Delft University of Technology
Master's Thesis in Embedded Systems

Interactive Design Studio: A spatial-computing framework for non-IT specialists

Agostino Di Figlia

embedded software

TU Delft Delft University of Technology

Interactive Design Studio: A spatial-computing framework for non-IT specialists

Master's Thesis in Embedded Systems

Embedded Software Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Agostino Di Figlia
a.difiglia@student.tudelft.nl

15th August 2012

Author
Agostino Di Figlia (a.difiglia@student.tudelft.nl)
Title
Interactive Design Studio: A spatial-computing framework for non-IT specialists
MSc presentation
15th August 2012

Graduation Committee
Prof. Koen Langendoen, Professor, Embedded Software, EWI
Dr. Stefan Dulman, Assistant professor, Embedded Software, EWI
Dr. Walter Aprile, Assistant professor, StudioLab, Fac. Industrial Design
Andrei Pruteanu, Phd student, Embedded Software, EWI

Abstract

In recent years, application domains outside information technology such as architecture have shown an increasing interest in the capabilities of networked systems such as localized computation, reduced power-consumption, and distributed interaction. These capabilities incline the design of interactive environments towards the IT-domain. In particular the employment of large scale distributed systems in interactive designs bring along new challenges for architects/designer such as distributed algorithm design, programming skills, embedded platform knowledge. As a consequence of this new trend, adequate software tools to bridge the gap between the IT-world and the field of interactive design are scarce. In order to fill that gap we propose a framework called Interactive Design Studio(IDS) which aims to provide the necessary tools to hide the technological aspects from the end-user when designing interactive environments. To tackle the problem of handling large scale networks of embedded platforms we are convinced that spatial computing is a promising paradigm. The main reasons are the scalability(network size does not influence node behavior) and resiliency to network dynamics(network is hidden by space abstraction). Therefore we present a spatial computing framework for non-IT specialists. We provide a way to specify agent-level behavior and to generate the corresponding code for a specific embedded platform. In our case, we use the eLua VM enhanced with the necessary spatial computing capabilities. We show that spatial computing is a good methodology for Industrial Design and Architecture. Moreover, we show that spatial computing is possible using off-the-shelf virtual machines for embedded platforms. In conclusion, we demonstrate that IDS succeeds within a certain extent in abstracting or hiding the underlying technological aspects(spatial computing) from end-users.

Preface

My bachelor thesis at the University of Palermo introduced me to the world of embedded systems and I immediately knew that this was the area I would like to specialize for. In addition, I wanted to give a boost of enthusiasm to my life by going abroad. The consequence was to apply at the TU Delft to pursue the Master of Science in Embedded Systems. During my two years at the TU Delft I experienced the exciting international community and Dutch lifestyle which enriched me profoundly in terms of social and cultural integration. Besides that, I was able to have a glimpse at the various aspects in embedded systems. My favorite area of interest became embedded software applied to large scale distributed embedded systems. I believe that technology and research is useful if it can be made accessible to domains outside information technology. Therefore, I decided to join the embedded software group and in particular the Snowdrop Project which aims to provide access to embedded systems and its benefits to non-IT specialists such as architects and designers. The project was ambitious and had everything I was looking for: interdisciplinarity, embedded software design, and engineering.

The thesis work was characterized by ups and downs. Usually we are thankful to people helping us in difficult times. I will do that by giving special thanks to my supervisors Stefan and Andrei for giving me advice and ideas. However, I want to thank them also for telling me "good job" and creating a friendly atmosphere within the Snowdrop Project, the Industrial Design and Architecture collaborators. I also want to thank my family for giving me the opportunity to study abroad and supporting me in difficult times. Last but not least, I want to thank the 'family' of friends created here in Delft for the wonderful moments together.

Agostino Di Figlia

Delft, The Netherlands
15th August 2012

Contents

Chapter 1

Introduction

1.1 Context

Application domains outside information technology are more and more attracted by the capabilities that networked systems bring along such as localized computation, reduced power-consumption, and distributed interaction. In fact, architects show an increasing interest for intelligent and interactive building environments [20]. Current state-of-the-art includes designs such as the Ada floor [22]. Ada consists of mesh of tiles that interact with the users stepping on it by means of light patterns. Healing pool by Brian Knep [33] is another example. It consists of a projection of organic patterns on a floor that self-heals after being torn apart by people walking on the floor. These projects emphasize the growing trend of designing complex interactive spaces in private or public buildings [27, 45]. This trend has been also noticed in architecture faculties where master tracks and courses are focused on interactive architectural components. An example is Phototropia [47]. Phototropia is part of an ongoing research on the application of smart materials in an architectural context realized by the Master of Advanced Studies class at the Chair for Computer Aided Architectural Design(CAAD), Department for Architecture at ETH Zurich. For this project smart materials are fused with embedded platforms to create an interactive experience by means of light patterns.

Based on the mentioned examples we notice that even though interactivity is achieved with various technologies a common property is occurring in all applications: computational elements are spread out and fill the design space. They interact with each other and the users in various ways leading to complex behaviors. Moreover, the relationship between architecture/design and the technology domain is becoming stronger and the necessity for an architect or designer to be acquainted with programming and hardware engineering is becoming a rule instead of an exception.

A commonality noticed when surveying the current deployments is the

employment of some form of centralized control. When the installation consists of a small number of embedded elements that is not an issue but when we have to scale up to large installations centralized control becomes almost impossible and the distributed interaction is difficult to create. When trying to mimic distributed systems such as in the case of the healing pool application [33] the used technology (projectors, cameras and image recognition, a limited-sized deployment area) requires the need for a specific carefully controlled environment, considerably limiting the design freedom.

We are convinced that the design problem faced by architects inherent to interactive environment design represents the killer application for the field of spatial computing. We can describe the solution at hand in relation to either bottom-up or top-down design of complex systems:

- Computing devices equipped with sensing and actuation are becoming ubiquitous. This triggers the research for feasible and interesting interactive designs that make use of embedded platforms. A side effect of this trend is having to deal with the underlying technological complexity related to communication protocols, programming languages, operating systems, embedded virtual machines, and hardware platforms etc inherent to embedded platforms. Therefore, a way is needed for non-IT specialists to fast prototype ideas on large-scale systems while abstracting from the technological aspects.

- Secondly, the top-down design uncovers the complexity that arises in such distributed systems which consists in the translation of specifications for system behavior into local rules (also called global-to-local compiling). This problem has been addressed before for different application domains [42] and it is an ongoing and challenging research question.

The aforementioned problems are open for research in both the field of complexity theory in general and the field of spatial computing in particular [11]. From what we know, solutions to both problems include human expertise [23].

In recent years, a number of spatial computing domain-specific languages (DSLs) targeting IT-specialists were made available such as Proto [13], Kairos [26], Meld [10], and TOTA [38]. These DSLs attempt to specify global system behavior via spatial computing constructs. Our framework is built on top of the Lua programming language to be able to provide the constructs inherent to spatial computing. We further elaborate on this in Chapter 2.

In the architecture and design domain there exist software tools to interface design with technological devices. The current most popular software tools that designers use are Max/Msp [21] and Rhino/Grasshopper [4]. They can be interfaced with Arduino boards. These tools have too many

constraints when it comes to large scale design and there is still no support for a distributed way of thinking. We further investigate these frameworks in Chapter 2.

In this thesis, we present a framework called *Interactive Design Studio*(IDS). We provide the non-IT specialists with a friendly tool chain to facilitate the creation of interactive experiences for various purposes such as education, entertainment and leisure, playground etc. To make that possible we exploit the capabilities that come with the spatial computing paradigm in tackling design issues inherent to large scale distributed embedded systems. We, therefore, present a holistic framework providing a way to specify algorithms, test them, and finally deploy them on distributed embedded systems.

Figure 1.1: ProtoSpace 3.0

We target ProtoSpace [28](Figure 1.1) at TU Delft, Faculty of Architecture, Hyperbody Group [17]. The space comprises an interactive floor, protoDeck, consisting of 189 tiles each equipped with a micro-controller, RGB LEDs and a pressure sensor (Figure 2.15). Moreover, ProtoSpace 3.0 [28] is enhanced with other multimedia devices such as beamers, speaker sound system and various interactive objects are under design. All the components of ProtoSpace are thought to form an ecosystem capable to create interactive user experiences. As user experiences we imagine use cases such as leisure and entertainment, teaching, art exhibitions, dance performances etc.

1.2 The Snowdrop Project

This thesis work is part of the *Snowdrop project* at the Technical University of Delft. The project aims to provide a tool-chain for the creation of spatial computing applications. In Figure 1.2 we show its components and the relations between them. The project can be divided into two areas of concern: high-level specification of spatial constructs and low-level software components that unleash the possibility to apply the spatial computing paradigm on a specific hardware platform.

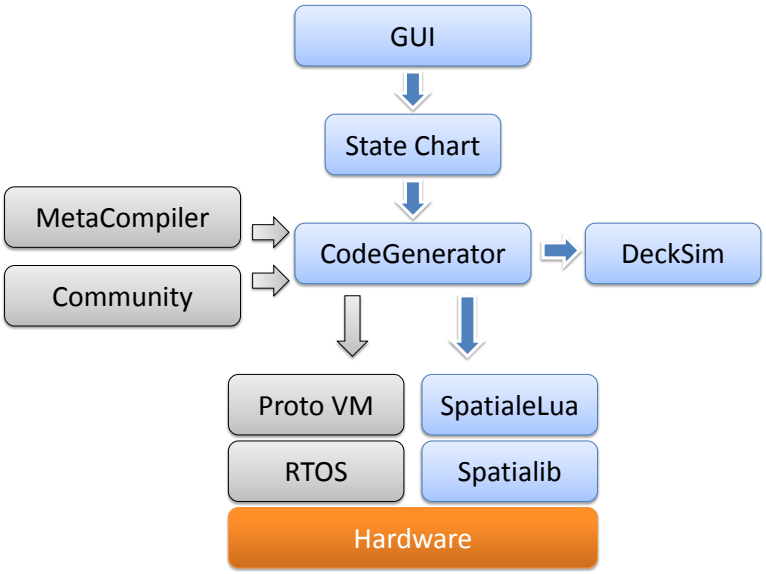


Figure 1.2: The Snowdrop project block diagram.

There exist two levels of specifying the system specification. One level is represented by a Meta-Compiler or Community, and the other by a GUI. The Meta-Compiler attempts to find spatial computing primitives(local rules) from global system specifications. On the other hand, the GUI gives to non-IT specialists a mean to specify individual node behavior by using a library providing spatial constructs. The applications can be tested on a simulator for algorithmic validity. The output of both the Meta-Compiler or the GUI is deployed on the embedded platform which comes with two different approaches to provide spatial computing functionality: Proto and eLua. The Proto platform is based entirely on the spatial computing paradigm and comprises the DelftProto virtual machine which executes Proto language scripts. My approach is to use a general purpose virtual machine, eLua VM, enhanced with spatial computing capabilities. Both platforms run on LPCXpresso 1769 hardware platforms. The focus of this thesis work ranges from the high-level specification of spatial behaviors by means of a GUI or State Chart Diagrams to the porting and adaptation of the eLua

VM for spatial computing purposes. This includes the specification and generation of spatial applications respectively by State Charts via a State Chart Compiler, and the realization of a simulator to validate those applications. The framework targets the field of interactive design and architecture and aims to deliver all the necessary features and functionality for that purpose(Chapter 3). Nevertheless, the framework can be adapted to serve other purposes and applications for spatial computing as we see later on(Chapter 4).

1.3 Problem Statement

The thesis introduces a framework in the form of a software tool-chain that makes use of distributed computing, sensing, and actuation. It targets designers and architects - the non-IT specialists - and aims to help them explore various interactive design ideas via spatial computing constructs. The main components of the framework target all the aspects that cover the design process of an interactive space: algorithm design, behavior specification, simulation, and ultimately deployment to the hardware platform.

More specifically, this thesis aims to answer the following research questions:

- *Is spatial computing a good methodology for Interactive Design and Architecture? To what kind of extent is spatial computing suitable? What are the possible limitations?*
- *Is spatial computing possible using off-the-shelf virtual machines for embedded platforms? If yes, what are the trade-offs, if any?*
- *Is it possible to abstract or hide the underlying technological aspects (spatial computing) from end-users? How and why?*

1.4 Outline

The thesis has the following structure. We start with presenting the background study on which the thesis work is based(Chapter 2). In Chapter 3, we identify the software framework design requirements. Afterwards, in Chapter 4, we present our solution to the problem statement and the design requirements. We then show example applications for our framework(Chapter 5). In Chapter 6, we show and discuss the platform specific experiments and benchmarks. We conclude by drawing our own conclusions and by identifying areas for future work(Chapter 7).

