subTextile: A Construction Kit for Computationally Enabled Textiles

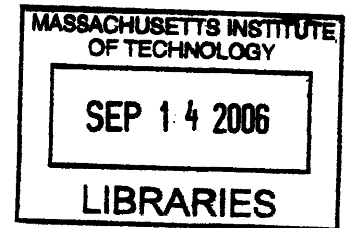
by

Sajid H. Sadi

B.S. Columbia University in the City of New York, New York, USA (2003)

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning
In partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[September 2006]
August 2006



© Massachusetts Institute of Technology 2006. All rights reserved.

ARCHIVES

Author ....
Program in Media Arts and Sciences, School of Architecture and Planning
August 11, 2006

Certified by ....
Patricia Maes
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences, School of Architecture and Planning
Thesis Supervisor

Accepted by ....
Andrew B. Lippman
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences, School of Architecture and Planning

subTextile: A Construction Kit for Computationally Enabled Textiles

by
Sajid H. Sadi

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on September, 2006, in partial fulfillment of the
requirements for the degree of
Masters of Science in Media Arts and Sciences

abstract

As technology moves forward, electronics have enmeshed with every aspect of daily life. Some pioneers have also embraced electronics as a means of expression and exploration, creating the fields of wearable computing and electronic textiles. While wearable computing and electronic textiles seem superficially connected as fields of investigation, in fact they are currently widely separated. However, as the field of electronic textiles grows and matures, it has become apparent that better tools and techniques are necessary in order for artists and designers interested in using electronic textiles as a means of expression and function to be able to use the full capabilities of the available technology.

It remains generally outside the reach of the average designer or artist to create e-textile experiences, thus preventing them from appropriating the technology, and in turn allowing the general public to accept and exploit the technology. There is clearly a need to facilitate this cross-pollination between the technical and design domains both in order to foster greater creativity and depth in the field of electronic textiles, and in order to bring greater social acceptability to wearable computing. This thesis introduces *behavioral textiles*, the intersection of wearable computing and electronic textiles that brings the interactive capability of wearable electronics to electronic textiles. As a means of harnessing this capability, the thesis also presents subTextile, a powerful and novel visual programming language and development. Design guidelines for hardware that can be used with the development environment to create complete behavioral textile systems are also presented. Using a rich, goal-oriented interface, subTextile makes it possible for novices to explore electronic textiles without concern for technical details. This thesis presents the design considerations and motivations that drove the creation of subTextile. Also presented are the result of a preliminary evaluation of the language, done with a sample chosen to represent users with varying capabilities in both the technical and design domains.

Thesis supervisor: Pattie Maes, Associate Professor of Media Arts and Sciences

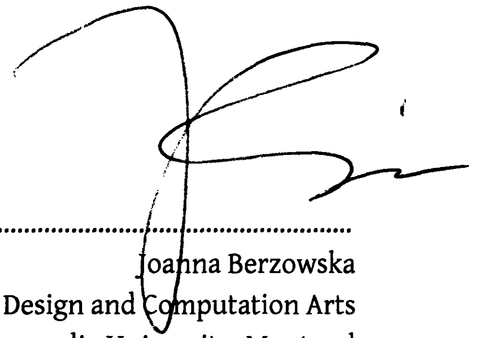
subTextile: A Construction Kit for Computationally Enabled Textiles

by

Sajid H. Sadi

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning
In partial fulfillment of the requirements for the degree of
Master of Science in Media Arts and Sciences
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2006



Thesis Reader ....

Joanna Berzowska
Assistant Professor of Design and Computation Arts
Concordia University, Montreal

Thesis Reader ....

Joseph A. Paradiso
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences, School of Architecture and Planning

acknowledgements

"It is not so much our friends' help that helps us as the confident knowledge that they will help us."

— Epicurus (341 - 270 BC)

Many people have contributed to this work. Their friendship, understanding, help, and critique was invaluable in making shaping this thesis and the thoughts that precipitated it. Therefore, I would like to acknowledge their help, and the confidence they gave my with their presence:

Pattie Maes, my advisor, for accepting me into the Ambient Intelligence Group; for letting me get away with doing what I wanted; and for always encouraging me to reach further and higher.

Joanna Berzowska, for showing me that electronic textiles can be more than what I had seen before, and for taking away the jadedness that kept me back.

Joe Paradiso, for his help, encouragement, and confidence; and for showing me that the difference between theory and practice, between ideas and reality, is merely will and passion.

A special thanks to the **members of Ambient Intelligence**. In particular, thanks to **David Bouchard, Enrico Costanza,** and **David Merrill** for stimulating discussions in a too-cramped office about everything under the sun. Thanks for **Orit Zuckerman** for helping me think about art and the needs of artists, for letting me collaborate on some great projects, and of course for putting up with me as an officemate. Thanks to **Hugo Liu** and **James Teng** for interesting and inspiriting discussions. Thanks to **Polly Guggenheim** for working her magic and taking care of us through thick and thin.

Thomas Hayes, for teaching me process and patience for electronics, and showing me what is possible with dedication and persistence. All I truly understand of electronics, I owe to him.

Bill Mitchell, for showing me what it is like to boldly state the vision of things that others only dare whisper.

Federico Casalegno, for letting me work on interesting projects on my own terms, and putting up with my somewhat bigoted dislike of sociology.

Thanks to the **eLens crew**, and in particular **Enrico Costanza, Jon Gips, Mirja Leinss**, and **Aaron Zinman**, for being themselves. There can be no others, and there can be no recovery from *zero-chi*.

Assaf Feldman, for teaching me how to swim in the Media Lab ocean, for being my first "colleague," and for getting me through my first year here.

David Gatenby, for showing me how to "noodle around" while getting things done.

Adam Boulanger, for his crazy schemes to build gigantic things far beyond our capabilities.

Jason Alanso, for letting me meddle with his projects, and for being the quintessential MIT guy. So this is what it means to be responsible, huh?

Ari Benbasat, for timely and appropriate sarcasm, encouragement, and actual sage advice.

Orkan Telhan, for sticking around too late into the night to talk about philosophy, design, and other things that are too difficult to talk about when alert. A special thanks also goes to **Arzu** for putting her husband out on loan for us.

Thanks to **my friends and colleagues at the Media Lab**, a community of incredible people. In particular, thanks to **Vincent Leclerc, Oren Zuckerman, Cory Kidd, Christine Liu, Nick Knouf, Gemma Shusterman**, and **Ayah Bdeir**.

Thanks to all the **UROPs and subjects** of my study who worked with me at various times.

Linda Peterson and **Pat Solakoff**, for making sure that all the administrative wheels kept rolling when I wasn't paying attention, and for the extra magic for when they threatened to run over me.

Thanks to the sponsor relations folks: **Sarah Page, Felice Gardner, Deb Widener, Lisa Lieberson,** and others for keeping the sponsors at bay, and waiting patiently while I ran over time.

The Media Lab sponsors, without whose resources, encouragement, and dedication the Media Lab and this thesis could not exist.

And finally, a very special thanks to **my parents** and **my sister**, for their tireless support; for their faith; for their prayers and hopes; for carrying the heavy burdens when my eyes were fixed on my dreams; and for making me the person I am today. You are always in my heart.

Thank you!

table of contents

abstract 3

acknowledgements 7

table of contents 11

1 **introduction** 13
- 1.1 contributions and roadmap 15

2 **state of field** 17
- 2.1 electronic textiles as materials 17
- 2.2 electronic textiles as expression 18
- 2.3 wearable computing 20
- 2.4 review of other related work 22

3 **the subTextile system** 29
- 3.1 development environment 30
- 3.2 hardware 31
- 3.3 software-hardware interaction 32
- 3.4 interface description 33
- 3.5 prototype application: ambient remote awareness pillow 37

4 **design considerations** 41
- 4.1 designing a toolkit for electronic textiles 42
- 4.2 aspects of the design of subTextile 43
- 4.3 design goals 44
- 4.4 prototypes 47

5 **software environment** 55
- 5.1 language goals 57
- 5.2 language description 61
- 5.3 language design choices 67
- 5.4 ui design choices 69

6 **physical layer guidelines** 71
- 6.1 communication 72

7 **evaluation** 77
- 7.1 experimental method 78
- 7.2 results 80

8 conclusions and future work 83

9 bibliography 87

appendix a: supported operators 91

appendix b: evaluation documents 93

9.1 documentation attachments provided 93

9.2 tasks 95

9.3 post-task questionnaire 96

1 introduction

With the advent of computationally-enabled textiles or "electronic textiles," it is becoming possible for textiles to have expression and a "mind" of their own. While a number of groups both commercial and academic have proposed solutions for applications as widely separated as fashion and battlefield biomonitoring, prototypes thus far have been limited to one-off systems with limited customizability, or systems which differentiate only in form, but not in concept or function. The wearables field is currently fragmented between material sciences, fashion & expression, and wearable computing & biomonitoring. However, the space between expression and wearable computing has seen virtually no exploration. Little work has been done in making the behavior of e-textiles easily changeable by designers and artists, and to allow expressive e-textiles to have computational behaviors.

E-textiles applications continue to be one-shot applications. There already exist many examples of such single-shot applications, but other than wearable computing platforms, very few examples that can be customized or programmed have been explored. Much of the work in the field of wearable systems, which is split between the camps of "computation that is wearable" (ie, "wearable computing" in the sense of the ISWC conference) and "wearables that can compute" (ie, "computationally-enabled fabrics" or "e-textiles"), is either highly general or highly specialized, but difficult to modify in either case whether due to design or due to the complexity of the programming environment. Of course, good designs should in fact be highly specialized, not merely blank chalkboards of capability. It is therefore a challenge to provide capability which encourages creativity, without producing constraints that turn designs into cookie-cutter copies lacking real substance.

It may be best to describe the problem by analogy. The current state of e-textiles is such that if regular textiles were in the same state, one would be required to weave cloth from fibers, die and cut them to fit, and sew them, in order to have a shirt to wear. Moreover, there is very limited crosstalk between the fields of wearable computing and e-textiles, and resultantly, a lack of programmable e-textile components. It remains generally outside the reach of the average designer or artist to create e-textile experiences, thus preventing them from appropriating the technology, and in turn allowing the general public to accept and exploit the technology. There is clearly a need to facilitate this cross-pollination between the technical and design domains both in order to foster greater creativity and depth in the field of electronic textiles, and in order to bring greater social acceptability to wearable computing. The term "behavioral textiles" will be used in this thesis to describe this intersection.

The term "behavioral textiles" applies the definition of "behavior" more strictly than is used in the vernacular. Instead of simply suggesting reactivity, behavior requires that the action be in response to stimuli. This in turn suggests a dependence on input and more complex processing of internal and external state. While it is entirely possible to demonstrate computational expression through the use of randomization [1], ultimately moving beyond the prototype stages to more widespread use requires a deeper foundation in real-world input and true behavior-oriented interactivity that comes from complex programmed behaviors.

The gaps between the wearable computing and electronic textile fields extend in multiple directions. Clearly, there is a technical gap that needs to be filled. However, this is perhaps the most superficial of fissures. More importantly, there is a sizable gap in access to the technology. "Access" is a deceptive term, because it does not merely require physical access, but also a "conceptual interface" that matches the capability of the user to the technology required to make ideas become reality. Lastly, the deepest fissures exist at the level of "knowledge of capability." In working with designers and artists, the author has found that consistently, the problem is not merely the technical knowledge and capability, but rather the knowledge of the possibilities afforded them through their collaboration with the author. Often, the most difficult task in collaboration has been to convince collaborators to expand their creativity to fill the entire length and breadth of the technology, while at the same time understanding the boundaries presented by the technology. In most cases, the boundaries have only been expanded by these encounters. This suggests a deep need to show artists and designers what is truly possible and feasible, not merely to show them tools to access technology.

This thesis presents subTextile, a system that supports the entire workflow within the domain of behavioral textiles, addressing interfaces to allow motivated but novice users to modify and behaviorally program computationally-enabled fabrics, a task previously considered the domain of wearable computing. The goal of the work presented is to nurture creativity without the distractions of technical details that are relatively uninteresting in context. The subTextile consists of two components: a visual language and associated development environment, and a hardware specification for creating hardware that can be controlled by the language. The subTextile language is designed to be as expressive as traditional programming using standard textual languages, while providing high-level primitives that succinct and easy to learn. The language allows extension to any hardware device that follows the specifications, but does not require creation of binary extensions to do so, thus allowing the user to operate independent of the design of language internals. The hardware specification is centered on a master node that is capable of running the subTextile virtual machine and capable of communicating to attached devices. Devices can be created to do as much as the user wishes, and do not demand any particular choice of hardware.

In order to motivate the thesis, a lifecycle scenario for the system is helpful: Ralph is a creator of e-textiles. His company produces a number of textiles incorporating the subTextile hardware specifications. This not only simplifies design of the hardware interfaces for the textiles, but allows artists and designers to use his textiles and systems, and program the textile in a uniform way instead of having to learn about the minutia of the programming interface and

hardware each time. Lauren, an artist starting off with e-textiles as a way of portraying self-knowledge, picks up several of Ralph's textile components, and once she has created a dress with the components, she connects the parts together, and then simply connects the system to her computer, which runs the development environment, and programs the textile to express her own ideas. These same components can be used by Brad, a designer, to create a new cubicle wall which can remember the visitors who came by, without any "deep" hardware modification. All these people are essentially using components of the same system, and are able to work seamlessly through it.

To provide contrast, the current workflow essentially requires that Ralph, Lauren, and Brad all operate independently, except in the unlikely event that they know each other already and have technology that can be reused. Each of them must develop the electronics themselves, or collaborate with someone who can do so. The materials are very raw (ie, conductive ink and thread, organza, etc.), and even for more finished products, there is no standard. Each must develop the necessary software, and though most of their software will be boilerplate code, they will have to write it again themselves and spend time debugging and correcting the code, taking time away from the actual task. Also, unless they are capable of doing the hardware and software work themselves, they must spend time communicating their vision to their collaborators, instead of playing with ideas with their own hands. Once the pieces are complete, the hardware and software will likely be one-offs, since it takes more effort and resources to create reusable code and hardware. Moreover, since the hardware is monolithic, any problems that are discovered late would require building new circuitry and software that accomplishes the new goals.

As the scenarios suggests, there is need for both a uniform hardware subsystem and a software development environment, as well as generalized support for a wide variety of electronic textiles. As such, the subTextile system is divided into two distinct yet interrelated components. The subTextile hardware specification describes the protocols and execution environment that allows behavioral control over electronic textiles. More importantly, the subTextile visual language (and integrated development environment), the primary focus of the work presented herein, allows the creation of programs that make behavioral textiles accessible and simple to create. This language and development environment forms the other half of the equation, bringing with it not only access to the technological capabilities of the hardware subsystem, but providing an avenue for the user to discover the true creative boundaries of behavioral textiles by easing the translation of concept into reality.

1.1 contributions and roadmap

The contributions of this thesis are outlined below, approximating the layout of the thesis itself. This work presents:

- A definition of a specific subclass of electronic textiles geared towards deeper and more interesting interactivity and reactivity
- A survey of the state of wearable computing and electronic textiles

- A survey of existing techniques used in order to create behavioral textiles
- A set of prototypes created in order to discover the needs of electronic textiles hardware and software
- Needs analysis for the creation of behavioral textiles
- An analysis of programming techniques for novice audiences, with particular focus on approaches for analyzing visual languages
- A novel visual language and development environment for programming electronic textiles with particular focus on fit-to-problem and ease of use
- A detailed specification for a hardware environment for the execution of aforementioned language
- A goals-oriented approach to the creation of toolkits that foster creativity and behavior-oriented design, while empowering adults to understand the true outer limits of available technology without becoming mired in technical minutiae
- A study of the design strengths and weaknesses of the proposed toolkit
- An evaluation of the design correctness of the language and analysis of shortcomings and capabilities is also presented.

2 state of field

This chapter, as well as the following one, will present a short overview of related work in this field. This chapter is primarily concerned with addressing the state of the field of electronic textiles and on-body computing, while discussion of languages and approaches is delayed to the next chapter. Despite the relative youth of the field of wearable computing and electronic wearables, the area has been the subject of considerable discussion as of late. In addition to an overview of the existing work, each subsection offers a critique of the current approach with particular attention to the both the ability of the design community to appropriate the technology in a meaningful way, and the ability of the general public to accept and appropriate a product designed on these bases.

2.1 electronic textiles as materials

Research into wearable computing has been accompanied by considerable advances in material sciences. Textiles, as electronic materials, exist as a bridge between the fringe technologies of the wearable computing world and the more concrete needs of commercial technologies. Pioneering work in the field of electronic textiles was done by Post, Orth, and others at the MIT Media Lab as early as 1997 [3]. Though the work eventually turning toward expression, it initially focused on the material aspects of integrating electronics into textiles. The Firefly

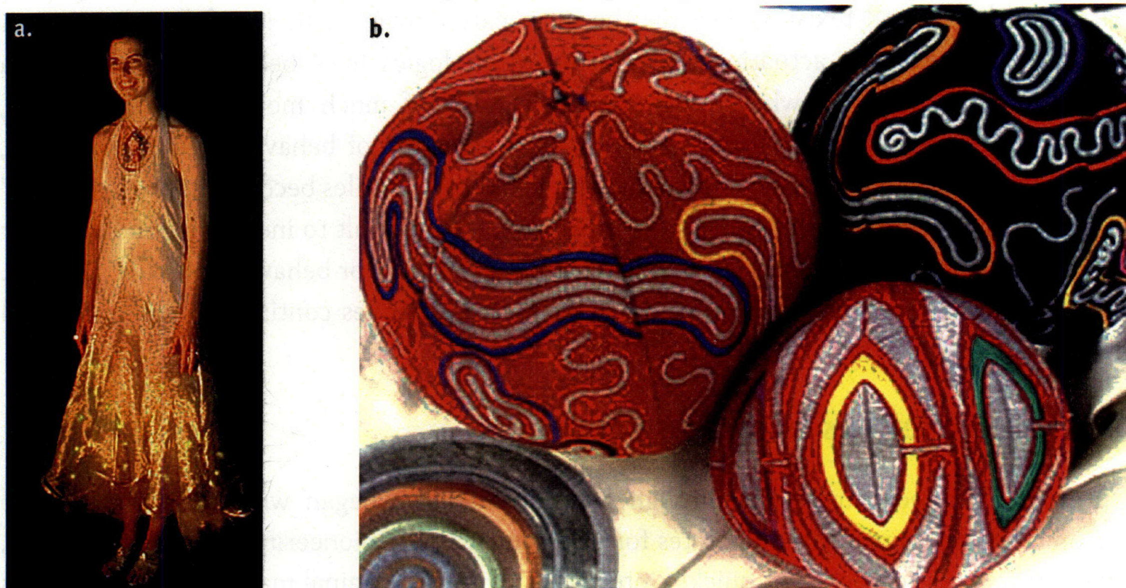


Figure 1 a. Firefly Dress by Maggie Orth (source: [2]) b. Music Balls by Post et al (source: [2])

Dress by Maggie Orth demonstrated the use of the material qualities of electronic textile for expression [2]. While the dress did not include true input processing, other works from the same time included the Music Balls, which served as inputs to a substantial music system that could be easily manipulated via a fabric interface [4]. At the same time, commercial ventures began investigation of embedded circuitry in textiles [2]. While many of the discoveries in the commercial sector remain highly confidential, the military has consistently shown a vested interest in technology embedded in combat-wear, and the materials technology continues to be pushed forward by commercial forces along this path.

More recently, a number of commercial concerns such as Sensatex [5] and Softswitch [6] have started marketing textile systems bases on innovations in this field. Sensatex offers custom sensing fabrics capable of monitoring various biological activity of the wearer, while Softswitch incorporates functional controls for portable music players and similar devices into clothing. A number of groups have also begun work on integrating high-density pixel-oriented addressable displays into textiles. One example of this is a textile from Philips Research with integral output capability integrated into a pillow [7]. For the time, most of these projects do not include collocated input capabilities, thought that is certain to change as the technology improves.

It should also be noted that a large body of research also exists in the field of biomedicine in terms of wearable devices, sensors for biomonitoring, and the use of e-textiles in that context, and is summarized by Capri et al. [8]. Additionally, innovative work in being done which is expected to remove the need for hard circuit boards and create clothing which is truly capable of native computation. Attempts are being made to both create circuits into fibers used to create textile [9] and to create flexible substrates that can be seamlessly sewn or concealed in textiles with no outward side effects from their inclusion [10]. Additionally, interfaces that allow input and actuation to and from these "integrally smart" textiles are being developed by a number of teams [11-14].

While these latter technologies will doubtless have a great impact on the future of wearable devices and behavioral textiles, they for the most part remain in their infancy in terms of computation power. The actuation and sensing technologies have been investigated for a longer time, and are somewhat more mature. Though of much more immediate utility, actuation and sensing textiles form only half of the equation of behavioral textiles. Indeed, without the computation portion of the equation, behavioral textiles become merely "reactive" in their behavior. Until greater robustness is possible, it is difficult to include technologies for embedding computation in textile in either wearable computing or behavioral textile systems,. As such, much of the electronics used to drive electronic textiles continue to use traditional materials and designs [15].

2.2 electronic textiles as expression

As previously stated, expression through electronic textiles began with the research into electronic textiles as material substrates for electronics. In the pioneering work by Post, Orth, and others, expression becomes quickly intertwined with the original material science aspects

of the research [2]. However, as the field of electronic textiles matured, the two have diverged considerably. More recently, however, these has been some convergence in the field with the work from XS Labs [16], Hexgram Institute [17] and individual artists and designers. The artistic process of fabric workmanship is once again being investigated alongside the intricacies of interactive textiles design. As the field matures into the mainstream, the artificial divide between electronic textiles as materials vs. as expressive medium is closing, mirroring the world of traditional textile work, where the two are intrinsically connected.

As electronic textiles move into the mainstream, there have been increasing attempts by both fashion designers and interested academic groups to include electronics as a means of expression. These forays can be roughly divided into conceptual pieces and pieces geared towards expression or fashion, though of course large overlaps exist. To a large extent, the fashion industry has been slow in its uptake of technology into the mainstream, though the novelty value of technology in fashion has not escaped notice. As the surge of electronic fashion shows in technology-oriented conferences such as the International Symposium on Wearable Computers (ISWC) and ACM SIGGRAPH indicate, this has of recent become a bridging area. Technology-oriented conferences receive the glamour of the runway in exchange for the novelty in design introduced into the fashion industry with the inclusion of electronics in fabrics. At the same time, the resultant injection of futurism into the field has in effect made it difficult to reconcile the vision of electronic textiles with current realities in the public psyche.

It is also interesting to note that the number of participants in this field are numerous, ranging from traditionally-trained designers experimenting with new forms to cross-disciplinary artists and technologists birthed on these technologies, as well as material scientists, researchers from the area of wearable computing, and of course those interested in fashion. Each of these groups brings their own views to the table, creating a chaotic field. Nonetheless, guidelines and analysis for the overall needs and capabilities of computationally-enabled textiles are now emerging. Baurley presents an overview of the field and its points out the facets it shares with information technology [18]. A recent publication by McCann et al. also presents an overall design methodology and concerns at each stage of the design process particularly for "smart clothing." It discusses not only the requirements for technological systems, but also for the human body itself [19]. Technologies such as Zuf [20] and bYOB [21] have also begun to address the need for extensibility and behavioral complexity in the domain.

A number of groups are currently involved in various aspects of augmentation of expression through textiles. The Topological Media Lab [22] is currently exploring the interaction between augmented spaces and augmented clothing [23]. Reach [24] and urbanheremes [25], as well as the iBand [26] explore the social aspects of computationally-enabled apparel. These projects give insight into wearable technology and suggest implications for e-textiles both on-body and in the environment. Apart from these pragmatic applications, e-textiles are also suited for their original artistic uses. At this intersection of computation and art, the "role of random" can vary from simple generative actions to lifelike "unexplainable" actions which show patterns of action while defying simple logical deduction [1]. It is no longer merely a method of introducing variability into programmatic structure, but a method of expressing the variability

found in art and nature. Berzowska's group has produced a series of dresses which emphasize such expression via incorporation of electronics in fashion in more playful ways [13, 27]. Such designs are also found plentifully in the fashion industry's forays into e-textiles [28, 29]. While powerful in their own right, these designs nonetheless leave ample room for more complex and deep behaviors that imbue textiles not only with animation, but with a form of will and intelligence.

Despite the relatively wide participation from a number of communities, the threshold for entry remains quite high for the average designer or artist interested in entering this field, mostly due to the need to "reinvent the wheel" in order to get anything done at all. The components that are used in electronic fashion are essentially the same parts one might use to build any other electronic circuit, and thus the information about these parts is designed for engineers. It becomes necessary, therefore, for traditionally trained designers and artists to seek collaboration in order to implement their ideas. Having collaborated on a number of such occasions, it is clear that this solution is not optimal. Ultimately, the imagination of the artist is bound by pragmatism, and oftentimes, this pragmatism is misplaced. In short, the beliefs of the artists about what is possible is bounded by what the artist feels he or she can accomplish, and thus oftentimes it is a tedious process to attempt to get an understanding of how much of the artist's dream is feasible, as opposed to how much of the artist's pragmatic guidelines are feasible. In practical cases, it has always been that while technology is hardly even sufficient to realize all of the artist's dream, it is capable of realizing far more than what the artist pragmatically believed was possible within a given time or budget. This limitation is perhaps the true shackles that hold back the field of electronic textiles in expressive terms, and is the problem that we hope to attack with this thesis.

2.3 wearable computing

The field of wearable computing started in the 1970's and has since established itself as its own branch of computing. The field of wearable computing is concerned with the technology and interface of devices that rest within the personal sphere. The ideas behind wearable computing have been variously reformulated in terms of human augmentation, with a focus on creating a mental prosthesis that can give the user "sixth senses" about their environment using the sensing and computational faculties of computers. Steve Mann, one of the pioneers of the field, has termed this "humanistic computing," wherein he describes this form of wearable computing as a human-machine system where the human is integrated into the control loop [30]. He has also described wearable computing systems as "smart clothing" which forms a conceptual middle-ground between device-centric world of personal computing and the space-centric approach of ubiquitous computing [31].

