software architectural mechanism and design principle, I draw from these findings.

Krottmaier and Helic (2002) consider issues and use cases for web-based knowledge work applications using transclusion. They are primarily interested in bidirectional text transclusions for cooperative knowledge sharing applications, e.g. university course materials and discussion fora. Bidirectionality of transclusions specifically allows authors and readers to discover how work is reused after publication. Krottmaier and Helic conclude that in a knowledge work context, transclusion is worth pursuing as an alternative to traditional reuse by copying.

Bernstein (2003) describes how the Tinderbox hypertext tool pursues the goal of helping people discover and document emergent structure in personal information spaces and electronic notebooks. Tinderbox is a hypertext system for editing, sharing, and analyzing notes. Bernstein uses the terms *montage*, *transformation*, *collage* for different linking techniques, where collages are composite documents arranging transclusions in a malleable layout.

He finds that a spatial hypertext collage approach is useful for establishing informal semantics of some subject of study which is documented by disorganized notes. Tinderbox additionally uses prototypical inheritance of user-written notes to support incremental formalization of notes as users develop taxonomies, and supports transclusion into the set of notes from any http-accessible document.

Di Iorio and Lumley (2009) take up transclusion as a mechanism for creating modular documents that are maintainable and reusable in a collaborative work setting. Their outset is that current XML implementations of document inclusion do not support Nelson's vision of "composite documents where each fragment could be permanently identified, retrieved and aggregated within the system" and consequently powerful navigation and manipulation of interconnected documents. They investigate the pragmatics of extending XML with support for transclusion, focusing on three issues for giving users control and power over modular content: richness of information about (nested) inclusions, the format of viewing and manipulating inclusions, and the liveness of inclusions. Di Iorio and Lumley argue that a good implementation of transclusion must include metadata about transcluded documents to support use cases where the structure of a document is the subject of interaction. They also consider pros and cons of static and live inclusions, and how they interact with greedy or lazy evaluation schemes for serving composite documents. In their proposed implementation, a viewer of transcluded documents may switch between transparent viewing (where all transclusions are flattened to a transparently cohesive document, as in Xanadu), and modes that support explicit manipulation of transclusions by highlighting sections of text and annotating them with metadata.

Maurer and Kolbitsch (2006) implement transclusion in a standard HTML-based environment, with no additional markup. They build a HTML- and Javascript-based client-server application which stores and recalls transclusions created in a custom authoring environment which supports fine-grained transclusion of text ranges in arbitrary webpages. Their design goals are to create a tool for transclusion which is comparable in ease of use to the standard copy-and-paste mechanism, applicable to any document on the web, and able to transclude any portion of text from the linked document. Maurer and Kolbitsch do not take up more advanced transclusive interactions than producing composite documents (Maurer and Kolbitsch, 2006).

L.-C. Lee, Lutteroth, and Weber (2010) investigate transclusion as an interaction for reuse and decomposition, applied to facilitate powerful end-user GUI customization. They present the Auckland interface Model (AIM) GUI specification and customization system. AIM is an open-source, cross-platform technology that represents GUIs as documents that can be loaded, saved, annotated, and changed by end users at runtime. AIM separates the concerns of layout, content (individual interactive elements that are present), and data (the types of data that users operate on in the interface) in GUIs. The model allows end-users to customize and decompose GUIs using transclusion. L.-C. Lee, Lutteroth, and Weber show that transclusion of GUI elements facilitates reuse, sharing, and consistency, and introduces different scopes of customization, e.g. for individual sessions or across

several different applications. Additionally, the AIM implementation of transclusion makes it evident to users where the configuration values of each GUI element are being loaded from, making customization scopes visible. AIM is evaluated using the cognitive dimensions framework, and the authors identify juxtaposability, informality and secondary notation, and premature commitment as issues of note for the model.

2.3 FROM HYPERTEXT TO SOFTWARE

Hypertext systems and ideas have influenced user software beyond knowledge work. ideals of software that is understandable and where users are free to make connections and interrelate. to make computing in general more like the vision of hypertext. Promising software systems which are relevant to my themes.

2.3.1 *Collaborative hypertext systems*

While Nelson's hypertext values supporting individual idiosyncrasies above all, other systems were often built with collaboration at their core. I now present a short survey of some interesting hypertext systems and research efforts, focusing on novel collaborative interactions.

NLS (oNLine System) was a first-generation hypertext system prototype for knowledge work, used mainly at Douglas Engelbart's Augmentation Research Center, and famously demonstrated to the computing community in 1968 (Barnet, 2013, p. 59). NLS was a collaborative word processor for documents and programs. In contrast to Nelson's conception of hypertext, it featured a hierarchical organization of texts as trees of documents, sections with headers, subsections, etc. Text links were used to bypass this hierarchy at will, with an addressing system allowing users to point at arbitrarily precise locations, from documents down to characters. "Link-jumping", navigating by links, was only one primitive interaction. NLS links could also be used as arguments for polymorphic editing commands (Engelbart, 1988, p. 220). Possibly the most interesting collaborative feature of NLS was its Mail/Journal subsystem. Conceptually, the NLS design team decided that messages and documents should not be differentiated. All documents produced within NLS could easily be "journalized", made available for reading, annotation, and linking by every user of the system. Combined with the fine-grained addressing mechanism, the NLS Journal became a hypertext documenting in detail, among other projects, its own development (Engelbart, 1988, p. 212).

HES/FRESS (Hypertext Editing System and File Retrieval and Editing SyStem) were hypertextual word processors developed at Brown

University, principally by Andries Van Dam (Barnet, 2013, ch. 4). As with NLS, the system did not distinguish between readers and writers, allowing editing by anyone of any document in the network. FRESS allowed authors to attach keywords to links, facilitating user filtering of the network based on their situated needs. Keywords could also be used to name text blocks for later referencing and searching.

> With these facilities, a student reader could choose to see only annotations left by the professor, examine only those links that led to literary criticism of a poem, ignore for the moment all the comments written by classmates, or select all poems written by a certain poet or on a certain subject.
> *(Yankelovich, Meyrowitz, and Dam, 1985, p. 23)*

2.3.2 *Hypercard*

Hypercard was an environment and toolset for prototyping and building applications for the Macintosh computer, designed by Bill Atkinson (Atkinson and Winkler, 1987). Hypercard models applications as stacks of cards with text and graphical content, links to define the structural relationship between cards, and scripts providing application logic. This model encourages users to develop applications in an interaction-centric manner, by first creating interface components, and then scripting them with behaviors. In a sense, Hypercard envisions user software as hypertext augmented with behavior. This notion was intended to make software a creative mass medium: "I think it is going to open up the software architecture of the Mac and let the nature of information on a computer go from text and graphics, to text and graphics and interaction." (Atkinson and Winkler, 1987).

Hypercard came with a set of pre-built stacks and interface elements, and user-made stacks were also distributed through public networks while the application was in wide use. Atkinson fully intended for users to learn programming and application design by investigating and adapting other stacks.

Halasz notes that the tailorability and scalability of Hypercard is a lesson for hypertext designers to absorb (Halasz et al., 1988).

2.3.3 *Hypertextual Software in Smalltalk*

The seminal programming language/system Smalltalk was developed by the Learning Research Group team at Xerox PARC as one experimental step towards Alan Kay's vision of *personal dynamic media*, which laid out the long-term goals for personal computing (Kay, 1993; Kay and Goldberg, 1977).

Smalltalk was designed as a tool to make computing palpable and useful to anyone. Kay considers computing literacy, in the sense that

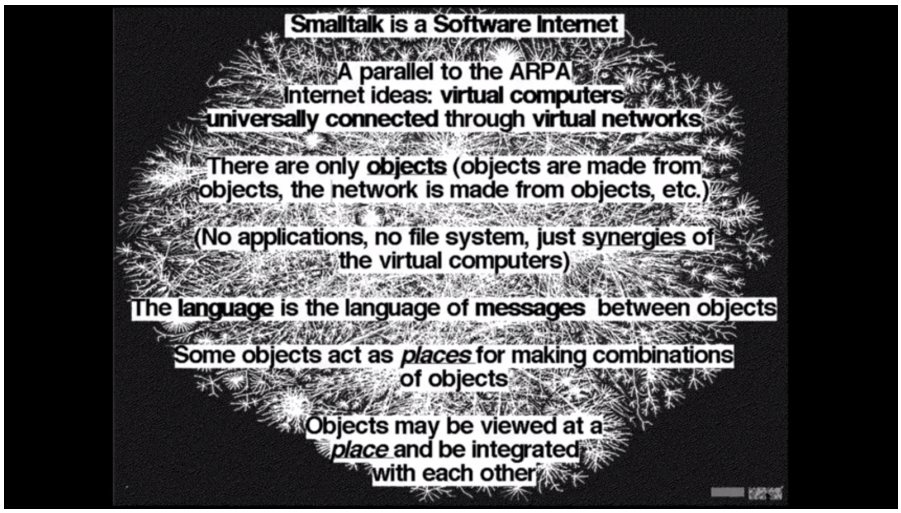

Figure 3: A frame from Kay's presentation of the Smalltalk system in his tribute to Ted Nelson. (Kay and MacBird, 2014)

computing should be as natural a part of daily activity and thought as reading and writing, to be the goal of personal computing. This is essentially different from pursuing an ideal of non-programmer-operable software. Smalltalk's proposal for a simple and powerful model of computing for human activity is object-oriented programming.

Objects have some private state, and can send and receive messages. Messages are the universal mechanism of querying state and manipulating objects. Kay's vision was of objects as virtual computers, i.e. the primitive unit of computing is itself a computer.

In a video tribute to Ted Nelson (Kay and MacBird, 2014), Kay demonstrates an emulated version of a Smalltalk system as it was internally at PARC in the 70's. The system appears to be a complete operating system, though it is simply a composition of objects:

> Smalltalk is in the form of an internet of software computers that is completely self-contained. There's no separate operating system, applications, etc., only software computers communicating with each other, each simulating some aspect of the personal computer system. Some objects simulate characters on the screen, some simulate pictures, some windows, some places where the users can do things. *(Kay and MacBird, 2014)*

The video presentation relates the work of Nelson and Kay, both in the methodological orientation of working on systems that illustrate the revolutionary potentiality of computing, and by the interesting notion of a *software internet*, described in Figure 3. This conceptual blend of hypertext and user software is more than a colorful turn of phrase. Kay demonstrates a hypertextual interaction in his demo

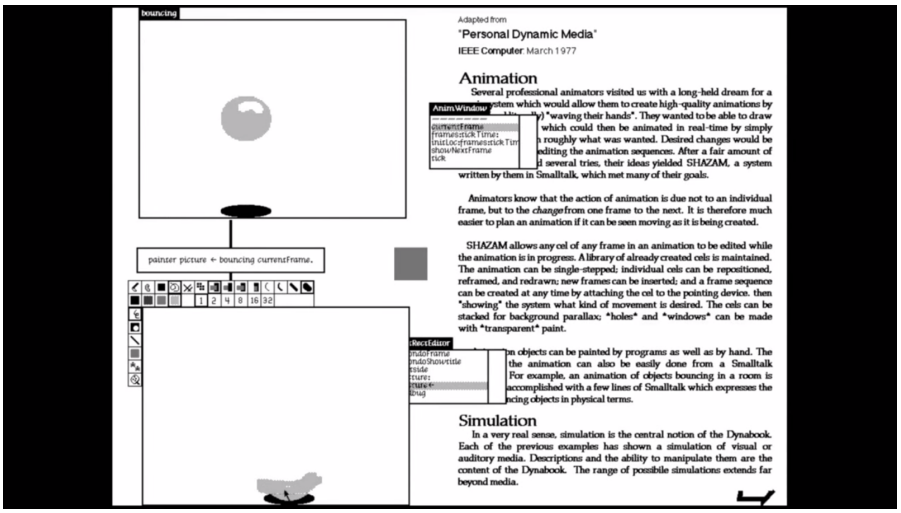

Figure 4: The emulated Smalltalk system. In the upper left window, an animation of a bouncing ball is running. In the lower left window, an image editing program is being used to edit a specific frame of the animation. The box connecting the two windows is added by a mouse gesture, and is filled out by the user with a line of Smalltalk (`painter picture ← bouncing currentframe`) linking the picture of the Painter window with the specific frame of the animation. (Kay and MacBird, 2014)

by gesturally creating a dynamic link between two window objects, which he programs to assign a particular frame of a running animation to be edited in a painter program (Figure 4).

(Meyrowitz, 1989) roundly criticizes hypertext researchers for designing systems which assume that users are essentially operating in a vacuum:

> Carefully examining the systems created to date, however, uncovers a single common thread that I contend explains why hypertext/hypermedia systems have not caught on: virtually all systems to date are insular, monolithic packages that demand the user disown his or her present computing environment to use the functions of hypertext and hypermedia. *(Meyrowitz, 1989)*

These systems include NLS, FRESS, etc. Meyrowitz argues that the fundamental potential of hypertext will not be fulfilled before the paradigm is integrated in a standardized manner with general software. Kay's illustration of software linking is an intriguing view into this vision of potentiality.

2.4 INSTRUMENTAL INTERACTION ARCHITECTURES

Beaudouin-Lafon and collaborators have developed *instrumental interaction*, a model for interactive software based on notions of ubiq-

uity and flexible multi-user systems. I take up this model as a design framework, because of its documented expressive leverage and fit with the kinds of interactive systems that I pursue in this thesis. Here, I review the instrumental interaction and some of the research applying it.

2.4.1 *Instrumental interaction*

In (2000), Beaudouin-Lafon describes *instrumental interaction*, an interaction model for post-WIMP user software, within which interactive systems consist of domain objects and interaction instruments. Domain objects play the role of data, being object and purpose of interactions. Instruments are the means by which domain objects are manipulated by users. They are mediating objects. Instrumental interaction is intended to provide a design space for new interaction techniques and a set of properties for comparing them. Beaudouin-Lafon illustrates this by applying the interaction model in the redesign of a text-search feature.

With Mackay, Beaudouin-Lafon concurrently considers design principles for large scale, complex visual interfaces (Beaudouin-Lafon and Mackay, 2000). These principles are applied practically in a redesign of the CPN2000 software for working with colored petri nets. The principles are:

REIFICATION Extending the notion of the object.

POLYMORPHISM Extending the power of commands with respect to objects.

REUSE Capturing and reusing patterns of use.

Beaudouin-Lafon and Mackay illustrate how these principles can be applied to create simple, powerful interfaces that re-mediate interaction issues of a complicated software system.

Beaudouin-Lafon further expands on the arguments and method of the 2000 articles in (2004). He argues that the only way to significantly improve user interfaces is to shift research focus from "point designs", GUI-oriented designs which seek to resolve specific issues within the WIMP framework, towards designing interactions, i.e. holistically reevaluating system design through new interaction models. Instrumental interaction is considered as a model for post-WIMP interactive systems.

Beaudouin-Lafon also argues that the underlying architectures of systems must change to support a more open interaction paradigm like instrumental interaction. Specifically, architectures must give more control to users, be reliable in a changing environment, and scale well. He enumerates these properties of post-WIMP interaction architectures:

REINTERPRETABILITY The ability of the user to reapply parts of the system in other contexts.

RESILIENCE The ability of the system to resist change.

SCALABILITY The ability of the user to interact with, and the system to handle, data at different scales.

Beaudouin-Lafon concludes that powerful interaction models and architectures are necessary for a move towards designing interactions.

2.4.2 *VIGO*

Klokmose and Beaudouin-Lafon (2009) argue that current application-centric approaches for software are inadequate for user interfaces in multi-surface environments, in which interaction spans the surfaces of several devices. They present the VIGO (Views, Instruments, Governors, and Objects) architecture, an architecture for interactive systems based on instrumental interaction.

They take multiplicity, dynamism, heterogeneity, and distribution as ideals for multi-surface interaction. Their goal is to support fluid interaction across mobile and stationary devices, as well as the ability to configure interfaces according to available devices and users' needs.

Klokmose and Beaudouin-Lafon are critical of the ubiquitous desktop interaction model WIMP and its architectural models such as the Model View Controller software pattern, which couple interactive objects with semantics for manipulation, and as a result does not support unanticipated interactions by users. This encapsulation is a purposeful feature of object-oriented software architecture, which supports system reliability. The VIGO architecture deconstructs the notion of an application and distributes the interface across surfaces to support the flexibility inherent in instrumental interaction.