Chapter 2

Background

In this Chapter we present the background study on which the thesis work is based. The study comprises notions inherent to the design world and to the engineering world. We first focus on previous work in the field of interactive environments emphasizing on both the aesthetics and the technical realization. Afterwards, we show the state of the art of current tools on the market for designing interactive design involving micro-controller. After that we focus on the designer's world by pursuing a study on the role of design and the designer. Moreover, we focus on the aspects concerning the collaboration and communication between designers and engineers. The design related study ends by presenting a user survey conducted on a sample of architecture students from TU Delft and ETH Zurich. The second part of this Chapter targets the related work in the spatial computing field, the chosen platform and language for our framework. We conclude this Chapter by presenting the target platform of this thesis.

2.1 Interactive environments

In the recent years an ever increasing popularity for interactive environments has been observed [16]. In this Section we discuss three interactive spaces (*Ada* [22], *Healing Pool* [33] and *Hallway monitoring* [12]) and their main characteristics in terms of interactivity types as well as technologies.

Delbruck et al. have created a tactile luminous floor, *Ada* [22]. It is part of an interactive autonomous space comprising a floor, projection screens, microphones, ceiling cameras, speakers and theatre lights. The floor tiles are equipped with tactile pressure sensors and RGB lamps. They are networked in a mesh like topology and communication is achieved by using an industrial automation network called Interbus [25]. The floor's tiles are centrally controlled: the tile's local controller delivers the data to a PC that controls the individual node behavior. On the contrary, our approach aims to provide a complete distributed approach achieving interactive user experiences.

One more example of an interactive floor is the *Healing Pool* project [33]. *Healing Pool* is an interactive video installation that was presented at an exhibition in the Brauer Museum of Art (Valparaiso University). The exhibition space is equipped with video projectors, cameras, custom software, and a vinyl floor. The distinctive feature of the installation is the ability to project organic patterns that are torn apart by visitors walking on the floor. These pattern are rebuilding continuously in an always unique way. Even though artificial intelligence and imaging techniques are the main focus of this realization it shows the increasing interest in creating interactive environments. Their results are promising but we strongly believe that large-scale complex interaction can be achieved only by means of distributed systems of sensors and actuators via the spatial computing paradigm.

The Hallway Monitoring [12] project is technologically similar to ours. In fact, this project consists of a hallway floor equipped with wireless sensor nodes underneath. Each node is equipped with a pressure sensor and light actuators. These lights together with speakers placed on the hallway's wall are triggered by exerting pressure on the tiles. Nevertheless, the project presents limited complex interaction capabilities due to the limited space and the lack of direct feedback from the tiles. Moreover, apart of the hallway there are no extra objects to interact with and the movement is restricted to one person along one direction only. That translates into a poor expressiveness for designers and architects. On the other hand, it might result more interesting from a computer science point of view.

2.2 Interactive Design Software Frameworks

Since interactive design presents an increasing trend in architecture and design, a series of software frameworks that bridge the gap between software design and/or deployment onto embedded systems have been made available. We briefly describe the most important ones. Among those are Rhino3D's plug-ins Grasshopper [4] and Firefly [2], MAX/Msp [21], and Processing [6]. All the previously mentioned tools in most cases interface with an Arduino micro-controller. In the following paragraphs we describe each tool in more detail.

The first framework consist of the following parts: Rhino3D/Grasshopper, Firefly and Arduino. Rhino3D [7] is a modeling-tool that is commonly used for industrial design, architecture, rapid prototyping, CAD/CAM, and more. It features a Visual Basic based scripting language and it became popular in architectural design in part because of Grasshopper. Grasshopper is a graphical algorithm editor that provides a way to generate parametric forms by dropping scripting and using functional block design. Firefly tries to bridge the gap between parametric modeling and Arduino. It provides the necessary software tools to facilitate the mapping between software

and hardware. In other words, Firefly allows real-time data flow between Grasshopper and the Arduino board. The main feature of interest for our research is the Code Generator component provided by Firefly which attempts to convert a Grasshopper definition into Arduino compatible code(C++). The Code Generator checks the component ID against a library of custom C++ functions which will then be added to the code in case of a match. Grasshopper, Arduino, and Firefly represent a tool-chain that is capable to cover the whole design process that starts with the algorithm design and ends with the deployment onto an Arduino based design installation. At the moment, no distributed system made of Arduinos are supported leaving a gap that our framework aims to fill. Moreover, since only Arduino platforms are supported, it limits the design space and the design projects in terms of hardware capabilities. In fact, with our tool-chain we aim to facilitate the usage of distributed algorithms for design installations that make use of large scale embedded platforms. It adds an additional choice when it comes to choosing the most suitable embedded platform that best suits a given project.

Another framework is represented by Max/Msp and Arduino. MAX is a visual programming language environment for music and multimedia developed by Cycling'74 [21]. It is highly modular and is easily extensible. There are several extensions that allow to interface MAX to an Arduino: Maxuino [5], Arduino2Max [1] and Firmata [3]. Maxuino seems the most actively developed and it is a collaborative open source initiative for quickly and easily getting MAX to talk to the I/O of an Arduino board. This allows Max to read analog and digital pins, to write to digital and PWM pins, to control servos, to listen to i2c sensors, and more. All the aforementioned extensions are really basic and they only allow to access the Arduino I/O pins. It allows little design freedom for eventual interactive design projects. Firmata is different from the other two extensions since it is a generic protocol for communicating with micro-controllers from software on a host computer.

The last framework we discuss is called Processing [6]. It is an open source programming language and environment for people who want to create images, animations, and interactions. Processing together with Rhino3D is the most used software among architecture students (see User survey in Section 2.4). Also for Processing the Firmata firmware is used on an Arduino to establish communication between them. In this framework, Processing programming language (C++ like) is a prerequisite and the common usage is to use the Arduino and its sensors in order to display their readings. The communication is therefore mostly done from the Arduino board to Processing that is running on a PC. No actuation and interactivity has been explored so far with these tools.

We have noticed that the current software frameworks that interface to micro-controllers are still at a basic level when it comes to features required for creating interactive designs on large scale embedded platforms. Even

when it comes to single device installations in some of the frameworks the previously mentioned limitations still exist. However, the most promising of the aforementioned frameworks is Firefly since it provides a code generator and a basic graphical algorithm design interface. In our work, we try to tackle all these shortcomings in order to provide a way to specify distributed behaviors and generate corresponding low-level code that is executed on large-scale embedded platforms.

2.3 Designer's World

Since the activity for this thesis has involved a close collaboration with designers and architects, an extensive study had to be done to understand the designer's world and way of thinking. This effort has ultimately paid off when it comes to incorporating ideas and thoughts about the common interface that separates the design and the engineering domains. The collaboration regarded the possible use cases and scenarios in ProtoSpace and exploring the feasibility of a GUI-concept to specify local-agent behaviors targeting protoDeck tiles.

This Section is mostly inspired from "Sketching User Experiences: getting the design right and the right design" by Bill Buxton [18]. In his book, Buxton tries to point out the complex design problems and the ever changing requirements that designers have to cope with due to the pervasiveness of nowadays technology. Buxton emphasizes the importance of a proper design phase that has to precede the engineering and sales phase in the product development cycles as well as the importance of a close collaboration between designers and engineers. In the following sections we focus on the most relevant parts of Buxton's book that are meaningful for our project.

2.3.1 Design and the role of the Designer

Buxton does not give a formal and precise definition of what "design": *"design" is what someone who went to art college and studied industrial design would recognize as design.* Design is seen as "design for the wild". By this, the author tries to detach design from the mere object design. In fact, he uses a term introduced by Hummels, Djajadiningrat and Overbecke [29], "context of experience". With that he means that the output will not necessarily be a physical entity but an user experience that the design is able to provoke by generating behavioral, emotional, and experiential responses. The previous interpretation of design is mostly dictated by the current technological *status quo*. For example, designers will have to deal with objects and environments equipped with embedded systems. In fact, pervasive computing is more and more a reality. Most of the times it has interactive properties and requires an in depth study of ecosystems. By that, we mean the established relation between user/s and computers. Another example where this change in the

design process has already happened is the architecture domain. Buildings are not seen as static entities any more but are becoming active and reactive in response to the interaction with people in the interior or in the exterior. This behavior will contribute to the architectural object as much as the shape, materials and structures do. A perfectly tailored example is *ProtoSpace* 3.0 [28] where *protoDeck* and the the surrounding technology contribute to an ever changing and immersive experience by means of light shows and projections onto the walls. We conclude that design is going through a metamorphosis from an object centered activity to an experience centered design due to the advent of pervasive computing devices.

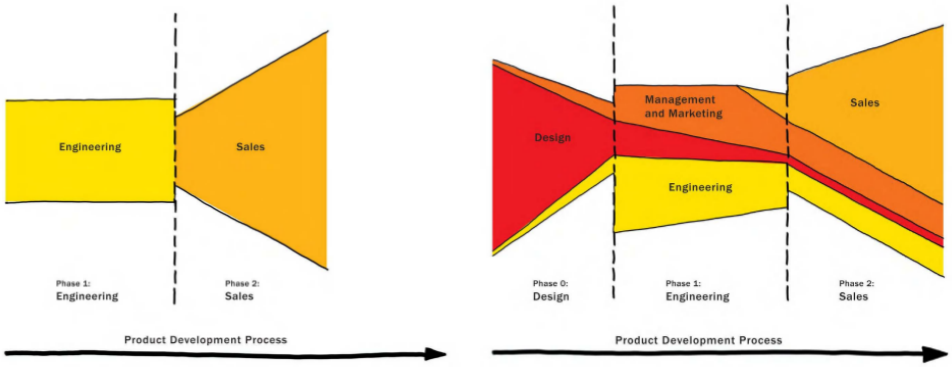
2.3.2 The role of Design in the Product Development Process

The thesis work requires the collaboration with designers and architects with the common goal to create a design tool-chain for interactive environments design. That tool-chain can be seen as product which has to go through a design phase and engineering phase.

Design and the designer have to face domains which are out of their expertise due to the pervasiveness of technology in their designs. As a consequence the role of design - seen as product design - and engineering in the whole product development(PD) process has to change. Figure 2.1a shows the *status quo* of the PD where the process consists of two separated phases: engineering and sales. The product design phase is completely left out and the product is created starting from the project requirements passed directly to the engineers. That translates into an engineering phase that has no feedback from the design domain but the requirements. The critical point of failure is that issues appearing in the production/engineering phase could have been predicted or foreseen by a proper product design phase. Doing so, time and resources could be better used.

Buxton promotes a more holistic approach(Figure 2.1b)where the different phases are mutually influencing each other to a certain extent. Designers and engineers interact with each other in order to clarify sector specific knowledge and different way of thinking during the whole product development process. In fact, it should include the consideration of engineering issues and product requirements as well as aspects related to management and marketing. The engineering phase is also influenced by the design/er, management and marketing activities. A close collaboration should somehow reduce incomprehension between the two "worlds" and lead to a better end result saving time and resources.

This approach has been used as general guide lines for the collaboration with the Industrial Design Faculty of the TU Delft. In fact, periodic weekly to monthly meetings have been organized during the product development phase in order to drive the process towards a common goal. In my opin-



(a) Status quo PD [18]

(b) Ideal PD [18]

ion, the key aspect towards a successful collaboration between engineers and designers is to have all parts with some background knowledge of the counterparts "world". The following paragraphs are exploring these concepts in more detail.

2.3.3 Designer's means of communication

As we stated above we promote a close collaboration between designer and engineers during the development process of the product. That means that communication plays a big role. The way of communicating is usually dependent on the domain. For example, engineers use modeling languages like UML to describe a system. In contrast, designers have different ways such as sketching, or prototyping. In this Section we describe the communication means of designers.