A different view of the field is outlined by Bradley Rhodes, another pioneer in the field. The wearable remembrance agent system proposed by Rhodes again augments the human being wearing it, but in the capacity of a personal assistant or librarian who not only catalogues but also intelligently considers and presents information [32]. At the same time, Alex Pentland has taken the idea of wearable computing in a further direction that is perhaps more aligned with

Mann, in using wearable devices in order to gain insight into social behavior in group situations, thus giving the wearer a social sixth sense based on the insights of the wearable device [33]. While all three authors suggest a change in the way that computers interact with humans, Mann suggests a more visceral modification of the senses than Rhodes, whose approach focuses on an improvement over existing user interface paradigms that focus on causal input-output relationships. Pentland splits the difference by allowing the wearer a sixth sense from an autonomous wearable inspector.

While the aforementioned investigators hope to alter the role of computing in daily life, there is also a large amount of research geared toward improving the technology of wearable computing itself. While wearable computing has goals that are different from the average desktop computer at the application level, these same applications essentially demand the same level of computational capability as desktop applications. The MIThril project is a particular incarnation of this family of wearable computing platforms [34]. A similar system, the aptly named Georgia Tech Wearable Motherboard, also focuses on similar abilities to provide a distributed on-body computing system capable of supporting a wide variety of wearable computing [35]. Perhaps not surprisingly, with the advent of more powerful portable devices such as PDAs and cellular phones, the importance of custom on-body computation has itself waned, with the focus now shifting to accessibility and interface technologies [34].

When considering wearable computing in the context of behavioral textiles, it is necessary to consider the ability of the designers and the general public to appropriate it. Even while the field has matured, the involved individuals have remained highly technically proficient and focused on moderate-to-long term goal of birthing a new paradigm for the interaction between humans and computers. Additionally, while social acceptance has always been an issue within the field, it has generally remained a secondary goal. More recently, this has started to change to an extent [36], though uptake remains relatively slow. Additionally, most of the applications created by the wearable computing community remain non-trivial to use, while requiring wearable devices that are generally difficult to disguise or integrate into everyday clothing and situations.

In technical terms, it is perhaps more sensible to speak of integrating, utilizing, or developing on wearable technology platforms than to speak of appropriating such systems, since in some sense they are the wearable counterparts to the average desktop system. Insofar as one may appropriate a device via personal markings, programming, and physical modification, wearable computing devices are reasonable appropriable. However, due to the relatively crude grafting of electronics to the wearable substrate, any changes require an understanding of the system at a software and hardware level, thus making them inaccessible to most of the design community. The threshold for entry into the field is heightened by the additional hurdle of creating hardware and software for the environment, a considerable task even for those versed in the technological underpinnings. Perhaps the greatest issue with considering wearable computing devices as an underpinning for behavioral textiles is that the devices are general in software capability while remaining fairly special purpose in hardware, which is essentially contrary to the requirements of a toolkit for generalizing behavioral textiles, which demands

fairly special-purpose software running on generalized hardware that can produce a plethora of effects with minimal engineering requirements for designers and users. Additionally, there is a subtle but critical divergence with particular segments of the wearable computing community due to the treatment of the hardware as a goal in-and-of itself as opposed to a means to achieving a conceptual or expressional end, which in turn leads back to the issues mentioned previously.

2.4 review of other related work

As stated in the last section, subTextile is not the first system to attempt to address the problem of assisting in the creation of behavioral textiles. In fact, many systems in existence today make no use of programmed elements at all. Many others make use of microcontrollers programmed using traditional circuit design and programming methods to create one-off circuits particular to one textile design. However, some forays have also been made into the field of programming tools, and will be described in the following subsections.

2.4.1 systems specific to textiles

Zuf, a system designed by Megan Galbraith, is perhaps the system that is most directly related to subTextile, and shares the same goals of allowing designers to transform conceptual expectations into tangible results [20]. The system uses "fuzzy logic," or algorithmic learning techniques, to transform natural language descriptions of input and output states into equivalent input and output on the textile itself. An example of the input can be seen in Figure 2. Zuf essentially uses a loosely constrained input parameter set to generate a set of rules for mapping inputs to outputs statistically. The advantage of this approach is its extreme simplicity. In fact, in the parlance of pattern recognition, what takes place is not programming but rather training of a linear system. Additionally, the web-based design, with upload of the "program" via TCP/IP allows for a user experience which is very different from the popular vision of what programming entails. This can be used to "trick" users wary of being able to program into programming textiles.

Unfortunately, the same algorithmic learning system that gives Zuf its capabilities also robs it of expressive power. Without a self-training system, it is nearly impossible for Zuf to handle state or memory. The behaviors that can be programmed via the highly template-oriented natural language selection system are also essentially unitary, allowing one input to affect one output. Of course, this can be overcome via the use of multiple rules that affect the same end-effectors, but the interaction between the rules in such a case are unclear, and can easily lead to user confusion when the system does not match the user's mental model of what the system should do in practice. Additionally, the Zuf system is biased toward continuous output methods which map better to the probabilistic outputs of the learning system than binary outputs would. While this does to some extent cause a bias towards binary inputs, properly manipulating the values assigned to input states can be used to correct for the problem. The cost of the flexibility is paid in greater requirement for user input and a requirement for users to have a better understanding of the system internals. As such, while highly empowering in

terms of electronic textiles as they exist today, the Zuf system is not well-suited to the creation of behavioral textiles as defined in the context of this thesis. A simple example of these shortcomings would be control over a grid of LEDs, which would be quite difficult due to the lack of proper primitive types within the Zuf "fuzzy logic." Likewise, it would be quite difficult to create a system that smoothly animates a fabric flower (as is done with Kukkia [13]), while adding a behavior to the flower to close when the user attempts to touch the flower.

While the Zuf system focused almost exclusively on software innovation, the bYOB system shown in Figure 3 chose hardware as the focus instead [21]. The bYOB system allows the creation of textile pieces that have a variety of capabilities using a quilting-like approach. The test case is a handbag. The bag is made up of patches that encapsulate particular behaviors. As the patches constituting the bag are changed, the capabilities of the bag also change. The approach taken by the bYOB system focused on accessorizing and physical construction and manipulation as primitives for creating an electronic textile that had malleable properties. Additionally, the thesis focused more on the end user and on preprogrammed behaviors built into physical elements than on freeform creation of behaviors by the designers of electronic textiles. The system used an ad-hoc networking approach to the problem of customization,

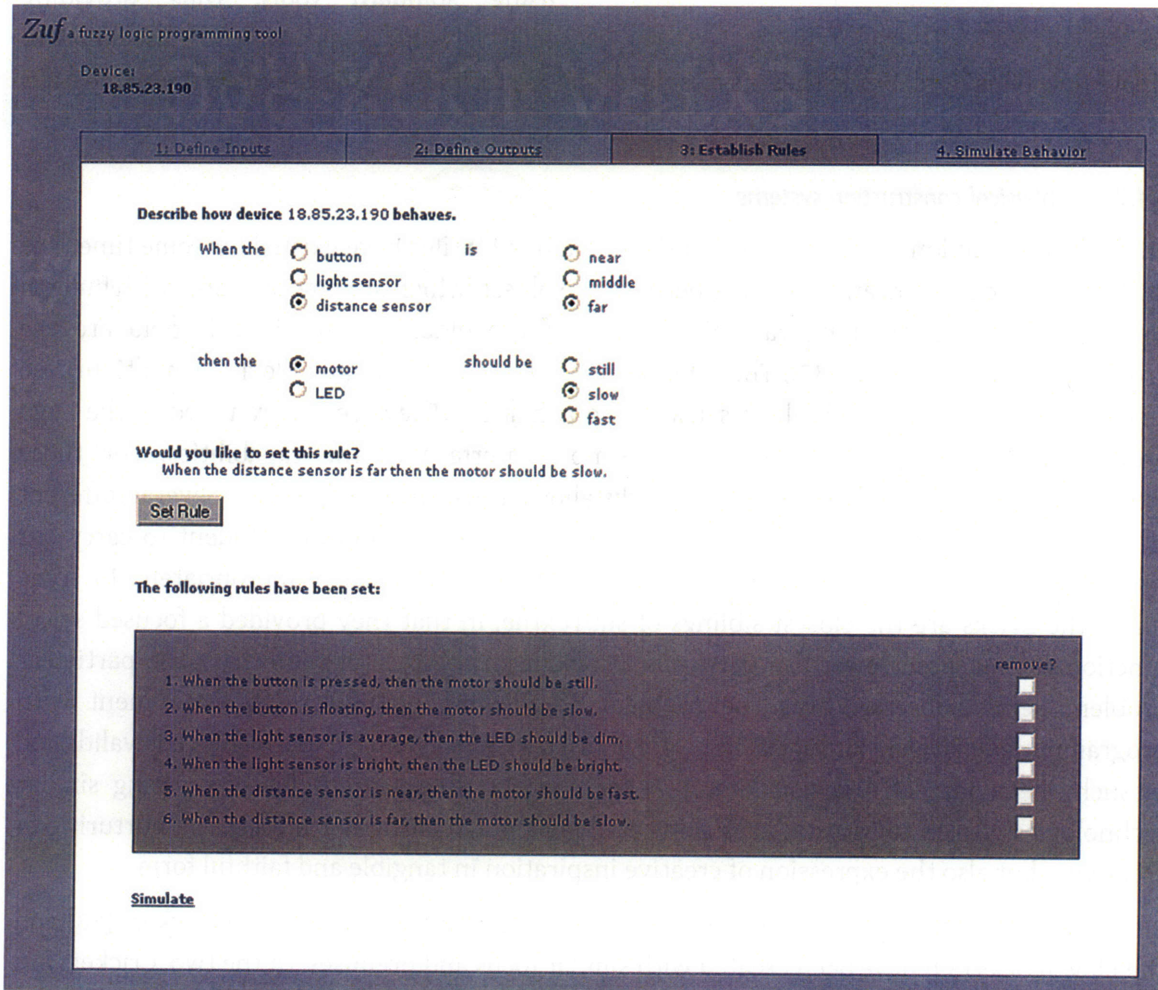


Figure 2 Zuf Screen, 3rd stage, showing the assignment of rules to input and output conditions (source: [20])

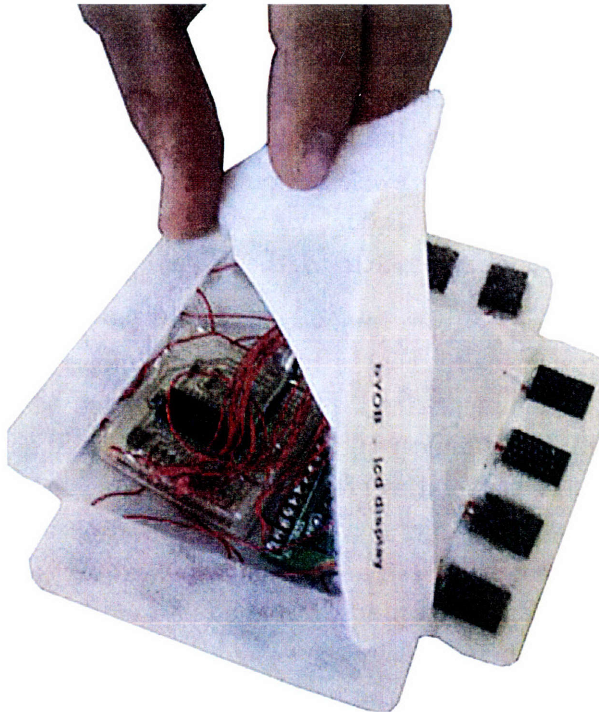


Figure 3 bYOB module (Source: [21])

with the electronic textile being constructed out of elements that each added their own capabilities to the whole in the form of a prepackaged behavior. While powerful in its simplicity and ease of use, the bYOB system is also faced with the critical problem of allowing appropriation when the lowest accessible level of primitives is highly designed in both form and function. In fact, the tight coupling of physical form with behavioral capabilities only highlighted the issue by tying not only physical but also reactive functionality to the physical form. Additionally, the behaviors offered by the system, while technically limitless, essentially break down into the creation of hardware and software using standard tools, thus providing relatively little aid to those seeking to create new behaviors. While these issues do not pose a problem to the use of electronic textiles as accessories, they are incompatible with the idea of a truly flexible behavioral textile system.

2.4.2 physical construction systems

Physical construction systems outside of the domain of textiles have existed for some time. The lowly Lego block and its analogues are perhaps the oldest, which are now available with built-in electronic and programming capabilities. One of the older systems in this field are the Programmable Lego Bricks [37]. These bricks are not so much building blocks as multipurpose development systems with built-in sensing and actuation. They are programmed in the Logo programming language, providing for a simple interface to the capabilities. For those interested in learning the language, the Bricks allowed reasonable capability. Several different "flavors" of bricks were made available, within the context of allowing student to carry out simple physical tasks or experiments, and capabilities were chosen appropriately. In some sense, the Bricks are the closest siblings of subTextile, in that they provided a focused set of functionalities geared towards a particular problem. In the case of the Bricks, the particular problem being addressed was the problem of allowing youngsters to experiment with programming fairly generic blocks to perform tasks within a simple experiment. The validation of such "behavior construction kits" [37] for children shows the value of creating similar technologies geared towards adults facing problems not relating not merely the nurturing of creativity, but also the expression of creative inspiration in tangible and faithful form.

Since the time of the Programmable Brick, several other systems such as the Crickets [38] and the Flow Blocks [39] have been created with similar focus and premises. Of the two, Crickets are a direct descendent of the Programmable Bricks, while flow blocks have diverged to use the

topology of the blocks themselves in the description of the language. Flow blocks use the layout of blocks with special functions as a method of teaching concepts like repetition [39]. An interesting hybrid use of the concept of blocks as both programming and processing elements was made by John Harrison in his thesis work, SoundBlocks [40]. In addition, the SoundBlocks system fed back to a visual programming system named SoundScratch that depicted the processing element organization in the Scratch programming language described in the next section. Also worthy of mention is the Nylon system developed by the Aesthetics and Computation Group at the MIT Media Lab [41]. Nylon is a device/programming system combination similar to the Programmable Brick, and is programmed using a simplified textual language with high-level primitives for various hardware input and output tasks. Nylon shares similar goals with the Bricks as well, though it is geared more towards adults and is more general purpose than the Brick.

2.4.3 programming languages and systems

There are a number of programming systems that have a basis in pedagogy and share traits with subTextile. They include both visual and textual languages, though in general textual languages have been considered too syntax-intensive to be introduced at as early an age as is possible with visual languages. Among the textual languages used for this purpose, Basic and Logo are possibly the oldest. The simple syntax of these languages belies their capability. While Basic was not originally designed for use with physical devices, with the advent of microcontrollers with integral Basic interpreters [42], they have made some headway in the field of hardware. The primitives offered by basic, however, tend to be at a fairly low level due to the generalized design of Basic. Nonetheless, the syntactic simplicity of the language has drawn a sizable community to it.

Logo [43], on the other hand, was originally designed explicitly to control hardware, much as subTextile is. Since then, Logo has been used as both a visual programming language and a language without hardware control capabilities. A variant of the language offered as a visual programming system, LogoBlocks [44], offers a fully visual development environment for Logo that does not require typing code. A more recent development with similar goals is Scratch, a visual programming language created in the Lifelong Kindergarten group at the MIT Media Lab [45]. Scratch support not only a fully-visual development environment, but also a large number of visual manipulation and animation primitives borrowed from Squeak [46], the language in which it is written. However, due to their educational bias, LogoBlocks and Scratch focus on the exposition of programming structures to young children. As such, a priority is placed on generation of control structures that mirror real textual programs, and focus on creating visual constraints that give the user a quick sense of correct and incorrect syntax, without necessarily exposing the underlying theory that defines linguistic syntax. As such, the languages are both visually "verbose", in effect mirroring the symbol-level verbosity of the textual languages being represented. Additionally, while visual programming techniques are used to reduce errors and provide a more intuitive interface, the power of the paradigm is restrained in order to achieve the aforementioned mirroring effect with the textual version. As such these

languages serve as excellent introductions to programming as a whole, but do not necessarily serve the needs of behavioral textiles well.

Processing [47] and its relatives, Nylon [41] and Wiring [48] are all textually-based languages very similar to the Java programming language in form, though using higher level primitives tuned for artistic uses. Processing in particular is geared towards visual primitives, while Nylon and Wiring both concentrate on physical sensing and actuation. All variants are fairly general purpose, providing great flexibility at the cost of time needed to learn the language. Additionally, the library-focused design increases the syntactic vocabulary, thus increasing the apparent complexity of the languages. Lastly, Nylon and Wiring do not have a focused domain. As such, they expose the functionality of microcontrollers at a very low level. This approach produces a language that is concerned with the hardware at a very low-level, and leaves much of the boilerplate work to the end user. Lacking these limitation, Processing has been a great success in interactive visual art, and is widely used.

In terms of dataflow languages with hardware interface capabilities, the most commonly used within this category is Max/MSP [49] and its video processing extension, Jitter. Max/MSP is currently the de-facto standard for musical effects processing, though it has been used for many other purposes. The architecture is based solely on dataflow, with no intrinsic support for branching or flow control. Additionally, the state memory of the language is highly biased to waveforms. While this is sufficient for creating audio effects, it is quite difficult to use it for generative tasks. Lacking any real alternatives to Max, artists have nonetheless learned to work around the limitations of the language. With respect to the interface, Max/MSP does not enforce any rules for placement and connection, leading to extremely messy flow graphs for even programs of minimal complexity. Furthermore, no assistance is provided to assist in error correction or functionality gisting. This, combined with its need for myriad workarounds makes Max/MSP difficult for newcomers to start with. Max/MSP is capable of controlling hardware over MIDI, and there is a device available that translates to and from MIDI levels. This approach the bus specification-centric design of subTextile, and allows for great flexibility in peripheral constructions. The requirements of Max, combined with its reliance of MIDI, makes it nearly impossible to use in size-constrained applications.

In closing, it is also necessary to mention some of the remaining programming paradigms. One of these, already mentioned in the description of Zuf, is programming by example. While the approach of Zuf was to use a fairly textual system, the same ideas have been explored using tangible input and output in Topobo [50]. Topobo is a system that uses mimicry combined with collocated input and output systems to replay activity. However, as mentioned in the discussion of Zuf, this is in fact a subset (and not necessarily a majority subset) of the behaviors imaginable in behavioral textiles. In the realm of more straightforward and expressive approaches, electronic textiles thus far have been programmed using traditional languages such as C/C++ or assembly. While these languages are extremely powerful and expressive, they require a very low-level understanding of the components, as well as a large amount of syntax. The syntax is also biased in favor of the compiler rather than the end user, thus increasing the

barrier to entry and oftentimes limiting the imagination of the designers and artists involved by obscuring the forest of capabilities behind the myriad and rigid trunks of syntax.

3 the subTextile system

This chapter describes the components of the subTextile system and gives an overview of approach and actions needed for program creation, execution flow, and interaction with hardware. The subTextile solution consists of two components. The development environment (*subTextile DE*, shown in Figure 4) is used to visually lay out the behavioral software. The subTextile visual programming language is highly integrated into the development environment, requiring no intermediate forms to be generated for compilation. While this may change at a later date, the current approach uses a virtual machine to run the compiled code. User "code" is compiled into bytecode which can then uploaded to the hardware via the proposed on-the-fly update mechanism. The software becomes active immediately upon completion of transmission, and the behavior can be tested in the actual fabric. For this thesis, the development environment has been implemented for testing and analysis of approach, while a comprehensive hardware specification has been created that acts as a counterpart, and

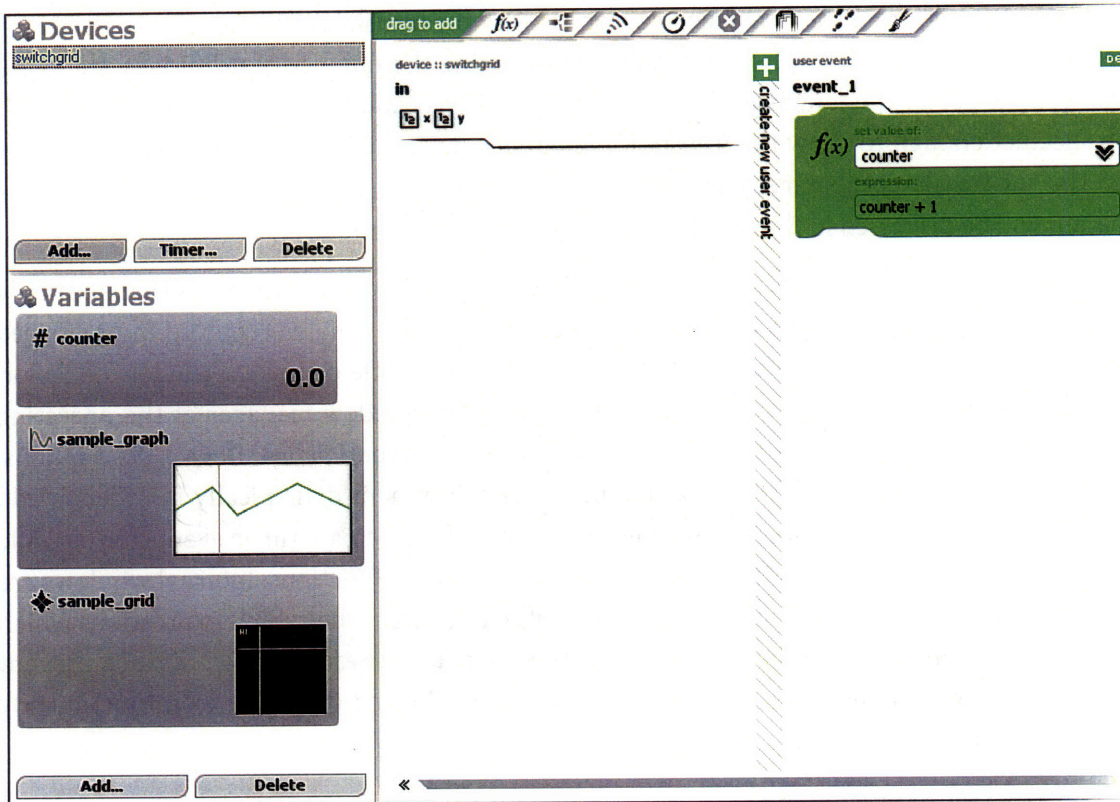


Figure 4 subTextile Development environment

will be completed in the upcoming months. The plan is to produce a central node containing the communication and debugging subsystems, as well as a processor running the virtual machine and non-volatile storage for the bytecode. A number of devices that may find common use in textiles will also be created to provide a starting point for users of the system.

3.1 development environment

It is generally more convenient to consider the subTextile language and the development environment in which it exists as a unit rather than separate entities. Due to the nature of visual languages, there is in fact little to differentiate the two conceptually. In fact, unlike textual languages, the *subTextile DE* is integral to providing the low floor and intuitive interface that allows subTextile to reach for its goals.

The subTextile language is completely event-oriented, where all actions are taken on the basis of some event. Events may be triggered by other events, and can be used to form arbitrary state-machine-like code paths. This decision was based on analysis of existing textile-based systems, as well as the hardware prototype designs done by the author for this particular purpose. Also of note is the relatively small number of primitives provided. Since this language is designed for adult users, the software makes use of components such as freeform expression. Additionally, no distinction is made between functions, if/else statements, and switch statements, which are all supported via a generic switch statement that supports all of the above seamlessly. In keeping with the event-oriented design, the language also does not uses zero-cost event firing in lieu of looping operators. The goal with all these choices is to remove "syntactical sugar" from the language in order to allow the translation of concepts to behaviors in the most direct and uniform way possible. By reducing the number of primitives, the *prima fascia* complexity of the language is curbed. Though the vocabulary of subTextile is rich, it is also succinct enough to allow the user to rapidly break down high-level concepts or behaviors into the subTextile vocabulary. The details of the language and design decisions made in its creation will be discussed in the chapter dedicated to this matter.

The development environment uses only a single window, with captive windows that exist only within the main window of the program. Internal windows can only be removed in prescribed ways, overcoming issues of occlusion. This allows for more subtle control of window behavior, and stems difficulties resulting from interactions with the window manager of the operating system. The window is divided into device, variable, and event panes. Device descriptions, written as simple XML files, can be loaded into the environment dynamically in order to add their events to the main event area. In addition, it is possible to create timer events for periodic tasks, as well as custom events that the user can use as callable functions. Internal windows are used to add and edit variables, but more importantly, to contain the switch expressions which allow for decision making capabilities within the language. These windows do not prevent interactions from taking place between windows, thus allowing drag-and-drop functionality between windows.

The UI design of subTextile DE breaks with common UI layouts intentionally and systematically in order to establish its own visual language and identity (see Figure 4). This is meant to allow users to easily recognize when the UI demands interactions particular to subTextile vs. when a standard GUI interaction is expected. This is particularly evident in the docking bars of the event area, as well as the tear-off ribbon near the top of the event area that is a source of new primitives. At the same time, similar actions are reinforced by reuse of the same visually distinctive patterns. For example, each component that is subject to drag-and-drop (DnD) or editing actions has a depth component, while the UI components tend to meld into a plane, thus reinforcing the separateness of the DnD components. Likewise, the same custom button is used to indicate a location where the user may create new docking bars for primitives. The strong and consistent visual grammar is used to allow the user to easily understand new areas of the interface. Additionally, the interface avoids the use of context menus, which tend to hide functionality, and instead encodes these actions using visual means and by consistent use of gestures such as double-clicks.