Objects are passive entities which expose their state through directly accessible properties. They do not provide methods for interaction, nor do they have any responsibility for maintaining internal consistency. This allows shared and distributed objects to be implemented efficiently.

Views are similar to their MVC counterpart, being responsible for representing objects on individual surfaces in ways that are perceptible to users, e.g. rendering a vector graphics data as shapes on a screen. Views are strongly coupled with objects they represent, reflecting changes to objects continuously. Device-dependent views allow objects to be represented optimally on different devices.

Instruments implement the notion of mediating entities which transform user actions into commands on the targeted domain objects. They are used to transform objects directly, through a chain of instruments, or in negotiation with governors applied on the object.

Sometimes, it is helpful to reify instruments as objects, exposing their state and allowing them to be acted on directly.

Governors are the entities which implement system constraints and reactions. They are applied to objects. For example, a governor implementing the rules of a specific game may be applied to a game board object. Thus, when users use simple movement instruments to manipulate game pieces, the governor ensures that only transformations which are legitimate game moves are performed.

Such constraints should not be the responsibility of instruments, since that would limit the potential for flexibly reapplying instruments significantly, e.g. resulting in a drag-move instrument which only moves objects according to the rules of chess, rather than moving any object with position attributes.

The radical deconstruction of the architecture of interactive software to support flexible interaction in distributed systems is highly relevant to my themes, and can be considered as one concrete proposal for an alternative ecology of software objects. VIGO is one starting point for considering what content we are reusing when applying software transclusion.

2.4.3 *Shared Substance*

Gjerlufsen et al. (2011) continue the technical exploration of interactive systems in multi-surface environments. Gjerlufsen et al. present Substance, a data-oriented programming framework and Shared Substance, a middleware technology for distributed applications, as a flexible solution for multi-surface interactive systems. They apply these technologies to design systems featuring colocated multi-device collaboration where users analyze and compare large collections of potentially heterogeneous data. The prototypes are designed specifically for the *WILD room*, an interactive room featuring a large composite wall display, an interactive table, and a motion controller, along with users' own commonplace devices like laptops, tablets, and trackpads.

Gjerlufsen et al. suggest data-orientation as a powerful programming paradigm for distributed interaction, Much like VIGO, Shared Substance makes the point that objects modeled as exposed data rather than encapsulated class instances, along with a loose coupling of data and functionality, can be applied to support flexible multi-user and multi-device interaction systems.

Within Substance, data is the primary structuring mechanism in applications, as opposed to classes in object-oriented program architectures. *Nodes* and *facets* represent data and functionality, respectively. Shared substance models applications as trees of nodes with zero or more facets attached. Both primitive object types can be added or

removed at runtime. Facets are attached to nodes to provide some functionality for that particular node or nodes within its subtree.

Gjerlufsen et al. illustrate how, in addition to modeling more typical data structures, nodes can be applied represent shared objects, legacy applications, and devices in a uniform manner. Extending the notion of data-orientation to legacy applications and hardware allows Substance developers and users to dynamically compose the subsystems in the WILD room, e.g. hooking up the motion controller in the room to the distributed application by simply listening to the state of the node representing its input, and adapting the functionality of the highly specialized Anatomist application by interacting with a Substance tree representing its internal data structure.

Shared Substance's distributed application model allows these trees of data and functionality to be distributed across devices and shared at will. Nodes in Substance tree have a universal address, allowing them to be shared via either a remote procedure call-type protocol called *mounting*, or *replication*, where identical copies of shared objects are maintained in several locations. Objects which are marked to be shared are made discoverable on the local network by the middleware layer. Everything modeled by Substance is shareable since the tree model is consistent, i. e., arbitrary segments of applications, data, functionality, resources, and devices can all be interacted with in several locations at once. The replication method of sharing objects supports facets to be mounted on local copies of objects to provide some localized functionality, e.g. replicating complex graphical data on a tablet, with a facet attached to render the data in a suitable way for the device's speed and specification.

In concert, Substance and Shared Subtance encourage developers to *structure distributed applications for sharing*. This supports the design of a system which takes advantage of interesting opportunities inherent in distributed parallel interaction scenarios. The resulting design has high design-time and run-time flexibility.

2.4.4 *Webstrates*

Klokmose, Eagan, et al. (2015) investigate *shareable dynamic media*, influenced by Alan Kay's vision of personal dynamic media (1977). Shareable dynamic media are "*malleable* by users, who may appropriate them in idiosyncratic ways; *shareable* among users, who collaborate on multiple aspects of the media; and *distributable* across diverse devices and platforms." (Klokmose, Eagan, et al., 2015)

Webstrates (web + substrates) is a client-server layer for the web which makes real-time sharing a low-level feature of web technology. Pages served by the Webstrates server, webstrates, are shared objects. Changes to a webstrate's DOM, i.e., web pages with HTML content, embedded Javascript code, and CSS styles, are transparently

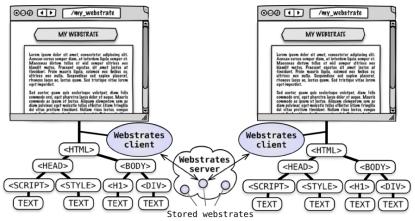

Figure 5: One webstrate open in two browsers. (Reprinted from (Klokmose, Eagan, et al., 2015) with permission).

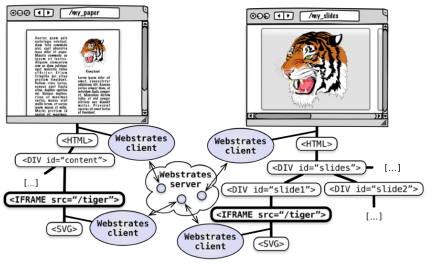

Figure 6: Two webstrates transclude the same image in different contexts (Reprinted from (Klokmose, Eagan, et al., 2015) with permission).

persisted on the server and synchronized across all clients viewing that particular webstrate. Klokmose, Eagan, et al. apply this to create multi-user and multi-device software on the web.

Shareable dynamic media are compositions of *information substrates*, software entities that act as applications or documents depending on use. The notion of substrates originates in the *paper substrates* developed in (Garcia et al., 2012), components of a system merging interactive paper and desktop applications. Mackay details the concept in a 2015 keynote lecture covering, among other work, (Garcia et al., 2012):

> This notion of a substrate is the notion of creating a structure or a way of organizing information that has constraints and rules that are identified by the person, that can evolve over time, and that allow [users] to build in details in various ways that they want. *(Mackay, 2015)*

The simplest application of Webstrates, where two users operate the same web application concurrently, enables collaborative editing of any web document, e.g., a collaborative note-taking webpage (Figure 5). Klokmose, Eagan, et al. apply transclusion to extend the design space of webstrates to include asymmetric collaboration. By applying `iframes` to include webstrates within webstrates, a shared document can be multiply present in different operating contexts (Figure 6). While `iframes` are, as a rule, unable to interact due to the same-origin policy of the web, webstrates are colocated on one server, circumventing this limitation.

Klokmose, Eagan, et al. thus introduce the notion of software transclusion, where interactive tools can be knowably in more than one place in addition to data. The authors describe and assess several demonstrations of shareable dynamic media designs, including:

- An asymmetrical text editing setup where a common document is transcluded into two web-based text editors which differ in their representation of the document and editing paradigms.

- A shareable citation-making instrument which a user copies from one editor, and embeds in another to instantly extend it with a new feature.
- A program editor which transcludes other webstrates and allows users to script and style them as they are in use.

These prototypes are evaluated as shareable dynamic media.

Since Webstrates adapts the DOM as a shared data model, it encourages a DOM-centric data-oriented application design paradigm. Webstrates is comparable to Substance in this regard, as they share the goal of supporting elegant and flexible sharing. The DOM does not provide for the hard distinction between nodes representing state and facets representing behavior. A similar paradigm can be achieved, however, by carefully separating concerns of state and behavior between non-script HTML and Javascript DOM nodes. The rationale for a data-oriented approach to designing information substrates is that, as with Substance, it allows uniform sharing of arbitrary elements of the system, be it data, behavior, or devices.

Software transclusion supports information substrate-based design by allowing cohesive substrates to be constructed independently, and flexibly composed. How to design and construct information substrates to take full advantage of transclusion is not dealt with at length in (Klokmose, Eagan, et al., 2015), but is taken up in the further chapters of this thesis.

(Bouvin and Klokmose, 2016)[submitted to HYPERTEXT '16] consider Webstrates as a hypertext technology. They argue that it supports classical hypermedia virtues as realized in first- and second-generation systems, and since abandoned (Section 2.2, Section 2.3). They use Webstrates to realize two-way linking, and annotation of shared documents in the browser. Bouvin and Klokmose argue that in addition to providing an a simple infrastructure for shared applications, Webstrates transforms the web into a hypertext-based hypertext authoring environment.

With Webstrates, it is possible to realize software prototypes for complex collaborative activities, drawing heretofore hypothetical (or complex and esoteric) technology into the realm of empirical research for anyone with access to a web browser.

2.4.5 *Previous work - Experimental prototyping*

Webstrates provides an environment for investigation of my research themes for user software. I used Webstrates as an experimental design tool in a prototyping study as part of (Tchernavskij, 2015).

I built two working software prototypes based on asymmetric collaboration scenarios, and analyzed their novelty, flexibility, and reliability from a software architectural perspective. From the analysis,

Software precedent (Type)	Concepts
Smalltalk (Language/System)	Chaining together objects, software internet
Instrumental interaction (Interaction model)	Instruments, reinterpretability, resilience, reification
Shared Substance (Framework)	Data-oriented tree-like applications, sharing by mounting/replication
VIGO (Architectural model)	Governors, objects
Shareable dynamic media (Conceptual framework)	Malleability, shareability, distributability
Webstrates (Design tool)	Software transclusion, information substrates

Table 2: The software precedents and associated concepts which I apply in the rest of the thesis.

I derived two proposed design patterns for reusable components implementing collaborative interaction features that fit into the conceptual framework of shareable dynamic media. I reframe these prototypes in the context of software transclusion in Chapter 4.

In my reflection on the two studies in (Tchernavskij, 2015), I argued that a practical design language could be built on top of the basic components of Webstrates to realize novel software applying shareable dynamic media.

In this thesis, I propose such a design language, centered around information substrates and software transclusion. I apply Webstrates as a reference implementation of these phenomena, from which I extract a proposal for a general model. I apply the model to describe, evaluate, and generate software constructs and systems which take up issues under my research themes.

3 DEFINITIONS

I present once again the main phenomena that will be the subject of the rest of the thesis, including a working definition of software transclusion, which I investigate thoroughly in the next chapter, and descriptions of what kind of user software and collaboration I will focus on in my design phase. Finally, I reiterate my research themes in view of the context provided in Chapter 2.

3.1 SOFTWARE TRANSCLUSION

The great virtue of hypertext systems is their ability to represent and handle complexity (Barnet, 2013, p. xx). Hypertext presents a view of the world where deep interconnections are ubiquitous, and interactive systems consequently directly support collaboration, non-hierarchical (re)organization, and plurality in use. The complexity of how texts and media are produced, consumed, manipulated, and applied are acknowledged and provided for. (Meyrowitz, 1989) argues that the power of hypertext and hypertextual interactions must be integrated with software in general rather than being limited to esoteric systems that delimit hypertexts as work artifacts delimited from other kinds of computer-mediated work. (Kay and MacBird, 2014) show how one hypertextual interaction, dynamic linking of variables between programs, can be applied directly in a model of software which dissolves the now-assumed isolation between individual applications and their associated work artifacts.

Transclusion is an interesting decomposition mechanism and user interaction because of its practical and theoretical flexibility, covered in Section 2.2. Transclusion originates in, and is especially tied to, Nelson's vision of hypertext. (Klokmose, Eagan, et al., 2015) show that transclusion is applicable to software expressed as *information substrates*, a deconstructive model not entirely unlike Smalltalk's object-orientation. In their experimental software environment, Webstrates, websites are automatically distributed and synchronized between viewers, augmenting the web with support for rudimentary real-time transclusion. They then build interactive software as information substrates with HTML/Javascript/CSS. Software transclusion is literally what happens when the content that is "knowably in more than one place" is interactive. Transclusion inherently supports a subjective viewing experience, and this aspect of Nelson's hypertext paradigm is extended to subjective interactions, e.g. asymmetrical collaboration

systems, by information substrates, which support the decoupling of content from representation and interaction.

Nelson defines transclusion as "reuse with original context available" (Nelson, 1995) and "the same content knowably in more than one place" (Nelson, 2012). In existing Webstrates prototypes, transclusions gravitate around shared virtual objects. This is perhaps the core notion of software transclusion: that data and interactions can be in several places and be transparently operated on and with as if they are not. My working definition of software transclusion is

SOFTWARE TRANSCLUSION

> the same content in more than one place as a shared virtual object, representing data or interaction logic.

As discussed in (Di Iorio and Lumley, 2009), transclusions can both be live and static, depending on the particularities of the implementation. Software transclusion as I will apply it in this thesis involves both types, but is mainly focused on live transclusions in real-time collaboration with volatile shared objects. (L.-C. Lee, Lutteroth, and Weber, 2010) call transclusion a decomposition mechanism, to emphasize that its primary power lies in allowing flexible reuse of arbitrary digital content. Software transclusion decomposes software so that interactions and data can be decoupled from any particular context, i.e. individual "applications", devices, and users.

In the last chapter, I established that software transclusion is a mechanism which has a theoretically cohesive background, has been practically demonstrated, and is novel. The rest of this thesis is dedicated to investing software transclusion as a user interaction and software composition mechanism through analysis and design. These two angles on the mechanism support different theoretical and methodological approaches, but are both necessary and interconnected, since one underlying goal is to design interactive systems which can be dynamically composed by users.

I will use Webstrates as my experimental platform. It has several advantages for this project: It supports economical experimentation, as it is a simple web server which extends the web with support for software transclusion without any need for specialized API's or tools, and runs in recent versions of the Chrome web browser on any platform or device. Klokmose, Eagan, et al. have illustrated its expressive power, and I have applied it to develop prototypes for asymmetrical collaboration software in (Tchernavskij, 2015). The platform comes with issues as well: Webstrates is fundamentally a hack, a reappropriation of the DOM as a model for distributed interactive software. The technical advantages are dependent on using standardized web technologies to develop software with features and properties that they were not designed for. The stated goal of this thesis is to develop software transclusion as a design principle and model, independent of

the particular implementation context of Webstrates. Webstrates is at once an experimental environment for examining software transclusion in practice, and a compromise with the ideal of the mechanism. I take advantage of this duality by developing a critique of Webstrates and the web as an emerging platform for software as part of my analysis.

In the following chapters, I will generally use transclusion to mean software transclusion as defined here.

3.2 USER SOFTWARE

User software is an informal notion, referring to those interactive systems which directly mediate human activity. Digital calendars, editing programs, terminals, and music players are all examples of user software, while operating system kernels, machine learning algorithms, and backend databases are nonexamples. There is no hard limit for what is or isn't user software, as the examples all involve operation by users at some point in their lifetime, and can be applied within user software, but the concept is easy to grasp as a pragmatic distinction.

A user interface is a view into user software, not a mandatory component of it. In Chapter 4 and 5, I view user software as highly decomposed systems that are not always actively instantiated in a particular user interface. I do not separate interface and underlying system entirely either. I assume instead that the internal models of a piece of software are as interesting as the tools it provides to manipulate those internal models.

3.3 THEMES FOR SOFTWARE

I collect themes and issues from my related work under three research themes, which define the problem space that I argue transclusion can support novel design solutions for.

3.3.1 *Asymmetrical collaboration*

Sørgaard defines cooperative work as a prototypical kind of work, satisfying, to some degree, four criteria:

> (1) People work together due to the nature of the task, (2) they share goals and do not compete, (3) the work is done in an informal, normally flat organisation, and (4) the work is relatively autonomous. *(Sørgaard, 1987)*

Star and Griesemer and C. P. Lee investigate and model cooperative work at a larger scale, with decoupled communities of practice overlapping through shared boundary objects. They illustrate a contin-

uum of collaborative work with axes including routine/non-routine work, hierarchical/flat organizations, and degree of standardization. In this thesis, my view of cooperation leans towards C. P. Lee's, where consensus is continuously negotiated, and standardization is not an end-goal for the participants.

I narrow my focus to cooperative work at a particular scale, where small, colocated or spatially distributed, groups interact concurrently with shared artifacts. Hence, I will not take up interaction in e.g. groups that are distributed in time, large and loosely connected groups, etc.