Designers use different techniques based on the specific design stage. The most important techniques are sketching and prototyping. Sketching is used in the early design phase where ideas come to birth in a continuous and always evolving manner. In contrast, prototyping is used in a later stage when an idea has already been consolidated. The time moment in the design cycle represents the detail of representation in the design object: low and imprecise in sketches and high detail and closer to the end design in prototypes. Nevertheless, both sketching and prototyping have a commonality: they are both used to help the designer explore the design space. This is done by creating a mock up of the target experience. The key point is to experience the interactivity of a product before the finalization of the design. In the following, we illustrate what is meant by sketching and its purpose. Finally, we will show the differences with respect to prototyping and a user feedback based technique called "The Wizard of Oz".

Sketching and its purpose

By sketching we mean both the classic drawing of sketches and sketching with a graphical tool on a PC. Example of the latter could be MAX/MSP or Rhino3D software tool. Sketching is used as a means to explore ideas and to communicate them in a graphical form. *Sketching is a graphic mean of technical exploration*, McGee [35]. Since it consists in drawing or modeling on a PC it reveals to be the perfect initial design technique. In fact, changing the initial ideas has to be easy and quick. All this allows to explore the new ideas in an efficient and accurate way. However, the most important attribute of sketching is its use as a powerful means of communicating ideas. With a sketch the designer gives life to ideas and communicate them in a "comprehensible" way. "Comprehensible" because in the end a sketch is not used to give precise instructions but to suggest roughly a certain vision. That means that it has to give rise to debates that help to refine the final design. *"Sketching is a catalyst for stimulating new and different interpretations that are fundamental to the cognitive processes of design"*(Buxton [18]). The most important aspect in the sketching phase is the ability to extract the meaning of the sketch. In fact as Alan Kay says: *"It takes almost as much creativity to understand a good idea, as to have it in the first place"*. If we connect this to our previous discussion in Section 2.3.2 about the importance of having interactions between designer and engineers we can see that it is necessary for an engineer to acquire or develop new skills that allow him to "understand" a designer(sketch in this case). We conclude by saying that sketching is an important tool for a designer both for his own design process and for communicating his ideas.

Prototyping vs Sketching

Another technique that designers use is prototyping. Often prototyping and sketching are erroneously seen as synonyms. As already mentioned before sketching and prototyping differ in the granularity of detail by which they represent the design object. In contrast to sketching, prototyping is closer to the end design than sketching: the whole iterative process described for sketching has been performed and a more finalized idea exists. In Figure 2.1 we can see the differences between a sketch and a prototype.

As we can see, prototyping represents something that is closer of being "ready" for usage. It does not has to suggest, question, or provoke but it gives certainties and facts to be evaluated.

The "Wizard of Oz" technique

The "Wizard of Oz" technique is a popular technique to investigate user experience during the design process before the actual product is ready for the consumers. The technique is inspired by story of "The Wizard of Oz".

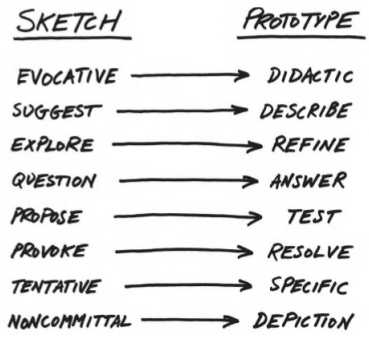


Figure 2.1: Sketch vs Prototype [18]

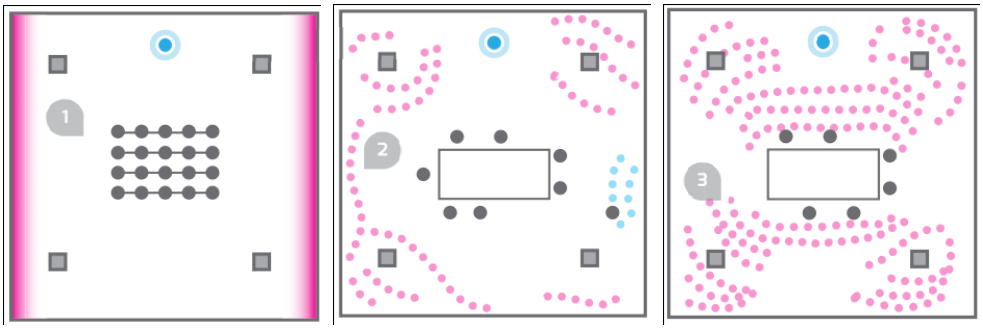


Figure 2.2: Sketch of a workshop use case in ProtoSpace by Industrial Design collaborator. It shows the desired light patterns.

As everybody knows, in the story the protagonists fear the wizard of Oz due to its "appearance". In the end, it turns out that the Wizard of Oz is a tiny, old, and scared wizard. The moral of the story is that appearance can deceive and deceiving can be, in our case ,used to fake user experience. As an example, we imagine the ticket vending machines in an airport. In order to test these systems the design company simulates the back-end mechanism with human operators that are hidden from the user. By that way, the user-machine interaction is recreated in order to retrieve useful data about the user experience. We introduced this technique to point out that a designer sometimes needs to know in advance the user before a final implementation is created. That shows that a designer uses user experience, touch, and visual representations in the design process. These are important tools with which a designer has to be provided with or has to provide for himself.

2.3.4 Design Process Iteration

From the previous paragraphs we conclude that a designer follows a particular design process. Transforming an idea into a sketch or prototype is a continuous iterative process involving creation, realization, and/or adaptation. The outcome is a prototype used for proving the validity of the idea. We call this process *Design Process Iteration*. As mentioned, it has a dynamic and empirical nature. In Figure 2.3 we see a state transition diagram that shows the design process iteration.

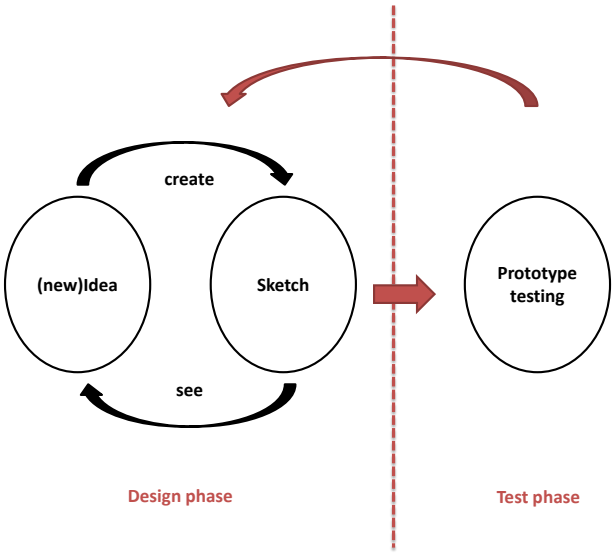


Figure 2.3: Design Process Iteration

The main characteristic of the design process iterations is the possibility to iterate from the creation and/or sketching phase to the prototyping and testing phase in a quick manner. Moreover, during various phases the designer has to be provided with specific means and tools such as the wizard of oz technique. In this thesis work we try to provide as much as possible the tools necessary for designing user experiences.

2.4 User Survey

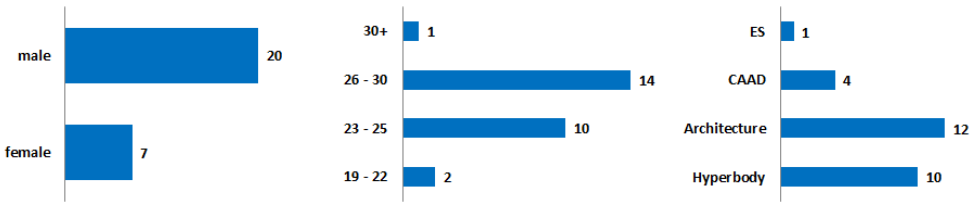
In order to be able to create the right features when designing the IDS framework and to highlight the need of such a framework we conducted a survey that targets a group of students from the faculty of architecture at the TU Delft and the Computer Aided Architectural Design department for Architecture at ETH Zurich. The survey was split in four areas of interest:

1. Interactive design in architecture
2. Prototyping tools
3. Programming skills and hardware knowledge
4. Design process and tools

This Section is organized as follows: we first show the subject profile and then discuss the topics targeted by the survey.

2.4.1 Subject Profile

We conducted the survey on architecture students from TU Delft and ETH Zurich. The former are students which currently exploit the capabilities of *ProtoSpace 3.0* and related projects in work of the *Hyperbody* group. The latter are architecture students which participated at the Master of Advanced Studies in CAAD at ETH in Zurich. It is a one-year postgraduate program with a focus on computer-based architectural design.



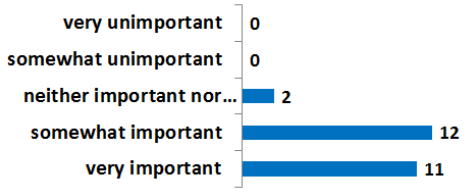
(a) Gender (b) Age distribution (c) MSc Track

Figure 2.4: Subject Profile.

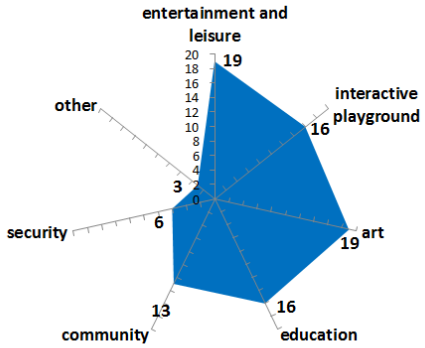
As we can see from the survey, the sample group consists of two thirds males (Figure 2.4a) while the age group is mostly from 23 to 30 years (Figure 2.4b). The MSc track (Figure 2.4c) is variegated and consists mostly of general architecture students and students that are part of either the CAAD group of ETH Zurich or the Hyperbody group of the TU Delft.

2.4.2 Interactive Design in Architecture

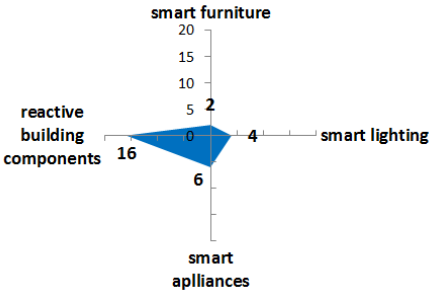
In this part of the survey we investigate the importance of interactive design in architecture. Furthermore, we ask them for the type of interactive components they think will be mostly used in interactive architecture. We inquire them about what kind of purposes they think interactive architecture is suitable for. Finally, we ask about the project they are working on or would like to work on and that involves some kind of interactiveness.



(a) Importance of Interactive Design in Architecture



(b) Design purpose



(c) Preferred "smart" components

Figure 2.5: Interactive Design in Architecture.

In Figure 2.5a we see that students imagine that interactive design is and will be a strong area of interest and future development for architectural design. Moreover, we notice that the main areas of interest in terms of design purpose are entertainment and leisure, and art but also educational as well as playground purposes. Little interest is given to security information for example (Figure 2.5b). When considering what kind of "smart" components the students would like to deal with the majority would like to create reactive building components(see Figure 2.5c). Finally we asked the students to describe a project they are working on or they would like to work on that involves interactivity and embedded technology. In the following we describe a creative and ambitious example of one of the Hyperbody project group [46].

> *The first element is the Lighting system: a grid of points. This reactive system encourages people to explore the world around them. The design is part of the research on how to discover new systems helps people in changing their surrounding environment. How appealing it is for people to adjust their surroundings to their own needs and how to interact with the lights. It is impor-*

tant to see how the building changes when more and more people adapt it in accordance to their individual needs. The Lighting

Figure 2.6: Inspire me 2 change [46]

system is used in four scenarios:

- *Automated reactive control of the grid of lights by people that move around in the building.*
- *Automated reactive control of the grid of points by a sinus movement.*
- *Making corridors, walkways, and spaces based on predefined shapes.*
- *Allowing people to control the lights with an application on their smart-phone (99 cents).*

2.4.3 Prototyping Tools

We want to know from the students what kind of software tools are currently used for prototyping interactive design. To do that we pursued them to assess these tools in terms of specific criteria that in our opinion are important for designing interactive environments. Finally, we asked to suggest eventual improvements or limitations of the current tools.

In Figure 2.7 we see that Grasshopper is clearly the most used tool among the survey subjects for creating interactive designs. A relatively strong presence is Processing which is more popular in students from the ETH Zurich group. That can be explained by the fact that Grasshopper was partly developed at the TU Delft and therefore it is mostly used among the TU Delft students.