The software component is written in the Java programming language (version 1.5). The choice of language is motivated by a number of issues. Java is a well-supported, widely used language available on all major desktop operating system platforms in use. It is also one of the languages that the author has a low-level understanding of, which was integral to the creation of the prototype within the time constraints of the thesis. Finally, Java offers a very complete set of UI primitives with good support for programmatically guided layout and display, which makes it an excellent candidate for the task. Graph oriented layouts, found in many other visual programming language to depict control and/or data flow, and not used within subTextile. Instead, the automated layout capabilities of Java are leveraged for the task. This is essence allows the subTextile interface to be directly translated into bytecode without the more usual approach of translating to an intermediate form and then compiling to final machine code.

3.2 hardware

The subTextile hardware specification calls for a main board with a powerful microcontroller that manages behavioral and communication activities, as well as input and output (I/O) boards that follow a particular specification and can be used to gather input and output according to the specification. The I/O modules are connected to the main board via a 5-wire serial bus that can be daisy-chained up to the limits of the microcontrollers used. However, this limit can be extended with an additional repeater board. An additional limit in this matter is the capability of the main board to power the number of devices connected to the bus. However, it is possible for devices to be self-powered with only a shared ground line as long as all devices on the bus are tolerant to the I/O voltages of all other devices. The devices use an extension of I^2C two-wire protocol developed by Philips for communication [51]. Additionally, the main board is capable of discovering and assigning addresses to the I/O boards even in the presence of multiple copies of the same device on the bus using a single-ended token passing scheme controlled by the fifth wire on the bus. This scheme was utilized due to the clear necessity for an addressing scheme with completely deterministic behavior.

The main board must support sufficient RAM and flash memory in order to run the virtual machine that executes the bytecode produced by the *subTextile DE*. It must be noted that while the bytecode itself is quite compact, the language promotes the use of variables which can at times become sizable. In addition, there is a fixed per-device and per-event overhead that must be taken into account. These constraints demand that the main board have relatively large amounts of RAM relative to the average microcontroller. Despite the sacrifice made in terms of raw speed, several limitations of physical hardware can be ignored by using a virtual machine instead of raw machine code in implementing the hardware layer of subTextile, including limits in stack size, limitation in interrupts and number of timers, and interference between the behavioral user code and the core communication and control code that handles event queuing and dispatch. This in turn allows for a highly robust system where the code can be updated quickly on-demand without resetting the system, which would in turn require a time-consuming re-discovery of connected devices.

The specification supports the transmission of any number of variables with an event, as well as the transmission of full grids of values in an encapsulated form without the use of individual input and output types. TAs mentioned above, devices are represented to the software environment using XML files which fully describe these behaviors and associated identifiers. There is no constraint on the number of input or output events a single device may include, though realistically this is limited by the logical segmenting of devices by function. It should also be noted that subTextile does not espouse the idea of creating one module which contains a large number of functions, but instead promotes the creation of small single-purpose modules that can carry out one task well. In addition to simplifying hardware and software design, it promotes systematic approaches to problem-solving instead of the grassroots approach often used in microcontroller-based designs today.

Lastly, in order to allow partial device sets to be debugged, it is possible have devices in the physical hardware network that are not described in the software. Such devices are catalogued, but are not used. If there are multiple of the same device on a network, the first *n* devices references by the subTextile program will be used, where order is defined as monotonically increasing from the closest to the farthest unit relative to the main board on the daisy chain. Commands sent to nonexistent devices are simply ignored. This can serve as an important debugging aid, where some parts of the module chain is disconnected in order to isolate a problem in the behavior or hardware.

3.3 software-hardware interaction

The software must be able to interact quickly and efficiently with the hardware in order to maintain the feedback loop between user action and textile behavior. At the same time, the constraints of textiles require that the connection be simple, efficient, and low-profile. With the gradual disappearance of serial ports and its replacement by USB interfaces, USB is a preferred candidate for connectivity. The choice of USB is bolstered by single chip USB to serial interface chips from a variety of sources. However, this is not explicitly a part of the hardware

specification, since the actual communication method is essentially immaterial as long as the instance of *subTextile DE* is able to contact the hardware.

Once properly connected and configured, the *subTextile DE* is capable of interrupting the VM and updating the bytecode directly. Once the bytecode is uploaded, the VM is re-initialized and execution continues immediately. Currently, there are no explicit debugging services available within the VM or the DE. However, it is entirely possible to include a "debug" block on the hardware bus which is capable of either directly examining the bus, or capable of outputting values sent to it for debugging. Such a block is described in the hardware sections, and can help in debugging complex behavioral conditions which require understanding of the change in values.

As with any language, debugging support is an important requirement. Debugging support in subTextile is planned at a hardware level. The debugger device will be designed to attach to the main module and listen on the bus for particular messages. The software environment will support setting the message filter to any number of messages. By allowing in-place debugging, problem situations can be more easily found. The debug device will also be capable of receiving messages directed to it, thus allowing debugging of internal values. By focusing on messages, the multi-threaded nature of the software can be hidden, allowing for easier comprehension. The software specifically avoids synchronization and syntax issues at compile-time, thus reducing the runtime error to errors of intent in most cases, and the in-place debugging is well-suited to targeting this family of error conditions.

3.4 interface description

This section will fully describe all of the interface components of subTextile DE. The interface can be broken down into three sections: the left sidebar (Figure 5), the object ribbon (Figure 6), and the main event area. Each of these areas serve to compartmentalize functional units of the programming interaction. The left sidebar contains the least-used portions of the language: the variable and device declarations. While it is used more during the initial setup, afterwards the area can be collapsed to dedicate the entire window to the event area, which is the primary user workspace. The ribbon serves as a constant companion to the event area, and provides a "tear-off" capability to create new program primitives.

3.4.1 sidebar

The sidebar is divided into two sections: the device list and the variable list. The device list shows all devices currently loaded into the environment. Each device is described by a XML file which specifies the device name and numeric identifier, as well as the name and identifiers of all events that can be produced or consumed by the device, and their parameters. The user may add new devices by clicking the add button, which opens a operating system-dependent file dialog to select the device file. In addition to devices that will be added via the communication bus, timers can be added as devices. The delay between timer ticks is specified by the user at creation time.

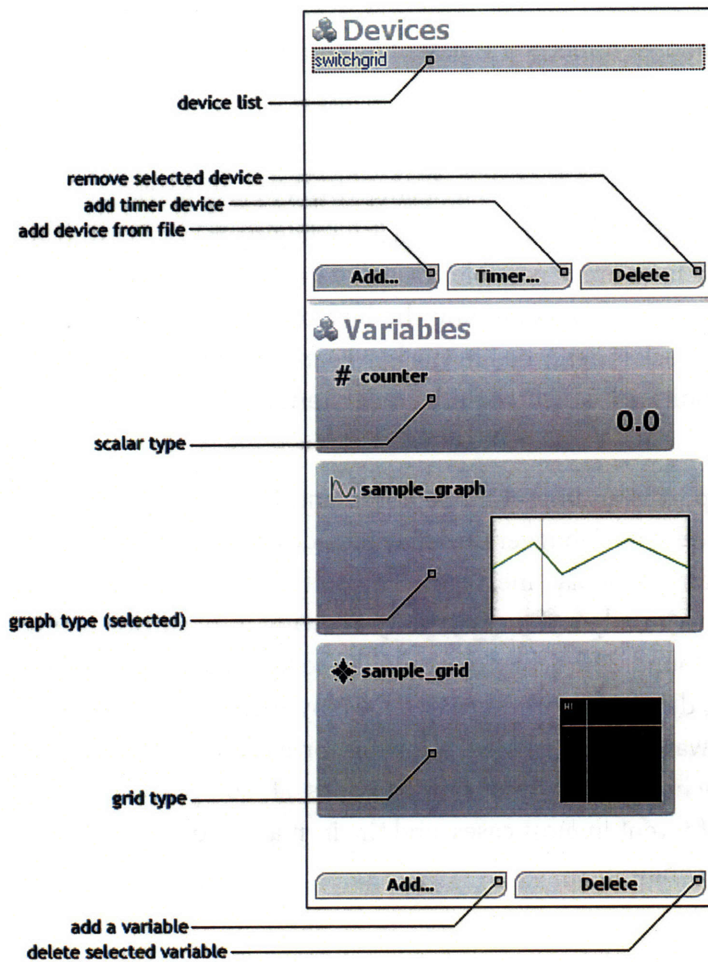


Figure 5 The left sidebar of subTextile DE showing functionality of various components. The sidebar can be collapsed when not in use, providing more space for the adjacent main event area.

The variable list contains the user-defined variables for the current program. Three types of variables are supported in the current version: scalars (denoted by a floating point value), graphs, and grids. Graphs are essentially one-dimensional arrays with integrated cursors, useful is creating animations, but not writable within the program. Grids are essentially grayscale images (range = 0..255). They can also be used as arrays. The size of a grid can be specified at creation time, or edited within the interface.

Each variable type can be edited by double-clicking the visual representation. Under no circumstances can the type of a variable change within the program. Editors for variable are displayed in popup windows, and lock the rest of the interface. Since there are no semantically feasible operations that require the operation of the variable editor in tandem with other popup window, this helps reduce clutter without reducing functionality.

3.4.2 object ribbon

The object ribbon is analogous to a toolbar in that it holds diagrammatic representations of objects to be created. However, a different presentation was necessary since the primary interaction with the ribbon is a "tear-off" dragging action. Since the interface presents multiple locations where a operation object (primitive) can be inserted, a click interaction would in general tend to produce incorrect results. By using a dragging action, the user can place the object correctly on the first try, reducing the number of steps to the desired result. Once the primitive is dragged onto the event area, an insertion mark is shown where the object would appear. The primitives supported are shown in Figure 6.

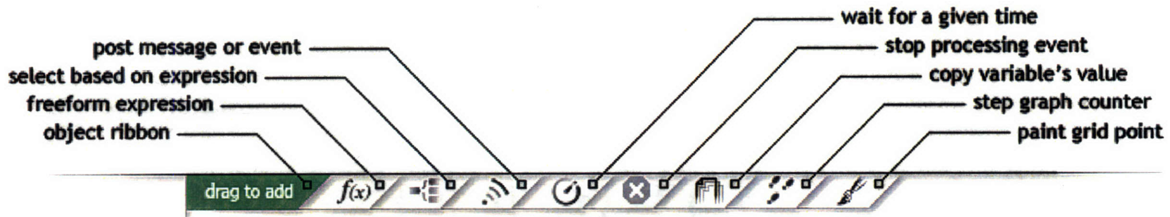


Figure 6 The object ribbon, showing primitives that can be created by "tearing off" icons from the ribbon

3.4.3 choice of operations

This subsection describes the capabilities of all primitives supported by subTextile. The primitives can be roughly subdivided into expression, message, flow control, and variable manipulation types, and the types will be noted after the names of each primitive. The parameters for each primitive. An expanded version of this section is available in section 5.2.3 with notes on parameters and operational details.

Operation: *Expression*
Description: Evaluates freeform expressions
Parameters: Expression string

Operation: *Switch/select*
Description: Evaluates a series of freeform expressions and interprets the results as boolean values. If the expression is a boolean truth, then the corresponding primitive stack are executed. If none of the expressions match, the default primitive stack (if present) is executed.
Parameters: Expression strings, primitives

Operation: *Message*
Description: Causes the firing of an event
Parameters: Event to fire, as well as parameters. Parameters may be variables or constant values.

Operation: *Stop*
Description: Stops the evaluation of an event handler
Parameters: none

Operation: *Wait*
Description: Waits for the given number of milliseconds
Parameters: Variable or constant (selectable)

Operation: *Copy*
Description: Copies one variable to another
Parameters: Two variables. Typing is forward-enforced, allowing graphs to be treated as scalars for assignment to scalars, but requiring that the source be a graph if the destination is a graph.

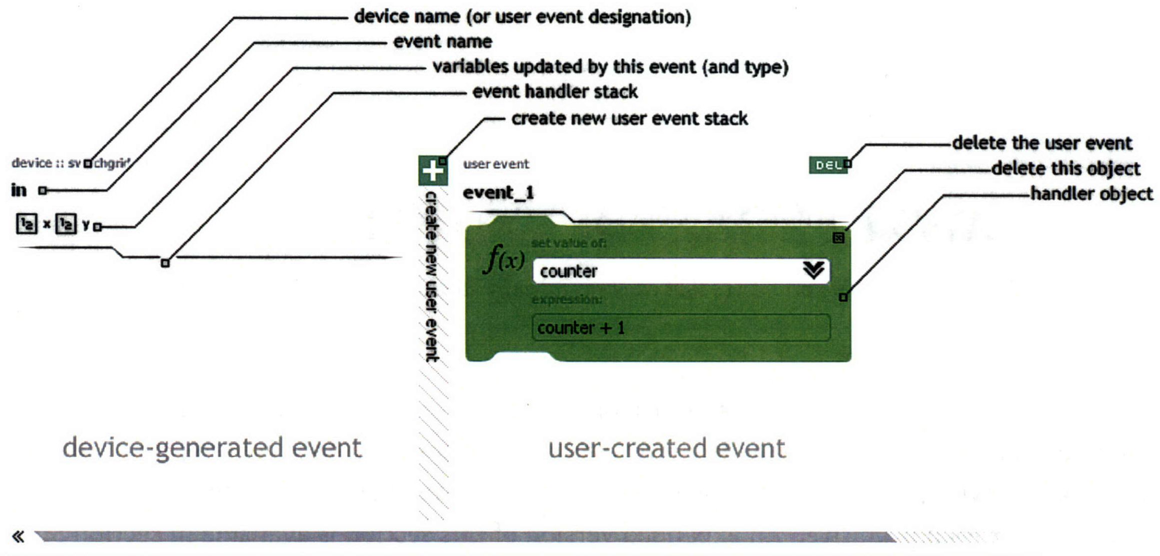


Figure 7 The main event area, showing event hander stack. The object ribbon above the event area is not shown.

Operation: ***Step graph***
Description: Moved the graph cursor in the specified manner
Parameters: Specify how the cursor should be moved. If required, a variable to set the value from or a user-specified constant value is prompted for.

Operation: ***Paint Grid***
Description: Programmatically paint a location on a grid to a particular value
Parameters: The grid to paint, x and y coordinate, and new value. All can be specified as user-supplied constant values, or as variables that evaluate to scalars (scalar and graph type).

3.4.4 main event area

The main event area is the primary workspace for the user, and is scrollable in vertically and horizontally as needed. As shown in Figure 7, the area consists of any number of event handler stacks. Each stack can contain any number of primitives stacked vertically. The primitives in a stack are executed top-to-bottom. Device events (caused by external conditions) and timer events are listed on the left. Execution can only be triggered by these device-generated events. A vertical button for creating user event follows the device events, and any user-created events are placed to the right of the button. User events serve as a method of code reuse, and allow the same code to be executed via various input paths. Icons from the object ribbon can be dragged onto this area to form new primitives. Primitives can also be dragged freely between all event handler stacks.

Of particular note are the event variables shown on device event handler stacks. These appear only when an incoming event provides data in addition to notification of an update. User events cannot have such variables. Once the device is added, the variables appear in the global variable namespace in decorated form. For example, if a device names *switchGrid* has an event name *switchPressed* with variables *x* and *y*, the variables would appear in lists as

switchGrid_switchPressed_x and *switchGrid_switchPressed_y*. The decoration allows the user to determine where the variable comes from and which particular device and event the data belongs to. These variables are available from *all* event handler stacks, and retain their values at all times. The values are merely updated by events, not created in the scope of the events as is the case with most imperative languages. In essence, this provides persistent state for all events, remembering what their condition was at the last event arrival. This matches the planar view of the language the interface provides, and simplifies many common tasks while reducing the number of temporary or state variables the user has to create.

Event execution is atomic at the primitive level and parallel across event handler stacks. Additionally, there is no concept of "calling" an event. When the message primitive is used to fire an internal event, the execution of the event begins immediately and in parallel with the handler stack that launched the event. If the launching stack has unexecuted primitives remaining, it will continue execution in parallel. Execution of primitives is round-robin across handler stacks.

3.5 prototype application: ambient remote awareness pillow

While the above sections provide an abstract overview of the language, it does not fully demonstrate the use of language *per se*. To better illustrate the mental process model and interactions the language is designed around, this section will walk through the creation of a subTextile program to run an ambient awareness pillow. This pillow was created in actuality as a prototype project, and is described fully in section 4.4.3. The hardware used in this scenario matches as closely as possible the actual capabilities of the hardware created for the prototype, and the division of labor between components is maintained.

Problem statement: Create the software to drive the ambient awareness pillow. The pillow has a switch grid containing a 16 × 16 grid of switches. A grid of 16 × 16 LEDs is layered on the switches. When a switch in the grid is pressed, the matching LED should be toggled. The pillow has a communication device, which connects it to another pillow. The pillow should notify the other pillow of any change in state. When buttons 1 and 2 on the first row are held down, the display should be saved. When button 1 and 3 on the first row are held, the display should be restored to the last saved state.

Device details: The switch grid sends a notification event for each button. The button specifies the x and y coordinate of the press. When buttons are held down, the same event is repeated at intervals. The LED grid has an internal representation of its state. On power-up, this state is set to "all off." The grid can be asked to toggle a given point. Additionally, the internal state can be read back as a grid data type by sending a read event, to which the LED grid will respond with an event containing the values. An entire grid can also be written to the LED grid. The communication device sends and receives messages with either 2 values or a grid.

Expected mental task breakdown: First, the devices need to be added. After that, various conditions need to be taken care of. Since subTextile is geared towards breaking down tasks in

terms of conditions or inputs taking place, that is the place to begin. In this case, the user can press a button, choose to save, or choose to recall. The remote user can do the same.

3.5.1 device setup

There are 3 devices necessary: a switch grid, a LED grid, and communication device. The device definition files in this scenario are pre-created, though if these devices were designed by the user, the user would write the definition files. As such, the add button in the device section is used to create the devices. The device list at this point is shown in Figure 8. At this point, the event area has the necessary event handler stacks from the device added automatically as shown in Figure 9.

Figure 8 Device list with devices added

3.5.2 handle user button press

When the user presses a button, a the switch grid will send an event with the position. This event must toggle the correct LED, and communicate the press to the other pillow via the communication device. The message primitive is used to take care of both of these functions. The first toggles the LED grid using the updated values from the event. The second sends a message to the communication device. The configuration of handlers is shown in the first event handler stack of Figure 9.

3.5.3 handle button press from other pillow

When the other user presses a button, this pillow receives a communication event. The appropriate LED must be toggled by messaging the LED grid. This is shown on the right side of Figure 9. Thus far, the variables passed in via the events themselves are used. As shown, the event and variable names are decorated with contextual information to allow the user to distinguish the source and destinations.

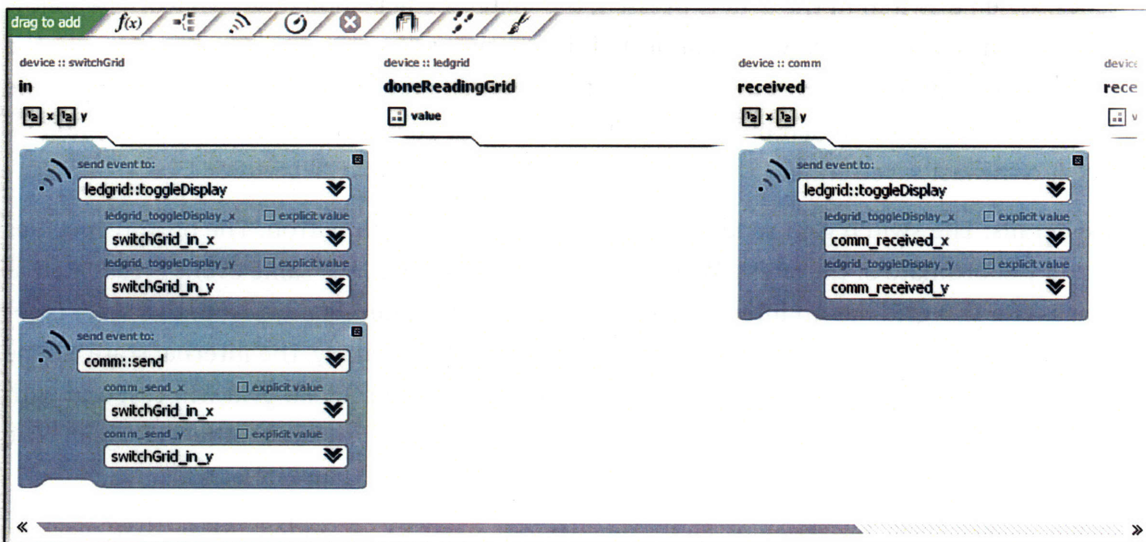


Figure 9 Main event area with local and remote user button presses handled, showing all device handler stacks.

3.5.4 handle local button combinations

When two buttons are pressed down, their values will be repeated continuously. Even though we don't know when two buttons are pressed simultaneously, the buttons are correctly handled, if we "remember" one, we can wait for the other to happen again. Note that this portion of the task requires critical thinking about the problem domain, and is essentially "algorithm discovery" that cannot be easily simplified by merely linguistic means.

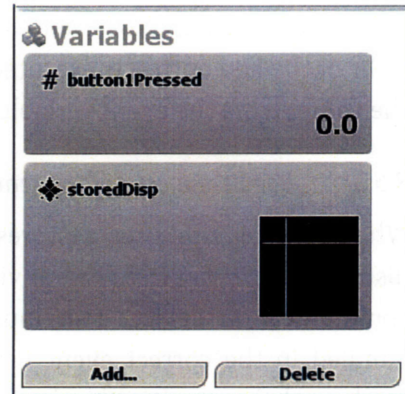


Figure 10 Variables needed for scenario are now added.

First, we need to remember one button press. This requires a variable (a flag). Next, we must use an expression to evaluate when the first button on the first row is pressed, and set the flag. Once the flag is set, if the other button is pressed, action is taken. Since we can't read the LED array in one step, we must send the read message, and then handle storage in the correct handler stack. Storage requires another variable of the grid type. In order to restore, we must send the stored grid to both the LED grid and the communication device. The flag must be cleared when any other button is pressed. The flag management is done using the expression primitive. When the LED grid's state is received, it is copied to storage using the copy primitive. Figure 11 shows the result of the operations. Note that the switch is edited in a popup window with its own per-expression stacks and

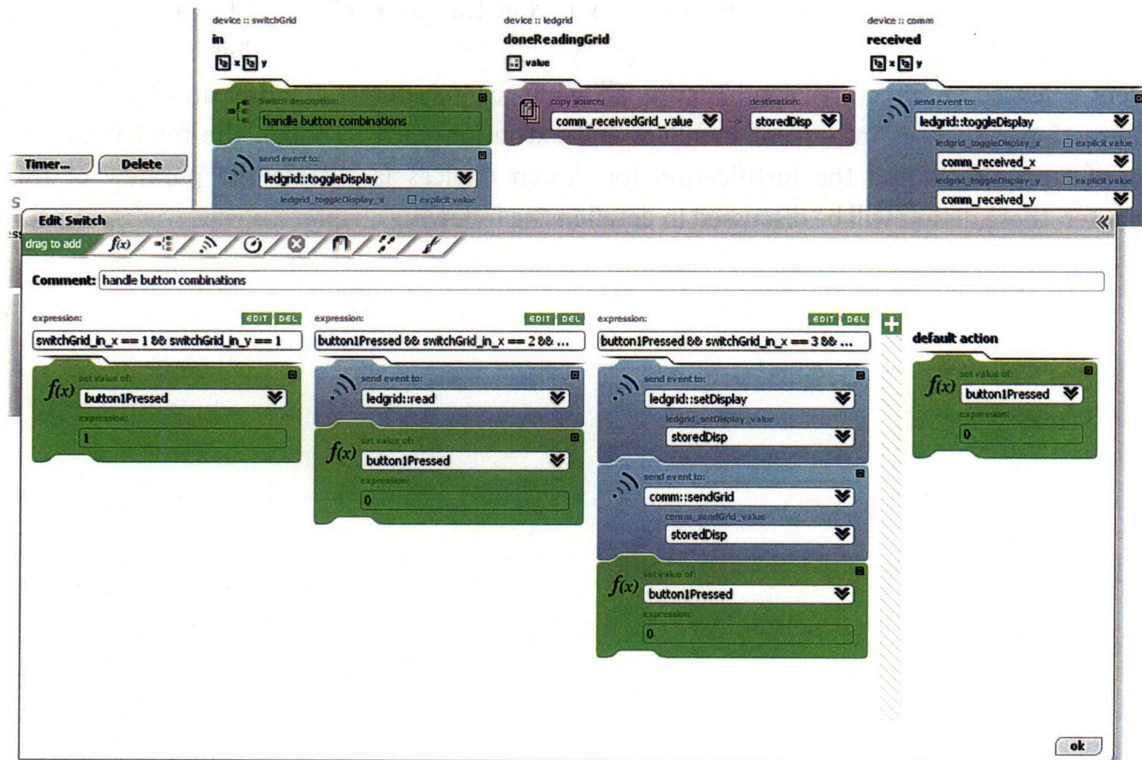


Figure 11 Switch/select editor showing setup for handling local user button combinations for saving and restoring the display.

ribbon. The ribbon metaphor is carried through as the resting place of primitives, while the reason for execution of each stack is as again given at the top. The add button in this case adds another expression to be evaluated. If all expressions fail, the default expression shown on the right of the add button is executed (in this case, to clear the flag). Note that in the expressions, the values from the event variables are used.

3.5.5 handle the case of the remote user restoring the display

When the remote user restores the display, the display state is sent via the communication device. This must be handled in the correct event handler stack, and the resulting grid must be sent to the LED array. The resulting change is shown in Figure 12. Since all the grid sized match, the data can be sent directly to the LED array from the communication device. At this point, the entire task is complete.