The collaborations I am interested in are dominated by asymmetry, involving multiple devices, skillsets, backgrounds, goals, etc. An especially interesting type of asymmetrical collaboration in computer-mediated activity is collaboration with heterogeneous work artifacts.

Asymmetrical collaboration is a common phenomenon in "real" life but presents many problems for traditional user software, which is typically built from a one-size-fits-all philosophy. This is especially the case when interaction involves professionals who have complex and idiosyncratic practices that cannot be easily readapted to a standardized software ecology (Klokmose and Zander, 2010; Sørgaard, 1988; Tchernavskij, 2015).

Fundamentally, software artifacts tend to be bad at moving between unequal collaborators. The flexible reconfiguration of work artifacts described in Section 2.1 is ill-supported. Boundary objects, syntonic seeds, etc. are defined by changing structure and role in activities. As a rule, user software is purpose-built to support a well-defined set of interactions, which handle a specific subset of data types modeling a category of domain objects. In practice, software artifacts change in one direction, from malleable, dynamic objects of work to immutable and opaque products (Klokmose and Zander, 2010; Tchernavskij, 2015).

The case study from my previous project is a good motivating example of the complex issues of asymmetry in computer-mediated collaboration, and the value of software with better inherent support for it: At Creuna, large design projects were distributed among "T-shaped" practitioners with idiosyncratic, overlapping, changing, and non-standardized skillsets.

In the formalized work organization of the agency, the work of individual practitioners was separated into "silos" of visual design, system design, software architecture, user interface programming etc. The agency applied groupware and structuring artifacts to model a rationalized, waterfall-model style design process.

This organization of a complex practice is dialectically related to the way the software tools in use model work. The plan-then-execute paradigm of the design process isolates design feedback and resultant learning as error states, rather than adapting to it and supporting reit-

eration. The designers and developers at Creuna worked around this by short-term ad-hoc interdisciplinary cooperation to resolve design breakdowns. These collaborations were evaluated by practitioners as rich in learning and fundamental to successful design work, yet they were ill-supported by the available tools and the organizational structure.

The ideals for asymmetrical collaboration are systems that

- expand to accommodate more users.
- can integrate collaborative interactions with personal systems.
- model shared material in a way that does not require users to agree to a common set of tools and constraints.
- can mediate interactions between unequal collaborators, like teachers and students, in a common space.

3.3.2 *Reconfigurable software*

The theme of reconfigurable software focuses on flexibility for individual users. Reconfigurability covers the properties of reinterpretability (Beaudouin-Lafon, 2004), malleability, and distributability (Klokmose, Eagan, et al., 2015).

(L.-C. Lee, Lutteroth, and Weber, 2010) exemplifies some challenges for designing highly reconfigurable systems. The AIM customizable GUI model allows users to recompose their personal interface at will, reusing configurations or writing their own. L.-C. Lee, Lutteroth, and Weber make efforts to design the model to ensure consistent results, to empower users with expressive power at several levels of abstraction by supporting reuse and sharing of customizations, and to support users' understanding of the underlying model with scope visualizations.

Designs that take up this theme support interactions like

- users (re-)composing their software at will, adding, removing, or modifying data and tools.
- users adapting and reapplying data and tools in ways not explicitly designed for.
- users distributing software across devices.

3.3.3 *Ecology of software artifacts*

Asymmetrical collaboration and reconfigurable software define patterns of use that are ill-supported by user software. The theme of software artifact ecology covers the nouns and verbs of software which can be composed into collaborative and reconfigurable systems.

The VIGO architecture (Klokmose and Beaudouin-Lafon, 2009) and paper substrates (Garcia et al., 2012; Mackay, 2015) are examples of proposals for alternative ecologies of software components that support multi-surface instrumental interaction and creative practice, respectively.

In Chapter 4, I define a model taking information substrates and transclusion as fundamental building blocks of software.

4

A MODEL OF SOFTWARE TRANSCLUSION

Following from my working definition of software transclusion, I present an analysis of the phenomenology (in the meaning of a classification of phenomena) and architecture of software transclusion and develop a model of *transclusive software*.

In this chapter, I answer questions like:

- In software transclusion, what exactly are shared virtual objects?
- Which software architectures work with transclusion, and which don't?
- What kind of interactions does transclusion support?
- How does Webstrates implement software transclusion, and how might it be differently implemented?

I treat Webstrates as a reference implementation of software transclusion. On one hand, the concept originates with Webstrates, and interesting properties of the mechanism can be studied through it, and on the other, there is value in separating software transclusion from this particular implementation. This is the principal concern of this chapter. The correct place to begin is describe the reference implementation in detail.

4.1 WEBSTRATES IN DETAIL

In this section, I summarize the pertinent technical aspects of Webstrates, which was introduced in Section 2.4.4.

Webstrates consists of a server and a transparent JavaScript client. When a Web page is loaded from the Webstrates server, the Webstrates client is served to the browser viewing the page. The client establishes a socket-based connection to the server, through which the data of the given webstrate is loaded based on its URL, populating the Document Object Model (DOM). The DOM is the programming interface for HTML documents, and models the structure of a Web page as it is instantiated in the browser. The Webstrates client observes local changes to the DOM, passing them on to the server, and subscribes to changes from other clients of the same page. The server persists the DOM in a database. Synchronization of the DOM is implemented using operational transformations. (Bouvin and Klokmose, 2016)

Any interactive page running in Webstrates is thus an implementation of the what-you-see-is-what-I-see paradigm of collaborative software, where several users can access an interface which is synchronized completely between them (Figure 5). Since only the DOM is shared between users, different views of a page will not synchronize state like the position and movement of the mouse cursor, or the JavaScript runtime, but will synchronize any modifications to the structure and content of the page. This difference is sometimes subtle, e.g., inline `style` tags are synchronized, and changes are instantiated in real-time as they are modified, inline `script` tags are also synchronized, but the behavior of the page will not change until it is reloaded (since the JavaScript runtime at the client only evaluates the scripts then), while the standard HTML checkbox `input` element is not synchronized between views of a page because its state is not modeled in the DOM.

This sharing scheme presents a good default for the collaborative software prototypes developed in (Bouvin and Klokmose, 2016; Klokmose, Eagan, et al., 2015; Tchernavskij, 2015). It is rarely considered desirable to have user input tools be controlled by several users at once, and it is hard to imagine the benefits of forcing all users to have completely identical browser sessions at all times. However, we can arbitrarily extend the notion of what is shareable by representing it as DOM, e.g. adding a `class` attribute to a checkbox element which is toggled between "checked" and "unchecked" by a script which subscribes to changes to the checkbox, and ensures that the attribute and the actual state remain synchronized. Inspired by (Beaudouin-Lafon and Mackay, 2000), this technique of modeling data in the DOM to integrate it in the Webstrates model of interaction is called *reification*.

Transclusion extends Webstrates from what-you-see-is-what-I-see to generalized shareable dynamic media, as it enables asymmetrical collaborative systems through composition of webstrates.

Transclusions in Webstrates function by applying the standard `iframe` (inline frame) HTML tag, which includes another Web page within a document by reference to its URL. In the context of ordinary Web activity, content in `iframes` is not expected to interact with the hosting page. This is due to the same-origin security policy, which states that scripts running on Web pages cannot access data from pages with different originating servers. This policy does not apply within Webstrates, since all webstrates are expected to be colocated on a single server. Another side effect of working exclusively on colocated documents is that CSS styles can be applied to transcluded content, so that it is possible to display local and transcluded completely uniformly.

Klokmose, Eagan, et al. employ transclusion in a few different modes. A bare document may be added to a webstrate by transclusion, and modified with the interaction tools at hand in the transcluding webstrate. Interactive logic may also be added to a webstrate by

transcluding a document holding a script. In this case, the authors style the inline frame to not be displayed, producing an invisible transclusion.

Another limitation of `iframes` is that they only ever include a whole document, i.e., by default it is not possible to transclude only part of one document within another. The simplest way to work around this limitation is to decompose webstrates into small, coherent documents storing content that is usefully shared, e.g., as in Figure 6, where a single image has its own substrate. We can imagine interaction scenarios where it is meaningful to have substrates containing data which is not expected to be transcluded all at once, either because it is an aggregate of some sort, or because it is very large. To support these cases, a mechanism for partial transclusion is needed. This kind of mechanism is not currently part of Webstrates, but one viable approach may be to augment the addressing scheme of transclusions with CSS selectors. Where a URL locates a document by its location on a particular server, CSS selectors are a query language which locates subtrees and individual nodes of documents by tag type, identifier attributes, and structural relations.

4.2 SOFTWARE TRANSCLUSION AS A DESIGN PRINCIPLE

In (2000), Beaudouin-Lafon and Mackay define three design principles for visual interfaces. Reification, polymorphism, and reuse are normative principles that can support the design of internally consistent and powerful interactive systems. For instance, the principle of reification extends the notion of an object in a system, supporting uniform interaction with groups of objects, abstract properties of objects, interaction instruments, etc.

Software transclusion can be considered as a design principle in the same vein. Software transclusion turns software objects into shared artifacts, and turns a definite point in an interactive system into a variability point which may change at runtime. Transclusions may involve both data and interaction logic.

As a normative principle, software transclusion states that users should be able to dynamically compose software as they see fit, and that data and interaction tools should be adaptable to many different contexts. This property of supporting recontextualization by users is congruent with Beaudouin-Lafon's post-WIMP software quality of *reinterpretability* (Beaudouin-Lafon, 2004). As in shareable dynamic media, software transclusion assumes that software should have low-level support for collaboration, i.e., all software artifact are always potentially held in common between multiple users, devices, tasks, etc. The three key properties of shareable dynamic media laid out in (Klokmose, Eagan, et al., 2015) function as ideals for the expression of the design principle in software. Shareability is directly enabled by

software transclusion. Transclusion for reuse and customization is the subject of (L.-C. Lee, Lutteroth, and Weber, 2010), and can be considered as supporting the property of malleability at a large granularity. This almost necessitates more fine-grained customization capabilities to be available to sophisticated users of a system, since malleability that only extends to recomposing a set of unchangeable, standardized elements is severely limiting. Distributability is a pragmatic necessity for software transclusion if we consider use cases featuring multiple devices and platforms at play.

The power of Beaudouin-Lafon and Mackay's design principles lies particularly in their combination. I have already mentioned how transclusion interacts with reification, which "extends the notion of the object". Where reification makes e.g. abstract commands into user interface objects, the addition of transclusion makes abstract commands shareable. Polymorphic tools are necessary in a context where users may freely transclude heterogeneous data into a document, and can potentially support a fluid interactive experience between devices, data-types, etc. (L.-C. Lee, Lutteroth, and Weber, 2010) illustrate how transclusion can support reuse in the software architectural sense of applying one thing flexibly in several places, e.g. transcluding the same GUI specification in several documents. This transclusive reuse is a variation on one kind of reuse described by Beaudouin-Lafon and Mackay.

A data-oriented approach to system architecture best exploits the potential of software transclusion for novel interactive systems. Gjerlufsen et al. develop a data-oriented architectural framework for distributed interactive systems, and argue that it provides a flexible foundation for sharing application state and behavior (2011). Similarly, the design principle of software transclusion is most powerful in software structured around hierarchical data which can be freely modified and is loosely coupled with behavior. It is not as expressive in a system where there is limited opportunity for decomposing software components, and the shared object primitive encapsulates data and behavior at design-time.

This short rumination on software transclusion as a design principle supports an argument that it can be a useful mechanism in designing systems with high run-time flexibility, supporting, e.g., reinterpretable software components, user customization, and cooperative-by-default systems. Software transclusion as defined here is loaded with assumptions about ideals for interaction, the context of use and the architecture within which transclusion is happening. A design principle is not an appropriate tool for answering questions like

- What architectures occur in systems centered around software transclusion, and what are their trade-offs?

- Can software transclusion support non-trivial asymmetrical collaboration scenarios, where the model of the shared artifact changes over time, or two users concurrently work on an artifact constrained by different models?
- How do users manage dichotomies like private/shared and persisted/ephemeral in interactive systems where software artifacts are by default commonly accessible and persistent?

To answer these questions, pertaining to software transclusion in the context of systems of interaction, a model outlining the interactive and architectural design space of software transclusion is needed. A model allows us to describe, evaluate, and generate interactive systems which are designed to express the potential of software transclusion. In the next section, I describe a model for *transclusive software*, which takes up challenges under my research themes, and applies transclusion as a central mechanism.

4.3 A STANDARD MODEL OF TRANSCLUSIVE SOFTWARE

As I argued in the previous chapter, software transclusion occurs in the interaction between traditional document publishing-oriented transclusion and systems which formulate software as documents. This model is derived from variations upon the theme of transclusion and information substrates, and reflection on the systems covered in Chapter 2. Recall that information substrates are software entities that act as applications or documents depending on use. Substrates are both the objects and mediators of interaction in transclusive software. In the context of this model, a substrate is a hierarchically organized document containing arbitrary data, structured as a tree. Substrates are held in common by all users of a system, and act as shared virtual objects, as opposed to files residing on particular devices. This means that substrates are persisted and synchronized in real time. Each substrate is accessible by a unique address. Users access and manipulate them directly through devices, and indirectly by transclusion. Substrates are expected to be modified continuously by users of transclusive software.

Transclusive software systems are software networks used in collaborative distributed interactions. These networks of information substrates change over time as users add devices or collaborators, move between work situations, refine their practice, etc. Transclusion is an essential composition mechanism in these networks.

Software transclusions are inclusions of substrates within substrates by reference. Transclusions are themselves a kind of node that is supported in the substrate model, which attaches the referenced substrate as a subtree in the referencing substrate. Limiting substrates to tree structures supports conceptually simple inclusion and decompo-

sition, since any subtree of a valid substrate is also a valid substrate. All views of a substrate are synchronized, so transclusions support asymmetrical work on shared material. A transclusion may include a whole substrate by reference to its unique address, or part of a substrate by additionally including a query which resolves to some particular subtree of the transcluded substrate.

Transcluded data is by default transparently part of the transcluding substrate, in the sense that it is for all intents and purposes equivalent to non-transcluded data. This transparency can be subverted, as transclusions are explicit, and any particular substrate may decide to treat transclusions differently.

The architecture of transclusive software swivels around shared data. A simple collaboration where two users simultaneously manipulate the same virtual object is a network of substrates where the shared data is the hub, and each user's personal substrate is a spoke. Passive data substrates are the objects of interactions and the logical mediators of collaboration. It is in this sense that transclusive software is data-oriented. This data-orientation does not preclude the programming of transclusive software in the object-oriented languages typically used for graphical user software. Both Shared Substance and Webstrates are written in OO languages. Rather, they support data-oriented architectures.

The scope of the model is the architecture of layered and interconnected information substrates, and how users and devices interact with them. The model is agnostic towards the particular infrastructure supporting shared objects.

How substrates model software and what the precise semantics of sharing should be is not immediately obvious. The fact that the data represented by substrates is persisted and synchronized between users can mean many things for the actual experience of using transclusive software. Substrates encode the structure and data content of the software, and that is what remains consistent between all views. In the user's experience, this means that representation of data and the local interactive state (any state and computation that does not affect the structure and content of the substrate) remain local to their particular view of a substrate. This separation of concerns allows users relative freedom in how they wish to interact with shared substrates, and clearly delineates what the nature of shared virtual objects is.

As with Webstrates, the mechanism of reification extends the virtual object to include more state. A good example of reification in Webstrates is inline `style` and `script` tags. Being tags in a HTML document, they represent in the structure of a document its representation and behavior, respectively. Structure and content of data is the only *inherent* property of substrates, whereas behavior and representation can be added or left to be specified by a (transcluding)

Transclusive software	Interactive software consisting of networks of substrates which change over time. Transclusion is an essential composition mechanism in these networks.
Substrates	Hierarchical, tree-like documents containing heterogeneous data.
	Both objects and mediators of interaction, depending on use.
	Can model passive data, or be augmented with behavior and representation for interactive logic and representation.
	Are shared virtual objects, whose structure and content is persisted and synchronized between all views.
	Are conceptually not local to any particular device, and can be accessed by any user by a unique address.
	Are malleable, shareable, and distributable.
Transclusions	Are a special kind of substrate node which embeds a non-local substrate within its host substrate by address.
	By default act transparently, as if they are local to the host document, but can be represented explicitly as well.
	Are a mechanism to decompose transclusive software units.
	Are an always-available user interaction, i.e., users can open and change transclusions at will.
Reification	The mechanism of representing something in a substrate to integrate it with a transclusive software system.