In Figure 2.8 we see the column diagrams which show the four aspects of assessment of the preferred software tool: interactive design features, scal-

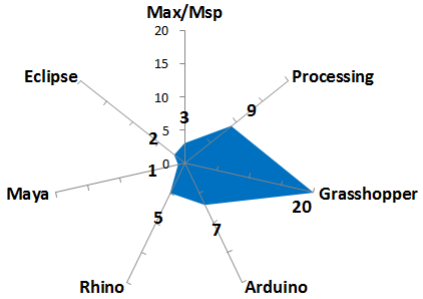
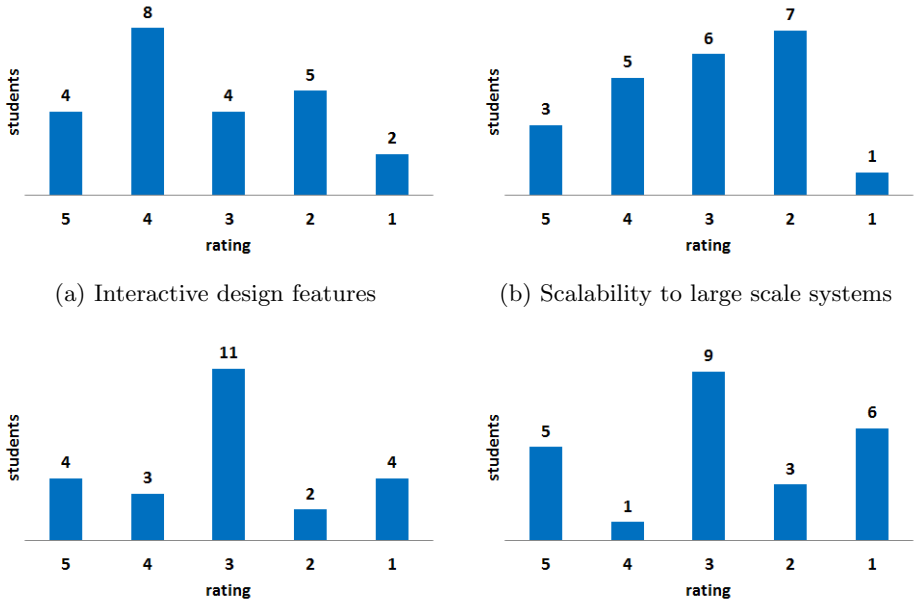


Figure 2.7: Design tools



(a) Interactive design features

(b) Scalability to large scale systems

(c) Required programming skills

(d) Deployment onto hardware

Figure 2.8: Software tool assessment

ability to large scale systems, required programming skills and deployment onto hardware. We ask the students to rate their favorite tool based on the aforementioned criteria with 5 being the highest value and 1 the lowest in terms of suitability to the corresponding feature. Each Figure shows the rating distribution. A better understanding of the software tool assessment can be seen in Figure 2.9.

In Figure 2.9 we observe that on average the current design tools are not suitable when it comes to the deployment of the code onto the hardware

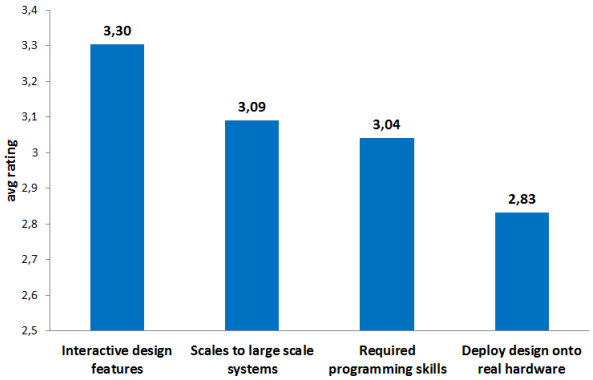
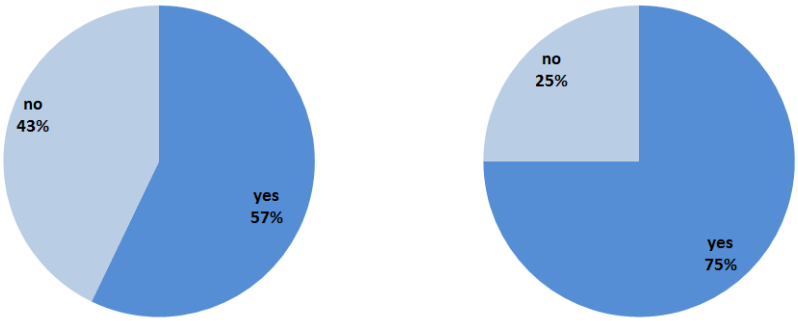


Figure 2.9: Design tool average rating

platform. Furthermore, we notice that both scalability and required programming skills have a low average rating, meaning that the software tools still require adequate programming skills and large scale systems are not yet of interest for software developers.

2.4.4 Programming Skills and Hardware Knowledge

Since interactive design involves the use of technology such as micro controllers and thereby require programming skills we were highly interested in the programming skills and hardware knowledge of the subjects.



(a) Programming knowledge

(b) Micro-controller programming experience

Figure 2.10: Programming skills

In Figure 2.10 we see that only 57% of the students have programming skills (Figure 2.10a). Out of these, 75% have experience in micro-controller

programming. As we can the from Figure 2.11, Arduino represents the micro-controller and programming platform. A quite high amount of students are also acquainted with the Processing programming language.

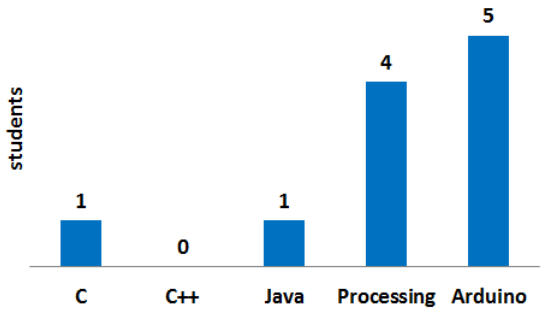


Figure 2.11: Known programming languages

2.4.5 Design Process and Tools

In Section 3.1.1 we described the importance of the design process and its most important phases. The design process is of peculiar importance for our framework in order to target the designer's or architect's requirements and needs. Therefore, we reserve a part of the survey to investigate the most important phases of the design process. In order to do so, we split the design process in the following phases: sketching, prototyping, simulation, testing, and deployment. Based on this classification, we asked the subjects to give an opinion on the relative importance of each phase in the design process and, moreover, which would be the main obstacle to overcome according to their skills.

Figure 2.12 shows the average ranking of each phase. We can see that the students did not choose unanimously one of the phases or a similar ranking of the phases. If we would extract some meaningful trend we can say that prototyping and testing are the activities with the highest average ranking. Strangely, deployment onto the real hardware is seen as an easy task.

Another important aspect we investigate is knowing which are the areas where the subject would encounter most issues. The areas we are interested in are algorithm design, hardware knowledge, programming, and deployment. In Figure 2.13 we see that on average the area with most difficulty would be algorithm design followed by programming. We see that hardware knowledge is not seen as the main issue as well as the deployment onto the embedded platform.

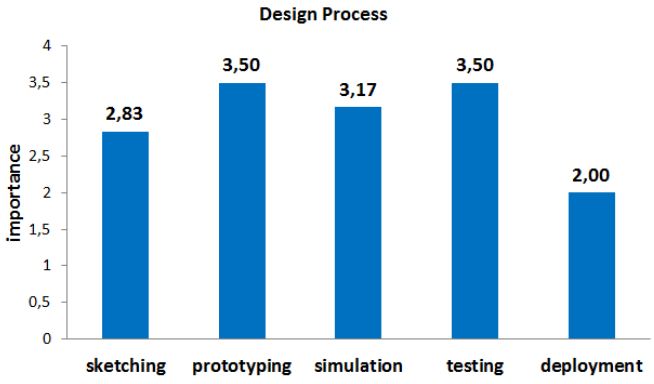


Figure 2.12: Design process ranking

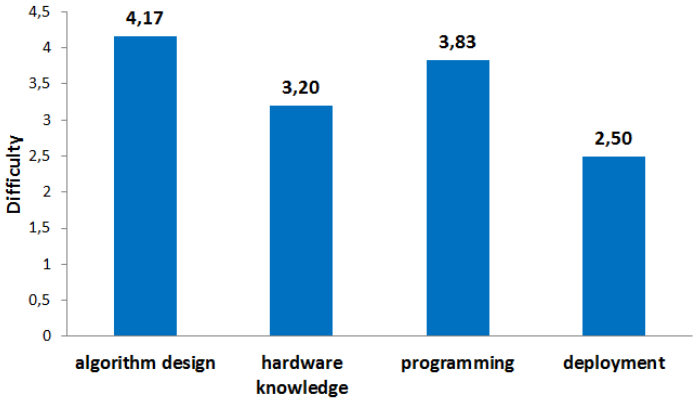


Figure 2.13: Design key issues

2.4.6 Conclusions

The survey shows that interactivity is a desired feature of architecture design. The sample group shows to be acquainted with programming languages although they all do not target embedded platforms. Most of the students have used an Arduino; hence, they have hands-on experience with microcontrollers. Nevertheless, most students struggle when it comes to algorithm design and programming. Moreover, there is a lack in dealing with large-scale distributed devices in terms of hardware and as a general concept, as we can notice by the fact that all the used software design tools do not support more than one device. The survey outcome is used to fill the gap that current students have to face when designing interactive environments that imply the use of adequate software tools and hardware knowledge.

2.5 Spatial Computing

When we talk about interactive design, interactivity is achieved by an interconnection of many devices spatially distributed which react to external events(ProtoSpace). Usually, an architect or designer has an idea of the aggregate behavior of the system but the the programming environment they are presented with operates mainly at the level of single devices. This boils down to the need of a global-to-local compilation [8] which allows to specify the aggregate behavior and to compile it to local behavior.

The specific domain of this thesis is interactive design for a space filled with a collection of local computational devices, *spatial computer* [15]. Moreover, the two main properties of this setup are the platform's spatial and temporal properties; therefore a way is needed to explore and use these two properties. Spatiality and time are strongly correlated to the spatial computing paradigm which makes use of spatial abstraction to unleash the potential of using the concepts of space and time in distributed systems programming. To exploit the benefits of spatial abstractions researchers have built domain specific languages(DSLs) aiming to simplify the problem of programming aggregates. In the following we will first present the spatial computing paradigm. Afterwards, we discuss the research that has been pursued in the field of spatial computing in terms of domain specific languages and platforms. Finally, we discuss our approach and its rationale.

Zambonelli et. all [52] highlights the inadequacy of traditional approaches to tackle modern distributed systems scenarios in relation to their key characteristics: large-scale, network dynamism, and situatedness. These characteristics are not fully handled and supported by traditional approaches such as transparent distributed computing or network-aware computing. To do so a new way of abstracting the network is proposed which is based on "spatial abstractions". The paradigm is denominated spatial computing and presents the following key characteristics:

- The concept of network - a discrete system of variously interconnected nodes - evolve into a concept of "space" which is a metric continuum.
- Distributed components are context-aware and they can perceive and influence the local properties of space.
- Spatial abstractions used in communication and application level.

The spatial computing paradigm is based on adaptive and automatic creation of an virtual metric overlay over the physical network where nodes are assigned to a specific area and are logically connected based on spatial neighborhood relations. Zambonelli identifies four levels which can provide "spatial services" and puts them in a spatial computing stack. From bottom to top the stack presents the physical level (communication), the structure

level (localization), the navigation level (local interaction/behavior) and the application level (global/system behavior).

In literature their exist several classes of spatial computers: amorphous computer [8], multi-agent systems, pervasive computers, and parallel computers. Each of them has different spatial abstractions which are based on computational fields. The later are mostly inspired by biology(chemical gradient) [43], chemistry(chemical reactions) [49], geometry/mathematics (manifold concept) [14], or physics(force fields). While the aforementioned techniques provide a means to abstract space from local interactions of the system's components, programming languages are necessary to describe goals by high-level abstractions. In the following we describe several programming languages and platforms providing the necessary spacial abstractions: GPL [19], Meld [10], Kairos [26], TOTA [39], and Proto [13].

Growing point language(GPL) is a language developed by Coore [19] for pattern formation in amorphous computer [44]. GPL is used to specify topological patterns consisting of lines of various thickness; these are compiled into a local agent program. Patterns are specified in terms of a botanical metaphor of "growing points" -a locus of activity in an amorphous mediumthat lay down material which secretes pheromones, and tropisms which determines the trajectory of the growing point. Complex patterns are the result of local rules.