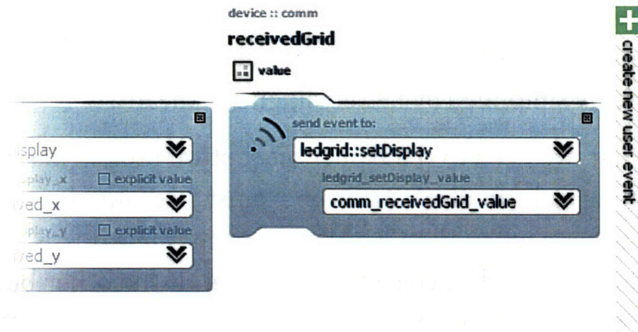


Figure 12 Incoming restoration event is handled

While this task is relatively simple, the replication of the prototype system's behavior provides some insight into the model of mental behavior, or metacognitive model, on which subTextile is based. The design is intended to simplify the initial approach to the problem by providing a clear framework for thinking about a problem from the point of view of the environment causing some change to the system which the system then must react to. This is echoed by the event-oriented architecture of subTextile, allowing for a systematic and clean breakdown of interaction responses into event handlers and execution paths. Though the finer details of specific primitives and the justification for design choices is outside the purview of this chapter, these details will be discussed in detail in the following chapters.

4 design considerations

To preface the following general discussion of the goals and design considerations of subTextile, it is perhaps proper to discuss the state of electronic textiles and how subTextile hopes to change the status quo. Certainly, to discuss the state of things and to design is in effect to proffer opinion. A certain subjective quality thus pervades this chapter. In this chapter, we attempt to state the motivations behind the design choices that will be detailed in the next two chapters.

To begin with a simple premise, the state of electronic textiles has "settled." Conceptual flux, while present, appears to be failing to penetrate the community at large. This is certainly not to be construed as disregard for the excellent work still being done, but it must be admitted that the vista has overflowed with sound-sensitive skirts, "defensive" garments, and parades of models outlined by swirling fogs of LEDs as electronic textiles have both attempted to breach the mainstream and failed to do so. After the initial burst of activity, change has been slow despite relatively high interest from a variety of sources. The reason for this can be stated in terms of the background needed to work in the field. To become a clothier, one can go to school and study the properties of clothes and the intricacies of stitches and design principles. The same cannot be said of electronic textiles. To work with electronic textiles, one must simultaneously be a programmer, an electrical engineer, a materials technician, and still fit in an interest in design. Of course, it is not necessary to be very good at all of these fields, but while it is possible to enter into e-textile, to create behavior-oriented, interactive textiles ultimately requires a deeper understanding of one or more of these fields.

At the same time, electronic textiles, unlike regular textile, requires an unusually high pre-commitment before the results can be seen. It is difficult to see how things look with the parts just pinned in place, because the textile must be whole both physically and electrically. This cripples the feedback loop between the hand and the eye, raising the cost of iteration and increasing the distance between cause and effect. With electronic textile, in short, it is difficult to "just try things out." This break in the feedback loop particularly affects any form of design, be it aesthetic, hardware, or software. All designers must rely on the ability to try out ideas in order to refine them, and this condition remains true regardless of the level of mastery.

Put simply, subTextile is an attempt to reduce the "background requirements" of electronic textiles. Just as languages like Fortran, COBOL, and LISP and others changed not only the way that computers were programmed, but what was even considered possible from the days of machine and assembly language, subTextile is an attempt to change the possibilities open with

electronic textiles regardless of level of skill, though with particular regard to non-technologist users. The hope is that by removing many of the hurdles to ease of learning and rapid prototyping of interactive and complex behaviors for electronic textiles, subTextile can change not only how people develop electronic textiles, but what they do with it. By allowing experts to design hardware devices to suite their needs using common standards for communication, subTextile decouples the behavior from the end-effectors and allows endless variety and creativity in physical implementation. At the same time, by providing a fast and efficient environment for describing behaviors, it allows for rapid prototyping of ideas, thus renewing the connection between tweaks in behavior and their realization in fabric.

In the long term, the goal of subTextile is to encourage development of behaviorally complex electronic textiles in both expressive and interactive domains. That said, even given fairly wide adoption, the chances of these initial iterations of the design being the correct one are low, simply due to the nature of the endeavor. However, the goal is to achieve a sufficient design thoroughness to allow users to experience the positive effects of the system and thus incite discussion which in turn guides the evolution of subTextile not via fiat but rather by feedback. The prototypes, research, and evaluations of the system are therefore explicitly geared towards gaining this level of completeness in the portions tested for this thesis.

With these general considerations in mind, the high level system design is articulated in this chapter. The following chapters discuss the general design goals, hardware prototypes created to probe the topic, and the lessons learned from these prototypes. The results of the probe become relevant to the ensuing chapters on the finer details of the hardware and software components of subTextile.

4.1 designing a toolkit for electronic textiles

One of the interesting aspects of designing a behavior construction tool is that ultimately, the actions taken in fabric, though modified by the medium, essentially can be generalized to any other medium as well. This brings up the obvious question: why call it a toolkit for textiles at all? This question can be addressed in several ways. Firstly, the inspiration for subTextile came from the field of electronic textiles, and the needs seen within it. Working with electronic textiles also provided the insights into the design, especially of the hardware, that is embodied in this thesis. The hardware specification was intentionally created to be extremely open ended, with explicit awareness of the fact that while subTextile has the potential to open doors for behavioral textiles, it also has the potential to closes them. The hardware specification is meant to be open-ended enough to be infinitely extensible without any need for intervention from the author of subTextile. Additionally, the experience in working in the field showed that construction is a large part of the effort, and can easily subsume all other tasks. The modular design was created to allow maximum encapsulation, so that each piece could be tested individually as needed. Additionally, as the hardware evolves, the most specialized modules are expected to be best suited to use with electronic textiles, and the form factors will reflect this.

However, intent alone does not define the function of a product. As with any other human creation, subTextile exists as a conversation between creator and users. As has happened with almost every programming system, toolkit, and utility, subTextile will be appropriated by its users to whatever they see fit as its best use. At times, it will certainly also be used for cases where the fit is ultimately poor. These factors cannot be controlled for via declarations or inner intent. That is not to say, though, that intent is unimportant. A study of existing special-purpose languages clearly shows that the declaration that a language is meant to address a particular need is instrumental in honing the language to the domain, as well as helping to coalesce a community around the language that is capable of holding a much larger volume of experience and knowledge than any single document. This community, in turn, can help to truly explore and guide the path of future developments.

As such, subTextile is by intent and origin, a system for creating behavioral textiles. The language in itself is designed to be as expressive at the behavioral level as lower level systems, and is essentially a high-level hardware control language at heart. The design philosophy, however, prioritizes condition that are more likely to be found in interactive and expressive behavioral textiles. As the goals of the user deviate further from these roots, tasks become progressively harder. The hardware design is likewise inspired localized designs found in electronic textiles. As the language evolves, the hope is that the choices made are flexible enough for longevity, while rigid enough to guide the user towards solutions.

4.2 aspects of the design of subTextile

The primary design goals of subTextile are simplicity of use and clarity of design. This goal is more easily reached in the hardware design than in the design of the language designed to control that hardware, particularly in highly specific contexts. Designing a language is inherently different from other design tasks in that in language design there are very few absolute "right answers" *a priori*. This problem stems from the fact that, even among what may be considered general purpose languages, the choice of emphasis and the expected usage deeply affects the constructs used. Simply put, a language is designed around principles and goals, instead of empirical formulae.

In addition to the language design issues, visual languages introduce also an integral user interface and interaction design component into the fray. While at first glance this may appear to negate the impact of syntax, in fact this simply moves syntax from the literal to the conceptual realm, in that instead of knowing and spelling correctly the commands that make up a language, the user is now required to understand the interconnections between components and the flow of information between them. This requires careful design of data and control flow, a requirement that does not exist in imperative languages due to the implicit flow of control and explicit flow of data within a program.

Lastly, an important part of the overall experience with subTextile is the interaction with the hardware itself. As stated previously, and noted by Berzowska in her analysis of computational expression in general, an artist or designer operates in a closed loop between action and

outcome [1]. While the initial concept is important, it is this feedback loop that allows for mistakes to become brilliant flourishes, and for accidents to morph into inspiration. It is therefore important to maintain this loop and provide as short a path between the creative action and its output in the physical world as possible. This brings into consideration the ideas of emulation and continuous update, and benefits and pitfalls of this approach.

4.3 design goals

4.3.1 interaction design criteria

In designing interactions for subTextile, the intent was to retain a low floor while providing a high ceiling, or in other words, to allow the user to produce meaningful output as quickly and intuitively as possible while allowing ample room for growth of skills, proficiency, and complexity of design. As with any learning exercise, proficiency in the use of software is an investment, and in the long term it is therefore essential to provide a high enough upper bound that the user does not fear outgrowing the language. Traditionally, however, interfaces with high ceilings have generally, though often unintentionally, been at odds with the concept of a low floor. An example of this is Microsoft Word, where the growth of features has led to no less than 20 predefined toolbars and 9 menu trees, disregarding dialogs and modal behaviors [52]. While the goal is no doubt to allow the average office worker to easily use the software, the floor has been raised to the point that it is now commonplace to find certification courses for the use of MS Word. This is caused by two interdependent problems, one more subtle than the other.

At first glance, the problem is caused by the endless addition of features that allows Word to function as everything from a drawing program to a spreadsheet, as long as the user is willing to work around the hodgepodge of functionalities. This suggests a need for careful design and integration of the feature set so that the users can accomplish what they intend, without needless "workarounds" for deficits. For the sake of brevity, I call this "expressional clarity." Expressional clarity demands that the user be able to translate goals into representation as cleanly as possible. If there is a common operation that requires a trick, then that operation should be supported natively as a primitive.

While requiring brevity, expressional clarity also requires that composite operations break down into primitive actions cleanly. As Bonar points out, this clean translation of expectations to syntax is essential for quick acquisition [53]. This is, in some sense, contrary to the hybrid approach that most modern imperative languages take by declaring large foundational libraries of commonly used code snippets. This leaves the user to deal with low-level code in the gaps, thus forming sharp transitions between high level function calls and low-level "glue code." While snippet libraries are needed for general purpose low-level programming, reducing the number of black boxes is certainly more effective in a language designed for beginners [54]. As an added benefit, expressional clarity also lowers the barrier to entry by reducing the initial cost of learning syntax and the foundational library contents.

At a deeper level, the proliferation of features and associated toolbars indicate unplanned "feature creep," or the addition of incremental features that do not integrate into the workflow and purpose of a program. This process is not self-limiting, and eventually leads to the product we see today. Feature creep is a pitfall that is difficult to avoid with a library-oriented approach to managing components. The monolithic construction of libraries discourages interconnected structure required for integration of features. However, by rejecting libraries, changes to subTextile become not changes to an ancillary library, but changes to the language proper. This, along with the principle of expressional clarity, forces cognizance of additions to the language that tend to introduce code-snippet-like features into the language. It also demands that there be a well-defined way of extending the capabilities of the language, as well as a definition of what can be extended and in what ways.

Despite its interface, MS Word does have an implicit advantage in terms of initially perceived and expected functionality. The functionality of a word processor is well known and understood, thus allowing users to immediately grasp at least the basics of the interface like the large typing area immediately. This ensures that a first-time user is never fully lost, and that is an advantage that subTextile does not enjoy, especially in light of the expected audience. As such, it must endure a tenuous introductory period while the user is acquainted to the system. This introductory period must be taken into account in the design of the interaction, thus further putting limits on the complexity initially presented, while opening the gates of the user's imagination with a simple demo or tutorial.

Content aside, it is important for the environment itself to be "clean." When placed in a new environment, whether physical or virtual, one of the first approaches used to assess the conditions is a visual survey. The human perceptual system is well designed to segment and cluster surfaces by complexity, which in turn is ascertained using contrast and the frequency of contrast change [55]. Modern UI's often run afoul of this by using a "nested boxes" approach to display, with a tendency to blindly highlight not only the edges of importance, but also the edges that exist incidentally due to the rectilinear layout. In order to reduce visual complexity, the subTextile interface therefore must use this principle and reduce visual noise and clutter to provide an interface that suggests freedom and simplicity. Admittedly, aesthetic concerns tend to a low priority in much of technical research. A visual language, though, provides no distinction between visual and internal considerations, and thus for the beginning users, it is important to consider all factors in easing adoption.

As mentioned earlier, a large part of the final form of subTextile will be the interaction between the software and hardware platforms. Nominally, the goal is shorten the loop between creative action and its outcome. At the same time, it is necessary to remember that unlike painting or computer-aided modeling, where the relation of action to outcomes is closer to unity, using *subTextile DE* is really a form of remote control, where there is mental dissociation between action and effect. Additionally, the remote control is not singular, in that multiple actions and primitives on *subTextile DE* combine into one conceptual action on the hardware. With this in mind, an on-the-fly, but on-demand update mechanism is desirable. This has the advantage of not transmitting incomplete actions to the hardware device, but at the same time allowing the

user to update very quickly to test micro-changes and experiment with parameters once the action is complete.

A final consideration in interaction design is the design of the primitives and the interaction surrounding them. The goal in designing the interaction is to allow the user to easily manipulate units, while maintaining a strong concept of chaining and self-containment within units. As stated previously, expressional clarity is meant to ensure that each primitive is effectively indivisible as a side effect. This in turn suggests visual and action semantics for primitives that implicitly convey the conceptual qualities of the object. In terms of interactions, the goal is to lean towards well-designed, consistent defaults over configurable options. While the design choices of the author may not be optimal for everyone, practical experience shows that a consistent and effective interaction mode will be successful regardless of optimality, since consistency allows for ease of learning and deduction of functionality. An excellent example of this is CAD programs, where interfaces have traditionally remained very idiosyncratic, with the "goodness" of an interface being based on consistency over *prima fascia* complexity. Consistency allows for a fast learning curve and fulfillment of expectations, both important considerations in allowing a user to gain proficiency and efficacy quickly within and interface. Additionally, emulation was considered as an alternative to immediate feedback. However, upon further consideration, this idea was discarded in favor of the technical design criteria described in section 4.3.2. While emulation has the possibility of shortening the feedback path, the problem remains that it can, at best, only show the electronic components of the electronic textile in isolation, while in the actual case these components often work in unison physically and conceptually. Additionally, the disparity between generic emulation and the actual effect only serves to highlight the discontinuity between the software and hardware environment, while the benefits can be alternately realized with an efficient on-demand, on-the-fly update system as described previously.

4.3.2 technical design criteria

The technical design can be split into two parts, the physical hardware specifications and the *subTextile DE* development environment. While the hardware layer is designed to be as generic as possible, this generality is a liability to the software environment. For the hardware layer, a generic interface specification allows for flexibility in material design, which is the forte of the primary audience of designers with limited technical background. If this were translated directly into the software environment, though, a complex plug-in would be required to depict the unit. Since it is the expectation that the end-designer would be greatly involved in designing the textile components of the system, the onus of creating the plug-in would fall on the designer, adding a complex and ultimately unnecessary step to the development process. Therefore, the goal of the technical design prototypes is to summarize the wide possible variety of input and output systems into a small and convergent set of primitive types that can support open-ended development. This convergence is further supported by the fact that there are relatively few ways for driving and sampling the myriad actuators and sensors that are possible to incorporate into fabric. Ultimately, however, the final level of constraint is provided by the fact that subTextile is not a system which treats textile as a carrier, but rather as an active

composite material that is itself the sensor or actuator. This in the realistic case suggests a level of sparseness and simplicity. Although it is certainly possible to incorporate traditional displays into fabric, it is my belief that such displays are best served by much more general purpose systems than subTextile claims to be.

In terms of software, the goal is to have a system that can operate on multiple OS with minimal changes, and this guides choices of languages and techniques. Fortunately, this requirement is more easily met today than even 5 years ago. In addition, the software must be modular and maintainable, due to the expected lifespan of programming languages relative to most software tools. It is also necessary for the software to be able to interface easily to the hardware components. In order to allow greater flexibility in hardware design, the language needs to be fairly agnostic of the actual hardware. While this is achievable using pluggable back-end modules in much the same way as the GNU Compiler Collection (GCC), it requires an increase in complexity of the internal architecture of the software system. An alternative to this is a virtual machine where the actual hardware is abstracted. With the advent of high speed microcontrollers with substantial on-board RAM and non-volatile storage capacities, it is now conceivable to implement such a system. It should also be noted that behavioral textiles operate not on machine time scales, but on human ones. Therefore, extreme speed can be de-prioritized in relation to other gains. While emulation does require a second level of abstraction, the more limited and yet more high-level nature of the subTextile language lends itself well to this form of abstraction, while simplifying software architecture considerably.

4.4 prototypes

In preparation for the work to be done for this thesis, some hardware prototypes were created in order to explore the needs of the area of behavioral electronic textiles, as well as to gain an understanding of the technologies and capabilities within the field. Two technologically different, but conceptually similar prototypes were created, and the design and programming process was analyzed in order to identify requirements. While books, papers, and journals can provide technical and conceptual guidance, it is also necessary to have an understanding of the practical difficulties of a field in order to be able to successfully craft a solution. Particularly in an exploratory domain, this problem is compounded by the lack of ability among expected users to articulate needs. Therefore it is further necessary to ferret out details and analyze lessons learned from personal, hands-on experiences. In this section, these initial prototypes, the concepts driving the creation of behavioral textiles, and the lessons learned will be discussed.

4.4.1 i/o fabrics

In deciding on the particular form of electronic textiles to investigate, there is a wide range of possibilities, though these possibilities are not equivalent in value as probes into the field. The intent was to choose a prototype that allows for a greater understanding of the intricacies of the goals and requirements for textiles that were designed for expression, and yet supported interaction, which can evolve into behaviors as the complexity of the interaction grows. This

differs from purely expressional uses of electronic fabrics in that it implies an explicit use of input. I classify these set of electronic textiles as I/O fabrics, a terminology meant to separate the use of textiles for practical purposes or for expression from the use of electronic textiles as interaction devices.

An important aspect of this research is the strict definition of behavior used throughout this thesis. Behavior is not defined merely as activity, but activity that involves response to interaction, however indirect. Since interaction is such an important concept within this work, it seemed appropriate to investigate its presence in electronic textiles further. However, the probes conducted for this thesis were not meant to be, in and of themselves, behavioral pieces. While such an accomplishment would have been personally interesting to the author, ultimately the goal was not the creation of individual artifacts. Rather, the goal was to discover hurdles and difficulties in creating relatively simple software and hardware. The relative behavioral simplicity of the prototypes is also an artifact of the author's limited experience with the design of electronics explicitly for the field of textiles prior to embarking on this endeavor. Therefore, the development of a technically complex textile system allowed for maximum exposure to the field before venturing into the creation of a design aid for the field.

Lastly, while the I/O fabrics were incorporated as a technical probe into the field of electronic textiles at a later stage, it also proved to be an interesting avenue of augmenting objects with digital capabilities and behaviors, which has been the interest of the author for some time. This intersection of expression and digital behavior instigated the creation of the subTextile system and provided many insights into the shortcomings of existing electronic textile systems in terms of behaviors and complexity.

4.4.2 use of collocated i/o as bridge towards greater interaction

One of the problems faced with simple interactive systems meant to bring digital behaviors to an existing physical object is the placement of input and output nodes. In general, it is simpler to use "invisible" input systems due to the difficulties in placing traditional input systems on fabrics. Additionally, even if a traditional input device such as a keypad could be placed on an existing artifact, the matching of the input device to the surrounding textile would be difficult, since such electronics continue to be fairly rigid in design. At the same time, if the input device is made invisible in the immediate sense, it is necessary to provide some sort of indication within the design to allow the end user to locate the input device. Though the input device may in fact be completely ambient in its activities, this also adds the additional problem of creating a behavior where the apparent cause is invisible, and thus apparently random. This has interesting repercussions for expressive uses of electronic textiles [1], but does not serve well the purposes of this prototype.

A solution to these issues is the use of collocated input and output, with the input system completely hidden. Since the input and output are collocated, it is possible for the output system to signal the location of the input system without additional help from terms of textile markings, etc. Such a device then becomes inherently interaction oriented, since the output invites the user to interact and provide input. Additionally, sensors and actuators can be layered used to increase the density of I/O assemblies. This also allows for modular materials that mirror the modular design of subTextile hardware by bundling related functions. Lastly, this large scale functionality exposes needs for control structures that can support large numbers of input and output units efficiently within the subTextile language and hardware systems. Collocated I/O therefore provides a specific avenue of electronic textile designs that can be examined using prototypes for valuable lessons in the design of interactive and behavior-oriented textiles. Two prototypes were created in order in order to gain understanding of the needs that subTextile was eventually designed to fulfill.

4.4.3 first prototype: intimate messaging pillow

The first of these prototypes is a set of augmented pillows (Figure 13). The purpose of the pillow was to allow for a low-demand, "ignorable" line of communication and interpersonal awareness that borrowed intimacy and camouflage from the textile aspects. Two pillows can be connected at a distance via the internet, and reflect each other's displays. Interaction is provided via touch, with collocated output. Touching each point toggles its state from on to off, or vice versa. It is also possible to send a number of pre-designed messages by pressing key combinations. Since the pillows can behave as regular throw pillows when off, and can be deformed freely, it is possible for the devices to blend into the environment, and provide an

Figure 13 Pillow created with I/O textiles

ambient means of communication.

The pillows use a collocated array of light emitting diodes (LEDs) and contact switches which can sense pressure. Each pillow has a total of 256 I/O positions at 0.75" intervals. Each pillow is capable of being paired with another pillow over any form of available networking connected to the pillow, and causes the other to mirror its own display. Therefore, a low-fidelity communication channel is opened between the two locations, allowing for a low-demand, background method of communication. By incorporating the design inside a pillow, it is possible to exploit the perception of a pillow as being a soft, inviting, personal artifact that is normally outside the realm of conscious notice. At the same time, it provides a different form of communication than most digital communication channels, such as cell phones, email, and pagers. Unlike these more common systems, it does not demand foreground attention and provides plausible deniability, which suppresses the sender's implicit expectation of receiving immediate attention. A similar project [7] has recently been completed by Philips research that uses the pillow merely as a deformable display for pager messages instead of capitalizing this alternate, more personal and yet less-demanding form of communication.

The internal structure of the pillow can be seen in Figure 14. The pillow uses a flexible LED matrix permanently bonded to a stretch-resistant fabric layer in order to reduce wear on the matrix wires. This layer is placed on the switching layer, which is created out of furniture-grade foam. The foam is perforated by conductive stiffeners which allow the pillow to both deform easily in all dimensions, and support bulk compression or displacement without accidental switching. The aperture in the foam above the stiffener allows an ultra-light steel mesh above the foam to make electric contact with the conductive stiffeners, forming a switch. The mesh itself is supported and kept under tension by the foam, which prevents it from accidentally sagging and contacting the stiffeners when pressure is removed. This design was arrived upon after much experimentation, and created a surface that was soft and yet damage-resistant and produced the minimum number of accidental switching events.

The pillow electronics consists of three modules, which each support a particular purpose. The first module is dedicated to scanning the display at 30 kHz dot rate, and contains the constant

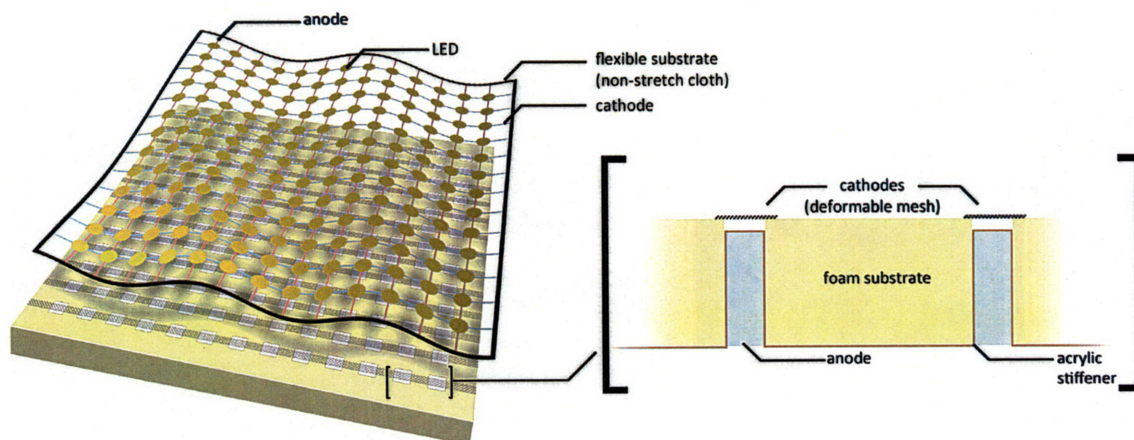


Figure 14 Internal structure of pillow, display and sensing layers

current sources and power management electronics to accommodate the matrix, which can draw pulse currents up to 0.5 amperes in operation. A second module scans the switch matrix and provides de-bounce and communication facilities. The switch sensor module can be connected to an optional network module that can transmit the data via whatever network protocol it chooses. In addition to this, the network module also provides power to the other boards. This design decision was made in anticipation of possible use of Power-over-Ethernet (PoE). In the absence of a network module, a simple crossover cable with a connected power jack can be used to connect nearby pillows. In this case, the power is supplied from a bench power supply. Each module contains its own self-contained microcontroller running code asynchronously. The display and switch modules communicate over serial peripheral interface (SPI), while the switch module communicates with the communication module over a 115.2 kilobaud bidirectional serial interconnect.

The design of the pillow exposed many of the benefits of modular and layered designs both for I/O devices and control electronics. In the original prototypes produced, the electronics were combined, requiring complex layout and code. Performance was poor due to the interleaving of various pieces of code at arbitrary intervals. The high current drain of the LED drive electronics also caused complicated circuit design, since the drivers are easily capable of creating ground bounce (and increase of the ground voltage above zero volts) sufficient to corrupt microcontroller memory and cause it to enter an arbitrary state. With the decoupling of the various segments, not only did the code for each segment become considerably simpler, but each circuit could be designed according to its own needs.