Table 3: A summary of the components of the standard model of transclusive software.

use context. Whether or not a substrate reifies these aspects affects what type of substrate it is. A substrate which only specifies structure acts as passive data, whereas a substrate with attached behavior and representation acts more like an application.

Reification is very flexible. For example, it can be applied to user presence, but also to represent devices as substrates, or to share the state of interaction instruments between views.

4.3.1 *Example: A transclusive presentation system*

To concretize the model, I describe an example of what transclusive software looks like in use. For this example, I have adapted case study 2 from (Klokmose, Eagan, et al., 2015) of a Webstrates-based slide presentation system.

The system supports moderated presentations by speakers to audiences, centered around slide decks in the vein of common presentation software. The roles in the system are that of the presenter, who authors and controls the slides while speaking, the audience, who view the slides and write questions, and the moderator, who curates and edits the questions for the presenter.

There are three pieces of variously shared data in the system: the set of slides, the presenter's script, and the questions posed by the audience. These are passive substrates operated on by the users in their respective client substrates.

There are a number of instruments supporting the manipulation of the data. The slide editor is used by the presenter to compose the presentation and the script. This can be a composite of many instruments. The slide controller simply allows the presenter to change which slide is currently in view. The question editor allows users to write and submit questions to the moderator during the presentation. The moderating tool allows the moderator to edit and select questions to show to the presenter when the presentation is over.

An example of a set of devices in use during a presentation could be: a tablet for the presenter to control the slides and view the script, laptops on which the audience can view the presentation and write questions, a large projection screen for a common view of the presentation and moderated questions, and a laptop for the moderator.

Each device accesses a substrate which combines particular instruments and data by transclusion to create a client for a particular role in the system. The presenter tablet transcludes the slide controller and the script, the audience laptops transclude the slide currently in view and the question editor, etc.

The operation of this system is an asymmetrically collaborative interaction between presenter, audience, and moderator. During the presentation, each user accesses a client substrate corresponding to their role, which attaches them to the software network centered around the presentation. The audience submit questions to the moderator while the presenter speaks. The moderator uses their client to pick out a set of suitable questions to guide a discussion, which are transcluded on the projection screen and presenter tablet when the presentation is over.

This example of transclusive software covers a rudimentary type of software-mediated interaction, but illustrates how the model con-

ceptualizes software as a common medium. As a network of substrates, over time the system moves from a single user (the presenter creating the slides and writing the script) to many, collaborating around shared data, interacting via substrates combining simple tools and views of the data. Transclusive software supports distributed and collaborative interactive systems by decomposing software into shared substrates which can be flexibly layered with transclusion, and support a system expanding by addition as more users, instruments, and devices enter the network.

4.3.2 *Applying the model*

As the primary reference implementation of transclusive software, Webstrates overlaps with the model to a large extent. Transclusive software is also a kind of flexible, dynamic media. The purpose of a standard model of transclusive software is to describe, evaluate, and generate transclusive software systems outside of the particular experimental context of Webstrates. The model ignores present limitations of Webstrates to expand the design space for transclusive software beyond what is feasible with this particular technology. This allows me to consider the potential of an ideal software transclusion, by exploratively applying the model.

I apply the model descriptively shortly, as I describe how a number of interactions that deal with issues under my main research themes can be formulated as transclusive software. I also draw in the VIGO architectural model for instrumental interaction-style distributed applications as an example of an architecture that is highly compatible with the virtues of transclusive software. In the final section of this chapter, I critically evaluate and the web as a platform for transclusive software. In the next chapter, I sketch a reformulation of InPlenary, an existing system for distributed collaborative computed-mediated lectures, as transclusive software. I then evaluate the flexibility, fit, and novelty of the design.

4.4 DESIGNING TRANSCLUSIVE SOFTWARE

In this section, I outline the design space for transclusive software. In Section 4.4.1 and 4.4.2, I focus on its software architectural values and describe some useful constructs for interactive software that are in line with these. In Section 4.4.3, I apply the model to describe a number of novel interactions taking up my research themes.

4.4.1 *Virtues and trade-offs of transclusive software*

Virtues

Transclusive software emphasizes particular software architectural qualities. My three main themes especially emphasize flexibility. In software architecture, flexibility is the ability of a system to be modified or extended by additive changes (Christensen, 2010, p. 35). Commonly, flexible architectures are mainly concerned with design-time flexibility, e.g., systems low maintenance costs and high tolerance to being modified and extended. While these values are not antithetical to transclusive software, they are not its main virtues. In malleable multi-user systems, flexibility means the ability of users to change the system at run-time as well as design-time. Transclusive software assumes some behavior in systems that is not a concern or considered exotic in most software, e.g., changing interaction methods, devices, and users, all while the system is in use. This level of volatility is what transclusive software needs to support.

The model of transclusive software posits that transclusive system architectures should support fluid expansion, modification, and cooperation by users. The values of malleability, shareability, and distributability are at the core of the model. Beaudouin-Lafon's post-WIMP property of reinterpretability, stating that users should be able to use a system in contexts it was not designed for, is another ideal for transclusive software.

Trade-offs

There are trade-offs to be made in designing transclusive software. The common measure of a software system's robustness is how reliably it can provide a service, i.e., under what conditions does it no longer do what it's supposed to do.

With transclusive software, it may be difficult to specify what the behavior of a system is supposed to be, since my themes explicitly encourage systems where interactions can occur that were not designed for. It may also be difficult to define where a particular system starts and ends, since transclusive software is so liable to change.

The ideal of users being able to dynamically compose their software presents a risk of incompatibility between substrates limiting or breaking the system. Structured and automated testing of user software systems typically works by testing components of the system in isolation, which presents challenges for transclusive systems. Beaudouin-Lafon suggests resilience, the ability of a system to resist change, as a reliability principle for post-WIMP systems. I do not take up the issue of designing and evaluating for reliability in transclusive software in this thesis, but the topic is critical for the development of real-life systems.

Transclusion supports decomposing software into reusable parts, but how and when decomposition is appropriate is not a trivial issue. For decomposition to support flexible reuse, decomposed components should ideally have low coupling, and high cohesion. Coupling is "a measure of how strongly dependent one software unit is on other software units", and cohesion is "a measure of how strongly related and focused the responsibilities and provided behaviors of a software unit are" (Christensen, 2010, p. 156,157). The VIGO architecture and the interaction mode of instrumental interaction are an excellent starting point for developing useful transclusive software components. Klokmose and Beaudouin-Lafon developed VIGO for local multi-surface systems. Software transclusion goes hand-in-hand with this application area, as it separates the digital objects from individual devices and applications. Transclusions can potentially expand the flexibility of VIGO applications. For example, the VIGO governor allows designers and users to flexibly change the rules a particular object is governed by. With transcluded objects, multiple users can simultaneously apply different governors to the same object. In the next section, I illustrate how the components of VIGO can be designed for and with in transclusive systems.

4.4.2 *Transclusive software constructs*

Transclusions of shared data

The most basic use of software transclusion is the sharing of a document substrate between several substrates for collaboration. Views of a shared document are structurally identical, but representationally independent. I illustrate transclusive sharing of an image, and introduce an informal visual language for transclusive software, in Figure 7.

Shared document substrates are congruent with VIGO Objects (Klokmose and Beaudouin-Lafon, 2009). Like Objects, document substrates are passive entities. The reason for this is that it maximizes the ability of users and designers to recontextualize the data, and for flexible interaction with the documents. While document substrates do not explicitly limit how they can be interacted with, they do act as shared material mediating collaboration, as in (Sørgaard, 1987, 1988).

While transclusions of data should be representationally independent, it is useful for documents to carry some degree of embedded representation (like the default rendering rules of HTML elements), so that users who have no preference for how, e.g., a text document should look can still interact usefully with one.

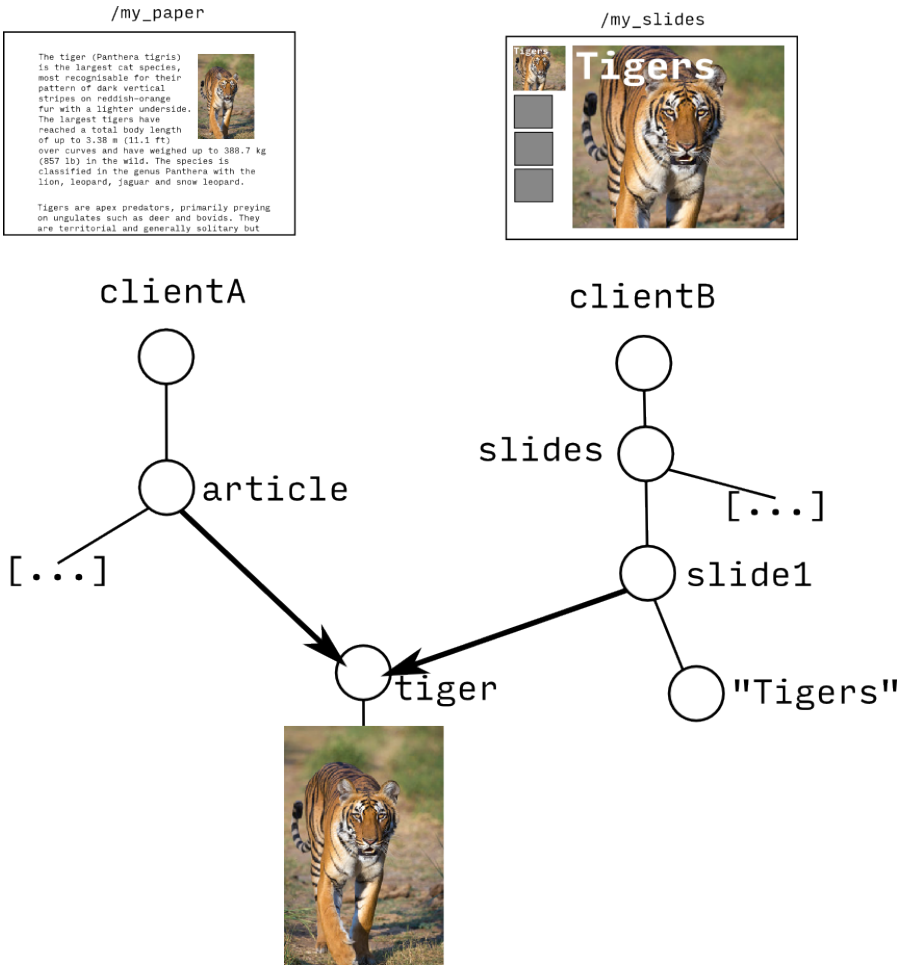

Figure 7: Figure 6 illustrated as a simple transclusive software system. `clientA` and `clientB` transclude an image of a tiger. Substrates are trees of nodes which may overlap by transclusion (full arrows). Nodes are arbitrary data, like text, images, programs, and containers of other nodes. Note that I illustrate transclusions as direct connections between two nodes. In all of these structural figures, transclusion nodes are implicit (Table 3). Photograph by Sumeet Moghe, distributed under CC-BY-SA 3.0 license.

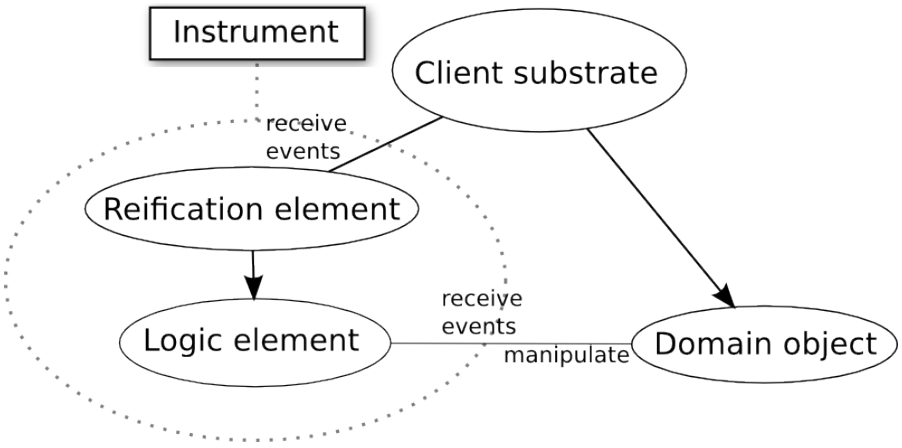

Figure 8: The components of an instrument, embedded in a client substrate transcluding a domain object for the instrument to interact with. Ovals are substrates, arrows are transclusions. The reification element is a subtree included directly in the client substrate (Tchernavskij, 2015).

Transclusive interaction instruments

(Klokmose, Eagan, et al., 2015) introduce shareable VIGO-like interaction instruments. In (Tchernavskij, 2015), I developed a proposal for a reusable design pattern for this type of instrument. The pattern is an attempt to formalize interaction tools decoupled from the context of an application, that can be shared between users, distributed between devices, and reapplied to different applications. The interaction logic of an instrument resides on a substrate by itself, as a script without a corresponding user interface. Instruments are instantiated by adding interface elements and a transclusion to the instrument script directly in the substrate the instrument is to be used in. The script substrate is called the logic element, while the combination of interface and transclusion of the logic element is called the reification element. The pattern is illustrated in Figure 8.

This separation supports two distinct methods of transcluding instruments. In the first case, a new instance of an instrument can be deployed by copying the reification element from one substrate to another. The result is two instruments which share function but not state, since their reification elements are copies residing in separate substrates. In the second case, an instrument can be transcluded, so that its state is shared between several context, by transcluding the reification element in several contexts.

The first case is described in (Klokmose, Eagan, et al., 2015), when one collaborator in an asymmetrical text editing scenario passes a citation instrument to the other collaborator by copy-and-paste. The second case can be considered useful for, e.g., multi-device software where some interaction should be continuous across several devices. A rudimentary example is a newsreader substrate which optimizes

content for the particular device it is being read on, but implements a shared scroll bar instrument which maintains a reader's place in the material across all devices.

Ideally, interaction instruments should express some degree of reinterpretability. It is useful for a reinterpretable instrument to only be loosely coupled to the data it is operating on, allowing the user or designer to substitute, add, or remove data without the instrument ceasing to function. However, the typical pattern for defining an instrument in object-oriented GUI systems is to explicitly program the instrument to listen to user events occurring on some well-defined data. This necessitates that instruments are tightly coupled to data, i.e., that changing one part requires changing the other.

My instrument pattern does not resolve the issue of ensuring loose coupling between instrument and data. There are methods to circumvent this issue, like providing a universal source of events on data that allows instruments to operate with no knowledge of what data is actually present in a system. In instrumental interaction, users conceptually operate a chain of instruments when interacting with the world. For example, in a graphical editing application, the mouse may be used to operate a scaling tool which finally operates on an image. This notion of chaining mediators does not correspond well to the event-driven instruments in Model View Controller-style frameworks.

Transclusive governors

If the primary components of transclusive software are passive data and interaction instruments, the behavior of a software system is predicated on what instruments users bring to bear on their world of shared substrates. Most real-life software systems require some notion of constraints on interaction that are tied to the model of an activity, rather than the tools in use, whether to prevent unrecoverable breakdowns, to guide users, or to comply with some agreed-upon set of rules.

The established circumstances of transclusive software, where users, devices, and practice are volatile, present the need for a flexible relationship with constraints on use.

In the VIGO model, governors are a mechanism for dynamically attaching rules to objects. The guiding principle for governors is that they constrain or guide data manipulation which they are explicitly interested in, and let everything else pass by. Klokmose and Beaudouin-Lafon's primary example is a governor modeling the rules of the board game Othello, which is attached to a checkerboard. In their conception of multi-surface interaction, the board can be moved about and drawn on by users as they play. Governors can also be circumvented at will, creating a soft relationship with rules in software that is more akin to socially constructed rules of behavior.

The life cycle of a governor is

- A governor is attached to an Object.
- An interaction instrument about to modify the object queries all its governors with its modification.
- The governor evaluates the modification and returns an appropriate response, either letting it go through unchanged, modifying it, rejecting it entirely, optionally providing a useful rejection message (e.g. a list of alternate, valid modifications).
- When the object is modified, the governor optionally reacts with additional modification (e.g. changing the checkerboard when an Othello move captures pieces.)

Governors can be attached to multiple objects, and an object can have multiple governors. In transclusive software, governors ought to be implemented as a type of substrate, so that they can be manipulated and shared easily.