Meld [10] is a declarative programming language for programming ensembles of modular robots which have to accomplish a distributed task. Meld is based on P2, a logic programming language. The global task of the ensemble can be described as a high-level abstraction by specifying facts and rules. Facts are situated in a single point in the network and they can either be independent(e.g. neighborhood status) or dependent on other facts. On the other side, rules specify a set of conditions and a new fact that can be proven. A process called forward chaining takes place: new facts are created from initial facts and which in turn satisfy additional rules and so on. This process ends whenever the common goal has been accomplished and all provable facts have been proven. The validity of Meld's approach has been evaluated only in simulation.

Kairos[26] is based on ideas from shared-memory parallel programming. It delivers three primitives: a node abstraction delivering the programmer tools to manipulate (lists of) nodes, a list of one-hop neighbors, and remote data access. Remote data access does not guarantee delivering the correct value, instead, Kairos relies on 'eventual consistency'. Eventually the system should converge to the correct solution of the problem at hand. Kairos's functionality is delivered through an API which can be accessed from imperative programming environments. However, it still remains in a proof-of-concept state.

TOTA[38] stands for 'Tuples over the Air'. TOTA is thought to be used for pervasive computing scenarios. It is based on the notion of tuple fields which

can be seen as information fields inspired from nature like force fields or chemical gradients. Each tuple consists of three elements: content, diffusion rule, and maintenance rule. The content element contains the tuple data; the diffusion rule specifies the policy by which the tuple is cloned and diffused; the maintenance rule specifies the policy through which the tuple should evolve in response to events or time constrains. The spatial neighbor concept is followed when distributing tuples through the network. Its limitation is the impossibility of aggregating tuples' information. TOTA come as a middle-ware and exposes a Java API to the end user.

Proto[13] is a functional lisp-like language that employs the concept of an amorphous medium abstraction[9], in which the discretization of space and time is hidden from the end user. Proto is based on the mathematical definition of manifold, a space that looks like an Euclidean space locally but globally might be more complex. In fact, Proto's primitives are mathematical operations on fields(functions that associate a value to each point in space-time). Communication and related services(e.g neighborhood discovery or distance estimation) are not incorporated in Proto programs. Moreover, the information about the network and neighborhood is presumed to be available and should be taken care of by the underlying layers allowing Proto programs to be very compact. Proto comes with a tool chain that includes a compiler, a simulator, and a virtual machine.

From the aforementioned programming languages, Proto suits best to the spatial computing paradigm in terms of both spatial and temporal abstractions. Nevertheless, we decided to use Lua as programming language. In fact, we use Lua's main features in terms of extensibility and tailor it to best fit the spatial computing paradigm. We will use Proto as reference for our evaluation.

2.6 Lua and eLua

For our framework we decided to use Lua programming language based virtual machine for embedded platform, eLua. In this Section, we first describe and present the Lua programming language and the Lua virtual machine. We describe its main features and most important characteristics of Lua. Finally, we describe the embedded version of Lua, eLua, and describe briefly its architecture.

2.6.1 Lua

Lua [32] is designed, implemented, and maintained by a team at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. It combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs

by interpreting byte-code for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping [30]. Lua is a multi-paradigm language, meaning that it supports more than one programming paradigm.

> *"The idea of a multi-paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms."*

The main characteristics and features can be summarized in the following:

Extensibility Lua is not a stand-alone package. It is both an *extension language* and an *extensible language*. It can be used as a library that we can link with other applications to incorporate Lua facilities into them. On the other hand, Lua is an extensible language since a program can register new functions in its environment; such functions are implemented in C (or another language), so that they can add facilities that cannot be written directly in Lua [31]. These two views of Lua imply two kinds of interaction between Lua and C. Lua as an extension language means that C has the control and Lua is the library, while in the other case Lua has the control and the C code is called library code.

Simplicity Lua is a simple and small language. It has few (but powerful) concepts. This simplicity makes Lua easy to learn and contributes to its small size [31].

Efficiency Lua has a quite efficient implementation. Independent benchmarks show Lua as one of the fastest languages in the realm of scripting (interpreted) languages [31].

Portability Lua is distributed in a small package and builds out-of-the-box in all platforms that have a standard C compiler. Lua runs on all flavors of Unix, Windows, mobile devices (running Android, iOS, BREW, Symbian, Windows Phone), embedded microprocessors such as ARM, IBM mainframes, etc. [30].

Free Lua is free open-source software, distributed under the MIT license. It may be used for any purpose, including commercial purposes, at absolutely no cost [30].

2.6.2 eLua

eLua [24] stands for embedded Lua and this project brings all the advantages and features of the Lua programming language and VM based programming

to the embedded world. For that purpose, eLua offers the full features of the regular Lua version and uses Lua's native mechanisms to extend with embedded development optimized and specific features. As a result, eLua allows to develop and run Lua programs on a wide variety of microcontrollers. The most important feature of eLua is the possibility to extend Lua with platform specific C modules and drivers. ELua's support for a wide range of embedded platforms is mainly due to its the particular structure. In Figure 2.14 is an overview of eLua's architecture.

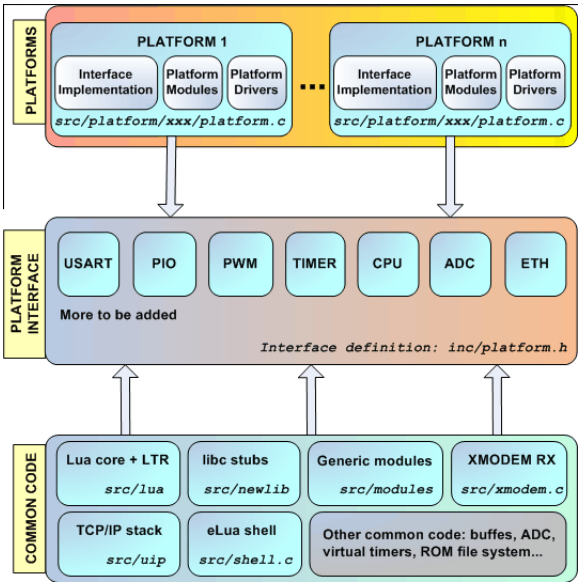


Figure 2.14: eLua logical structure [24]

For eLua, a platform is a group of CPUs that share the same core structure, in our case LPC17xx. Within that platform an eLua port implements one or more CPUs. As seen in Figure 2.14 eLua uses simple design rules in order to be as portable as possible:

- all the code that is platform-independent is common code and it has to be written in ANSI C as much as possible, this makes it highly portable among different architectures and compilers, just like Lua itself [24].
- all the code that can not be made generic (mostly about peripherals and CPU specific code) must still be made as portable as possible by using a common interface that must be implemented by all platforms on which eLua runs. This interface is called platform interface [24].
- all platforms (and their peripherals) are not created equal and vary greatly in capabilities. As already mentioned, the platform interface

tries to group only common attributes of different platforms. If one needs to access the specific functionality on a given platform it can do so by using a platform module. These are platform specific and their goal is to fill the gap between the platform interface and the full set of features provided by a certain hardware platform [24].

2.7 Target Platform - ProtoDeck

The target platform and application of this thesis is protoDeck. ProtoDeck is part of ProtoSpace 3.0 [28] which is a state of the art multi-purpose facility designed for the development of non-standard, virtual, and interactive architecture. It is developed by the Hyperbody team at the Delft University of Technology as a revolutionary real-time collaborative design environment.

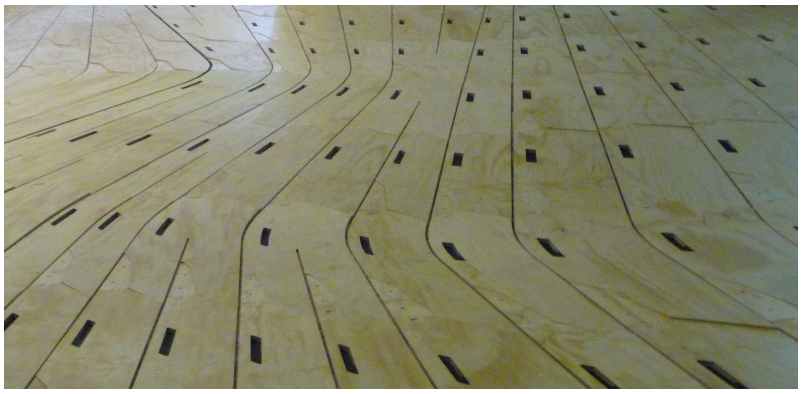


Figure 2.15: The protoDeck floor

The space consists of a set of distributed components/actuators such as protoDeck(Figure 2.15), beamers, sound sources. ProtoDeck is composed of a network of 189 tiles, each equipped with a micro-controller with a pressure sensor and RGB led actuator. All the components of ProtoSpace give rise to an interactive space for a new user experience oriented design. The main focus of this thesis is protoDeck and the underlying embedded platform. In the following we will present the hardware specification of the embedded platform.

	LPCXpresso 1769
Flash	512 kByte
CPU	32 bit, 120MHz
RAM	64 kByte
EEPROM	64 kByte
UART	4
CAN	2
Misc	I2C, SPI/SSP, USB Host, 10/100 Ethernet, on board JTAG debugger

Table 2.1: Hardware platform

Each tile on the protoDeck is equipped with an LPCXpresso 1769 by NXP with specification shown in Table 2.1.

Chapter 3

Software Framework Design

Having determined the void for a framework that would aid designers and architects in designing interactive spaces filled with technological devices, we aim to design and implement a software framework, denominated Interactive Design Studio(IDS). IDS aims to provide all the necessary tools required in the interactive design process. Ideally, these tools allow to design and specify the target system's behavior and the deployment onto the hardware platform hiding the technological aspects from the end-user. In order to do so, a series of requirements have to be met. In the following section we uncover these requirements (Section 3.1).

3.1 Requirements

The objective of the IDS framework is to bridge the gap between designers/architects and embedded software engineering. In this particular case, we have two sets of requirements that have to scope the entire process consisting of a high level and non-technical perspective of the system and an engineering perspective determined by the target hardware and software constraints. The requirements can be categorized based on whether they are inherent to the *Design process*, *Software Architecture* aspects or the *Target Platform and Application*. In the following we describe them singularly.

3.1.1 Design Process

As already described in Section 2.3 the iterative design process is strongly characterized by the iteration between design phase and test phase(see Figure 2.3). For that reason IDS should be able to speed-up the process that starts with the algorithm specification and end up with the deployment and testing on the target platform. The main issues to be tackled are summarized as follows:

- *Design process speed-up* - fast deployment and code updates dissemination onto the target platform.
- *Abstract algorithm specification* - modeling and specification language that abstracts away from a platform specific and language specific programming language.

3.1.2 Software Architecture

In order to allow further improvements and to support as many target hardware and software configurations, IDS must have a *modular software architecture* which allows to be easily extended in its functionality.

3.1.3 Target Application

Since the target applications may vary from one design project to the other in terms of capabilities of the embedded platform(sensor and actuator configuration), IDS should be able to cope with that as much as possible. To do so, IDS needs:

- Support for *distributed control* by means of spatial computing constructs.
- *Generalized system behavior description* - a behavior specification should be universal for all the supported target platforms. That means that a behavior description will have the same outcome for all the target platforms independently from being Netlogo, Proto or eLua.
- *Target application customization* - IDS should support embedded platform customization in terms of hardware configuration(sensors and actuators).
- *Hardware platform independent* - IDS should be able to support a variety of embedded platforms.
- *Lack of domain specific knowledge* - IDS should hide as much as possible all the technological aspects from the end-users.

Chapter 4

Implementation

This chapter describes the implementation aspects of the two main contributions of the thesis: Interactive Design Studio(IDS) and eLua VM for spatial computing. In Section 4.1 we discuss the IDS software framework and its main components. In Section 4.2 we show the adaptation and extension of eLua VM to the spatial computing paradigm.

4.1 Interactive Design Studio - IDS

In order to tackle the aforementioned requirements(Chapter 3) we designed IDS. IDS has two design goals. The first is the high-level behavior specification aspect which serves as an interface to the non-IT specialists community, while the second is the mapping and translation of these high-level specification onto binary code for the specific hardware platforms.