As to be expected of any project at this level of hardware abstraction, the code for the microcontroller is written at a very low level. It is therefore advantageous to limit complexity in order to promote the stability of the code. The idea of modular interconnected units was envisioned in analyzing the code of each of the units. While the pillow does not use a very complex internal network, the ability to divide module firmware into communication and drive sections simplified the program flow. Without the benefits provided by an operating system, this is a boon since this eliminates the need for emulated threading in the code. The primary program loop can be dedicated to the operation of the module, while communications can be hardware accelerated and handled with interrupts.

In terms of interactive portions of the software, the level of complexity is quite low. However, it became immediately clear that the behavioral code was event-oriented, with the main program loop scanning the switch matrix for user input, and then accessing a code path when such an event occurred. The code could be easily and cleanly separated by this criterion, with the behavior acting upon the event with access to a set of variables that could be clearly defined as an interface. Additionally, the actions of the behavior code could be defined as the creation of events such as a notification over SPI to the display and communication module conveying the change. In the final iteration, this was enhanced with the ability to send predefined icons for common messages in response to particular combinations of presses. In this case, the switch module sends a different kind of event (a command) to the other modules to cause the loading of the same icon on the display and the remote pillow. This exemplified

the need for individual modules to support both multiple incoming and outgoing event types. The design of the pillows have been completed, thought further work remains in perfecting the switching matrix and overall robustness.

4.4.4 second prototype: textured wall

The second prototype was designed as an attempt to use an all-textile display surface for a structural rather than mobile use. Textiles have been used since ancient times as wall hangings as well as structural upholstery, and provide a natural elegance to a space. While LEDs as used in the pillow can be visually pleasing, it involves introducing a new visual component that interferes with the nature of the textile in isolation. While this served to highlight activity in the context of the pillow, on a wall the same technique was far too invasive. At the same time, in the architectural context allows considerably more space for electronics and actuators, which in turn allows for more complex actuation scheme which can be used to produce repeatable fold pattern in a "wall" of textile to denote "pixels" (Figure 15). The wall can be used as a large-scale interactive or reactive architecture, capable of acting either as a simple larger-scale version of a single-sided pillow, or as an artistic installation actuated by some other program.

The surface of the wall is made of a highly stretchable textile which can both support numerous deformations and can expand by a large percentage in order to allow adjacent pixels to activate. A pulley system is used to distribute the force evenly along the surface in order to produce a reproducible deformation that folds the cloth along the perforations of the connecting threads. Like the pillow, the wall supports collocated input and output. Switches

Figure 15 Structural wall clad in I/O textiles

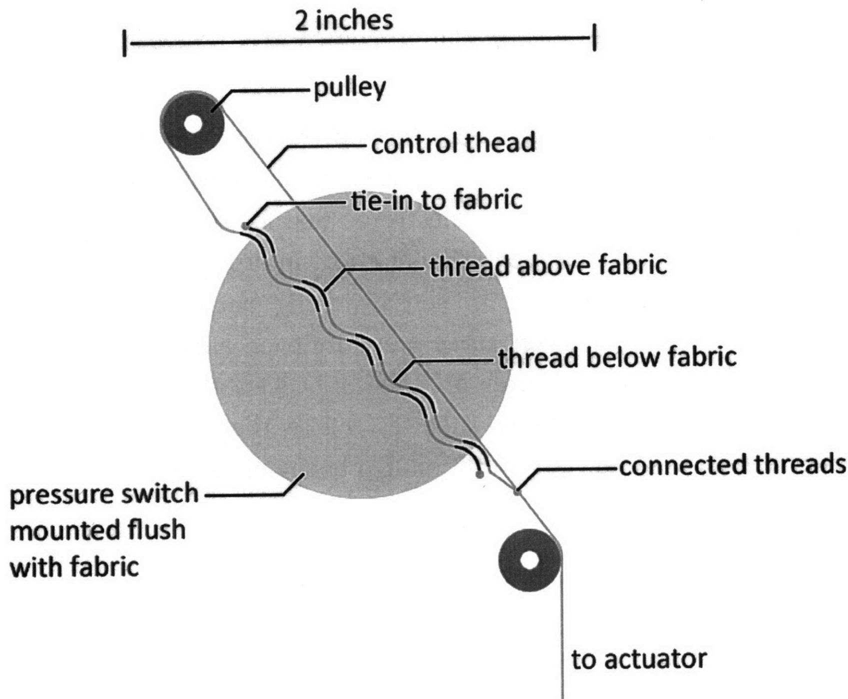


Figure 16 Interior structure of wall

embedded in the backplane of the structure provide input capability. Unlike the pillow, though, the wall is designed with much more demanding driving electronics (solenoid or shape memory alloy actuators) in mind. The use of these actuators, along with an order-of-magnitude less I/O dots, demands a different method of approaching the problem.

The wall prototype is constructed of a single input module and several output modules. The input module can be connected to the same communication module as the pillow in order to provide remote connectivity. Unlike the pillow's display module, which shares its energy source with the other modules completely, the high-current drives use an auxiliary 24 volt power supply to provide current to the actuators. This suggests that the communication criteria among boards needs to include I/O voltage tolerance as opposed to a fixed voltage, since it becomes apparent that not only may some of the boards require an alternate voltage, but each board may also need to operate at a different voltage (such as 3.3 volts) in order to accommodate all parts used.

The output modules are connected to the input/control unit via SPI. Each output device carries a unique ID which allows the input unit to address the boards. However, the inconvenience of programming each board individually with their own unique identifiers suggested the need for an automated discovery which was deterministic in its assignment. When using highly modular electronics, the possibility of having multiple units of the same type also becomes quite high, which requires that it be possible not only to address units, but separate identification from addressing where possible. Lastly, this suggests the need for a bus which is capable of multiple drops (or multiple connections to the same bus) and capable of arbitration among multiple senders. While that particular issue is skirted by the fact that each of the modules is dedicated to only input or only output, the sharing of the bus among modules of many types requires that

all modules be able to share the bus efficiently. The texture wall is currently in a prototype stage, with a minimal number of pixels to test out actuation and control schemes. A larger, more complex version is planned once the design is validated.

Designing, programming, and modifying these prototypes formed the basis for many of the design choices made in the design of subTextile. It was clear from these experiences that even without the hurdles of hardware and software issues, electronic textiles remained a highly challenging field, requiring a great deal of experimentation of effort in order to achieve even relatively simple effects. For example, the design of the soft switching matrix for the pillow required at least four major re-designs, of which three were fully or almost completely implemented, only to be discarded for one reason or another. Given the time cost of experimentation, it is doubly clear that systems such as subTextile are required to compartmentalize the hardware code and allow for prototyping if behaviors are to exist to any real extent in the field of electronic textiles.

5 software environment

At its core, subTextile is a system meant to empower novice users, not only in terms of capability, but also in terms of creativity, and the subTextile language is specifically designed to support this. Before delving into the details of the language, it is instructive to first consider the gamut of approaches taken over the years by language designers to meet these goals. Kelleher and Pausch provide a highly comprehensive analysis of approaches taken to simplify programming, along with case studies for systems that exemplify particular approaches. A summation of their findings is shown in diagrammatic form in Figure 17 and Figure 18. The treelike subdivision of capabilities in Figure 17, though nicely structured, is in fact highly deceiving. In reality, it is nearly impossible to create a language that is a "leaf node," just as it is impossible to create a language that empowers novice users without also taking into account the ability of the user to acquire the language, thus including a teaching dimension to the problem. When the diagram is restated as a dimensional view of the space of programming languages for novices, the structure becomes considerably clearer. As the dimensional view (Figure 18) suggests, the goals of the language are in fact better mapped as biases or escalations along multiple continua. While there are certain to be other dimensions that need to be

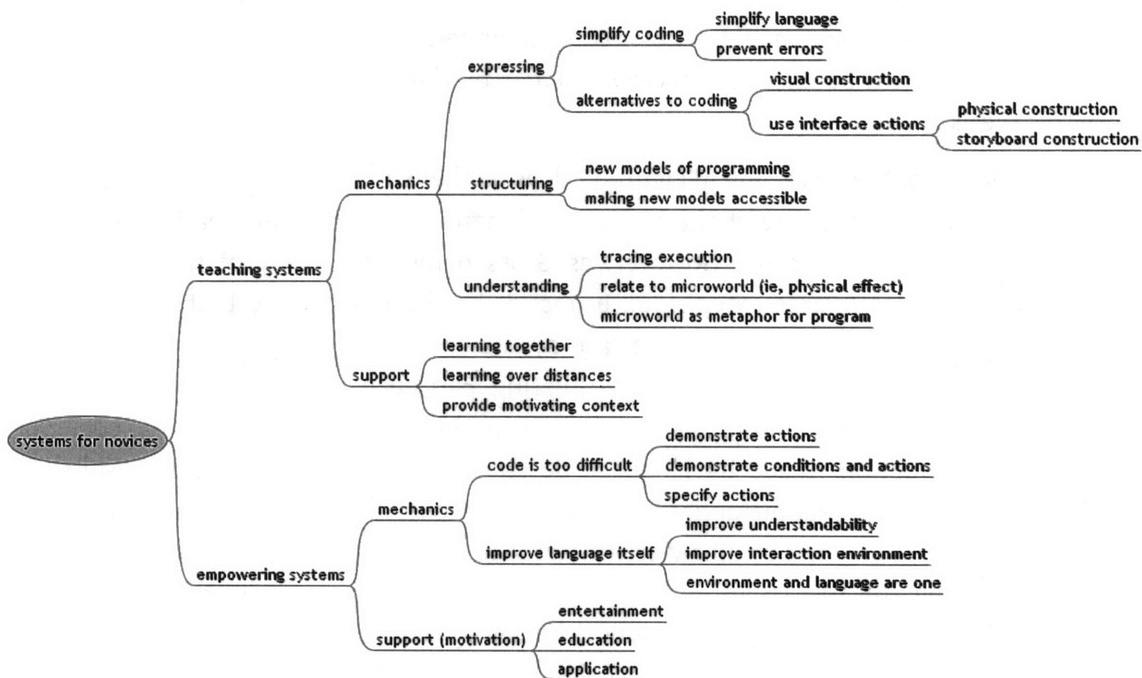


Figure 17 Taxonomy of approaches to programming and behavior creation for novices (summarized from [54])

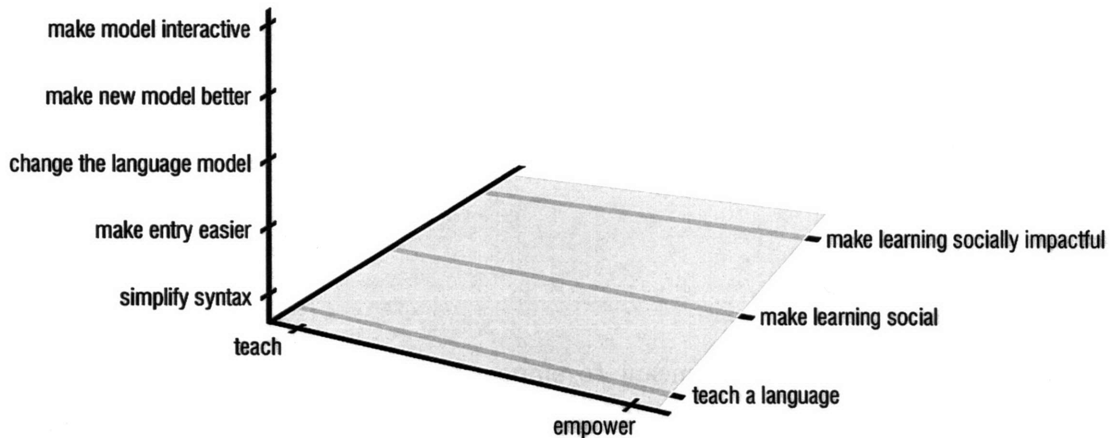


Figure 18 Approaches to programming and behavior creation for novices, reinterpreted as escalation along dimensions (original taxonomy suggested by [54])

considered, at the end of the day the given dimensions hold regardless of the addition of dimensions.

Within the taxonomy of languages designed for novices, subTextile can be easily classified as a system designed to empower users in order to make a social impact via its contributions. Admittedly, it also has the effect of teaching the user how to program (or at the very least, to create behaviors). However, pedagogy is a means towards the end of empowering users who are already proficient in the conceptual realm and allowing them to convert concept into reality. On the other hand, it is not quite as clear where subTextile falls on the vertical axis. While it certainly does define a new language model, it does not yet quite make the language interactive, though that is part of the near-term goals. It is unclear whether the language design makes the language model accessible and understandable to the end user. This particular question with be answered to some extent by the assessments that are described later in this thesis.

While the depiction of the taxonomy in Figure 17 does not help in classifying a language, it does provide a menu of tools that are at the disposal of language designers in creating languages that are accessible to newcomers. However, as Gross points out, many of these tools are themselves unproven to a large extent [56]. In fact, it has been suggested that standard HCI methods for analysis of interfaces and interface dynamics are of limited value when dealing with the wide scope and difficulty in encapsulating particular changes that characterize programming languages [57]. Regardless, progress in being made in analyzing some of metacognitive assumptions made by language designers, with some interesting results. In particular, studies suggest that metaphor, long used as a justification in design choices in visual and textual languages, may not be an aid to the learning of languages [58]. At least part of this may be attributable to the issues outlined by Bonar in examining "bugs" in common programming languages that cause a disconnect between what the user thinks is the case (ie, how matters are handled in natural languages) and what the structural requirements of the

language demand [53]. This disconnect can also cause metaphors to be in some sense harmful to the construction of a novice language, since an immense number of assumptions must be included in the language as "hidden dependencies," and these dependencies must be agreeable to all users of the language, or will themselves prove to be a pitfall [57].

With visual languages, an additional consideration is the visual design of the language itself. In general, each language encountered in reviewing the state of the field approaches this issue in a slightly different way. Burnett et al. suggest a classification system for visual languages, which breaks down the visual depiction into diagrammatic, iconic, and pictorial-sequence types [59]. Dataflow and control flow languages, such as Max/MSP [49], tend to visualize the flow of the primary element, which in their cases in the data that is being managed, and are thus quite diagrammatic. Form-based languages, on the other hand, focus on the static data itself, in most cases hiding the "program" underneath the form structure, and are essentially equivalent to textual languages. Constructivist languages such as LogoBlocks [44] and Scratch [45] have a completely different focus of allowing the user to transfer the knowledge gained in the environment to textual languages, and therefore the focus is on low-level primitives and the way in which parts fit (or do not fit) together to form a program. In general, this class uses an iconic representation. Lastly, in the realm of special-purpose languages, programming by visual example is also quite common, and tend to use depictions based on pictorial sequences of expected effects. This class of languages is closer to Zuf [20] than subTextile in approach and goals.

The goal of this chapter is to describe the details of the subTextile in relation to the overall goals of subTextile. In particular, this chapter goes into much greater detail in terms of language features and reasons for their selection while to some extent ignoring hardware interactions, which are further detailed in the next chapter. The first section describes the technical design goals and inspirations for the subTextile language. The next sections are dedicated to describing the language, the design choices made in the selection of features, and the tradeoffs and motivations for those choices. While all of these sections will tend to discuss the graphical user interface of *subTextile DE* to some extent, the last section is dedicated to the design choices made in terms of the user interface in particular, in light of the inherent intertwining of user interface and language in a visual programming language.

5.1 language goals

In designing a language such as subTextile with very specific goals, it is helpful to have an understanding of the cognitive and design tradeoff dimensions involved. Green et al. present such a set of such dimensions with respect to the usability of visual programming languages [57]. While the work itself is agnostic to the goal and approach dimensions presented by Kelleher [54] and outlined in the introduction of this chapter, it has many of the same base assumptions inherent in the domain of subTextile, and thus provides highly relevant principles for the design. In terms of the psychological and cognitive underpinnings of language, the first and perhaps most important truism is that all notational formats are tradeoffs [57]. Since it is impossible to highlight all information, by definition all notations must highlight some

information and obscure others. For example, the structure of imperative languages highlights instructional and causal flow, while dataflow languages such as Max/MSP highlight the flow of information. As a corollary to this, Green also points out that understanding presupposes a fit between mental model and notation, and that understanding is hampered by the lack of this fit [57]. This is also clearly suggested by the escalations outlined in Kelleher's taxonomy as a increase in the conceptual depth of assistance provided by a language [54]. These maxims require that the choice of primitives and notation within a language be consistent with the mental model that the language wishes to proffer.

In terms of the mental model of programming, programming itself requires the same feedback loop between mental and syntactical representation that art or design requires [1, 57]. A program is essentially a mapping from concept to syntax, and the language must effortlessly support this mapping not only in the forward direction, but also in the reverse direction. Without this reverse translation, it becomes extremely difficult to evaluate the execution of the plan or mental model that the user forms of the solution. The problem is made worse if the language makes it difficult for the user to see the results of an entire conceptual block without extensive searching, or if the language demands that that the blocks be expressed in a particular order [57]. As such, it is extremely important to support out-of-order editing combined with a syntax that matches the conceptual actions the user may be using in order to map the conceptual solution into the syntactic space of the language.

With these considerations in mind, Green proposes a set of cognitive dimensions that should be maximized in order to produce a language that is cognitively optimal. These dimensions are outlined in Table 1. Of course, when applied to real languages, these dimensions need to be traded off in order to meet other constraints. The domain, design constraints, usability concerns, level of complexity, skill of users, and countless other factors affect how well a language can adhere to the dimensions. Regardless, they form a set of ideal measures which can be used as underpinnings for various design decisions in creating a new language.

5.1.1 expressivity

As previously mentioned, subTextile is designed to be as expressive as the imperative C language it replaces as a vehicle of creating behavioral programs for microcontrollers. This goal can be broken down into several different scopes. First, the language must be effect-complete, or be able to carry out the same end effects that C can achieve within the context of programming behavioral textiles. The latter clause is an important caveat, in that subTextile is not designed to flip bits in registers or control hardware interrupts, but for the specific and particular task of dealing with creating programs that have to do with the behavior of a set of modules that *are* programmed in a lower-level language and capable of such low-level control of end actuators. This avoid the hard mental mapping issues encountered in attempting to code behavior in a constrained environment, while providing greater expressiveness and viscosity within the design domain.

Dimension	Description
Abstraction	What is the minimum & maximum level of abstraction? What is encapsulated?
Closeness of mapping	What tricks are needed to get the effects the user envisions?
Consistency	How much of the language can be inferred from partial knowledge?
Diffuseness	How verbose is the language?
Propensity for errors	What parts of the notation are prone to inducing careless mistakes?
Hard mental operations	Are there conditions where the user required aids to map a conceptual operation to language notation?
Hidden dependencies	Are all dependencies indicated?
Premature Commitment	Does the user have to make decisions with incomplete information?
Progressive evaluation	Can a partially completed program be tested?
Role-expressiveness	Can the user see how the parts fit into the whole?
Secondary notation	Can the user annotate the language beyond the notational semantics?
Viscosity	How much effort is required to make changes
Visibility	Can the entire program be viewed simultaneously (on an infinite display)? When the code is dispersed, does the user know in which order to read the code?

Table 1 Usability dimensions for visual languages (summarized from Green et al, [57])

At the same time, it is necessary for the language to be complete in terms of mappings and abstractions. For example, Max/MSP has no operation that can function as a loop, and therefore an internal capability is lacking which the user may want in order to achieve a goal. This does not mean that the same end results are not reachable within the context of the problem that Max/MSP deals with. However, it requires resorting to tricks when the required action is in fact a loop. Lastly, a language may need to be complete in terms of extensibility of core features. In this sense, Max/MSP and C provide examples of the extremes. Max/MSP provides only the very lowest level infrastructure and leaves all primitives to be created as extensions, while C has a strongly defined set of core features and syntax, with libraries that build upon it without affecting the core features themselves. The extension facilities need to be carefully managed when a language is based around extension (as hardware-oriented languages need to be at some level) in order to maintain coherence.

While subTextile aims to be as effect-complete as possible, the determination of whether it needs to in fact be operations-complete is more indistinct. SubTextile is not meant to become a generic language, and thus choices were made based on its domain. Along with these choices comes the inevitable outcome that some operations become more fluent and simple, while others are made relatively more complicated in order to provide the appropriate level of abstraction. The choices in design are made in order to present a coherent language model and promote best practices for the language. At the same time, the design explicitly attempts to avoid over-abstracting and adding functions which do the same action in multiple ways, instead opting for clear solutions to problem which do not burden the user with selection of syntax.

In terms of extensibility, the intent is to not require the end user to write complex extensions to extend core functionality, while still allowing the user to use a variety of devices, whether custom-made or not, seamlessly. This suggests the need for a strong core language that is lean

and yet capable of dealing with a wide variety of scenarios. The extension mechanism must also be sufficiently simple so that the hardware designer does not need to delve into the innards of subTextile and its internal APIs in order to be able to support new devices. In the initial mockup pilot tests of subTextile, it also became quite clear that the ability to modify the name of devices and variables was quite important. Since generic modules can oftentimes be used for wildly different purposes, and the ambiguity in names poses a significant hurdle to the novice user in terms of ability to reason quickly and efficiently about the problem. Concepts such as abstraction (ie, a thing is not what it says it is) have seldom, in the experience of the author in working with artist, been well received by them. Indeed, such abstraction is often contrary to the physically-situated realities in which these professions function, a fact that has been noted to the author multiple times by different pilot subjects. It is therefore ultimately necessary to natively support the ability to abstract naming from low-level information that allows the language to function in order to achieve the higher-level goal of simplicity and clarity of expression that subTextile strives for.

5.1.2 *expectation management*

When discussing a language with a low floor, or one meant to simplify development in a particular domain, it is common to confound simplification with simplicity. Indeed, most languages dealing with complex domains (such as Max/MSP or Squeak) are in fact complex languages that do have a learning curve. The goal is to reduce the learning curve, though it cannot be completely eliminated. On the other hand, there is considerable room for improvement in simplifying the development of the application types that the language is geared towards by using methodology well-suited to the task at hand and operations that complement the effects desired. That said, the goal with subTextile is to have as much simplicity as possible while simplifying development tasks. This simplicity can be achieved, for example, by streamlining variable types and primitives in order to reduce ambiguities and remove overlap. The outcome of this streamlining can then be supported with appropriate training to overcome the initial cost of using the language. Lastly, it is possible to portray simplicity regardless of the deep-level complexities of the language. This initial simplicity helps to create a high ceiling so that experts can create more complex behaviors without excluding the neophyte.

Aside from expectations of the language itself, expectations inside the user interface are also an important consideration. With a textual language, these expectations normally do not surface in dialogue, since the expectations are essentially conceptual. For example, a user may perhaps wish for simpler syntax within makefiles, or a particular way of managing code in a development environment. However, with visual languages, the possibility exists to articulate, understand, and manage these expectations about the interface integrally with the language instead of as a separate case. This can, for example, be used to reduce mistakes by highlighting particular blocks or constructs by semantic importance within a layout, or prevent the use of incorrect types. At the same time, this exposes a mapping problem where it is possible to write code which attempts to outsmart the user. In the particular domain that subTextile exists, it is in fact particularly important to prevent the user from feeing powerless, since oftentimes the

user will already be quite insecure about his or her abilities. With careful design, the visual nature can be exploited to achieve this balance through the integration of language and UI.

5.1.3 complexity management

Last but not least are the considerations with respect to complexity. As stated in the previous subsection, it is not logical to claim the ability to reduce complexity arbitrarily, and within the domain there exists a theoretical lower limit on simplification beyond which the expressive capabilities of the system are compromised for simplicity. Complexity for the particular case of subTextile can be broken down into syntax complexity and perceptual complexity. As discussed in the previous sections, the syntax complexity is a natural part of the requirements of learning the language, and can be attacked by minimizing and honing primitives. While this will not eliminate syntax, within visual language it can provide reasonable containment.

Perceptual complexity, on the other hand, stems from several factors in textual languages, including but not limited to code nesting, intricacies of the execution path, and variable scope. Since the user has to perceive and envision these factors in order to be able to understand the functions of a program, complexity in these areas negatively affect the efficiency with which the user can understand or formulate solutions. Most of these issues apply to visual languages as well in the general sense, though most visual languages tend to flatten one of more of the domains in order to achieve a simpler, more homogeneous visual representation. As suggested earlier, it is possible to directly visualize these problems, though in practical experience these techniques provide limited assistance since the visualizations themselves becomes as complex as the problem cases. With this in mind, the goal for subTextile is to create a visualization of the language that neatly encapsulates the complex segments and provides a method of exploring these segments in a uniform and intuitive way.

5.2 language description

5.2.1 event handling

SubTextile is an event-oriented language, and thus the only way for execution to be initiated in subTextile is via an event. Therefore, a large portion of the user interface and dynamics is dedicated to the management and processing of events. Events can come from one of three sources. They may be externally triggered events caused by devices connected to the main board, they may originate from a user-created timer, or they may be user-created. Of these three types, only the external and timer events can begin processing. The user events are used in a manner similar to invocations of functions in traditional languages, in order to allow code reuse and encapsulation. External events, in general, are the result of some change in a sensor reading or other change of state in a device, while user-created events serve more as function calls, aggregating behavior that is executed as a result of multiple incoming hardware events. User-created events allow for code reuse within the event framework. It should also be noted that device events can be "synthesized" to establish equivalence between various events.