In Figure 9 and 10, I illustrate two approaches to attaching rule-like objects to data in transclusive software. I adapt the checkerboard case from (Klokmose and Beaudouin-Lafon, 2009). Figure 9 is a governor-like construct where instruments must actively query any governors attached to an object. This couples the two constructs, creating a scenario where users may unintentionally circumvent the rules on an object by applying instruments which do not comply with the architectural model.

Figure 10 is an alternative construct where a *constraint layer* is a transparent intermediary layer for rule-bound interactions. The constraint layer is decoupled from instruments, as it only reacts to attempted changes on the object it transcludes. I differentiate between this behavior by calling governors explicit, and constraints implicit. This is a less powerful construct than governors, because it does not allow users to decide whether to follow the rules or not per-interaction. It instead enables rule circumvention by simply bypassing the intermediate layer and transcluding the underlying substrate directly. As with data and instruments, transclusion enables asymmetrical interaction with constraints.

4.4.3 *Transclusive user interactions*

Reifying devices

Reification can also be applied to operate on and with devices as substrates. In (Klokmose, Eagan, et al., 2015), the hand position of an actual analog clock is reified in a substrate, allowing configuration of the clock, or sharing of the clock between any number of physical or digital instances through different substrates transcluding the clock.

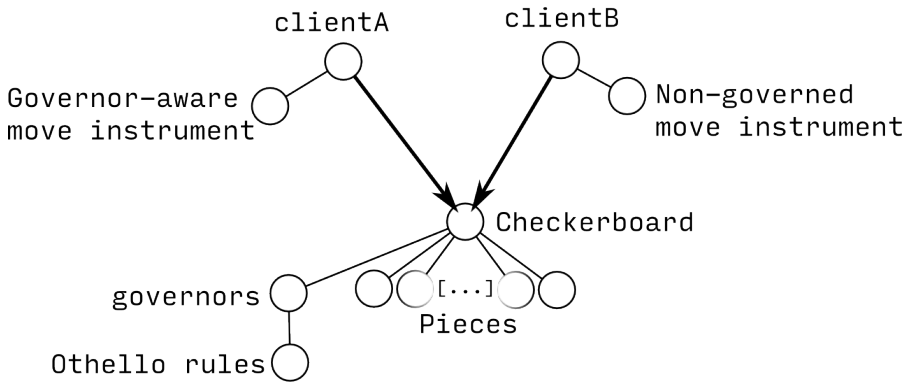

Figure 9: A VIGO-like governor object is attached to the checkerboard. Instruments query governors to ask if a proposed transformation of the governed object is valid, and the governor responds with the final transformation. In this case, `clientA` uses a move instrument which is compliant with governors, and follows the rules of Othello. `clientB` simply transforms the checkerboard directly without querying the governor.

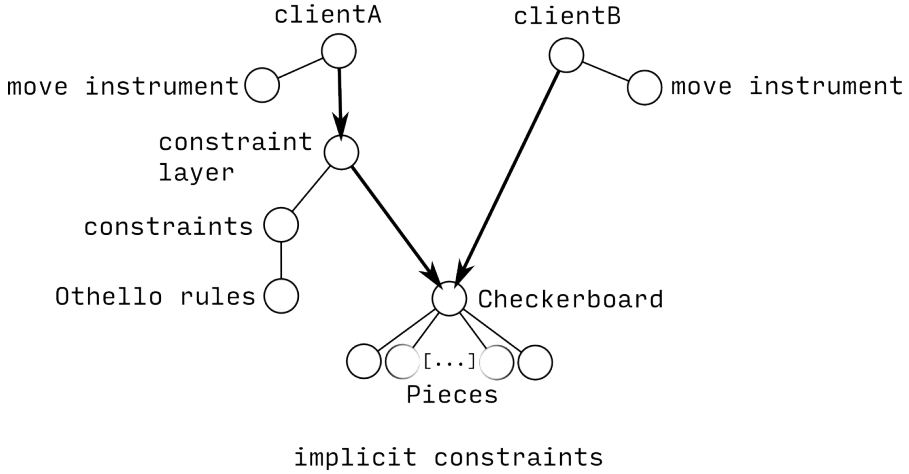

Figure 10: A transparent layer with the Othello rules as an attached constraint transcludes the checkerboard. The Othello constraint object subscribes to transformations on the checkerboard, and accepts, rejects, or alters them. Neither move instrument has any awareness of the constraints. `clientA` accesses the checkerboard through the constraint layer, and is thus subject to the rules, while `clientB` accesses the checkerboard directly.

In (Gjerlufsen et al., 2011), the input a motion sensor registers is reified in the Substance application framework so that any part of the distributed application can be made motion controlled.

Reifying user presence

Similarly, users (or aspects of their behavior) can be reified for, e.g., presence features in collaborative systems. In the Google Docs collaborative text editor, each user's cursor is color-coded and shared to guide collaboration. In transclusive software, achieving this is a matter of reifying each user's cursor in the substrate, and then providing an instrument to appropriately render the cursor in the document.

We can also imagine a substrate which transcludes a marker for each collaborator in a system, carrying a reference to whatever substrate the user is currently working in. This kind of user reification can be used to allow collaborators to easily access a particular substrate when a collaboration is begun, or to notify a user when there is activity in a particular substrate. Going further, user presence can be tracked over time, and collected in a substrate not unlike the standard browsing history feature in web browsers.

Editing substrates with substrates

Substrates are both mediators and objects of interaction. Substrates to edit substrates are a powerful illustration of this. (Klokmose, Eagan, et al., 2015) demonstrate a code editor substrate which works by transcluding another web substrate and exposing all its inline `script` and `style` tags for editing. Within Webstrates, modifying the behavior of a substrate in this manner requires reloading the modified substrate to reevaluate the scripts, while changes to the styling of a substrate are instantaneously effected. As a matter of course, this kind of substrate can be used to edit itself. (Bouvin and Klokmose, 2016) make the point that Webstrates supports a movement towards creating hypertext content within hypertext tools. Transclusive software can nominally be decomposed into mediators and objects, but particular substrates are expected to oscillate between those roles in different contexts of use.

Transclusive multi-device interaction

In a transclusive software environment, the locality of substrates is fluid, and there is ideally little distinction between workspaces running on one machine or several. The transparency of transclusions can be applied to facilitate multi-device interactions. An example of this is illustrated in Figure 11, where a drag-and-drop instrument supports direct manipulation between similar environments running on different devices.

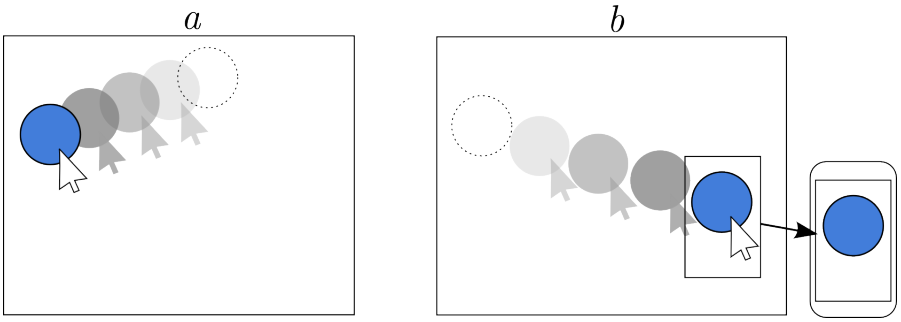

Figure 11: Multi-device interaction as an emergent feature of transclusive software. *a)* In a desktop environment, a drag and drop instrument is used for direct manipulation style interaction. *b)* By transcluding a substrate running on a mobile device into the desktop context, the drag and drop instrument facilitates moving an object between devices.

Combining fluid interaction across devices with transclusive interaction instruments produces distributed instruments. An example application of distributed instruments could be a system where a single user "collaborates with oneself" by working on the same substrate at one table using both a desktop computer and tablet, hosting instruments that take advantage of the available interaction methods (e.g. as in Figure 12), or a system where many users collaborate in a shared environment on a large screen and interact with mobile devices.

These two scenarios, additively constructing a working environment from multiple devices and self-collaboration, are novel interactions within my three themes. They both apply the aspect of substrates that they are conceptually independent of the devices they are viewed on.

Collaborating with different models

The second architectural pattern I sketched in (Tchernavskij, 2015) attempted to deal with asymmetrical collaboration where users work on different types of data. This was motivated by the multidisciplinary collaborations frequently occurring between the developers at the web development agency that was the subject of the case study.

I experimented with creating a layer of abstraction on top of an unstructured shared object, which combines a constraint layer (Figure 10) and an abstract logical object. The constraint layer synchronizes the unstructured object and the abstract object, and prevents interactions on the unstructured object from breaking the synchronization.

The result is a tiered architecture, where the abstract object transcludes the unstructured object, and user substrates transclude either object to work on them simultaneously.

The example I developed realized direct collaboration between a user whose working object is a vector graphics canvas, and a user whose working object is a mathematical graph. The prototype allows the drawing user to freely change the canvas by annotating, moving, or modifying elements of the graph.

The abstraction layer maintains the structure of the graph in the canvas, and creates the abstract graph object. The graphing user can then interact with the same visual representation of the graph as the drawing user, which his editing substrate interprets as events on the non-rendered, abstract graph.

To some extent, the abstraction layer prototype successfully mediated concurrent work on the canvas and the graph with neither the users nor their tools needing to be aware of the other user's working material. Figure 12 and 13 illustrate this collaboration scenario and its underlying transclusive software structure.

However, the architecture of the prototype was messy, and had several unresolved issues. Maintaining synchronization between two objects of different types representing the same abstract object is a non-trivial problem in theoretical computer science and distributed systems which doesn't have widely available practical solutions yet. For example, a theory in development is the concept of *edit lenses* which translate edits between connected data structures (Hofmann, Pierce, and Wagner, 2012).

Constrained transclusions

An interesting idea emerging from the abstraction experiment is that transclusions can be applied not only to expand access to an object, but to constrain it as well. The structure of the constraint layer emerged from that prototype.

We can imagine scenarios where a shared object is distributed to collaborators not by directly sharing its origin substrate, but by sharing a substrate which transcludes the shared object with an attached constraint layer. This kind of unequal sharing can be applied in collaborations with less-capable peers, implementing a set of "training wheels" to prevent them from breaking shared material, or with users that expect different rules, e.g., want to draw on a grid rather than freehand in a shared canvas.

Constraint layers can also be used to implement sharing with access control. One example is a user owning an object who provides read-only access to others by sharing a substrate transcluding the object under a layer that prevents any modification of the object.

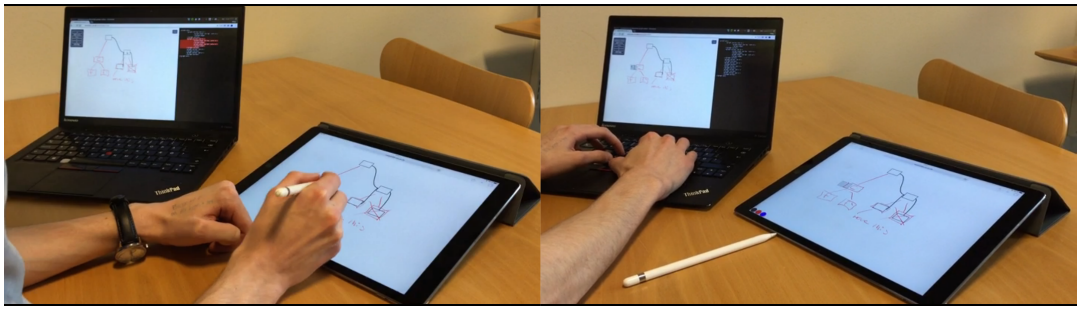

Figure 12: Concurrent editing of a graph as vector graphics and a mathematical graph data structure, realized in the prototype developed in (Tchernavskij, 2015).

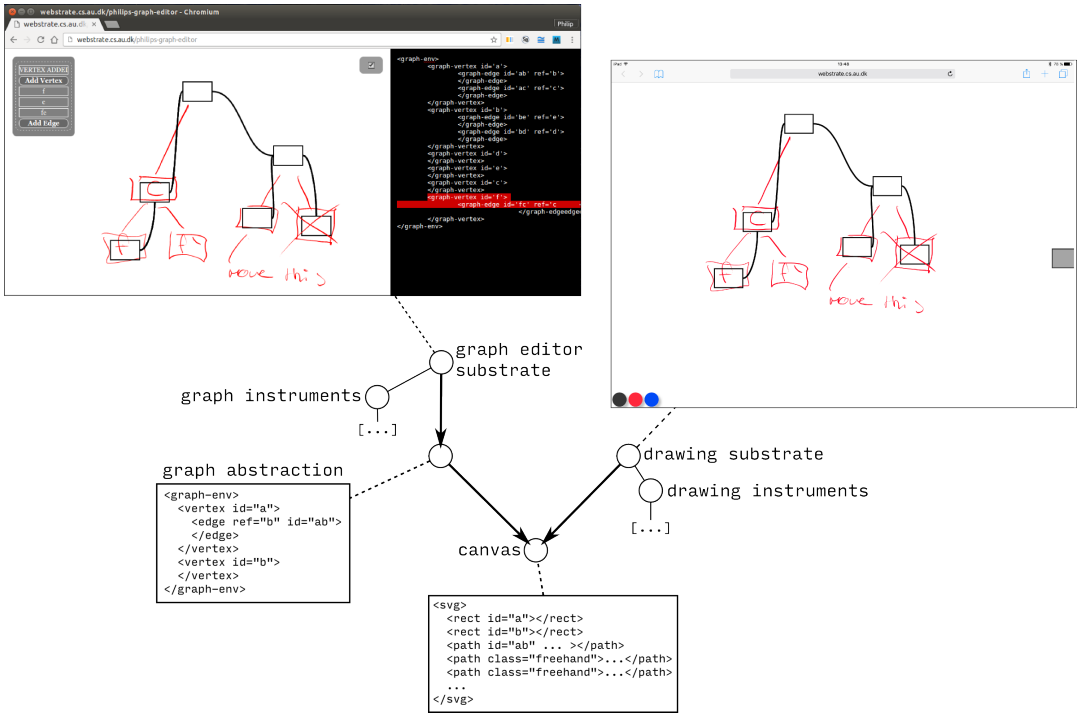

Figure 13: The structure of the transclusive software system in Figure 12. the graph editor substrate is open on a laptop, and is manipulating the custom HTML graph data structure provided by the graph abstraction substrate. The drawing substrate is running on a tablet, and is manipulating a Scalable Vector Graphics element.

Lossy transclusions

Rather than limiting how users interact with substrates, a different type of constraining transclusion could provide transformed substrates to collaborators.

For example, imagine a collaborative system in which some users have access to a substrate collecting location data for a number of people. This kind of data can has many applications, but should not be shared to use contexts where there is risk of privacy violation. To deal with this, a substrate transcluding the data can provide an anonymized or otherwise lossy version of the data to other components of the system.

Transclusions with anonymized, lower-resolution, or scrambled data could have many applications where the asymmetry between collaborators is in privacy, privilege, or responsibility.

4.5 CRITIQUING AND CRITICIZING WEBSTRATES

In this section, I evaluate how Webstrates and web technologies in general comply with transclusive software.

Webstrates deviates from the standard model of transclusive software in several respects, and necessitates many workarounds to provide transclusive software functionality. Some of the design issues that occur with Webstrates also illustrate limitations of the proposed model for transclusive software.

While `iframes` provide the functionality of mounting a whole document within a substrate, they are extremely simplistic. For transclusion to be useful in a wide array of circumstances, some specific behaviors need support.

For example, Webstrates has no support for transclusions that are cross-references within a single document. The de facto transclusion element cannot partially transclude a substrate. This behavior can be achieved by augmenting the `iframe` with a script that hides everything in the transcluded document but the needed subtree, but this requires users to write a custom script for each case. Local and partial transclusions are natural features of transclusive software because they further support the transparency and flexibility of transclusions.

Creating a non-live transclusion, a non-synchronized copy of a remote object, should ideally be an option. This type of transclusion can facilitate, e.g., concurrent experimentation on the same data by multiple users. Shadow DOM, a web technology currently in the experimental stage, can be applied to circumvent Webstrates automatic synchronization. Creating intentionally ephemeral objects also requires actively circumventing Webstrates functionality.