In Figure 4.1 we show the block diagram of IDS. IDS consists of four main components: *GUI*, *CodeGenerator*, *DeckSim*, and the *Embedded Software platform* which in our case is represented by *eLuaVM* [24].

The diagram shows the four components that correspond to the four stages of the *design process*. In the following sections we discuss each component and its implementation. Moreover, we show how it meets the requirements of Section 3.1.

4.1.1 Graphical User Interface - GUI

From the framework perspective, the GUI represents the component which serves as an interface to the designer for the ProtoSpace user experience creation. The design of the graphical user interface(GUI) is the outcome of the collaboration between the correspondents of the Faculty of Industrial Design and EEMCS of the TU Delft. The design as well as its concept are to be credited to Sandro Macchioli, MSc student at the Industrial Design Faculty. In Figure 4.2 you can see the structure of the GUI. It is a concept

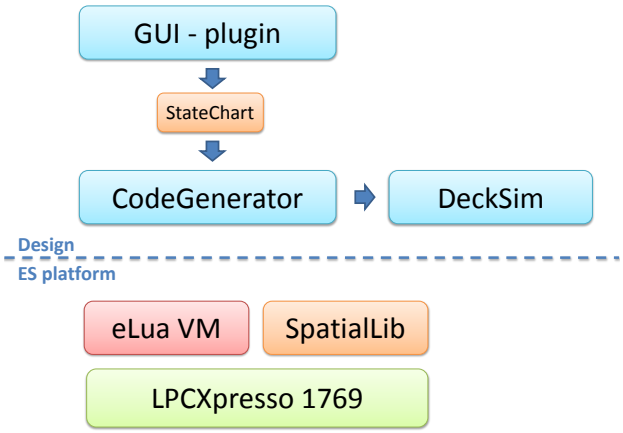


Figure 4.1: Block diagram of IDS framework

on how a possible user interface can be used for specifying a tile's behavior in the protoDeck floor.

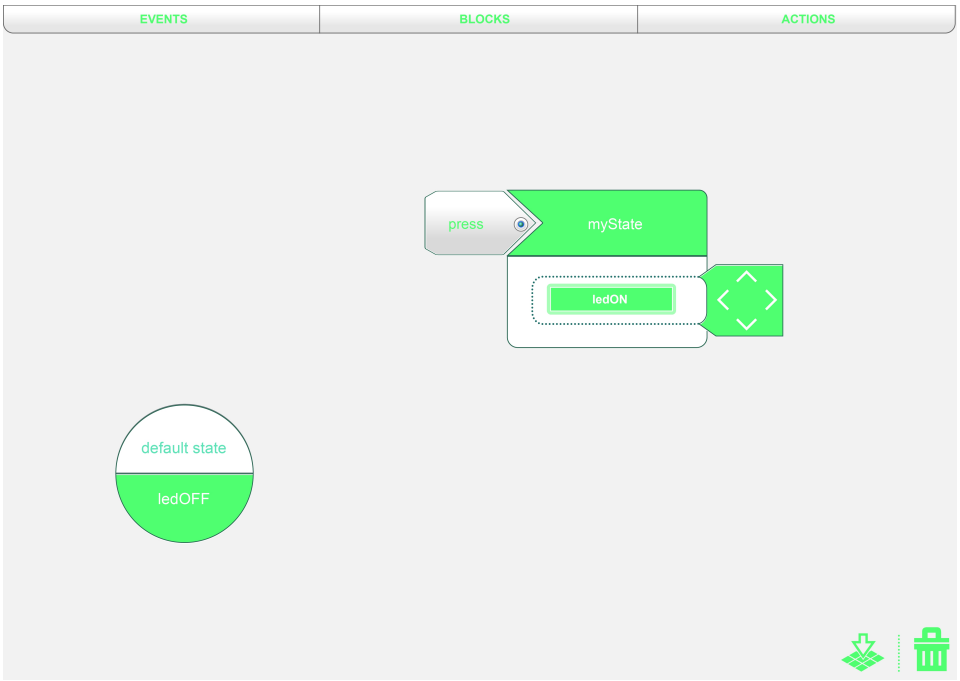


Figure 4.2: Concept GUI Industrial Design collaboration

Since the described GUI is only a concept, the actual GUI consists of a graphical state chart editor [41]. It allows non-IT experts to design a state chart representing the desired behavior of individual tiles. The state chart describes the program state transitions. The ultimate ambition is to

design a user friendly and easy to use GUI plug-in for an already existing modeling and design software such as Grasshopper. The plug-in should allow the specification of system level and node level behaviors that will hide the cumbersome design of a state chart.

A commonality of the aforementioned GUIs is the fact that they produce an SCXML, structured according to the W3C State Chart extensible Markup Language(SCXML). The SCXML is discussed in more detail in the following Section.

4.1.2 StateChart - XML for spatial computing

As stated in Section 3.1.1 we need a way to abstract algorithm specification that is independent from any programming language. For that purpose the choice was to use the StateChart - XML format. However, the default SCXML format is not suitable for our purpose. We thereby decide to extend the standard with custom fields which would facilitate its usage for spatial computing purposes. An example code snippet is shown in Figure 4.3.

```
<scxml initial="ledOFF" version="0.9" xmlns="http://www.w3.org/2005/07/scxml" xmlns:my="http://my.custom-actions.domain/CUSTOM"><!
 <state id="ledON"><!--   node-size-and-position x=340.0 y=240.0 w=140.0 h=50.0  -->
  <onentry>
      <if cond="sense1">
      <my:statevariable name="becomeSource"></my:statevariable>
      <my:actuator name="actuator1" action="1" args="LED hopcount 0 100"></my:actuator>
      </if>
    <if cond="minNeighborHopcount &lt; hopcount">
      <my:statevariable name="getHopcount"></my:statevariable>
      <my:actuator name="actuator1" action="1" args="colorMinHopNeighbor hopcount 0 100"></my:actuator>
   </if>
  </onentry>
  <transition cond="minNeighborHopcount &lt; hopcount"><!--   edge-path [ledON]  x=190.0 y=-60.0  --></transition>
  <transition cond="sense1"><!--   edge-path [ledON]  x=60.0 y=10.0  --></transition>
 </state>
 <state id="ledOFF"><!--   node-size-and-position x=10.0 y=250.0 w=150.0 h=40.0  -->
  <transition event="sense1" target="ledON"><!--   edge-path [ledON]  x=250.0 y=180.0  --></transition>
  <transition cond="minNeighborHopcount &lt; hopcount" target="ledON"><!--   edge-path [ledON]  x=230.0 y=340.0  --></transition>
 </state>
</scxml>
```

Figure 4.3: SCXML example

We extended the format in order to be able to specify actions that are either state variables changes or actions performed by actuators of the target platform. The rationale behind the use of a state chart representation for the tile's behavior is the following. A state chart is based on automata with the addition of hierarchical model support as well as concurrency. In addition, it includes a limited way of specifying time [40] which is enough for our purposes. The system under design is reactive and its evolution is based on local events and conditions. These are all aspects that can be modeled with a state chart. Another important reason to use state charts is an implication of being an extension of a finite state machine [40]: it is widely acknowledged that a FSM can represent any algorithm abstracting away from a specific programming language. In fact, a state machine can be described and ma-

nipulated with ordinary mathematics; therefore, they provide an uniform way to describe computation with simple mathematics [36].

4.1.3 CodeGenerator

CodeGenerator is the core component of IDS. It is a Java based tool that parses a SCXML file and produces platform specific code. In our case, the supported languages are *Netlogo*[51], *Proto*[13], and *Lua*[30]. In order to be able to support as many target platforms as possible, the SCXML has a set of events, conditions, and actions that are mapped to specific platform code. These specifics are stored in platform specific XML library files(Lib.xml). They implement spatial computing primitives that are used by the applications. The CodeGenerator module has a modular software architecture that aims to be fully extendible with new modules and add-ons to further improve its capabilities. In Figure 4.4 we show the CodeGenerator diagram showing its modules and the respective inputs/outputs.

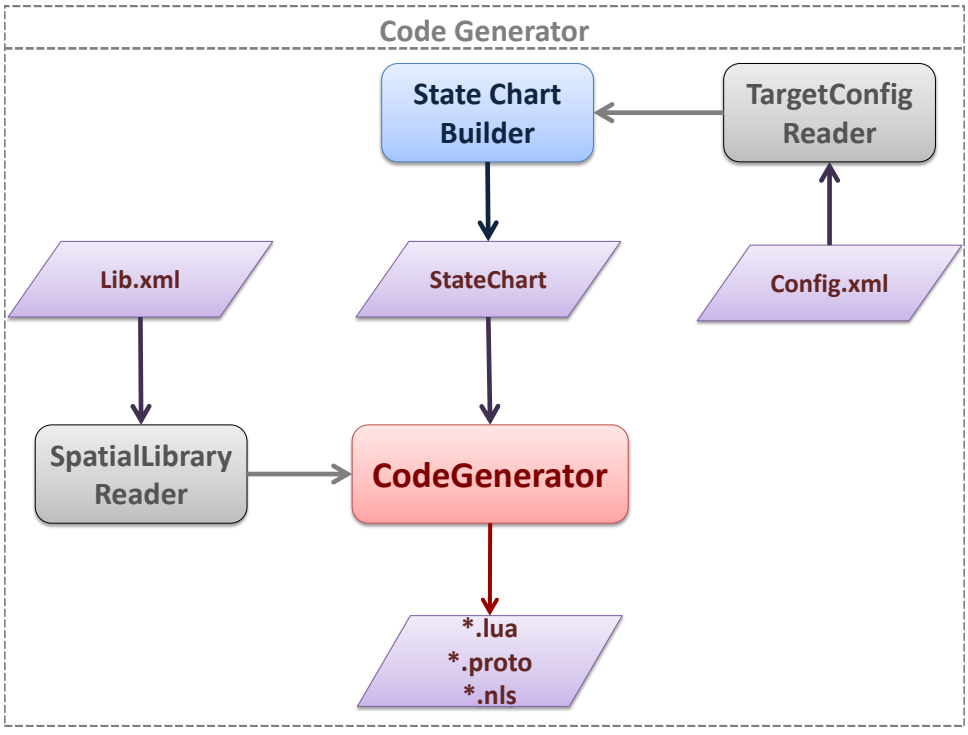


Figure 4.4: CodeGenerator modules and data flow diagram

CodeGenerator has four modules: *SpatialLibrary Reader*, *TargetConfig Reader*, *StateChart Builder*, and a platform specific *CodeGenerator* module. The SpatialLibrary and the TargetConfig Reader are both an XML file interpretation module which respectively takes as input Lib.xml and Config.xml. The library XML file contains the platform specific spatial

computing primitives and the relative utility functions. The platform specific configuration file(Config.xml) specifies the embedded target platform configuration in terms of sensors and actuators. For each actuator a set of actuation modalities are specified. The StateChart Builder takes as input the *.scxml and the output of the configuration file reader. The output is a state chart object containing the SCXML file translation with the corresponding platform specific changes. The output of both the StateChart and the library reader module are fed into the code generation module. The CodeGenerator module differs for each target platform: there exists a module for Netlogo, Proto, and Lua. The CodeGenerator ultimately translates the SCXML and platform specific directives into the target platform's code. The translation process differs for each platform since they differ in syntax and/or in programming paradigm(Proto is functional;Lua and Netlogo imperative). The module maps the scxml specifications with the platform specific library and creates the syntactically correct platform code. Besides the application code, the CodeGenerator creates the spatial library for eLua and Proto. The focus was mostly on the eLua module for which we provide a customizable way to generate the spatial library. The library file specifies the data to be exchanged between nodes and the means to interpret that data(more detail in Section 4.3).

4.1.4 DeckSim

DeckSim is a Netlogo based simulator for *protoDeck*. More precisely, it models the tiles' topology and their connectivity. Each "hosts" a software agent which can communicate with its immediate neighbors of a cellular automata like topology. Each agent is equipped with a LED and a pressure sensor.

The simulator's purposes are manifold:

- investigate space-time evolution of the system
- test algorithmic correctness
- visual feedback - emergent behavior
- test tool for Industrial Design colleagues

In Table 4.1 we show the main characteristics of the simulator.