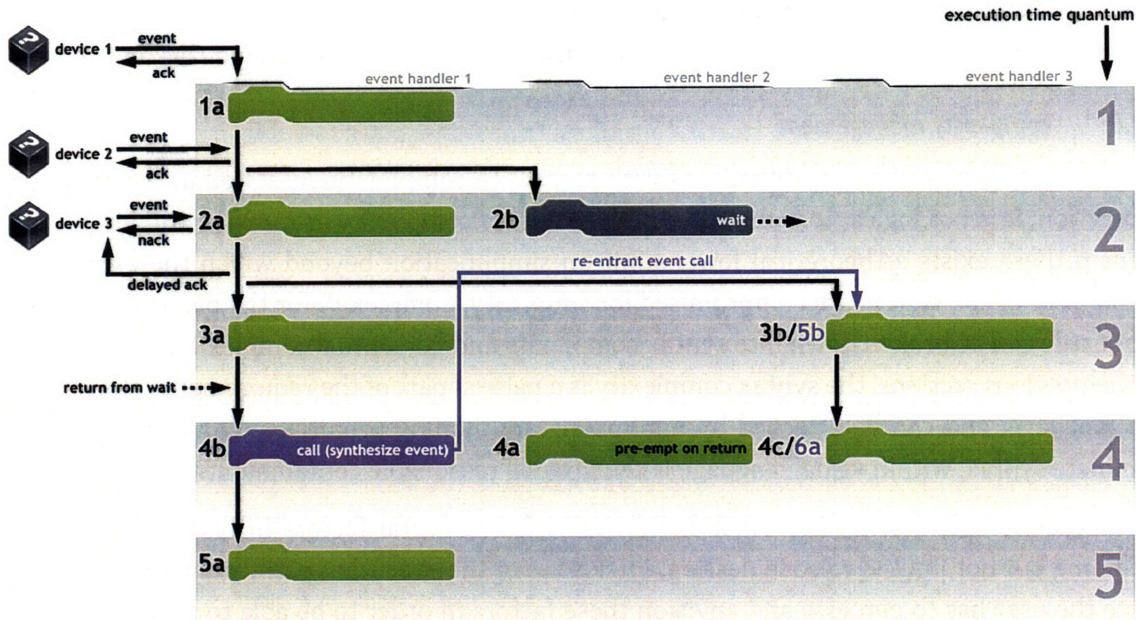


Figure 19 Thread execution and external event interaction in the subTextile visual language

All events exist in a flat top-level namespace, and can be triggered via software. The existence of an input event is signified by the presence of an event stack in the event area of the UI. The stack is identified by its source (ie, device name, timer name, etc), the name of the event, and the list of values passed in during an event. Each event can carry an arbitrary number of variables with it. These variables created by the execution of events are pre-allocated and exist in a top-level global namespace. This is in fact equivalent to persistence of state. Once the system has declared its state via an event (before which that state is accessible but undefined), the state is remembered and made accessible without any further effort towards storage or querying. It should be noted that user events cannot have any variables attached to them. However, this is in fact not an issue, since the variables from the event source are defined. It is also important to note the difference between an event handler and a callback. Unlike a callback function, which operates like a normal function called outside the normal flow of a program, activation of an event handler is a one-way operation. Once activated, the event handler essentially operates in a vacuum and in parallel with other handlers, and does not provide any indication of completion *per se*. Also, since event handlers are essentially like threads in most other programming language, the semantics accommodate multiple executions of the same code path at the same time.

The event handling model for subTextile, more conveniently thought of as a threading model for the rest of this subsection, is atomic and fully synchronous. Unlike languages such as Java or C that require intricate locking methodologies for managing thread interactions, subTextile automatically guarantees that all state modifications will happen synchronously. The interaction between external events, thread processing, and timing is shown in Figure 19. As indicated in the figure, all primitives in subTextile are atomic. While devices do operate asynchronously, this is handled at a hardware level by "locking" the data bus for input, but

deferring input acceptance until the current event is complete. If the current primitive requires access to the bus, time is "turned back" in order to free the bus first, and the event is re-executed after the bus is serviced. This requirement is necessary since the state variables created by events exist in the global namespace which can also be modified by an incoming event, and thus poses a danger to unsynchronized operations taking place at the same time that may use the same variables.

Since there is no stack associated with events, the scheduler simply maintains a set of index pointers into the bytecode that specify the location of the next execution point. This essentially allows for zero-cost event firing, since the event queue is pre-allocated. In addition, event triggers appearing at the end of an event handler are scheduled in $O(1)$, allowing the use of events as loops. Additionally, these semantics make the creation of arbitrarily long and complex state machines simple and intuitive. Unlike many languages used for the purpose of creating behavior for electronic textiles, the subTextile language does not have implicit return semantics (ie, transitions in the program's state machine are by default unidirectional), thus removing the need to keep track of states or return paths. This has the effect of allowing the user to treat the program itself as the state machine, instead of emulating a state machine with the program. This task-oriented approach is designed to further simplify complex behavior development within subTextile.

The scheduling quantum in subTextile is therefore of variable length, but always encompasses the execution of one primate operation from each event handler currently active. The quantum may be extended by the arrival of external events, which are handled between the atomic executions of primitives on an as-needed basis. The response time for events in this model is effectively the same as response time in a cooperatively multithreaded environment. In particular, it is possible for user expressions to take considerable time to execute. Therefore, there are no real-time guarantees on timer and sleep statements, which can only cause the queuing of addition threads to be serviced. However, in practice this has not proven to be a major problem for behavioral segments of code. Since the subTextile model separates effectors and sensors from the behavior-oriented main board, it is unlikely to affect the functionality of the system as a whole in any case. With exception of wait statements, all other events are treated equivalently. Wait statements are always scheduled as the next event to be processed since they are considered "altruistic" in terms of processing time consumption. The performance of wait statements is considerably improved by this scheduling method, since their execution is only deferred to the end of the current atomic operation, instead of the start of the next scheduling quantum. Except for the exceptions described above, all other new events are added to the end of the event queue, thus ensuring that their first instructions will be executed within the same execution frame. Completed handlers are marked dead if necessary during the processing time quantum.

Due to the constraints of microcontroller environments, it is necessary to use $O(n)$ techniques in manipulating the scheduler queue. Dead handlers are discarded at the end of each time quantum to reduce processing overhead. While the hardware specification does not limit the number of concurrently-executing handlers, in the general case it can be expected that there is

a limit imposed due to the cost of supporting an arbitrary number of handlers in a microcontroller environment. Additionally, it is expected that the performance will drop below acceptable limits long before the handler limit is reached. For the sake of processor efficiency, subTextile does not provide facilities for detecting these conditions, though they would be fairly apparent in physical debugging. Despite these limitations, the event oriented nature of subTextile allows it fluent access to capabilities without requiring additional effort.

5.2.2 *data types*

SubTextile uses two primitive and one hybrid data type. The first primitive type is a scalar value, and is represented by either an integer or floating point value. Floating point values are available for convenience. While the floating point functions can be emulated on a microcontroller, the performance is extremely slow, and their use is not recommended. For most normal mathematical operations, integer-only variants are available. The second primitive type is a two-dimensional array. The size of the array must be declared in advance, and cannot be changed programmatically. The primary purpose of the grid type is to hold a large number of preset values in a concise form. However, they can be used as arrays to store calculation results from expressions as well.

The only hybrid type available to subTextile is the graph type. The graph is a linear array of values accompanied by a cursor. The cursor can be manipulated programmatically to point at any valid position. In addition, the value at the cursor can be read back as a scalar. The purpose of the graph type is to simplify animations. Instead of writing loop-like structures, the user is free to create a graph of the animation that is automatically interpolated smoothly. The availability of the graph type not only allows for easy animations, but the flexible programmatic capability to manipulate the cursor allows the user to support a interactive behaviors that can be easily tweaked using methods and an environment that is simple and familiar to the user from exposure to drawing programs in the desktop world.

As stated previously, all variables in subTextile occupy the same global scope, including state variables introduced by incoming events. This is a natural side effect of using stack-less event handlers, which in turn leaves no place for scope to reside. The high-level nature of subTextile precludes the use of most variables that one would normally consider as candidates for use in local scope, such as loop variables. With subTextile's high-level primitives, variables are meant to be used as overall program state, and as such the global scope is a good match for the expected usage pattern. It should also be noted that subTextile does not use named constants, though the user is free to treat any variable as a constant. The graph type is essentially constant within the program, but only in that the actual graph cannot be modified. This functionality can be easily duplicated using the grid type. For memory efficiency, static analysis is used to tag grids that are written to, and these grids are copied to RAM. Grids which are not written internally become constants and are not copied to RAM.

5.2.3 *choice of operations*

This subsection describes the capabilities of all primitives supported by subTextile, before delving into the particular peculiarities of the more complicated primitives. The primitives can be roughly subdivided into expression, message, flow control, and variable manipulation types, and the types will be noted after the names of each primitive. The parameters for each primitive

Operation: ***Expression***

Description: Evaluates freeform expressions

Parameters: Expression string

Notes: Details of the supported functions in expressions and syntax is discussed separately in section 5.2.4. The freeform expression operates on all existing variables. The expression is pre-parsed and validated at compile-time.

Operation: ***Switch/select***

Description: Evaluates a series of freeform expressions and interprets the results as boolean values. If the expression is a boolean truth, then the corresponding primitive stack are executed. If none of the expressions match, the default primitive stack (if present) is executed.

Parameters: Expression strings, primitives

Notes: This primitive can be used as a switch statement (as seen in C, Java, etc) or as an if/else-if/else statement. In practice, it resembles the latter to a greater extent. It has the effect of selecting a primitive stack and inserting that stack into the handler stack at its own position. Depending on size, this primitive can be quite expensive, since the selection of primitive stack (and therefore the evaluation of expressions) takes place in the scheduling quantum of in which the primitive is invoked. More than one expression may be true at a given call, but the first expression to evaluate to true is used. The expression may be either a boolean or scalar expression. If the expression produces a scalar, zero is interpreted as false and all other values as true.

Operation: ***Message***

Description: Causes the firing of an event

Parameters: Event to fire, as well as parameters. Parameters may be variables or constant values.

Notes: If the event is an outgoing event, the parameters need to be defined. If the event is internal, the parameters are assumed regardless of whether the event is user-created or device-generated. The message primitive does not surrender its execution context like a function call in an imperative language. The operation following the message primitive is executed in the next scheduling quantum. As previously noted, an internal exception is made if this is the last primitive in a handler, but this does not affect the perceived effect.

Operation: ***Stop***

Description: Stops the evaluation of an event handler
Parameters: none
Notes: The stop primitive only affects the hander it is in. Since there is no caller or return semantics in subTextile, this is the only logical option.

Operation: ***Wait***
Description: Waits for the given number of milliseconds
Parameters: Variable or constant (selectable)
Notes: The wait statement causes the current event to give up execution. Handler stacks returning from waits are given increased priority to provide better timing.

Operation: ***Copy***
Description: Copies one variable to another
Parameters: Two variables. Typing is forward-enforced, allowing graphs to be treated as scalars for assignment to scalars, but requiring that the source be a graph if the destination is a graph.
Notes: The function of this primitive can be emulated for scalar values by the expression primitive. However, since the function is sufficiently common, a dedicated primitive is provided. For graph types, the primitive copies the values from one graph to another. This is the primary function of the primitive, since doing so by emulating a loop is not an encouraged practice. For graph types, the primitive causes the destination graph to have the same cursor value as the source.

Operation: ***Step graph***
Description: Moved the graph cursor in the specified manner
Parameters: Specify how the cursor should be moved. If required, a variable to set the value from or a user-specified constant value is prompted for.
Notes: The step function allows forward and backward wraparound cursor manipulation. The cursor can also be set to a specific value specified by either a variable or but with a user-specified constant number.

Operation: ***Paint Grid***
Description: Programmatically paint a location on a grid to a particular value
Parameters: The grid to paint, x and y coordinate, and new value. All can be specified as user-supplied constant values, or as variables that evaluate to scalars (scalar and graph type).
Notes: The use of this primitive causes the grid to become internally "writable" and copies it to RAM. This function allows the use of a grid as a 2-dimensional array for various array-based storage needs.

5.2.4 freeform expressions

In subTextile, freeform expressions were chosen over fully-visual layout for both switch/select operations and for the expression primitive. The purpose of freeform expressions is

constrained to the modification of a scalar variable, which is specified separately in order to further reinforce the constraint. This is departure from a completely visual environment, in that there is relatively little visual layout assistance provided to the user in creating these expressions, and marks an intermediate path between the approaches taken by fully-visual languages such as scratch and visual layout languages such as Max/MSP. The choice of freeform expressions is perhaps the most significant and powerful design tradeoff in the design of subTextile.

The power of freeform expressions is that mathematical formulae can be entered in a familiar form without additional work imposed by creating an equivalent syntax tree in the way that Scratch demands. This allows for complex effects to be calculated on the fly without devolving the language into scripting. In addition, the use of freeform expressions prevents the clutter of low-level operators from entering the language, which keeps the primitive operators simple and elegant. However, in terms of *prima fascia* perceived complexity, these freeform units are perhaps the most challenging to the user. To some extent, the complexity is mitigated by using a special editor that assists the user in selecting variable names and operations. Additionally, as the user gains experience in the environment, these expressions gain the advantage of speed and simplicity over visual expression assembly techniques, which can be difficult to manipulate rapidly, and in general takes longer to transcribe from the common written forms. In addition, as the expression gains in complexity, visual layout gains sufficient complexity to overcome the initial gain in construction ease due to visual verbosity caused by the tree-like growth pattern inherent to closed-form expressions. Therefore, in the long run, function-constrained freeform expressions provide a reasonable compromise in ease of construction and comprehension.

In addition to perceived complexity, freeform expressions also have certain other caveats. Perhaps most importantly, syntax errors in freeform expressions cannot be prevented preemptively as is done in other subTextile primitives. While other primitives can use filtering to prevent syntactical errors completely, expressions must be checked for correctness. This problem can be mitigated with just-in-time checking, but compile-time checking is also necessary in order to find errors caused by the deletion of variables after the creation of the expression. Additionally, the evaluation of freeform expressions is an expensive task. In the interpreted environment, it is necessary to pre-expand the expressions and evaluate the expression is RPN form for efficiency. While more optimal solutions are possible, the interpreted nature of subTextile demands a general solution. The evaluation may therefore take considerable time to perform relative to most other functions, and the user needs to cognizant of the issue and break apart operations as needed to compensate. Fortunately, this is expected to be an issue with advanced users who will most likely have a reasonable understanding of the language at that point.

5.3 language design choices

SubTextile borrows features from imperative, dataflow, and visual languages alike in order to provide these features within one body. The dataflow aspect comes to fore in the event-oriented design, which allows the processing to be interactive instead of busy-loop based. In

analyzing the software of the design prototype, it became clear that most of the effort was spent in designing proper interleaving of actions and in gathering of data from input devices. By automating these aspects of programming behavioral textiles, the complexity of the boilerplate code is minimized or removed, leaving the end-user to focus on the behavior of the system. The visual layout of the primitives and primitive stacks, on the other hand, allows the user to quickly understand the program flow, again with a bias towards interaction and input rather than on synthetic constructs such as classes or functions. Lastly, the freeform expressions borrowed from imperative languages allows for powerful variable manipulation without the time-consuming ritual of decomposing the common algebraic forms into visual layouts. Each of these features has been selected for its demonstrable efficiency and power, and is sometimes balanced by tradeoffs inherent to the techniques used. However, many of the design choices were also made after considering the interaction of the various portions of the language with each other, which led to changes as the language evolved into its present form.

In the preliminary design, subTextile was designed with handler stacks that were populated with entities similar to functions. This model suggested a one-to-many sequential distribution of events where the handler stacks were processed somewhat like callback functions. However, in later analysis, it became clear that in light the event generation semantics, this approach added additional overhead both in terms of interface and language semantics, without a corresponding improvement in the expressivity of the language. Since these handlers acted as functions (which were not used elsewhere in the language), they were essentially an anomaly that disrupted the consistency of the language. Additionally, it required that each handler be housed in its own popup editor, which significantly reduced the user's ability to "skim" a simple program, since much interaction was needed to delve into each handler and understand its actions. The current model uses a single handler stack which contains primitives instead of pseudo-callback functions, and allows for quick scanning without reducing the power of the language. Additionally, the stop function could be simplified by this choice. In the previous model, the stop function could either stop the current callback, of all processing for that event. In the new model, it needs only to stop the event processing, thus simplifying flow analysis for the user.

A similar choice was made with the switch/select primitive. Originally, separate if/then and switch statements were planned. However, it became clear that in most cases, the most useful clause was in fact and if/else-if/else clause. The implementation penalizes switch statements (which are the least commonly used form) in favor of this more common form by requiring an expression to be computer for each choice. In the end, the decision was made to support the non-optimized form in order to both maintain consistency and provide a simpler, more uniform solution. This choice also improves the legibility of the switch/select primitive over traditional switch statements by explicitly stating the conditions under which an expression stack is evaluated rather than requiring the user to recall the half of the expression that forms a switch clause. As with each of these design choices, it is important to note that the choices made were geared towards the goals that subTextile subscribes to, and the particular granular choices were made based on secondary goals of simplicity and expressional clarity within the

language. It is therefore unlikely that the choices hold in the general context beyond the reasoning that prompted them.

5.4 ui design choices

The UI for the subTextile development environment is designed not only to ease development of behavioral programs, but to limit the types of errors possible as much as possible. In the case of a fully-graphical language such as Scratch, it is possible to remove syntactical errors altogether, thus limiting the types of errors to purely logical errors, which in general can only be discouraged, but not altogether eliminated using visual constructs. Since subTextile uses imperative elements in some elements, it is not possible to completely prevent syntactic errors. As explained in Section 5.2.4, errors in the free-form expressions are constrained in scope by design. In addition, the UI provides the user with features that assist in creating robust expressions. Additionally, in-place and compile-time checking is done in order to ensure error-free bytecode. It should be noted that while visual construction cannot prevent logical errors (ie, failure to handle a particular case or event), the dataflow features of subTextile are in fact quite robust against such errors. The event-oriented design makes certain that the user addresses or is at least cognizant of all events and cases that need to be handled. This limits the possible errors to errors of intent or of conceptual understanding, where the user's mental model of the actions taken by the language is not congruent with the actual actions. Issues in this domain can only by tested using experimental techniques, and the details of this testing is available in the evaluation chapter.

In general, the only syntax errors in subTextile originate from the possible improper use of variables. In order to prevent such errors, combo boxes containing matching types are provided wherever possible. Where explicit input of value is required, the value is validated in-place in order to reduce errors. Additionally, a centralized model is used to guarantee that any changes or deletions to the variables list are propagated immediately. If a variable in use within the program is deleted, the change is propagated to all primitives using the variable. Additionally, instead of selecting an arbitrary value (which can easily lead to logical errors), a null selection is set in order to force an error during compilation. In order to further enhance robustness, variable types are set at the time of creation, and become immutable from that point. This prevents an additional class of errors and simplifies the language's handling of variable types.

In order to improve coherence within the *subTextile DE*, the choice was made to create a single-window tool with the ability to have multiple internal children windows. This model was chosen over having multiple windows due to the inherent problems in keeping multiple windows in focus in most normal window managers or desktop environments, which is necessary to support drag-and-drop between editor windows used to edit switch/select statements. Additionally, it allows the user to focus on the work being done by allowing more fine grained control of window positioning and focus behavior than is possible using the external window manager features alone.

Lastly, though the windows and UI elements in subTextile are a part of the user interface, they also are part of the language itself. In short, unlike textual languages, visual languages do not have an alternate form *per se*. In addition to preventing user errors, it is therefore part of the design goal of the UI elements to reduce the possibility of inconsistencies and errors in language and compilation backend. To this end, subTextile does not treat the visualization as a rendering of an invisible data structure, but instead treats the visual elements as the data structures themselves. This is in contrast to the model-view-controller (MVC) design pattern where the representation and the visualization (model and view) are distinct. The two-phase update of model and view is replaced instead with a hierarchy based approach, where visualization is largely offloaded to an abstract base class, while individual primitives focus on the handling of their own parameters and editor setup. This setup has the advantage of discouraging errors in the code of the language itself, thus helping ensure that the platform itself is consistent and error-free.

6 physical layer guidelines

The physical layer guidelines for subTextile are meant to provide sufficient guidance to an implementer to allow them to reproduce the devices described on arbitrary architectures (within the limitations of the design). The hardware specification supports communication of all subTextile data types and event blocks, and incorporates features that allow the software to operate efficiently at the protocol level. Additionally, it incorporates details of the virtual machine and suggests implementation specifics that retain the execution characteristics of subTextile and allow for easy reproduction of the subTextile environment on any platform from a regular PC to a reasonably complete microcontroller. While physical boards were not implemented for the work presented here, support is planned in the very near term. Once this work is complete, a master node, as well a variety of sensor and actuator nodes will be available for validation, testing, and to serve as examples for building other types of nodes.

While the communication system used for subTextile uses a multi-master daisy chain network topology, the logical layer does in fact have a master unit. This topology was chosen in order to minimize difficulties encountered in achieving connectivity in textile applications, where the concealment of still electronic units as well as the creation of durable interconnects is difficult to say the least. Additionally, the interconnect was chosen in order to minimize the number of cables required, while still producing deterministic addressing behavior that allows for the use of generic nodes. The purpose of the master unit, whose behavior has been described to some extent in the preceding chapters, is to interact with the *subTextile DE*, run the virtual machine that executes the subTextile bytecode, and manage address allocation on the interconnect bus. The master node is designed to encapsulate the software complexity of the subTextile hardware, thus allowing for other nodes to be implemented with much simpler hardware, since they need only maintain a minimal amount of state in order to interact with the master node.

Since the master node must have the capability to connect in practice to most mainstream systems, the communication to the master node must be uniform both on the computer and the microcontroller side. In order to accommodate this restriction, the *subTextile DE* uses serial communication to talk to an attached master node. The advantage of the serial connection is that serial connections can be emulated over USB easily. Several single-chip solutions encapsulating the entire USB protocol stack and providing a serial interface on both the computer and MCU side are now available, and can be used in place of a traditional serial cable connection. The data rates offered by serial connections are ample for the uploading of programs to the master node, while the protocol itself is well documented and supported. It is necessary for the microcontroller to support some method of self-programming or other non-

volatile storage in order to retain the subTextile program. Oftentimes, this can be handled by writing to flash located on the microcontroller itself, though another option is to utilize an external EEPROM or flash memory module to store the data.

The hardware guidelines presented in this chapter are intended to be agnostic to the selection of MCU, though the Atmel AVR series of microcontroller unit (MCU) is referenced for platform-agnostic details, in large part due to the author's familiarity with the platform. Additionally, certain features of the architecture are necessary in order to allow the system to operate efficiently. Since subTextile is not directly executed as binary code on the MCU, proper operation of the system requires that the subTextile main processing loop have a reasonable share of the processing time, rendering polled communication solutions untenable. There is also an additional memory cost associated with the use of the interpreter, and this should be considered in the selection of the MCU. Fortunately, it is now fairly common to find low-power, low profile, high speed MCUs that carry sizable amounts of RAM and flash memory on-board. Most of these units can operate with nothing more than an additional decoupling capacitor, though for maximum speed most will require a crystal or ceramic resonator and associated parallel capacitors.

6.1 communication

6.1.1 i/o configuration

Communication is an integral part of the way that subTextile operates, and is thus a requirement for any subTextile hardware node. All devices are connected to each other using a five-wire bidirectional interconnect. The interconnect carries communication, synchronization, and power cables. The maximum number of devices on the bus is theoretically limited to 127 devices due to limitation in addressing. An additional limit is placed by the maximum allowable bus capacitance, which makes for a practical maximum much lower than the theoretical addressing limits. It is possible to create repeaters to extend the limitations in addressing and bus capacitance, though it may be more prudent to support additional bus channels on the master node.