Room for these kinds of interactions inevitably becomes necessary in real systems, and it is necessary to consider, in Webstrates and in

Transclusive Software feature	Available in Webstrates?	Method/Workaround
Document transclusion	yes	`iframe` elements
Partial transclusion	no	Faked with styling
Local transclusion	no	
Ephemeral transclusion	yes	`iframes` inside shadow DOM trees
Representational independence	with scripting	Programmatic CSS rule injection
Transclusion transparency	with scripting	Recursive listener propagation
Malleable substrates	yes	Editing DOM with web inspector or scripting

Table 4: Webstrates' compliance with transclusive software features. There are three categories of compliance. I count a feature as available in Webstrates if it has first-class support through adapted web API (e.g. `iframes` implement transclusion at the granularity of whole documents). "with scripting" means that the feature can be realized with additional scripting on a webstrate. Features that are not available require modification of Webstrates or wholly new API to be realized.

transclusive software generally, how systems elegantly support users who need non-default behaviors.

Transclusions are not transparently part of the hosting document. Instruments which work on data local to the document are liable to break if transcluded data is substituted. This is due to a couple of factors. The first is that user events on transcluded data do not reach the host document, resulting in part of the substrate being essentially invisible to instruments. The second, more broad reason is that the event model of the browser does not inherently support decoupling interaction instruments from the objects they operate on. As I mentioned in the section on transclusive instruments, there are workarounds for this, but it is worth considering whether there exists an interaction architecture which provides better fundamental support for interaction instruments.

Another side effect of the non-transparency of Webstrates transclusions is that creating representationally independent transclusions is somewhat difficult. The CSS styles in a transcluding substrate do not automatically override those in the transcluded document. Applying local styling to a transcluded document is instead achieved by using a script to explicitly override the styling on the transcluded document at runtime.

It is worth reiterating that the whole mechanism of transclusions in Webstrates depends on not being subject to the same-origin policy implemented by web browsers. Non-localized transclusive software systems are outside the scope this thesis, but this limitation certainly poses an issue for them.

While it is a stated advantage of Webstrates that it introduces only a minimum of new API for the web, the obvious way to bolster its support for transclusive software is to provide some of these workarounds as standardized mechanisms available in all Webstrates. For example, a proper transclusion element which wraps an `iframe` with a script providing a better default behavior and interface. Creating a standard library for transclusive software in Webstrates is an interesting project to investigate the subject further, which would provide opportunities to experimentally develop the model.

Webstrates is a transparent addition to the existing web infrastructure. In theory, anything that is possible in regular web development is applicable in a Webstrates environment, but a many web frameworks only have limited compatibility with transclusive software, e.g., because they implement an MVC type architecture.

The web is a growing environment for user software, but modern web applications mostly adopt the assumptions of desktop application-style software. Though users can view and manipulate the source code of any web page they interact with, web apps can hardly be described as knowable. The source code is typically machine-generated and compressed, so that it is effectively unreadable. Newer Web apps also depend extensively on server-side computation. No major architectural frameworks for the web apply the DOM as a simple, uniform, and flexible medium in the same vein as Webstrates.

The difficulty of achieving novel interactions in Webstrates in a standardized manner arguably supports the need for a model of transclusive software. While Webstrates enables novel transclusive interactions, creating useful systems is in practice rarely as simple as programming the user-facing components. Instead, many detours occur when this and that bit of the working material do not act as you expect.

5 A TRANSCLUSIVE SOFTWARE SYSTEM

In this chapter, I apply my model of transclusive software to describe a transclusive reformulation of InPlenary, a research prototype for co-located active learning. I analyze this sketch of a transclusive system to investigate whether and how properties of transclusive software affect the flexibility and power of the system with respect to my research themes, and what limitations it exposes in the model.

5.1 INPLENARY

InPlenary is a proof-of-concept system for co-located active learning activities in university lectures (Korsgaard, Klokmose, and Caspersen, 2016, in preperation). The authors experiment with designing a system that leverages the technologies that are readily available (but not always conducive towards learning) in the situation of university lectures: slideware, lecturing halls with projectors, and student devices including laptops, tablets, and mobile devices.

The system is used used in classrooms or lecture halls during lectures. The lecturer's slides act as a shared resource accessible by everyone in the room through a local network. The lecturer controls the slides from a web client that provides an interface not unlike common presentation software. Students are served a client which lets them follow along with the slides and take notes on their personal device. The computer controlling the room projector also accesses the slides on the network, so that they automatically update when changed by the lecturer or students. The lecturer can augment an ordinary slide deck with learning activities like polls or quizzes, which students participate in through their clients.

InPlenary is built on four design principles formulated with an outset in the design space of related technologies for learning: *physical space as a first class digital entity*, *integration with existing practices*, *focus on learning activities*, and *focus on commonness*.

InPlenary is tied to location. The system differentiates between slideshows and lectures, the latter being a situated instance of the former, which collects interactions and data created during a particular lecture. Lectures exist with the existing spatial, temporal, and social delimiters of university lectures, i.e., an InPlenary lecture can only be started in a lecturing hall, and later access to the digital lecture depends on having been present during the physical one.

InPlenary integrates with existing slideware by allowing lecturers to create slides by their preferred method, and then uploading it to

the system via web client, where it can be augmented with activities. Since it is implemented for web browsers, it runs on all the relevant devices with no need for a custom environment or software to support it. This also means that InPlenary readily coexists with whatever software students and lecturers otherwise apply in or around lectures.

In the web-based slide editor, lecturers can add activity slides, which facilitate different active learning activities during the lecture. In the proof-of-concept implementation, there are five available activity slides:

- Polls,
- Clicker questions (Quizzes),
- Discussions, which gather student input,
- Rating widgets, which augment existing slides and allow students to rate their understanding of their content, and
- Reflection slides, which present a question/prompt for reflection, and then allow students to navigate a subset of the slides which are the topic of reflection.

InPlenary lectures behave as a shared resource during lectures. Students connect to a Wireless Access Point to access an audience client, where the slides and a simple note-taking area are available. There are additional views for the lecturer and projector screen. All three views are of a shared lecture, and are updated as the slides change.

In their study, the authors find that there is potential for supporting co-located and cooperative active learning using personal devices and systems that support distributed presentations and learning activities.

5.2 INPLENARY AS TRANSCLUSIVE SOFTWARE

InPlenary is designed for users in a particular situation delimited in time and space, whose interactions are mediated through shared data that they access with different tools fitted to their nonequal roles in the system. InPlenary fits with the model of transclusive software because of its scale, multiplicity of users and devices, and collaboration around a common digital artifact.

Korsgaard, Klokmose, and Caspersen's stated design goals also overlap and interact with my research themes: Reconfigurable software should readily integrate with existing and adjacent practices, lectures are a type of asymmetrical collaboration, and a system designed to adapt personal devices for a co-located activity is a novel configuration of software ecology.

In this section, I describe InPlenary in terms of transclusive software. I do this top-down, by moving from the activities and goals of

InPlenary users, to their working material and tools, to the substrates that layer shared data and instruments to service users.

I focus on a subset of InPlenary functionality, where I can apply the design language developed in the Chapter 4 at a level of complexity that lets me look into its expressiveness, flexibility, and limitations.

ROLES InPlenary sessions are collaborations between one lecturer and many students[1]. The lecturer creates the slideshow before the lecture, and controls slides and administrates learning activities during. Students view the slides, perform learning activities (which give them a limited ability to manipulate the slides), and access the slides after the lecture.

SHARED DATA All interactions in InPlenary center around the shared slides. A distinction is made between the slideshow as composed by lecturer, which can be instantiated in multiple distinct lectures, and the product of the lecture, which is a slideshow that is accessible by a specific subset of students who participated in the lecture, and embeds their Discussions, Poll and Clicker answers, and Reflections. One way to implement this as a transclusive design is to consider the lecture as an overlay on the slideshow, like the document annotation overlays demonstrated in (Bouvin and Klokmose, 2016), which transcludes the slideshow and collects data that is separate from it but rendered on top of it. Students are expected to directly manipulate activity slides, however, so that is not the correct design abstraction. Instead, a copy or prototype instantiation mechanism is needed, which creates a new substrate that is a deep copy of the previously created slideshow substrate. Access to the lecture afterwards can be mediated by simple access control. The users who are allowed to read the lecture substrate are the ones logged on to the local network when that particular substrate is created. Instantiated lectures should maintain a connection to their prototype slideshows, e.g., a node storing the prototype substrate's address, for the benefit of lecturers who may want to access all lectures produced from a particular slideshow.

INTERACTIONS The interaction instruments in the transclusive InPlenary should model all the domain-specific user interactions needed, while decomposing them into individual instruments to a degree that supports reuse and flexibility.

Activity slides provide for an interesting application of the instrument pattern described in Section 4.4.3. The function of the pattern is to package behavior and data to create distributable and shareable instruments, with which can essentially extend their software. Activity

1 All the examples in (Korsgaard, Klokmose, and Caspersen, 2016) assume one lecturer, though this is not necessarily a hard limitation. The number of students is presumed to be between a classroom (10-25) and a lecture hall (100-250).

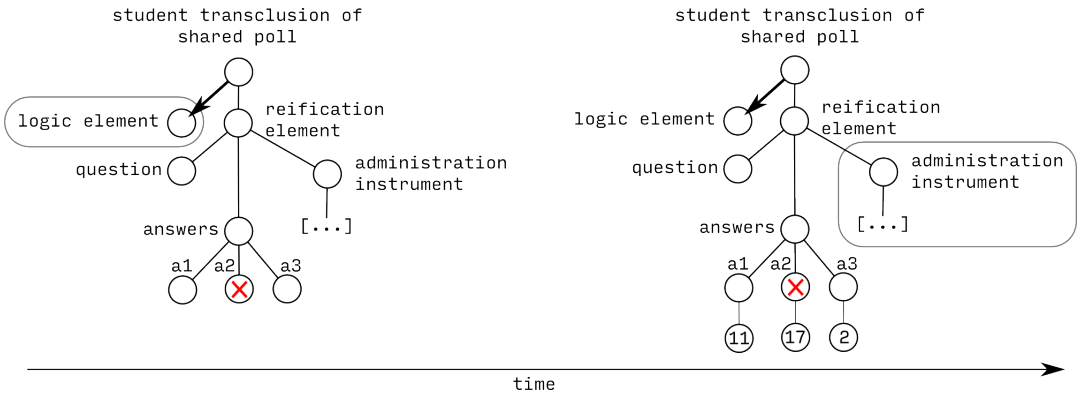

Figure 14: The structure of the poll activity object over time. A student interacts with a poll activity slide through a transclusion of the shared slides. The poll's primary behavior is to change the representation of the poll answers to according to which answer the student has selected (the red ×).
When enough students have answered the poll, the lecturer uses an instrument embedded in the activity slide to collect the answers. This administrator instrument queries all student clients for the identifier of the answer which is represented as selected, and convert this data to an answer distribution (e.g., by a map-reduce operation). Finally, the collective answers are reified on the activity slide by attaching the number of respondents to each answer.

slides similarly package behavior and data, but are applied to embed collaborative interactions within an ordinary document.

As shared objects, the activity slides present some issues for obtaining the behavior of InPlenary in use. For example, the poll activity must allow each student to personally select an answer to a question, and later collect all those answers to show them collectively. The problem is to allow some interactions to only have client-local consequences, and some to be propagated to the shared object.

According to the model of transclusive software substrates, the *structure* of substrates is shared, while their *representation* can be individual to views. We can take advantage of this by designing the poll object to model individual answers representationally within each particular student client, and then, on request by the lecturer, reifying all answers in the shared object. A structure for this design of the poll activity is illustrated in Figure 14.

As in Section 4.3.1, the lecturer uses representational changes that travel to all clients to change the slide. This representational change can be freely overridden by students in their local view.

Similarly, the Reflection activity slide can provide students with an instrument that locally changes their representation of the shared slides, which is deactivated when the lecturer closes the activity.

The matter of localizing changes to the slideshow made by students could also conceptually be the responsibility of a constraint layer, one which could be applied to all student transclusions to provide a safe-

guard against certain modifications to the slideshow, e.g., transcluding a text editing instrument and changing the slides maliciously.

The Clicker question activity slide presents another issue for transclusively shared data. Consider that the activity should let students pick an answer, collect all answers to show them to the lecturer, and reveal the answer distribution and correct answer to all students. The naive shared data approach to achieving this behavior would be for the activity slide to embed all its data within its structure, including the correct answer.

This does not harmonize with the malleability and representational independence of substrates. If a question embeds the correct answer, it is a trivial matter for a student to inspect the data, and represent it in a way that reveals the correct answer.

Instead, it is necessary to provide some notion of fine-grained access control on particular subsets of data which can be modified at runtime. This mechanism should support students being able to modify shared data, (provide answers to the Clicker question activity slide), while only being able to read a subset of it (the question, available answers, and their own choice), and for that restriction to be lifted when a privileged user decides to do so.

The Clicker question then becomes a piece of shared data with fine-grained access control that changes to a transparently shared piece of data when the lecturer decides to reveal the answer and statistics. The poll, discussion, and rating slides do not present added complexity for activity slides, and can be modeled similarly to the Clicker question activity.

The final interesting aspect of transclusive interaction I will cover here is the lecturer view on learning activities. The life cycle of an activity slide is 1) The lecturer picks an activity and places it in the slideshow, 2) The lecturer configures the activity, filling in its content and contextual data like which answer is correct, 3) During the lecture, students manipulate data or perform some interaction as made available by the activity slide, and 4) the lecturer takes some administrative action to end the activity and possibly incorporate activity data into the lecture.

When the lecturer adds and configures activity slides, they are operating on them, whereas students performing the activities are mediated by them. Lecturers can create, configure, and administrate activity slides as objects. In the parlance of transclusive software, this means that they have access to a set of instruments that operate on instruments.

An instrument which creates activity slides is very simple to design within the model of transclusive software and associated design principles. Since activity slides are broadly instruments, which are reified as substrates through their reification element, meaning that they are shareable, distributable, and malleable objects. A *spawner*

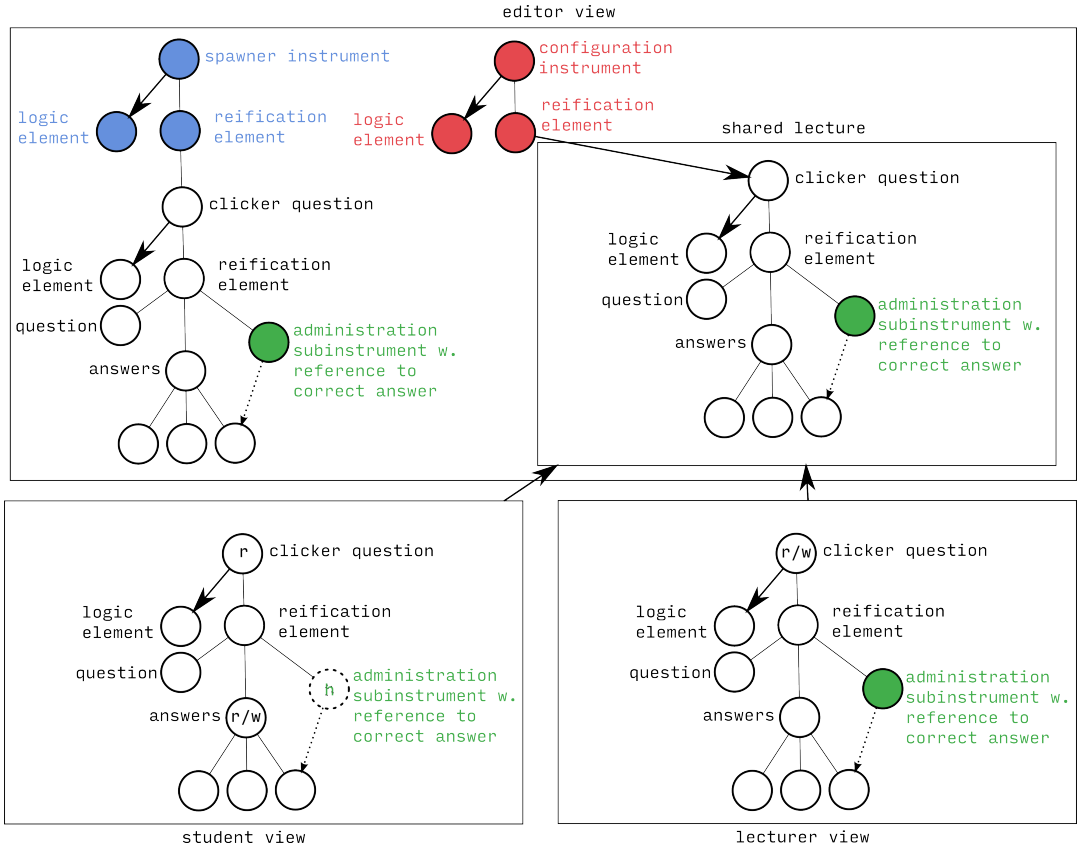

Figure 15: A snapshot of the instruments used to completely model the life cycle of the Clicker question activity slide in the transclusive InPlenary system, and how they appear in different client substrates. The instruments are color-coded to distinguish their roles. "r" and "w" indicate read and write access to the subtree of the marked node. "h" indicates that a subtree is hidden, i.e., not readable or writable. Full arrows are transclusions.

instrument, which contains some data that it produces copies of on demand, can be "fed" a prototypical activity slide to create an activity slide spawner.