The iterative design process consists of a design and test cycles that are usually performed during sketching or prototyping. By providing a simulator which models the target system, protoDeck, a non-IT specialist is able to iterate from the algorithm specification phase to the test phase and back before deploying the code. In Figure 4.5 we show the DeckSim architecture. The simulator uses the *StateChart.nls* which contains the application generated from the SCXML. The application uses the spatial computing library,

Context	Type	Description
Time	ticks	Time is represented by an internal counter for each agent.
Execution	synch. round	Agents' state is stored before each round. New state takes effect at the end of the round.
Neighborhood	mesh-like	Each agent has four neighbors(except border agents).
Distance-metric	hopcount	Hopcount from a pressure sensor triggered source used as distance metric.
Events	discrete events	Possible events are: Pressure sensor pressed or particular neighborhood states.

Table 4.1: Simulation environment characteristics

SpatialLib.nls. As a result, we are able to test the generated code in terms of its algorithmic correctness and its space-time behavior. In Figure 4.6 we see the front-end of the simulator in the form of a Java applet which can be embedded in a website.

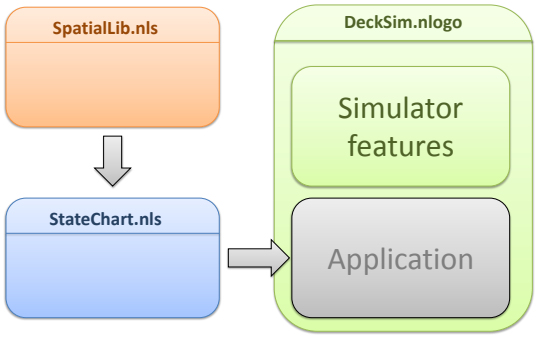


Figure 4.5: DeckSim Architecture

Even though the simulator tries to model the system as true to reality as possible, it is obvious that the shift to the real hardware platform is not straightforward in some cases. In Section 6.4 we discuss the issues that might emerge when this shift happens.

In our design process, our Industrial Design collaborator used the simulator for testing purposes. In this test cases the simulator was remotely accessed with an iPad and used as an interactive test medium to explore

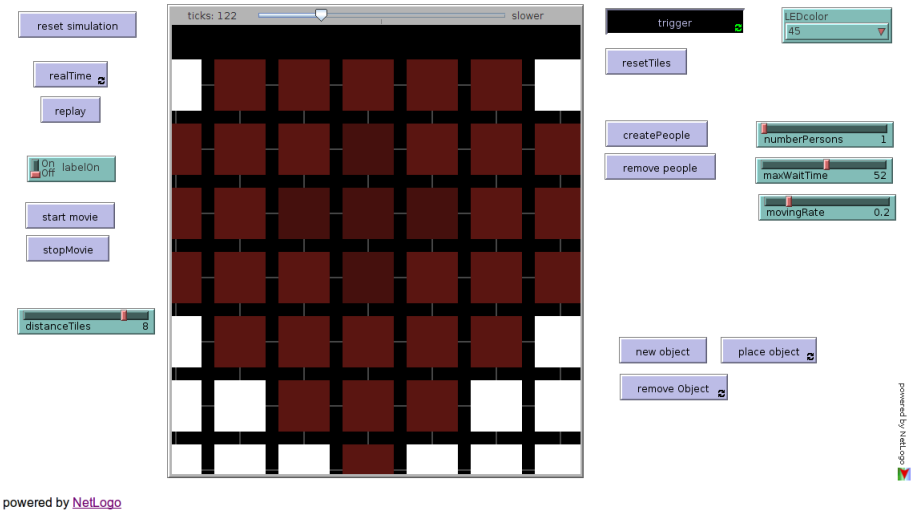


Figure 4.6: DeckSim front-end

and assess the GUI's capabilities(example of "Wizard of Oz" technique, Figure 4.7).

4.1.5 Conclusions

In Section 3.1 we listed a series of requirements that in our opinion are necessary for our goal. In this paragraph we show how IDS tries to tackle those requirements. The code generation module aims to tackle the requirements discussed in Section 3.1. By dividing the code generation process in separate module tasks we tackle the need for a *modular software architecture* that is easily extendible and customizable. Moreover the use of xml standard for the configuration and library files gives the possibility to easily bind them with a GUI. By using platform specific configuration files we are able to drop constraints that are platform and target application related. More precisely, we are able to tackle the requirement of *Target application customization* with the use of platform specific configuration files where we configure the embedded target platform's sensor and actuator configuration and features. The code generation itself avoids *domain specific knowledge* in embedded software from the end-user. Finally, we provide a simulator, DeckSim, which models the floors behavior that contributes in speeding up the design iteration process.

4.2 SpatialeLua Platform

The last peace of the puzzle is SpatialeLua, an adaptation of eLua to the spatial computing paradigm. As mentioned in Section 2.6.2 eLua comes with

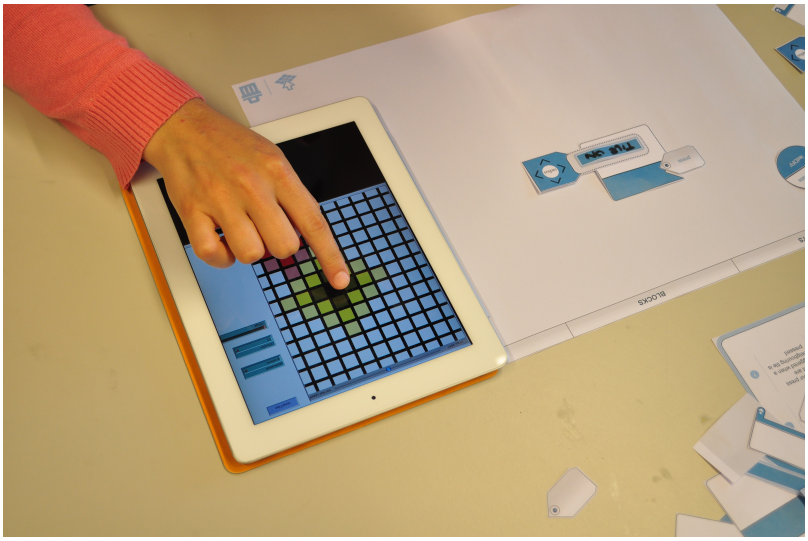


Figure 4.7: Example use case of DeckSim for the 'Wizard of Oz' technique

a series of supported embedded platforms. Out of the box there is a port to the Mbed LPC1768 platform which is similar to the used LPCXpresso 1769. Therefore, we adjust the peripheral's port mappings and update the drivers to port eLua to the LPCXpresso 1769. With that being done, we are able to run the eLua VM on our embedded platform. In order to make eLua 'spatial' we have to provide the following features:

Neighborhood: Neighborhood discovery and information retrieval.

Spatialib: Spatial Computing primitives.

ViralCode dissemination Code dissemination features to allow rapid prototyping and application updates.

Communication Inter-node communication protocol.

Each requirement can be mapped into the corresponding building block. In Figure 4.8 we show the software architecture and the relation between the various blocks of *SpatialeLua*. In the following we describe each layer.

4.2.1 Platform Layer

The platform layer consist of the platform specific drivers. In our implementation we used the platform drivers supported by eLua for the LPC17xx series. Since the provided eLua UART driver did not support buffered transmission and reception we added this feature to eLua's platform interface. In more detail, the UART driver uses a software FIFO buffer to allow asynchronous sending and receiving of bytes. The mechanism is interrupt-based:

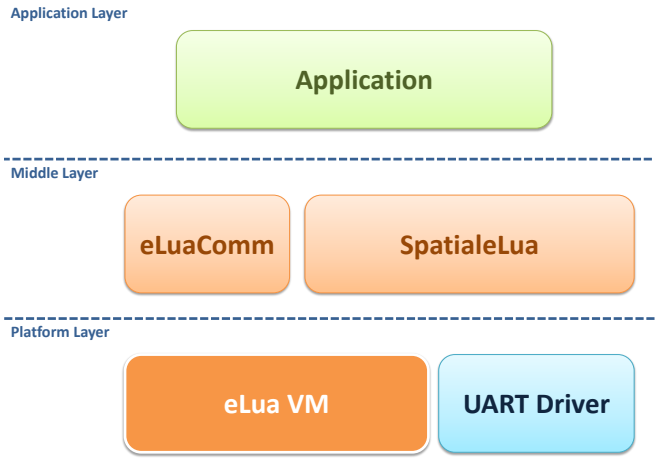


Figure 4.8: SpatialeLua platform

bytes are placed into a software FIFO buffer by the interrupt routine. Transmission is initiated by placing the bytes into the hardware buffer until it is full. When the buffer is empty again an interrupt is raised which triggers the interrupt routine to check whether the software buffer contains more bytes to send. If that is the case, sending is restarted until the software FIFO is empty.

4.2.2 Middleware Layer

The Middleware Layer consists of two components: eLuaComm library, SpatialeLua VM. The main task of the Middleware Layer is to provide eLua with spatial capabilities. These are achieved by providing the necessary data structures and manipulation means, and an adequate communication protocol. In this Section, we first describe the software components responsible for the neighborhood data manipulation and, afterwards, discuss the communication protocol.

eLuaComm supplies the VM with neighborhood information in a best-effort way. The basic services of eLuaComm are reading the raw data byte stream coming from the UART driver, scan the software buffer for valid packages, and deliver them to the VM. The library is implemented in C and it exposes an API for sending and receiving packets respectively from and to the eLua VM. Further details about packet structure and the communication protocol follows in Section 4.2.2.

SpatialeLua VM, eLua VM for spatial computing, provides the features that are necessary for the spatial computing paradigm. We adopt a modular approach and divide SpatialeLua in the following sub-components (Figure 4.9): Neighborhood, Spatialib, and ViralCode.

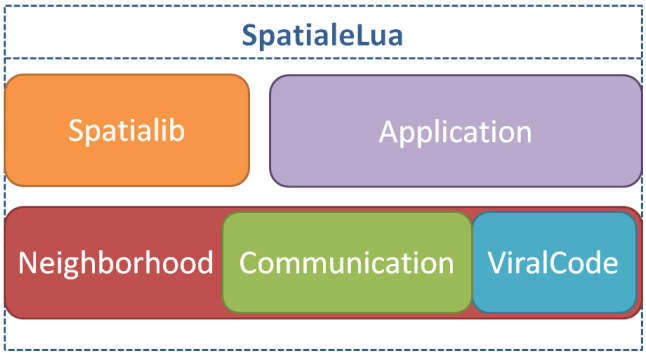


Figure 4.9: SpatialeLua modules

The *Neighborhood* module is connected to the Communication module since it is the core part for providing spatial capabilities. The main responsibilities of the Neighborhood module are: maintain neighborhood information, neighborhood discovery, and packet processing. This includes the active participation in the communication protocol. The module maintains a data structure containing the neighbor data and the temporary data structures used for the viral code dissemination. At each virtual machine execution round the UART buffers are checked for the availability of valid packets. They are processed and the payload(state variables) is stored. Neighborhood discovery is performed by using time-out mechanism based on the last time neighbor information was received.

The *Spatialib* is the spatial library module generated by the IDS Code-Generator. Spatialib maintains the node specific exports which represent the information that is sent to the neighbors. In addition, all the spatial primitives(see Section 4.3) are declared in this module. Finally, the module also provides access to sensing, actuating, and local clock access of the node.

Communication Protocol

The communication protocol is implemented in Lua and provides the following features:

- neighborhood discovery
- state updates
- code updates

Neighborhood discovery is achieved in two ways: beaconing and state/code updates. To avoid the use of erroneous neighborhood information caused by faulty communication links we constantly update neighborhood information. The mechanism is based on a periodic beaconing. If a node has not received

any beacon or valid packet within a specified time interval the neighbor is removed from the neighborhood data structure.