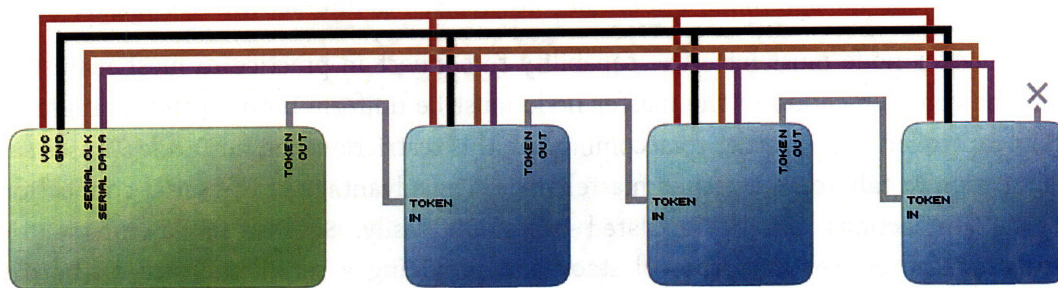


Figure 20 Connection diagram for the subTextile module interconnect bus

Master Node

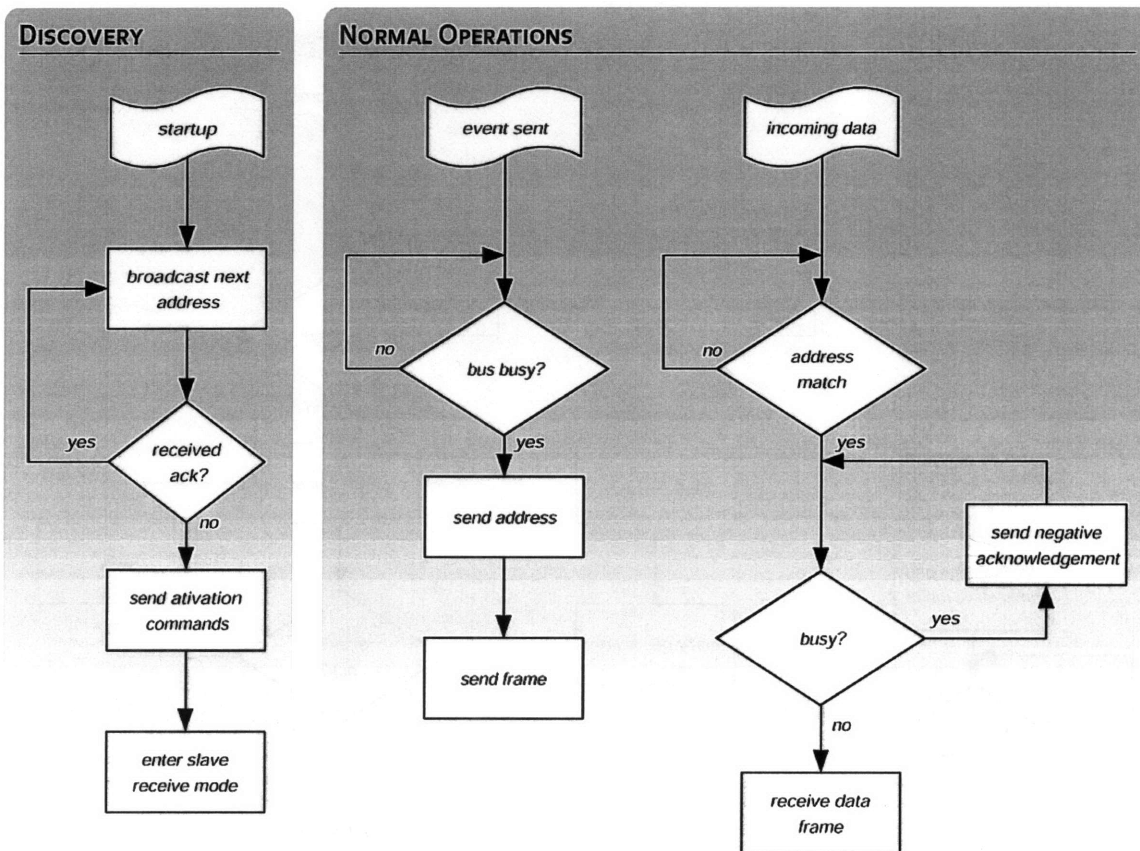


Figure 21 State diagram of I²C mode transitions during communication (master node)

The subTextile interconnect uses a two-wire multi-master-capable Inter-Integrated Circuit (I²C) bus originally pioneered by Philips Semiconductors [51]. The bus architecture is widely documented and supports a number of advanced features such as collision-avoiding (as opposed to merely collision detecting) multi-master arbitration and integral addressing capabilities. While it is certainly not the only bus protocol with these capabilities, the availability of the technology with considerable hardware-level acceleration on a number of platforms was a strong motivation for this choice. Two additional wires on the bus are dedicated to power and ground. The final wire is not a bus wire, but rather an inter-device token passing wire used to allocate addresses to devices. The connection diagram for the bus is shown in Figure 20. The token is not passed in a complete loop. Instead, the I²C bus is used to detect the end of the token loop.

The choice of I²C over a single-wire technology such as the Dallas 1-Wire or iButton [60] was based on the proven reliability and maturity of the I²C technology. While a single-wire power and data bus would have reduced the interconnect by two wires (clock and separate power), practical wisdom suggests that the single-wire technology is not sufficiently mature at this point, either in terms of speed or robustness. Additionally, the wide scale availability of the I²C implementation in microcontrollers and peripherals ensures a wider selection for MCUs than would be afforded by the newer combined power and data solutions.

Other Nodes

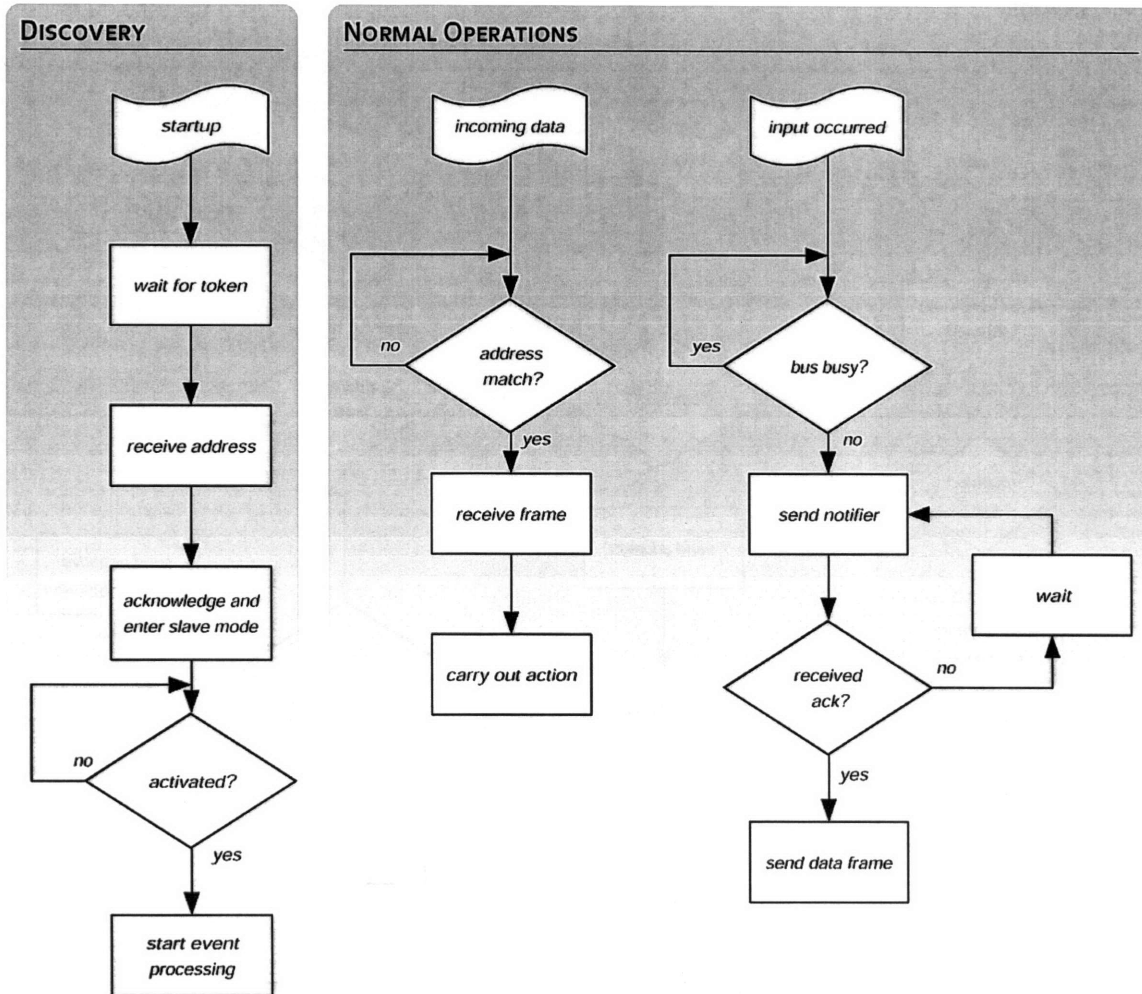


Figure 22 State diagram of I^2C mode transitions during communication (device nodes)

The I^2C transceiver can operate in one of 4 selectable modes under application control [51]. The application can transition between master or slave, and transmitter or receiver, on demand. The mode transitions during an interchange are shown in Figure 21. In general, after initial address allocation, all nodes including the master node operate in slave receive mode, and transition to master mode when outbound communication is needed. Regardless of operational efficiency, it should be noted that the bus has practical limits in terms of the amount of data that it can handle without causing erratic behavior. The design expectation is that there will always be sufficient aggregate bandwidth on the bus over time for some inefficiency in communication. While some level of redundancy is built into the specification, it is not practical to support much buffering in the MCU execution environment, and it is left to user to constrain themselves in terms of bandwidth use.

6.1.2 discovery

Discovery within subTextile is handled by the master node. Once powered up, the master node should wait a sufficient amount of time to allow all other nodes to power up. All other nodes

must power up with their I^2C units at standby (ie, not acknowledging data). The master node initiates discovery by sending a pulse on the outgoing token line to transfer the addressing token to the first device in the chain. The device in possession of the token changes its client address to the global broadcast address (0x00). The master node then transmits the next available client address (starting at 0x02, with the master itself being address 0x01) to the global address. If the address is not acknowledged, it indicates the end of discovery, since no further nodes are now in global receive mode. If an acknowledgement is received, the master next reads from the address just assigned. The possessor of the token now transmits its device type (a 16-bit value) to the master, and transfers the token to the next device in the chain by pulsing the token line. Once a device has received an address and has given up ownership of the addressing token, it immediately switches to slave receive mode. Once discovery is complete, the master node communicates with each connected device and activates or deactivates the device based on whether the device is used in the current subTextile program that is loaded. While the deactivation of unused devices is optional, the activation of devices is not, in order to ensure that arbitrary data from devices does not cause a failure in the discovery process.

The discovery protocol handshake not only assigns a unique address to the devices connected to the master node, but allows the master node to deterministically associate connected devices and their device types with addresses. If multiple copies of the same device are in use, their order on the bus becomes the equivalent of their position in the device list within the subTextile ED. In this way, the user can always receive predictable behavior from the system without requiring modification to the firmware of each copy of a device connected to the master node.

6.1.3 node constraints

Once discovery is completed, all nodes are constrained to only output data in accordance with their event formats as set within the *subTextile DE*. Each outgoing packet is prefixed by the device's own address, the device type, and the event type as set in the device description used while creating the program. This is followed by an arbitrary number of bytes determined by the number of parameters and their types. All integer values are transmitted as 16-bit, while grid type data is transmitted in row-major order. No bracketing of data is used. When the master node sends an event to any other node, the master node uses its own address and device id of zero (0x00). However, it uses the event ID specified in the device definition file, and passes data in the same way as any other node. This uniformity allows for the easy addition of listening or debugging nodes, which can simply treat all communications as equivalent.

If busy, the master node may choose to acknowledge the first byte of data, but then send a not-acknowledge (NACK) to the second byte. This can happen, in particular, when the master node is in the process of executing a primitive operation, which must happen atomically. Once the operation is complete, the master will once again acknowledge data. If a non-master node encounters this condition, it must not relinquish the bus. Instead, it must re-transmit the second byte repeatedly with a fixed delay until the master has acknowledged it, and then proceed to complete the transmission. The delay is at the discretion of the sender, though it

must be non-zero in order to reduce loading on the master caused by interrupt handling. This ensures a form of interconnect bus locking, where the sending node and the bus itself become locked until the master node is ready. This additionally prevents state loss in any connected node within the subTextile system.

It should additionally be noted that the subTextile system does not explicitly provide a squelch or automatic transmission rate control functionality. It is therefore the duty of all nodes to operate at data rates that maintain the aforementioned aggregate bandwidth surplus. If a node encounters state changes at a very high rate, it should either discard, average, or select values for transmission at a rate that is not bus-saturating, as appropriate for the type of input. This cooperative design requirement is necessary since the inclusion of additional bus arbitration techniques in the subTextile system would generally negatively impact performance of the master module in a network with high data throughput and require more time to be dedicated to bus management on all nodes. Regardless, given the bandwidth of the I^2C bus and limitations in concealing electronics in textiles, it is expected that these issues will be seldom encountered by a novice user. If it becomes necessary, the bus arbitration guidelines will be amended at a later point to include further bus management strategies that prioritize outgoing messages from the master node to allow it to function properly regardless of bus saturation.

7 evaluation

Evaluation methodology for programming languages remains a contentious issue in academia, with the literature offering wildly different and yet logically valid experimental methods [56]. This fracture is mostly due to the complex interaction between reasoning and creativity that is necessary in order to approach the particular method of problem solving known as programming. Reitman introduced the term "ill-defined problem" to describe a class of problems (including programming) that lacks deterministic procedures for solving a given problem [61]. In particular, analysis of the application of HCI methodologies to the testing of interactions have shown that the results are heavily influenced by prior experience [57]. Although relationships exist between efficiency at a task and the contributing factors that precipitate the increase in efficiency, these relationships are apparently quite well obfuscated [57]. Additionally, it appears that there are well-understood but generally ill-articulated metacognitive reasons for the choices made within the field interested in the exploration of visual languages, as demonstrated by the successes of visual languages and effectiveness of stated metacognitive factors influencing design choices [62]. While an attempt has been made to articulate such metacognitive reasoning within the context of subTextile, this does not provide a *prima fascia* testable domain.

A side-effect of this lack of concrete methodology is the lack of concrete principles on which to base the design of visual languages. Indeed, design is perhaps the best term used to describe the process. in that there is an element of fiat influenced by the aforementioned diffuse metacognitive knowledge that dominates the design of visual languages. Difficulties in establishing reasonable baselines that apply across languages biased in different dimensions further complicate matters [56]. This issue is further compounded by the design of subTextile, which incorporates elements from multiple language classes, while eschewing diagrammatic representations, which is perhaps the best-studied class of visual programming languages. However, surveys of visual languages suggest that the presumptions made about the mappings used by users in working with visual languages are often different from what the user actually does [57]. This disparity may well be the result of the diffuse and inarticulate nature of the metacognitive factors as understood by researchers. As such, it is a prudent goal to attempt to discover and test the designer's expectations against a variety of users to determine flaws and points of contention in the design.

The particular method used in order to gather metacognitive information about the use of subTextile was pioneered by Bell et al., and uses a programming walkthrough in order to discover discrepancies and error-prone interactions between the metacognitive model used by

the designer and the cognitive model employed by a variety of end users by exposing the user model [63, 64]. The programming walkthrough presents the user with a programming task stated in terms of the expected behavior. The user is allowed any amount of questions of calls for assistance during the performance of the task. As the user performs the task, he or she is observed for signs of hesitation or confusion, and is asked to articulate the thought process. The combination of observation and articulation, along with stress on a learning-oriented scenario with plentiful available help, allows the observer to gain an understanding of the user's mental model of the language. This user model is then compared with the metacognitive model of the user which the language is based around, or in other words, the stereotype of the user's thought process that was presumed during the design phase of the language. Any discrepancies between the models indicate a false presumption, and suggest not only particular interactions that cause the user to stumble, but also lower-level expectations that the user has of the interface which are not fulfilled. The advantage of the technique is that it can be employed at any stage of design. However, with subTextile, the method is used against a fairly complete prototype in order to allow for a more complete evaluation and a better understanding of issues stemming from visual and interaction design, as well as deeper issues resulting from the low level language design choices themselves.

The particular capabilities measured are facility and expressiveness [64]. Facility is defined as the ability to solve a problem, while expressiveness is defined as the ability to state that solution in the terms provided by the language. A problem with facility, given a known-solvable problem, is generally a problem in communicating the capabilities of the language to the user. A problem with expressiveness, on the other hand, can be further compounded by the ability of the user to think in terms of the language. In the case of subTextile, expressivity of the language is of particular concern, especially in reference to the design itself. As a result, the evaluation was modified to elicit more data for expressivity.

7.1 experimental method

For a programming walkthrough, the measurement is proposed with respect to a set of problems and an overview the language. The test giver demonstrates the use of the interface in general terms, as well as the location of device files and the basics of program flow. The subject is then asked to complete a number of predefined problems, while thinking out aloud about the approach to the problem. Problems were proposed in pairs, with a different set of devices and activities for each pair. Example problems included re-creating the pillow created for the prototype as a pure communication device, and then modifying the design to include the ability to save messages; as well as creating a dress which is sound-sensitive, then modifying it to allow the user to enable of disable the functionality based on the proximity of certain people. Each problem is defined in terms of the set of inputs available, and the expected behavior. The breakdown of the problems scale in difficulty to match learning about the system, with the first focusing more on event handling, with the latter focusing on decision-making and interactivity. The task pairs also force the use to create and then modify behaviors, giving an understanding of problems caused during "tweaking" of behaviors.

In addition to documenting the thought process prior to the start of programming, the user is also asked to externalize the rationale behind a change, or the thought process at any time where the user is struggling with the next step. Once the user proposes a solution, the test giver then outlines the actual effects of the program (acting as a human emulator). The user is then free to continue the task. During the "implementation" portion of the exercise, the test giver offers three levels of fallback. First, the appropriate section of the written document is pointed out. If this does not prove to be sufficient, the test giver verbally explains the construct further. If the problem occurs in the actual approach to the problem, or due to incorrect breakdown of high-level tasks, the test giver then proceeds to help the subject reason through the problematic segment and reduce to be understandable. In order to further elicit useful information from the subjects, a freeform design exercise follows the predefined problems. This provides an avenue allowing the user exploration of a free-form problem that the user has defined, thus showing the flexibility of the language under ad hoc usage conditions. Once the freeform section is complete, the subject is given a short survey allowing them to voice any issues with the interface and the language in general, and to propose critiques of the approaches taken.

The sample population for the evaluation was specifically chosen to represent a cross-section of the possible combinations of skill and background that are most likely to affect an user's performance with subTextile: familiarity with technology and background in design-related tasks. The combinations of parameters tested is described in Table 2. One person was chosen for each pairing of competencies, resulting in 6 subjects. Since competency is gradated, each factor was tested against multiple people, thus allowing for some cancellation of per-subject variance. While this number is by no means statistically significant, the qualitative evaluation does provide a good understanding of the top-level changes needed in the system without delving into minutiae that would benefit from the statistical weight of more subjects. It should be noted that subjects with design background all formally hold degrees in various design fields. All subjects were in the age 20 – 30 bracket. Though the results of these evaluations are not quantitative, the relatively polar selection of subject backgrounds allows for observable gamut in user reactions.

Technical Background	Design Background	Description
Proficient	No design background	Able to program in other languages Understands hardware
Capable but not proficient	No design background	Able to do advanced user tasks Understands programming but not proficient
Not proficient	No design background	Able to do average user tasks
Proficient	Formally trained	Trained in media/fine arts Understands hardware and software
Capable but not proficient	Formally trained	Able to do advanced user tasks Understands programming but not proficient
Not proficient	Formally trained	Trained in media/fine arts

Table 2 Subject selection by background

7.2 results

During the study, additional behavioral notes were taken, and later correlated with the audio recording of the thought process. The users all successfully vocalized their internal problem solution process, providing insight into the approaches to solving problems. The users were evaluated offline along the various axes outlines in Table 1, and when possible their performances were analyzed to note differentiation due a particular background trait by excluding other backgrounds as being a factor.

In terms of understanding of abstractions used in subTextile, the primary effect came from proficiency with programming. Greater understanding of low-level details of programming at any level of proficiency caused subjects to initially disregard the abstractions provided by subTextile. Interestingly, once these same users fully discovered the abstraction mechanisms, they used the abstractions most successfully. This result is not entirely unexpected, since the proficient users of technology generally do not attach themselves to one particular tool, but instead try to determine the underlying principles that guide sets of tools. Closeness of mapping, or the relative ease with which the user can translate their mental model of a solution to the model used by subTextile, is closely dependent on the user's understanding of related abstractions, and the results echoed this correlation. It should be noted that as with any language, there is always a learning curve. The subjects all indicated that the design of subTextile allowed them to quickly achieve some effect. However, their performance clearly indicated that more than with languages with verbose syntax, the succinct representation of subTextile required a grasp of the underlying conceptual framework and ideals in order to successfully map their internal mental representations to the programming environment.

Just as interesting is the finding that while users with a higher level of design training did not necessarily find the abstraction as difficult to accept initially, they did have greater difficulty in transforming from mental representations to subTextile primitives when, for whatever reason, they had started from incorrect priors. In one particular example, one of the subjects came to the conclusion that one of the input event slots was in fact an input-output slot, and spent considerable time architecting a solution for the task at hand around this assumption. Once the subject had begun to travel down this lane of thought, it was quite difficult to switch to the correct path, even though the subject understood the problem correctly at a cognitive level. This effect was completely unexpected, and clearly needs to be investigated further. All subjects reported that subTextile was all time sufficiently changeable at the interface level, which at first flush excludes that source of problems.

The post-task questionnaires and discussions indicated that all users were satisfied with the level of consistency and verbosity in the language proper. However, the interface showed some of its flaws in this analysis, indicating that the presence of the variables and devices on the left side pane was inconsistent with the drag-and-drop operations that were used in the rest of the system. Likewise, some subjects found it difficult to fathom the purpose of the variable and

operator lists in the expression editor, occasionally mistaking a selection within the lists to mean the inclusion of that variable in the expression. Lastly, almost every subject attempted to access details about the devices by attempting to manipulate the device list, showing a clear need for some form of "device inspector." In terms of verbosity, subjects uniformly commented on the ability to accomplish tasks which they considered to be complex using only a minimum number of primitive actions. In the cases of the more technically proficient users, it often transpired that they made mistakes in performing a task simply because they expected the task to be more complex than what the subTextile representation of the task actually was.

Propensity for errors, as expected, was highest in the primitives borrowed from imperative languages. Users at all levels of proficiency spent more time determining how to use the freeform expressions and switch/select statements than any other primitives. In general, the number of errors was lowered by heavy use of assistive lists of variables and functions by all users. However, it does suggest that perhaps a greater level of assistance should be provided for the creation of freeform expression in order to reduce the cognitive loading of creating such expressions.

The lack of visual representation of output devices also proved to be a hindrance to performance for the subjects. Since outgoing messages depend on this "hidden" functionality, the subjects felt the need to prematurely commit to sending messages with incomplete understanding of the outcome, despite having full device descriptions on hand. Visual aids for distinguishing output messages and parameters would clearly provide an improvement to the user experience by allowing the user to forego recalling the exact details of particular attached devices. On the other hand, this problem may be less significant in field deployment, when the user would have much more intimate knowledge of the hardware in the system. On a related note, all users with design backgrounds expressed preference for a physical instantiation of the devices, indicating the clear importance of the interaction between input and output, but also between development and testing in the field of design.

Though subTextile has been designed from the ground up to avoid some of the most difficult mental operations in traditional programming, as previously pointed out, this has in some sense moved the problems to the conceptual domain. Accordingly, the subjects were observed to flounder while trying to understand how to attack the problem and break it down. However, when the users accepted that the system essentially automatically broke the problem down by providing events for relevant conditions, their efficiency improved dramatically regardless of background. Those with the least technical competency gained the most in terms of efficiency and performance speed from this understanding, and made these gains most quickly. Interestingly, an intermediate level of skill seemed to inhibit acceptance of the conceptual framework of subTextile, instead leading users to cling to their favorite approach to the problem. This may be due to the lack of a fully generalized mental model of programming capabilities within these individuals, and provides and interesting area of further investigation in terms of language design. At the same time, it underscores the need for clear documentation to accompany good language design, in order to afford the end user every chance to gain an understanding of the underpinnings of a language.

As a final matter of interest, the least technically experienced subjects in general produced the most elegant solutions in the least time, while the most technically experienced subjects created solutions at a higher speed (discounting issues mentioned previously), but approached the problem very incrementally, leading to more operations for the same problems. This echoes the thought processes observed, with experienced subjects approaching the problem in a much more sequential way than inexperienced subjects, who tended to think of the entire problem as an unit. This result suggest a need for a secondary "thinking scaffold" in languages like subTextile that allow users to create "skeletons" for behaviors. With the low number of primitives that subTextile offers, this problem is fairly minimal. However, this remains an issue for similar systems with more primitive operators.

8 conclusions and future work

In the immediate future, a number of improvements need to be completed in order to bring the work done for the thesis to a stage where it can be considered complete as a development environment. In addition to the issues discussed in the results section, the evaluation has supplied several new directions for the interface to subTextile. First and foremost, the display of device capabilities needs to be improved. As it stands, the device display, especially for output devices, represents a hidden dependency that impedes the problem-solving capabilities of the user to an extent. A new layout for the event area has already been formulated, and an early sketch can be seen in figure x. The new layout clearly specifies the device-event mapping, and allows in-place renaming and deleting of devices. To reinforce the fact that the event handler stacks exist only for incoming events, a large inward arrow now reiterates the fact. Lastly, outgoing events are also listed out under devices, causing output only devices to also appear in the event area. While this consumes some additional space, the tradeoff is well worth the gain in transparency. Outgoing events are marked with an outward facing arrow to reiterate their direction, and carry the name and type of the variables needed for sending a message to the device.

Additionally, it was noted that the semantics of the left-side vertical bar differed somewhat from the language proper. While indeed it makes no sense to drag an entire device or variable, their presence in a screen containing, for the most part, draggable components is somewhat confusing. Additionally, the pane serves no purpose other than the creation and editing of variables. It is unclear whether this pane should be moved to the less-dominant right side, or if it should be removed completely to a different tab dedicated to variables, thus making the viewing of variables separate from the viewing of the event area. Each method has its own

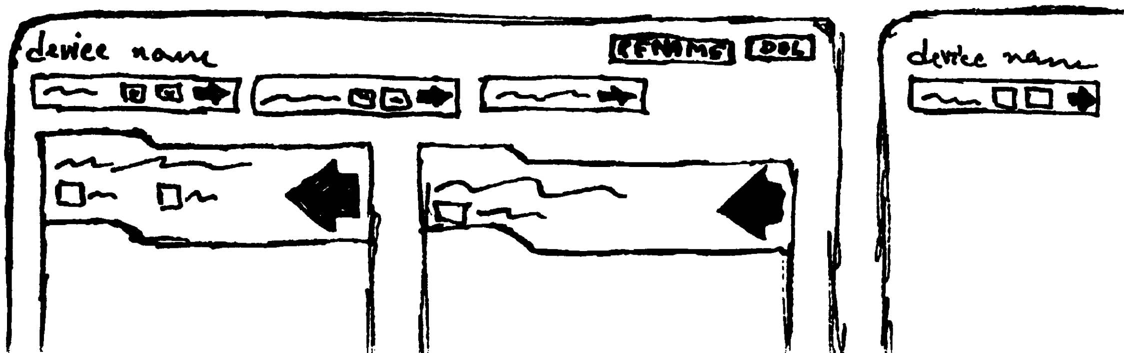


Figure 23 Preliminary sketch of changes to device and event handler stack representation

advantages and disadvantages, and it is likely that both need to be tested against users to determine the correct option in this case. Another similar problem exists in attempting to show the direction of flow of execution, which is left-to-right, and then top-to-bottom in the switch/select primitive. A number of possibilities are as again available, though the correct choice is far from certain.