An instrument to configure activity slides can be designed like the example of a code editor substrate, described in Section 4.4.3. The configuration instrument simply transcludes the particular slide it is applied to, and exposes its structure to be modified.

While spawner and configuration instruments can be made to be decoupled from the particular object they are operating on, the act of administrating a activity slide is likely dependent on its particular function. If we reapply the notion of fine-grained access control, we can consider simply embedding a sub-instrument in each activity slide which is only accessible by the lecturer. This instrument can model the lecturer's interaction with the activity slide, as well as containing any information that should be hidden from students.

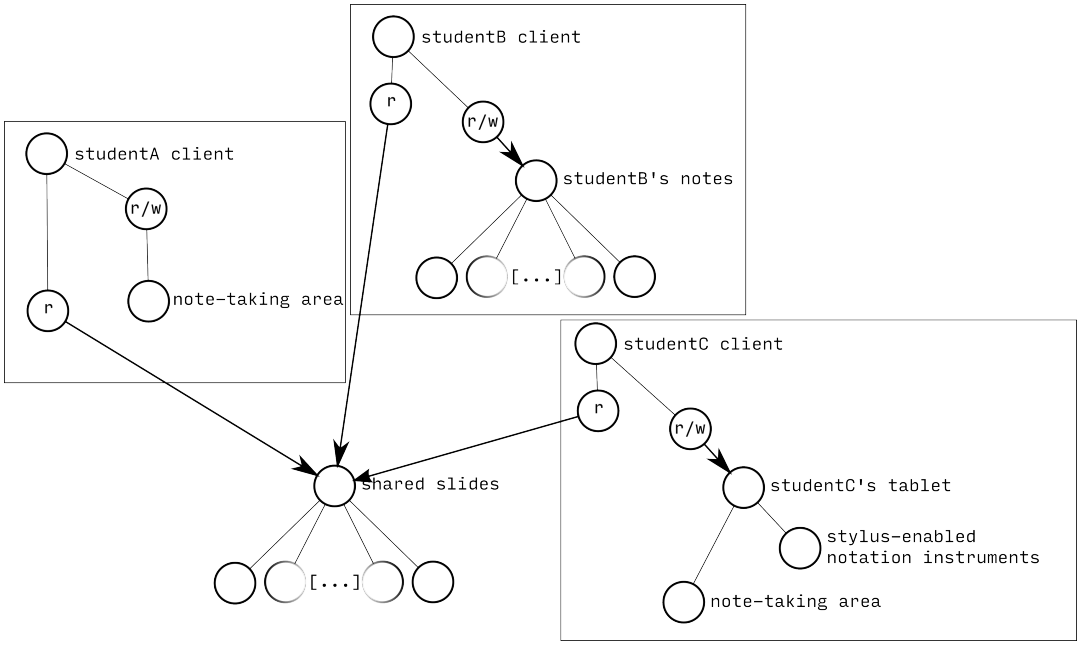

Figure 16: Three student clients with different note-taking configurations. Each student has read-only access to the shared slides, and read-/write access to a subtree of their client which is by default used for a simple note-taking area. StudentA takes notes locally, on her InPlenary client. StudentB transcludes a larger body of private notes. StudentC transcludes an environment for taking notes with a stylus on a tablet.

CLIENTS As in Section 4.3.1, data and instrument substrates are layered together in clients which collect the subset of system features a single role in the system needs. In InPlenary, there is one client for each distinct role: the lecturer's, the student's and the projector's. The lecturer may usefully have access to a complex editing client to create the slideshow, and a simple controller client to run the lecture. Every client transcludes the slideshow, while the editor transcludes the meta-instruments, the controller client transcludes the slide control instrument, the student client embeds a note-taking field, etc. The client substrate running on the projector simply shows the current slide, and has the same access rights as students, i.e., activity slides always show the student-facing information. Each client has access rights corresponding to their role in the system.

The system expects one lecturer and one projector client, meaning that they can be placed in pre-assigned substrates, i.e., the lecturer has a personal substrate and the projector substrate is tied to an address on the local network. The method of distributing clients to students needs to be slightly more elastic. A simple proposal is that students accessing InPlenary at a lecture's local network can either request a fresh client, a copy of the default student client structure into a new substrate assigned to the student, or call up their personal, customized client, and then transclude an address on the local network referring to the current lecture.

A snapshot of the structure of the transclusive InPlenary system is illustrated in Figure 15.

5.3 ANALYSIS

5.3.1 *What does InPlenary gain from transclusive software?*

Korsgaard, Klokmose, and Caspersen build their system on the design principles *physical space as a first class digital entity*, *integration with existing practices*, *focus on learning activities*, and *focus on commonness*. Transclusive software especially supports the latter three, and extends their power and expressiveness as well.

INTEGRATION WITH EXISTING PRACTICES InPlenary integrates well with existing student and lecturer practices by being highly portable, and not being a monolithic system. The system is designed to support many different practices, computer mediated or not, going on around its own functionality.

Transclusive software extends the potential for reconfigurability in the system by building the clients out of malleable, shareable substrates. In the original system, students have access to a simple note-taking field in their client. In a tranclusive implementation, students have the ability to transclude their own notation data and tools,

which may support idiosyncratic needs like connecting the lecture notes with a larger body of notes that a student maintains, or letting a student take notes with a different interaction method by integrating with personal devices. This example of transclusively integrating different note-taking methods with the student client is illustrated in Figure 16. This sort of additive modification of the system is an ideal interaction for transclusive systems. Of course, there is also room for reapplying the slides directly for note-taking by copying them into a student's note environment. The design principle of leaving room in the system for co-occurring practices fits with the research theme of reconfigurable software.

FOCUS ON LEARNING ACTIVITIES The original InPlenary system can be extended with more activity slides, since they are really self-contained applications. The transclusive InPlenary is also readily extensible in this regard. Since the lecturer meta-instruments can be shared, and the existing activity slide substrates are malleable, the system presents a low barrier of entry for lecturers and students to become designers of activity slides.

One higher level learning activity often applied in university classrooms is student-run presentations, where groups of students demonstrate their learning by disseminating a subject to their peers. Active learning methods are very useful for student-run presentations, because they can support participation by the students who are not currently presenting, and encourage the presenting students to think carefully about their teaching tactics.

While nothing prevents lecturers and students from exchanging roles in InPlenary as described in (Korsgaard, Klokmose, and Caspersen, 2016), a transclusive InPlenary supports piecemeal extension of student capabilities in addition to simply giving them access to the complete lecturer client substrate. For example, a simple creative exercise would be to have students create small presentations within the slideshow which already have a designated number of slides. An implementation of this exercise could be created by combining the Reflection activity with the slide editing instruments available in the slide editor client. This would allow students to modify a subset of the lecture slides as a learning activity.

As with the shared citation tool example in (Klokmose, Eagan, et al., 2015), shareable instruments allow users to extend and redevelop their own practice through collaboration. The T-shaped web practitioners at Creuna are a case where there is a legitimate business potential for software supporting overlapping and changing practices (Tchernavskij, 2015).

FOCUS ON COMMONNESS Shared digital artifacts are at the heart of transclusive software. The transclusive InPlenary should naturally

handle co-occurring collaborations with common artifacts during lectures, as well as the reapplication of the lecture objects before and after.

Shared-by-default substrates make room for new kinds of shared resources in the lecture situation. For example, it is simple to extend the examples in Figure 16 to one in which multiple students share notes. New kinds of learning activities could also be developed to take advantage of shared material, e.g., collaborative analysis exercise where students collect and sort observations together on a shared virtual whiteboard.

As InPlenary is now, the common resource is limited by the system. The lecture is only in common as long as it is viewed and manipulated through InPlenary. In a transclusive environment, shared objects are an infrastructural provision. In a transclusive software environment, InPlenary could be one of a number of collaborative systems applied in a university course. Lecturers could collaborate when producing the original slides, and students could apply the lectures for collaborative exam preparation.

5.3.2 *What does transclusive software learn from InPlenary?*

Formulating InPlenary as transclusive software effectively problematizes some aspects of the model as presented in Chapter 4.

InPlenary's shared learning activities present cases where the default sharing behavior of information substrates is overly naive, and the model must make room for case-by-case decisions about what data is shared when. Deep copies of shared objects and user-local and secret state on shared objects, which limit the travel of information through transclusions, all need to be accounted for in the of ecology of substrates and transclusion.

Giving users and designers of transclusive software awareness and control of sharing behavior is a significant challenge for a useful model. I sketch fine-grained access control on substrates, which introduces read/write privileges and hidden data at the level of subtrees and nodes, as a proposed mechanism for delimiting shared data and interactions.

As shareable dynamic media, substrates and transclusions are meant to be understandable and operable by users. It is implicit to the property of malleability that users can understand and work with the underlying structure of their software. Hence, transclusive software systems should be organizable and readable by their users. This is a challenge for the ecology of transclusive software.

InPlenary takes advantage of local networks to create a well-defined shared use context. Students always access the lecture by the same URL, transforming the web addressing scheme, which conceptually refers to servers and the files they hold, to a contextual reference.

In my sketch of the transclusive system, I assume that it is possible for instruments and users to query the system to answer questions like "who is participating in this transclusive system?" In the standard model, I state that information substrates are not located, like files on a computer, but are in common. In Webstrates, this means that they reside without organization on a server. In practice, users must exchange and keep track of the names of substrates that they are interested in. This was an issue in the early World Wide Web: discovery of new sites was dependent on passing along direct knowledge of each particular website, which has most obviously been resolved by search engines. Complex transclusive systems , e.g., where the amount of data is large, or many users are interacting, can pose similar problems for discovery and navigation.

The representation and navigation of information is of primary concern to hypertext researchers. (Bernstein, 2003; Di Iorio and Lumley, 2009; L.-C. Lee, Lutteroth, and Weber, 2010; Nelson, 1995) all consider hypertextual transclusion as a mechanism of organization, which supports the readability of bodies of information and software.

Di Iorio and Lumley implement transclusions in a way that allows them to modally be transparent or explicit, arguing that being able to switch between these configurations is optimal for users. L.-C. Lee, Lutteroth, and Weber implement *scope visualization* in their customizable GUI system. They make explicit to users by automatic annotation how their GUI is constituted from transcluded configurations and templates.

Software transclusions interlace the structure of layered information and of user interaction. An equivalent to scope visualization in a transclusive system context could be passively tracked user presence and access control visualization.

In this thesis, I am interested in collaborations between relatively small groups of people, but even small groups must make effort to organize their working material. Tools like repositories of substrates, user presence, and contextual queries are needed to support discovery and navigation with awareness of collaborators, history, user roles, etc., in transclusive systems.

(C. P. Lee, 2007; Star, 1989; Star and Griesemer, 1989) investigate how shared objects mediate collaborative work between different groups of practice, where co-awareness is low, and it is not practically feasible or useful for any one user to know the whole system of collaboration.

Star and Griesemer show how shared objects are tools for standardization, which support a robust network of practices where collaboration can go on in spite of ambiguity and contradictions between knowledge, models, and goals. C. P. Lee shows that boundary objects even apply to push, redefine, and negotiate inter-practice boundaries. The lesson to grasp from boundary objects may be that transclusive software systems can be designed to effectively mediate the complex-

ity of a distributed, malleable, multi-user system, rather than attempting to explicitly inform users of the structure and relations of the software network at any one point in time.

Fine-grained access control and constraint layers may support the design of transclusive systems which limit the modifications users can make to shared material, but allow them to freely modify private substrates transcluding it. This is one approach to the post-WIMP property of resilience (Beaudouin-Lafon, 2004).

What kind of guarantees should transclusive systems give users with regards to security, reliability, and privacy? Formal security models are concerned with properties like information and system integrity, availability, and confidentiality. A reliable software system is concerned with performing a service consistently under well-defined circumstances. InPlenary avoids a strong formal security model or reliability specification in favor of designing with the social contract of lectures in mind. Theoretically, malevolent students can derail lectures by flooding activity slides with trash. There is, in some respects, a dichotomy and inherent tension between reconfigurability/-malleability and resilient collaborative systems.

Korsgaard, Klokmose, and Caspersen find that this is a non-issue in a co-located practice where students are explicitly identified. The authors did see playful interactions where students tested the constraints of activity slides, but also found that student's actively used the down-voting mechanism to keep the shared data relevant and productive to the lecture.

The software constructs and design principles that I take up for transclusive software also take the position that users can and should be the final arbiters of system constraints. (Beaudouin-Lafon, 2004; Garcia et al., 2012; Klokmose and Beaudouin-Lafon, 2009) all emphasize systems where user-facing flexibility is a virtue over reliability and security. They develop models for software practices where constraints on users can be modified and circumvented at will. This informal idea of security can be described as "toilet door security": A public toilet door gives some level of privacy and even protection, but only to the extent that it can be unlocked, crawled over, or broken if necessary, say, if someone faints in a toilet stall.

The empirical subjects that exemplify issues under my research themes illustrate a spectrum of security and reliability needs.

In (Klokmose and Zander, 2010), laboratory workers concurrently expect their notebooks to be standardizing objects which track a complete and unambiguous record of their work, and to be unconstrained with regards to the structure and type of data it accepts and the activities it is applied in. The laboratory notebooks are an example of shared artifacts which include both formal security guarantees (there is only one, common notebook, no data is ever deleted or modified after it has been entered) and extremely flexible circumstances of us-

age (books accepts arbitrary data and usefully mediates experimental procedures, writing, shared equipment usage).

In (Tchernavskij, 2015), the web agency workers from the case study rely on the ability to collaborate orthogonally to their formal hierarchy and distribution of work when these cause breakdowns in the product design process. A recurring issue for the design process is that shared digital design artifacts can only be transferred between the workers' idiosyncratic work environments as immutable data. In the future workshop, the web agency workers envisioned transclusive systems where the underlying ecology supported design artifacts that remained malleable throughout the design process. These rethought artifacts acknowledge and take advantage of collaborations, by supporting concurrent manipulation and keeping the resulting design knowledge in common between the distributed team members.

These two studies document collaborative practices that widely differ with respect to type of organization and work, breakdowns, tensions, and goals. Nonetheless, their authors find that both practices have needs that can be met by systems applying the principles and constructs of instrumental interaction.

With the design challenges for flexibility and commonness documented in (Klokmose and Zander, 2010) and (Tchernavskij, 2015) in mind, we might attempt to design transclusive software systems for the laboratory and for Creuna. The security and reliability models of these hypothetical systems would likely be markedly different. Some transclusive software for flexibly collaborative systems may apply security models similar to InPlenary's, that establish best practices which can be subverted, and supply tools for users to enforce or relax constraints, and others may need to delimit malleability, sharing, and reintrepretation by formal security and reliability guarantees.

5.4 SUMMARY

Does transclusive software allow InPlenary to maintain the same functionality while being more flexible and powerful? Yes and no. It emphasizes InPlenary's design principles, but presents some problems for transclusive software.

This example of a transclusive system design illustrates how the model can be applied to my three research themes.

Malleable client substrates that can be reconfigured by transclusion are a reconfigurable software component which bolster the authors' design principle of integration with existing practices. The system makes room for students to compose their personal digital environments while supporting shared interaction with the lecture. The system extends lectures from objects that are in common within InPlenary, to being reapplicable in adjacent transclusive systems.

The situation of digitally mediated lectures is an asymmetrical collaboration. The transclusive system models the teacher/student asymmetry in how the clients apply the shared lecture object differently, in the activity slides providing student and lecturer interfaces, and in the application of access control, localized changes, and constraint layers to limit students' manipulation of the slides.

The application of transclusive software constructs in formulating the system validates the ecology of transclusive software by showing how they are composed to build functioning systems.

In replicating activity slides as transclusive instruments, I described two useful meta-instruments applying reification and malleability, spawners and configuration instruments. I also described how asymmetrical interaction between lecturers and students in learning activities presents a pragmatic issue for the semantics of shared objects in transclusive software.

This design exercise supports the claim that the model of transclusive software can be applied to generate software which is novel with respect to my research themes.

6 DISCUSSION

> In theory, theory and practice are the same. In practice, they are not.
> —Anonymous

6.1 METHODOLOGICAL LIMITATIONS

The motivation for this thesis was to investigate the unexamined phenomenon software transclusion, and to determine whether it is novel and useful in the design of user software. I decided that an experimental/empirical study would either be outside the practical scale of this project, or too small to make claims that related to the complexity of situated human activity. Instead, I have attempted to assume novelty and usefulness, and examine theoretically how systems extensively applying transclusion could work. My model of transclusive software is not constructed by inductive reasoning from many observations, but an extrapolation from principles and related work.

This limits the type of findings I can produce. I use my model to describe related software constructs, critically evaluate Webstrates as reference implementation, generate a sketch of a transclusive system and examples of new constructs. Finally, I reflect on the model's limitations in comparison to a real system that takes up challenges within my themes.

If there is to be a next step for software transclusion, I believe it should be an interrogation of it in use. One interesting direction to take is an inquiry into transclusive software as a material basis for common artifacts.

6.2 COMMON ARTIFACTS AND TRANSCLUSIVE SOFTWARE

In (Tchernavskij, 2015), in my reflection on the prototype development study, I proposed that a model or design language of software oriented towards collaborative work should be informed by CSCW models of collaborative artifact use. I called this idea "Boundary Object Oriented Programming". Is transclusive software a design language for common artifacts?

(Star, 1989) challenges the idea of intelligent systems pursued in the AI research community of that time. She argues that the intelligent systems cannot be reduced to machines programmed with a sufficiently precise universal model of the world and a large enough memory. Star instead calls for a social understanding of intelligence,

measured by a system's ability to adapt to the complexity, changeability, and ambiguity in communities of practice:

> So the Durkheim test would be a real time design, acceptance, use and modification of a system by a community. Its intelligence would be a direct measure of usefulness applied to the work of the community: its ability to change and adapt, and to encompass multiple points of view while increasing communication across viewpoints or parts of an organization. Such a test also changes the position of metaphors with respect to design and use considerations. In an open, evolving system, the boundaries between design and use, between technology and user, between laboratory and workplace, necessarily blur.
>
> *(Star, 1989)*

Star's criticism is equally valid in the context of user software. Software system designers generally embed the implicit or explicit assumptions that one size fits all, that high-quality systems are those that reliably support well-defined behavior, provide definite solutions to definite problems, and implement sufficiently advanced closed models of their domain.

In my analysis of how transclusive software might support InPlenary's design principles for active learning in lectures, I argued that the transclusive InPlenary creates an environment which can be reconfigured to integrate with students' and lecturers' adjacent activities.

During, before, and after lectures, lecture slides could be objects or mediators in activities like

- idiosyncratic note-taking practices (Figure 16)
- group learning activities between a subset of students at the lecture
- homework
- exam preparation

in which they could be copied, modified, annotated, etc.

C. P. Lee's boundary negotiating artifacts are used to negotiate and push boundaries (Section 2.1.3). Compilation artifacts bring communities of practice into temporary alignment. Structuring artifacts establish ordering principles. Borrowed artifacts are taken from one community of practice and used in an unanticipated manner by members of another community of practice.

Klokmose and Zander work with syntonic seeds, which are inscription artifacts that represent and sublate contradictions, and simultaneously and sequentially act as mediators and objects in many activities (Section 2.1.4).

Transclusive software can implement constructs like

- shared heterogeneous data which can be concurrently operated on by multiple users with personal tools.
- governors and constraint layers, which model common rules, and can be attached and circumvented as users see fit.
- instruments which can be freely appropriated.

I argue that transclusive software presents an ecology of software constructs which facilitate negotiating and pushing boundaries of practice and reuse in different modalities.

Across theories, common artifacts become complex mediators because they serve situated needs and can be generated, discarded, and modified by the communities of practice they occur in.

(Paper) substrates have already been studied as a design material for artistic practice (Garcia et al., 2012). (Bouvin and Klokmose, 2016) argue that Webstrates' transformation of the DOM into a malleable medium turns "the distinction between development and use or browsing and authorship [into] a phenomenon of use." I have presented transclusive software systems as networks of substrates which can support continuous change with regards to data, functionality, users, and devices.

The hypothesis that transclusive software can be a medium for common artifacts invites a participatory design study of a collaborative practice where experimental software can be applied.

7 CONCLUSION

I define software transclusion as the same content in more than one place as a shared virtual object, representing data or interaction logic. The phenomenon emerges in Webstrates by application of hypertextual transclusion to interactive software built in a rich hypertext medium.

I investigate from a theoretical perspective what its potentials, tradeoffs, and consequences are as an architectural composition mechanism and a user interaction.

This leads to me proposing a model for transclusive software, which brings together mechanism and medium, and defines a normative view of software taking up my research themes. Transclusive software systems are networks of information substrates that interconnect through transclusion, and are continuously restructured by users to modify functionality, data, and constraints, and to join and leave collaborations.

I apply this model to describe and incorporate concepts from related work, including interaction instruments, governors, reification, and substrate-editing-substrates. I find that transclusion has productive synergy with interaction instruments and reification in particular. I also develop new transclusive software constructs, including constraint layers for data and interaction, and spawner instruments.

I critically evaluate Webstrates and the World Wide Web as platforms for transclusive software. Webstrates lacks support for but can be extended to provide representational independence and transparency of transclusions, and has no support for partial and intra-document transclusions. I recommend the development of a transclusive-software-oriented library of custom web API that provides more fully featured transclusions than the standard `iframes` and provides a higher level of abstraction for design and use of web substrate based software.

WWW is concurrently the best infrastructure available for transclusive software, and a severely limiting one. On the positive side, there aren't really other monolithic distributed networks of cross-platform virtual computers running on an extreme variety of devices providing a healthy and extremely expandable API for interactive software. On the negative, the web as it is is not designed for shareable dynamic media and transclusive interactions. The state of the art of web application frameworks as a rule implements MVC-style architectures which rule out most integration with Webstrates.

I finally apply the model in a reformulation of InPlenary as a transclusive software system. I analyze the resulting design sketch to

examine the design consequences of a transclusive InPlenary, and to reflect on the limitations of and challenges for the model confrontation with a real system produces.

I argue that three of InPlenary's four stated design principles are significantly extended by transclusive software: *integration with existing practice*, *focus on learning activities* and *focus on commonness*. In the design sketch, I exemplify how transclusive constructs support reconfigurability and asymmetrical collaboration in the examples of integration with personal note-taking practices and shared activity slides, respectively.

InPlenary problematizes some naive aspects of my model. I reflect on the challenges of making the complex composition of systems readable, querying and navigation, giving users control and awareness of the limits of sharing and malleability, and the types of security and reliability models that are appropriate and feasible in transclusive software.

I propose fine-grained access rights, supporting read/write privileges and hidden subtrees, as a mechanism for achieving some more secure interactions compared to the default of complete lack of any security guarantees. This proposal is an ad-hoc consideration for the design sketch, though, and may be both complex and counter-intuitive as a general tool.

I argue that transclusive software systems may support informal security models that are in line with socially constituted security and privacy mechanisms.

In my discussion, I argue that the model of transclusive software describes digital artifacts which are in some respects congruent with boundary negotiating artifacts and syntonic seeds, and that a there is potential for a participatory design study which investigates transclusive software systems as common artifacts.

BIBLIOGRAPHY

[1] William Atkinson and Dan Winkler. *Demonstrating Hypercard*. Computer Chronicles (Interview). Archive.org. 1987. URL: https://archive.org/details/CC501_hypercard (visited on 02/08/2016).

[2] Belinda Barnet. *Memory Machines: The Evolution of Hypertext*. Anthem Press, 2013.

[3] Michel Beaudouin-Lafon. „Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces." In: *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '00* (2000), pp. 446–453. DOI: 10.1145/332040.332473.

[4] Michel Beaudouin-Lafon. „Designing interaction, not interfaces." In: *Proceedings of the working conference on Advanced visual interfaces - AVI '04* (2004). DOI: 10.1145/989863.989865.

[5] Michel Beaudouin-Lafon and Wendy E. Mackay. „Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces." In: *Proceedings of the working conference on Advanced visual interfaces - AVI '00* (2000), pp. 102–109. DOI: 10.1145/345513.345267.

[6] Mark Bernstein. „Collage, composites, construction." In: *Proceedings of the fourteenth ACM conference on Hypertext and hypermedia*. ACM. 2003, pp. 122–123.

[7] Niels Olof Bouvin and Clemens N. Klokmose. *Classical Hypermedia Virtues on the Web with Webstrates*. Submitted to HYPERTEXT 2016. 2016.

[8] Vannevar Bush. „As we may think." In: *The atlantic monthly* 176.1 (1945), pp. 101–108.

[9] Henrik Bærbak Christensen. *Flexible, reliable software: Using patterns and agile development*. Boca Raton: Taylor, Francis(Chapman, and Hall/CRC), 2010. ISBN: 9781420093629.

[10] Angelo Di Iorio and John Lumley. „From XML inclusions to XML transclusions." In: *Proceedings of the 20th ACM conference on Hypertext and hypermedia*. ACM. 2009, pp. 147–156.

[11] Douglas Engelbart. „The augmented knowledge workshop." In: *A history of personal workstations*. ACM. 1988, pp. 185–248.

[12] Jérémie Garcia, Theophanis Tsandilas, Carlos Agon, and Wendy Mackay. „Interactive paper substrates to support musical creation." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2012, pp. 1825–1828.

[13] Tony Gjerlufsen, Clemens N. Klokmose, James Eagan, Clément Pillias, and Michel Beaudouin-Lafon. „Shared Substance: Developing Flexible Multi-Surface Applications." In: *Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11* (2011), pp. 3383–3392. DOI: 10.1145/1978942.1979446.

[14] Frank Halasz et al. „Reflections on Notecards: Seven issues for the next generation of hypermedia systems." In: *Communications of the ACM* 31.7 (1988), pp. 836–852.

[15] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. „Edit lenses." In: *ACM SIGPLAN Notices*. Vol. 47. 1. ACM. 2012, pp. 495–508.

[16] Alan C. Kay. „The early history of Smalltalk." In: *The second ACM SIGPLAN conference on History of programming languages - HOPL-II* (1993). DOI: 10.1145/154766.155364.

[17] Alan C. Kay and Adina Goldberg. *Personal dynamic media*. IEEE. 1977. URL: http://ieeexplore.ieee.org/xpls/abs%5C_all.jsp?arnumber=1646405%5C&tag=1.

[18] Alan C. Kay and Bonnie MacBird. *Alan Kay's tribute to Ted Nelson at 'Intertwingled' fest*. 2014. URL: https://www.youtube.com/watch?v=AnrlSqtpOkw (visited on 12/25/2015).

[19] Clemens N. Klokmose and Michel Beaudouin-Lafon. „VIGO: Instrumental Interaction in Multi-Surface Environments." In: *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09* (2009). DOI: 10.1145/1518701.1518833.

[20] Clemens N. Klokmose, James R. Eagan, Siemen Baader, Wendy E. Mackay, and Michel Beaudouin-Lafon. „Webstrates: Shareable Dynamic Media." In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15* (2015), pp. 280–290. DOI: 10.1145/2807442.2807446.

[21] Clemens N. Klokmose and Pär-Ola Zander. „Rethinking Laboratory Notebooks." In: *Proceedings of COOP 2010* (2010), pp. 119–139. DOI: 10.1007/978-1-84996-211-7_8.

[22] Henrik Korsgaard, Clemens N. Klokmose, and Michael Caspersen. „InPlenary: Designing Systems for Co-located Active Learning in University Lectures." In preparation. 2016.

[23] Harald Krottmaier and Denis Helic. „Issues of transclusions." In: *Proceedings of E-Learn (E-Learn 2002)*. 2002, pp. 1730–1733.

[24] Charlotte P. Lee. „Boundary negotiating artifacts: Unbinding the routine of boundary objects and embracing chaos in collaborative work." In: *Computer Supported Cooperative Work (CSCW)* 16.3 (2007), pp. 307–339. ISSN: 1573-7551. DOI: 10.1007/s10606-007-9044-5.

[25] Lung-Chen Lee, Christof Lutteroth, and Gerald Weber. „Improving end-user GUI customization with transclusion.“ In: *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science-Volume 102*. Australian Computer Society, Inc. 2010, pp. 163–172.

[26] Wendy E. Mackay. *Interactive Paper: A Whirlwind Tour of Tangible Computing from the Digital Desk to Music Composition*. Keynote Lecture. ACM, 2015.

[27] Hermann Maurer and Josef Kolbitsch. „Transclusions in an html-based environment.“ In: *CIT. Journal of computing and information technology* 14.2 (2006), pp. 161–173.

[28] Norman K. Meyrowitz. „The Missing Link: Why We're All Doing Hypertext Wrong.“ In: *The society of text: hypertext, hypermedia, and the social construction of information*. Cambridge, MA: MIT Press, 1989, pp. 107–114.

[29] Theodor H. Nelson. „Complex information processing: a file structure for the complex, the changing and the indeterminate.“ In: *Proceedings of the 1965 20th national conference*. ACM. 1965, pp. 84–100.

[30] Theodor H. Nelson. *Computer Lib/Dream Machines*. 1st ed. Theodor H. Nelson, 1974.

[31] Theodor H. Nelson. *Literary machines: the report on, and of, Project Xanadu concerning word processing, electronic publishing, hypertext, thinkertoys, tomorrow's intellectual revolution, and certain other topics including knowledge, education and freedom*. 87.1. Swarthmore, PA: Ted Nelson, 1987. ISBN: 9780893470524.

[32] Theodor H. Nelson. „The heart of connection: hypermedia unified by transclusion.“ In: *Communications of the ACM* 38.8 (1995), pp. 31–34.

[33] Theodor H. Nelson. *Transclusion: Fixing electronic literature*. 2012. URL: https://www.youtube.com/watch?v=ohiKTVVtDJA (visited on 01/03/2016).

[35] Theodor H. Nelson and Moe Juste. *Xanadu transclusion Web-based demo*. 2014. URL: http://xanadu.com/xuHome--D41 (visited on 02/02/2016).

[36] Pål Sørgaard. „A cooperative work perspective on use and development of computer artifacts.“ In: *DAIMI Report Series* 16.234 (1987). URL: http://ojs.statsbiblioteket.dk/index.php/daimipb/article/view/7590/6435.

[37] Pål Sørgaard. „Object oriented programming and Computerised shared material.“ In: *ECOOP '88 European Conference on Object-Oriented Programming* (1988), pp. 319–334. ISSN: 0302-9743. DOI: 10.1007/3-540-45910-3_19.

[38] Susan L. Star. „The structure of ill-structured problems: boundary objects and heterogeneous distributed problem solving.“ In: *Distributed Artificial Intelligence* 5 (1989).

[39] Susan L. Star. „This is not a boundary object: Reflections on the origin of a concept.“ In: *Science, Technology & Human Values* 35.5 (2010), pp. 601–617. ISSN: 0162-2439. DOI: 10.1177/0162243910377624.

[40] Susan L. Star and James R. Griesemer. „Institutional ecology, 'Translations' and boundary objects: Amateurs and professionals in Berkeley's museum of vertebrate zoology, 1907-39.“ In: *Social Studies of Science* 19.3 (1989), pp. 387–420. ISSN: 0306-3127. DOI: 10.1177/030631289019003001.

[41] Philip Tchernavskij. „Asymmetric Collaboration and Webstrates.“ Unpublished manuscript. 2015.

[42] Noah Wardrip-Fruin. „What hypertext is.“ In: *Proceedings of the fifteenth ACM conference on Hypertext and hypermedia*. ACM. 2004, pp. 126–127.

[43] Nicole Yankelovich, Norman K. Meyrowitz, and Andries van Dam. „Reading and writing the electronic book.“ In: *IEEE computer* 18.10 (1985), pp. 15–30.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

All original figures were created with the Inkscape vector graphics editor, available for Linux, OSX, and Windows:

https://inkscape.org

Final Version as of May 2, 2016 (`classicthesis` version 4.2).