The next most immediate task at hand is the completion of the hardware prototypes for the subTextile physical implementation guideline. In order to help the adoption of subTextile, a feature-complete hardware platform is planned. This platform will provide a complete master module, as well as a set of commonly used input and output devices. It is expected that the prototype boards will be completed within the next four months, along with remaining implementation of the compiler. The language can then be tested in realistic conditions, perhaps by loan to designers hoping to make use of such technologies. Studies [63] also show that a language is hardly complete without the documentation that makes it accessible, and documentation of languages is in itself a complex area to consider. While the participants in the user study had the advantage of the author explaining the system to them, this is in fact another form of dependency, at a higher level than the original dependence on technological skill. In order to truly allow for open ended exploration, a comprehensive documentation must also be created. As the hardware prototype moves forward, this will become of greater concern.

There are also a number of improvements that can be made to the user interface in order to improve usability, adding features that make certain operations more simplistic. Currently, the *subTextile DE* makes no accommodations for code reuse. The only possible way to reuse code is to copy it visually. Code reuse has been found to be a double edged sword, creating passages of code which are difficult to process due to their visual simplicity. Nonetheless, in realistic scenarios where the number of attached devices is high, it is certain that code reuse will be less error-prone than recreating the same code passages repeatedly. As such, the group select, cut, copy, and paste operations need to be added to the UI. Group select in particular is problematic, since it requires adding a "hidden functionality" to the user interface, where either a key must be used along with the mouse in order to extend a selection, or the drag operation must be reloaded. Of these options, the drag operation is most likely more damaging to the user interface, but the changes will need to be studied to determine the best course.

The addition of cut/copy/paste, which can succinctly called "clipboard functionalities," also suggests the possibility of having a "scrap book" of code, either copied from the current program, or loaded as a library. These libraries, while supporting sharing and community development, also add a level of complexity which increases the cost of entry. Essentially, as tasks get subsumed into the library system, the user feels greater pressure to "learn the community" before delving into the task at hand. Additionally, it is unclear whether such capabilities will assist subTextile, which was intentionally designed to operate without libraries and other encapsulations.

The expression editor also provides a venue for greater capabilities. In particular, the ability to suggest completions and check for errors on the fly in the same way as a word processor would be helpful in finding problems early on. Additionally, there is currently no reverse dependency

checking in the environment, which prevents user notification when the deletion of a device or variable would affect expressions and primitives. This capability would reduce user frustration at the compile phase by providing relevant just-in-time feedback.

After the additional work is completed, it is the my hope to publish the software and hardware designs under a permissive license and foster a community which can freely use the system in order to further the field of electronic textiles. Ultimately, the adoption of a language or system is the true metric for its success. With a project such as this, it is only too common for the work to be left by the wayside, in particular because there are no convincing applications to highlight the capabilities of the system. It is therefore necessary to both create such applications and foster development. As subTextile matures, the focus will shift to this use.

Even as the work on subTextile progressed, it became clear that codification implied constraint and that as the design choices made in subTextile were a tradeoff that required discarding many other possibilities. As with any design task, the hope is to maximize the gains without curtailing liberties, and keeping overgeneralization at bay. It is perhaps best summed up by a quote from Maggie Orth [65]:

> *I think that computers have managed to remain neutral beige for so long because their function is based on their interiors, their software.*

Though this research is geared towards bridging wearable computing and e-textiles for novices, the generalization it engenders is a double-edged sword. If the language and interface designs constrain in form or function, then they cannot be judged a success. The goal of this thesis is not to generalize electronic textiles as a means of expression, but to generalize customization and programming, and to imbue electronic textiles with greater interactivity. Stated from the personal viewpoint, the purpose of this work is to make the author obsolete within the context of behavioral textile as a technical collaborator, and in some sense transfer to subTextile the capability to make ideas into reality. The work presented above is a first step towards understanding and exposing the behavioral aspect of the world. It is hoped that as the ideas behind subTextile, once developed, matured, and generalized, can be used to take a step towards a world where personalization does not stop at skin depth, but instead engages the behavior of smart objects and materials in order to create an environment is literally the embodiment of our will.

9 bibliography

[1] J. M. Berzowska and W. Bender, "Computational Expressionism, or how the role of random () is changing in computer art," *Human vision and electronic imaging IV*, vol. 3644, pp. 45-55, 1999.

[2] E. R. Post, M. Orth, P. Russo, and N. Gershenfeld, "E-broidery: Design and fabrication of textile-based computing," *IBM Systems Journal*, vol. 39, pp. 840-860, 2000.

[3] E. R. Post and M. Orth, "Smart Fabric, or Wearable Clothing," presented at Proc. Intl. Symp. on Wearable Computers, 1997.

[4] G. Weinberg, "Interconnected Musical Networks: Toward a Theoretical Famework," *Computer Music Journal*, vol. 29, pp. 23-39, 2005.

[5] Sensatex. http://www.sensatex.com/

[6] SoftSwitch. http://www.softswitch.co.uk/

[7] "An SMS at your pillow?," in *Password: Philips Research technology magazine*, pp. 4.

[8] F. Carpi and D. D. Rossi, "Electroactive polymer-based devices for e-textiles in biomedicine," *Information Technology in Biomedicine, IEEE Transactions on*, vol. 9, pp. 295-318, 2005.

[9] T. Healy, J. Donnelly, B. O'Neill, J. Alderman, A. Mathewson, F. Clemens, J. Heiber, T. Graule, A. Ullsperger, W. Hartmann, and others, "Technology development for building flexible silicon functional fibres," presented at Wearable Computers, 2003. Proceedings. Seventh IEEE International Symposium on, 2003.

[10] B. K. Rakesh and X. Yong, "A Novel Intelligent Textile Technology Based on Silicon Flexible Skins," 2005.

[11] R. Wijesiriwardana, T. Dias, and S. Mukhopadhyay, "Resistive fibre-meshed transducers," presented at Wearable Computers, 2003. Proceedings. Seventh IEEE International Symposium on, 2003.

[12] W. C. Bang, W. Chang, K. H. Kang, E. S. Choi, A. Potanin, and D. Y. Kim, "Self-contained Spatial Input Device for Wearable Computers," presented at Proc., 7th IEEE Int. Symp. on Wearable Computers, 2003.

[13] J. Berzowska and M. Coelho, "Kukkia and Vilkas: Kinetic Electronic Garments," presented at Wearable Computers, 2005. Proceedings. Ninth IEEE International Symposium on, 2005.

[14] E. Wade and H. H. Asada, "DC Behavior of Conductive Fabric Networks with Application to Wearable Sensor Nodes," presented at International Workshop on Wearable and Implantable Body Sensor Networks (BSN'06), Cambridge, MA, USA, 2006.

[15] M. J. Zieniewicz, D. C. Johnson, C. Wong, and J. D. Flatt, "The evolution of Army wearable computers," *Pervasive Computing, IEEE*, vol. 1, pp. 30-40, 2002.

[16] XS Labs. http://www.xslabs.net/

[17] Hexagram Institute. http://hexagram.org/

[18] S. Baurley, "Interactive and experiential design in smart textile products and applications," *Personal and Ubiquitous Computing*, vol. 8, pp. 274-281, 2004.

[19] J. McCann, R. Hurford, and A. Martin, "A design process for the development of innovative smart clothing that addresses end-user needs from technical, functional, aesthetic and cultural view points," 2005.
[20] M. L. Galbraith, "Computational Garment Design," Massachusetts Institute of Technology 2003.
[21] G. Nanda, "Accessorizing with Networks: The Possibilities of Building with Computational Textiles," Massachusetts Institute of Technology 2005.
[22] Topological Media Group. http://www.gvu.gatech.edu/people/sha.xinwei/topologicalmedia/index.html
[23] X. Sha, Y. Serita, J. Coffin, S. Dow, G. Iachello, V. Fiano, J. Berzowska, Y. Caravia, D. Nain, W. Reitberger, and others, "Demonstrations of expressive softwear and ambient media," 2003.
[24] M. Jacobs and L. Worbin, "Reach: dynamic textile patterns for communication and social expression," *Conference on Human Factors in Computing Systems*, pp. 1493-1496, 2005.
[25] C. M. Liu and J. S. Donath, "Urbanhermes: social signaling with electronic fashion," presented at Proceedings of the SIGCHI conference on Human Factors in computing systems, 2006.
[26] M. Kanis, N. Winters, S. Agamanolis, A. Gavin, and C. Cullinan, "Toward wearable social networking with iBand," presented at Conference on Human Factors in Computing Systems, 2005.
[27] J. Berzowska, "Memory Rich Clothing: Second Skins that Communicate Physical Memory," presented at Proceedings of the 5th conference on Creativity & Cognition, ACM Press, New York (2005), 2005.
[28] HearWear: The Fashion of Environmental Noise Display. http://www.absurdee.com/HearWear/
[29] NYX Wearable Displays. http://www.coolhunting.com/archives/2004/11/nyx_wearable_di_1.php
[30] S. Mann, "Humanistic computing: "WearComp" as a new framework and application for intelligent signal processing," *Proceedings of the IEEE*, vol. 86, pp. 2123-2151, 1998.
[31] S. Mann, ""Smart clothing": wearable multimedia computing and "personal imaging" to restore the technological balance between people and their environments," presented at Proceedings of the fourth ACM international conference on Multimedia, 1997.
[32] B. J. Rhodes, "The wearable remembrance agent: A system for augmented memory," *Personal Technologies*, vol. 1, pp. 218-224, 1997.
[33] A. Pentland, "Wearable Intelligence," in *Scientific American*, vol. 9, 1998, pp. 90-95.
[34] R. DeVaul, M. Sung, J. Gips, and A. Pentland, "MIThril 2003: applications and architecture," presented at Wearable Computers, 2003. Proceedings. Seventh IEEE International Symposium on, 2003.
[35] Georgia Tech Wearable Motherboard. http://www.gtwm.gatech.edu/
[36] A. Feldman, E. Tapia, S. Sadi, P. Maes, and C. Schmandt, "ReachMedia: on-the-move interaction with everyday objects," presented at Wearable Computers, 2005. Proceedings. Ninth IEEE International Symposium on, 2005.
[37] R. Mitchel, "Behavior construction kits," *Commun. ACM*, vol. 36, pp. 64-71, 1993.
[38] "Crickets," Lifelong Kindergarden Group, MIT Media Lab.
[39] O. Zuckerman and M. Resnick, "A physical interface for system dynamics simulation," presented at Conference on Human Factors in Computing Systems, 2003.
[40] S. M. J. Harrison, "SoundBlocks and SoundScratch: Tangible and Virtual Digital Sound Programming and Manipulation for Children," Dept. of Architecture. Program In Media Arts and Sciences, Massachusetts Institute of Technology 2005.

[41] Nylon. http://acg.media.mit.edu/concepts/volume10.html
[42] Parallax Basic Stamps. http://www.parallax.com/html_pages/products/basicstamps/basic_stamps.asp
[43] Logo. http://el.media.mit.edu/logo-foundation/
[44] A. Begel, "LogoBlocks: A Graphical Programming Language for Interacting with the World," Electrical Engineering and Computer Science Department, MIT, Boston, MA, 1996.
[45] Scratch. Lifelong Kindergarden Group, MIT Media Lab.
[46] Squeak. Squeak Foundation. http://www.squeak.org/
[47] Processing. http://processing.org/
[48] Wiring. http://wiring.org.co/
[49] Max/MSP. Cycling '74. http://www.cycling74.com/products/maxmsp.html
[50] H. S. Raffle, A. J. Parkes, and H. Ishii, *Topobo: a constructive assembly system with kinetic memory*: ACM Press New York, NY, USA, 2004.
[51] I2C (Inter-Integrated Circuit) Bus Technical Overview and Frequently Asked Questions. http://www.esacademy.com/faq/i2c/
[52] Microsoft Word. Microsoft Corporation. http://www.microsoft.com/office/
[53] J. Bonar and E. Soloway, "Uncovering principles of novice programming," presented at Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1983.
[54] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys (CSUR)*, vol. 37, pp. 83-137, 2005.
[55] M. C. Morrone and D. C. Burr, "Feature Detection in Human Vision: A Phase-Dependent Energy Model," *Proceedings of the Royal Society of London. Series B, Biological Sciences*, vol. 235, pp. 221-245, 1988.
[56] P. Gross and K. Powers, "Evaluating assessments of novice programming environments," presented at Proceedings of the 2005 international workshop on Computing education research, 2005.
[57] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages and Computing*, vol. 7, pp. 131-174, 1996.
[58] A. F. Blackwell and T. R. G. Green, "Does Metaphor Increase Visual Language Usability?," presented at Proc. 1999 IEEE Symp. on Visual Languages, 1999.
[59] M. M. Burnett and M. J. Baker, "A Classification System for Visual Programming Languages," Oregon State University Corvallis, OR, USA 1993.
[60] 1-Wire and iButton website. http://www.maxim-ic.com/1-Wire.cfm
[61] A. F. Blackwell, K. N. Whitley, J. Good, and M. Petre, "Cognitive Factors in Programming with Diagrams," *Artificial Intelligence Review*, vol. 15, pp. 95-114, 2001.
[62] A. F. Blackwell, "Metacognitive Theories of Visual Programming: What do we think we are doing," presented at Proceedings IEEE Symposium on Visual Languages, 1996.
[63] B. Bell, J. Rieman, and C. Lewis, *Usability testing of a graphical programming system: things we missed in a programming walkthrough*: ACM Press New York, NY, USA, 1991.
[64] C. Lewis, J. Rieman, R. Weaver, N. Wilde, B. Zorn, B. Bell, and W. Citrin, "Using the programming walkthrough to aid in programming language design," 1994.
[65] HorizonZero, "Interview with Maggie Orth," http://www.horizonzero.ca/textsite/wear.php?is=16&file=8.

appendix a: supported operators

Arithmetic Functions

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Mathematical Functions

Function	Description
abs(x)	Returns absolute value of x
acos(x)	Returns arc cosine of x
asin(x)	Returns arc sine of x
atan(x)	Returns arc tangent, not able to handle asymptotic conditions of x
atan2(x, y)	Returns arc tangent, quadrant-corrected of coordinates (x, y)
ceil(x)	Returns the smallest integer greater than or equal to x
cos(x)	Returns the trigonometric cosine of an angle
exp(x)	Returns Euler's number e raised to the power x
floor(x)	Returns the largest integer less than or equal to x
log(x)	Returns the natural logarithm of x
log10(x)	Returns the base-10 logarithm of x
max(x, y)	Returns the greater of x and y
min(x, y)	Returns the lesser of x and y
pow(x, y)	Returns x^y
random()	Returns a random number
round(x)	Returns the closest integer value to x
sin(x)	Returns the sine of x
sqrt(x)	Returns the square root of x
tan(x)	Returns the tangent of x

Constant Functions

Function	Description
e()	Returns value of the base of the natural logarithm
pi()	Returns value of pi

Boolean Operators

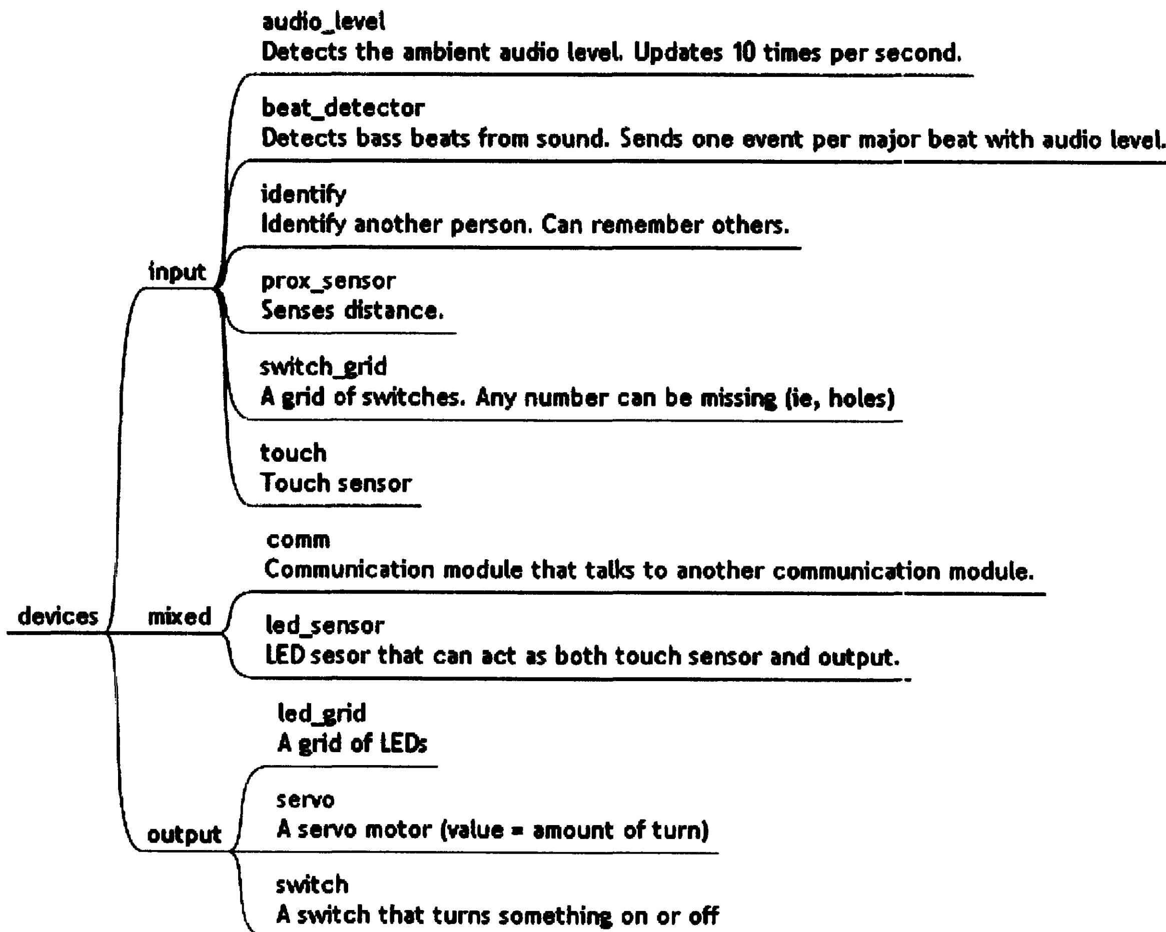
Operator	Description
==	Boolean equal-to
&&	Boolean AND
||	Boolean OR
!=	Boolean not-equal-to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!	Boolean NOT

appendix b: evaluation documents

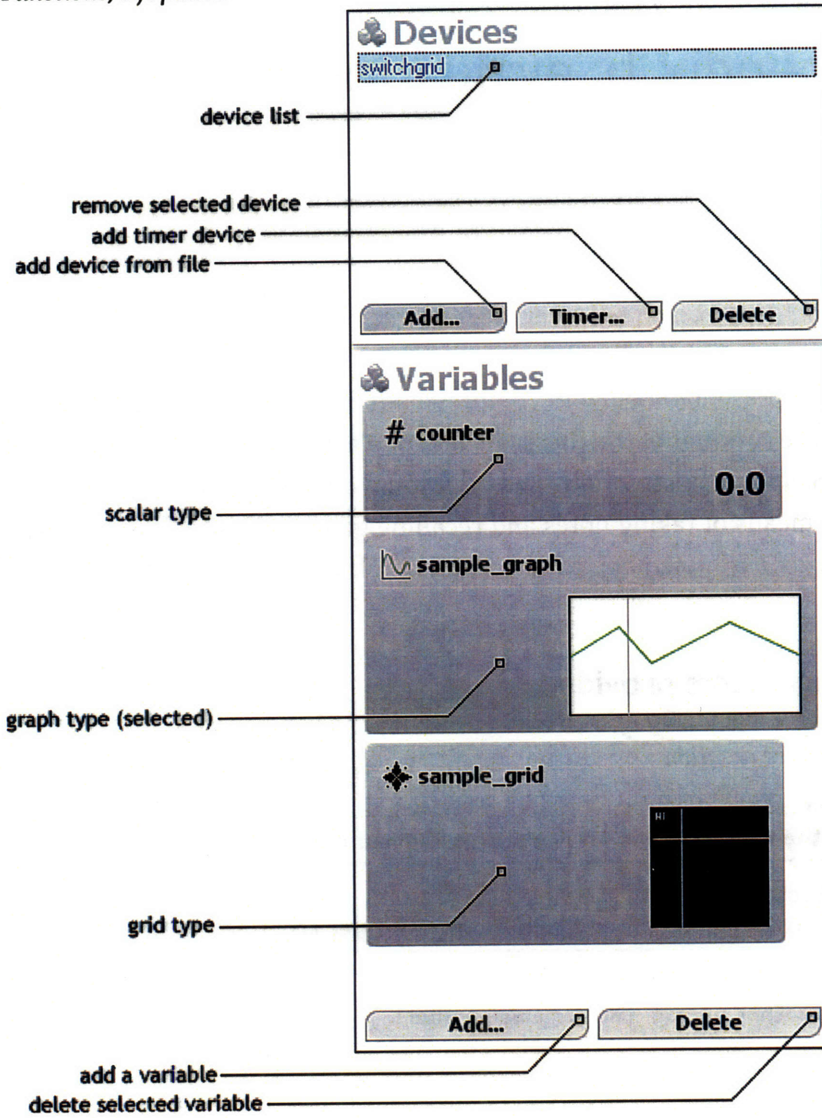
The evaluation documents are presented below in compressed form. The formatting has been modified to better fit within the confines of the thesis document. Each of these documents were supplemented with extensive spoken instructions and additional details. The references were provided to allow the user a means of taking notes and recalling general details. The post-task questionnaire was provided as a supplement to the analysis of verbalization of their own thought process.

9.1 documentation attachments provided

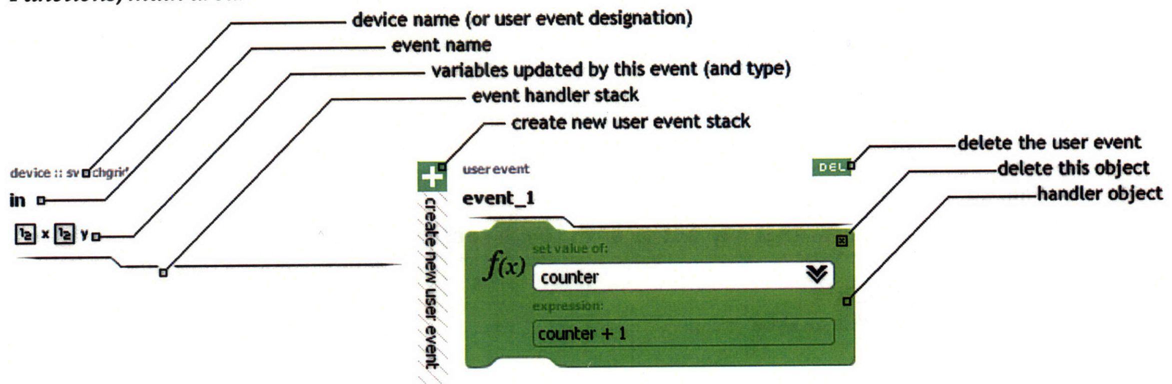
Map of device file location and device fucntion



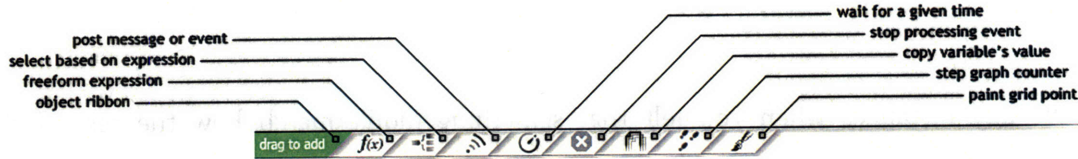
Functions, left panel:



Functions, main area:



Functions, ribbon:



9.2 tasks

Task 1:

Create a pillow that responds to touch. It will communicate with another copy of itself. The pillow has one LED grid output, one switch grid input, and a communication device. The LED grid keeps its own state. Each grid starts off in a known equal state.

Task 2:

Add the ability to save a particular display by pressing buttons 1 and 2 on the first row of the pillow. Rows and columns are numbered starting with 1.
Hint: the display can only be read by sending a message to it.
Add the ability to clear the display when button 1 and 3 on the first row are pressed. You can clear the display by sending an empty grid to it.

Task 3:

Create a skirt which switches a light built into it based on audio beats. The louder the beat, the longer the light should stay on. The skirt has one beat sensor, and one switch output. Beat sensor volume levels are between 0 and 100. The switch turns on when it is sent a 1. It turns off when sent a zero.

Task 4:

Modify the skirt so that it only works when a person you know is nearby. You will need to add an identification sensor. The identification sensor sends a number when it reads a person nearby. Allow the user to add or forget people. Use a touch sensor for this.
Hint: people are added by sending a remember event with the unknown ID. People are forgotten by sending a forget event with the known ID.

Task 5:

You have at your disposal a fabric flower (on a dress) that can animate using a servo. It has one servo output. Animate the flower (your choice of motion).

Task 6:

Modify the flower to be "shy." It should close when the use tries to touch it.
Hint: Add a proximity sensor, and check the reading. Presume distance are between 0 and 100 in 1/10 inches. When something leaves the range of the sensor, the sensor sends a 100 reading.

9.3 post-task questionnaire

Describe any particular tricks you felt you had to do to get things to work as expected.

Describe any instances when you felt that something didn't match how the rest of the environment worked.

Describe any locations where you felt that the language required more steps than you would have liked

----

Describe any locations where things worked contrary to the way you expected things to work.

Describe any locations where the system was not telling you something you needed to know.

Describe any locations where the system required you to make a choice with insufficient information.

Describe any locations where you couldn't understand how things fit together.

Describe any locations where it was difficult to make changes.

Describe any locations where you could not see the parts of the program that you needed to see.