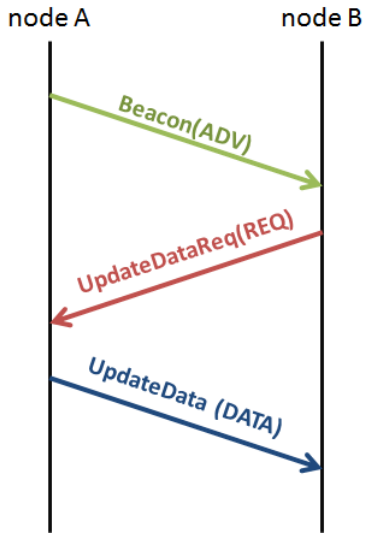


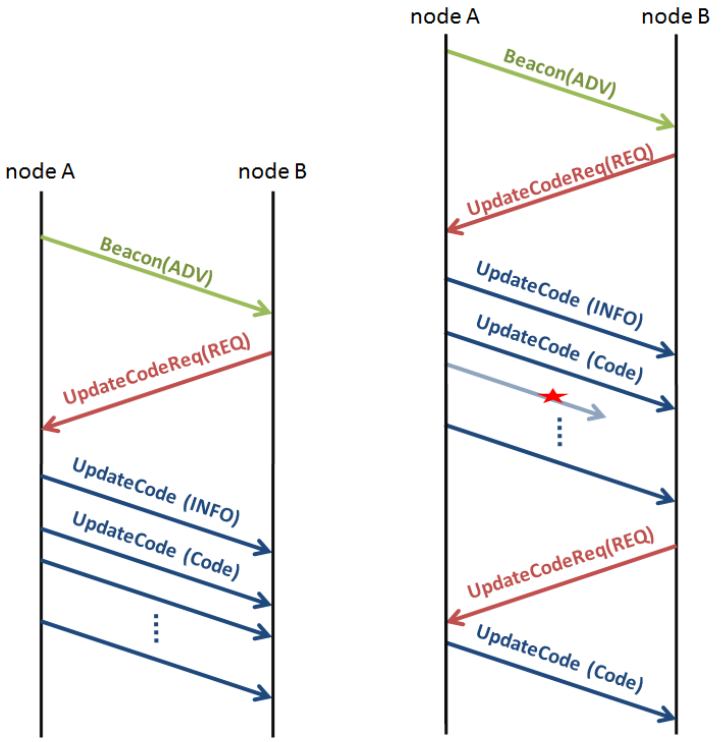
Figure 4.10: State update communication protocol

For both neighbor state and code updates dissemination we employ a negotiation based protocol [34]. In Figure 4.10 we show the negotiation phases between two nodes. Upon an ADV packet received, indicating that the neighbors state or code is new, the a node will send a REQ packet to the neighbor. Upon reception of that packet the neighbor will send back a DATA packet. The mechanism is slightly modified upon state/code update of a node. In that case, the ADV packet is replaced by a DATA packet. Only if this packet is missed the default negotiation mechanism takes place.

The code dissemination process is also negotiation based and is inspired by Trickle [37]. Upon receiving of an ADV packet a node(B) sends REQ packet. The sender node starts the DATA transmission upon reception of the REQ packet. The first DATA packet contains as payload the number of chunks the application is split. In that way, a node knows when the code update has been completed. If one or more DATA packets are lost, the receiver node(after a time-out) sends a REQ packet having as payload the indexes of the missing packets. In Figure 4.11 we show the protocol in both ideal(Figure 4.11a) and faulty case(Figure 4.11b).

Packet Formats

In Table 4.2 we show the UART packet format. The packet header is the same for each packet type. In contrast, the Payload section varies based on the packet type. An eLua packet can have the following types: Beacon, ExportUpdateReq, ExportUpdateData, CodeUpdateReq, CodeUpdateData,



(a) Correct (b) Missed code chunk packet

Figure 4.11: Code update protocol

and Invalid. PacketID, Application ID, and Export ID are used to indicate the sequence number of respectively the packet, the application code, and the information. These are used by the protocol to advert the current state of the sender node. The Payload size is variable and contains information only for ExportUpdateData and CodeUpdateData packet types. Each packet contains two bytes for the CRC that gets calculated from the packets content.

4.2.3 Application Layer

The Application Layer consists of the Lua code that is generated by the IDS-CodeGenerator. The application is a direct mapping of a state chart structure which takes advantage of the spatial capabilities provided by the underlying layers.

Variable	Size(bits)	Notes
Packet type	3	Beacon / ExportUpdateReq / ExportUpdateData / CodeUpdateReq / CodeUpdateData / Invalid
Packet ID	16	
Application ID	16	
Export ID	16	
Source Address	16	
Destination Address	16	
Payload Size	16	
Payload	*variable*	
CRC	32	

Table 4.2: Packet Format

4.3 Spatial Computing primitives

The main prerequisite for spatial computing is implicit local neighborhood communication and space abstraction. The former is achieved via the UART based communication and the Neighborhood module of SpatialeLua VM. The space abstractions as well as the neighborhood states are handled in the *spatialib* which is generated by the IDS-CodeGenrator. The spatialib is defined at design time by means of a configuration file. The provided spatialib targets protoDeck and provides the necessary features for demonstrating the feasibility of applications involving time and spatial primitives(Section 6.1).

In Table 4.3 we show the state variables identifying a specific state of the node; while, in Table 4.4 we show the provided neighborhood functions.

Variable	Values	Description
hopcount	[0-255]	Indicates the hops away from a source
flash	{0,1}	Indicates whether LED is on or off
sensor1	{0,1}	Indicates whether tile is pressed or not

Table 4.3: State Variables

The data exchanged in the neighborhood identifies node state variables, while the functions are used to extract the desired information from that data. By providing the aforementioned state variables and functions we prove that *SpatialeLua* is suitable for spatial computing.

With our framework we want to make spatialib customizable at design

Function name	**Description**
min/maxNeighborHopcount	Returns the minimummaximum hopcount value in the neighborhood
neighborsPressed	Returns the number of neighbors that have the pressure sensor active
neighborsLED_ON	Returns the number of neighbors which have their LED on

Table 4.4: Neighborhood state functions

time. In fact, we aim to provide a mean to exploit the software architecture of SpatialeLua VM to employ user-defined spatial abstractions. We make this possible by providing IDS with a configuration file for specifying state variables and spatial functions(in a specific domain specific language). In on our case for instance, we use the hopcount state variable to create a gradient based spatial abstraction. The provided functions in turn use this information to determine specific neighborhood states. In conclusion, we aim to ease the specification of other spatial abstractions inspired from mathematics, biology or chemistry. By doing that we do not limit the framework to use a fixed *modus operandi*, but we render the framework as general as possible.

Chapter 5

Applications

In this Chapter we present two applications for both *IDS* and *SpatialeLua*: Phototropia and Spotlight. The two projects were realised in different stages of the thesis work. Phototropia in the very early phase of the eLua VM platform; while, spotlight is an application targeting protoDeck and is used as a test case for the whole IDS design tool-chain. In the following we first present the Phototropia project experience. Afterwards, we present the Spotlight application for protoDeck.

5.1 Phototropia Project

Phototropia is a project of the Chair of Computer Aided Architectural Design (CAAD) at the Faculty of Architecture, ETH Zurich. Our collaboration consisted in providing the interactive experience by means of our eLua based embedded platform. As said before, the project was in the early stage of the eLua VM porting and it was the perfect challenge to face in that stage to extrapolate the necessary requirements for the thesis project. The installation consisted of two parts: custom active components and embedded devices. The active components where three: light-emitting electroluminescent displays, shape-changing electro-active polymers and energy-creating dye-sensitized solar cells, while the main structure was composed of eco-friendly bioplastics. As embedded devices we used LPCXpresso 1769 connected in a line. In order to provide interactive behavior we equipped each board with a proximity sensor in order to detect people approaching the installation and trigger an action. In Figure 5.1 we see the installation.

5.1.1 Setup and Application

The installation is equipped with six LPCXpresso boards running the eLua VM in its early stage. By that time *SpatialeLua* did not exist. We used eLua VM out-of-the-box after porting it to the LPC1769. The boards have

Figure 5.1: Phototropia installation

a line topology and each node is responsible for a region of the installation. The application for the installation consists in creating a wave pattern starting from a triggered proximity sensor. The algorithm is very simple and consists in notifying the neighbors of their state consisting in specifying whether their actuators are active or not, and the waveID. The waveID is used to start a new wave and the direction of the wave(from greater waveID to smaller). The wave pattern is created by delaying the information transmission. A wave can be a wave of 'OFF' states as well as 'ON' states. Since the eLua UART driver does not provide buffering mechanism(asynchronous sending/receiving) we adopt a polling strategy. This requires a careful timing of the application's timings. In Figure 5.2 we show the time subdivision of the application loop.

In order to not miss packets we decide to reserve the main part of the round to the UART's polling. The timings are approximate timings. This setup required a lot of tweaking and empirical tests since accurate measurements were not possible at that stage.

5.1.2 Experience

The Phototropia experience resulted very useful to determine the requirements of our framework and the extensions to be made on the eLua VM

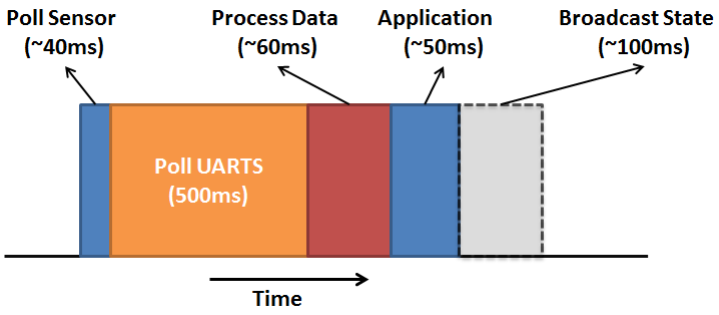
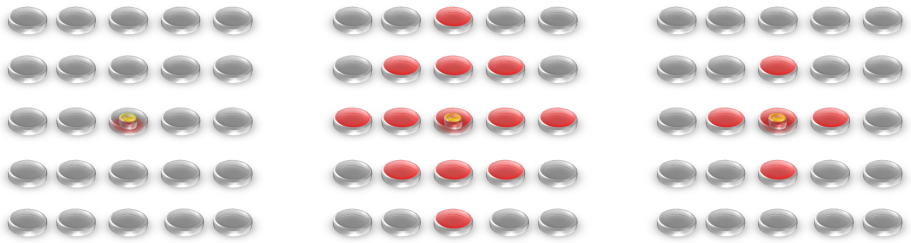


Figure 5.2: Application round timings

(Chapter 3 and Chapter 4): buffered asynchronous transmission/reception over UART, sensor calibration mechanism, and efficient neighborhood discovery and management module.

5.2 Spotlight on protoDeck

In conclusion of the thesis work, we use protoDeck to test IDS and SpatialeLua for the purpose of interactive design. For that we propose the application called *Spotlight*. As spotlight we mean a pulsating light circle activated by a node's pressure sensor. In Figure 5.3 we see the light pattern sequence.



(a) (b) (c)

Figure 5.3: Spotlight light pattern

In Figure 5.3a we show the initial state of the pattern with all tiles turned off. Figure 5.3b shows the pattern created when the central tile is pressed. In that precise moment the tiles light on with a specific delay until the limit hopcount is reached. Figure 5.3b shows the maximum circle radius after the central tile was triggered. The circle then decreases in steps of one hope at the time until the minimum one-hop distance from the source(Figure 5.3c). In that state there will be a small pulsating circle around the central tile.

The Spotlight algorithm is the result of using the whole design tool-chain. We use the State Chart editor and DeckSim to test the generated code. Afterwards, we deploy the generated Lua code onto protoDeck. As expected, the visual feedback differs when shifting from simulation to protoDeck. The main reason is the different timing properties between the simulator and the hardware platform. Nevertheless, the light pattern is quite similar.

Algorithm 1 SpotlightSC Algorithm

Require: $hopcount = 255,\ flash = OFF$

```
1:  state ⇐ start
2:  while true do
3:      if state = start then
4:          if pressed then
5:              hopcount ⇐ 0
6:              flash ⇐ ON
7:              LEDIntensity ⇐ hopcount
8:              nextState ⇐ fade
9:          end if
10:         if minNeighborHopcount < radius then
11:             nextState ⇐ neighbor
12:         end if
13:     end if
14:     if state = neighbor then
15:         wait for waitInterval
16:         hopcount ⇐ minNeighborHopcount + 1
17:         nextState ⇐ fade
18:     end if
19:     if state = fade then
20:         if #rounds mod rate = 0 then
21:             hopcount ⇐ hopcount + 1
22:             LEDIntensity ⇐ hopcount
23:         end if
24:         if clock > fadePeriod then
25:             resetStateVariables
26:             nextState ⇐ wait
27:         end if
28:     end if
29:     if state = wait then
30:         wait for waitInterval
31:         nextState ⇐ start
32:     end if
33:     state ⇐ nextState
34: end while
```

In Algorithm 1 we show the Spotlight state chart algorithm. The algorithm makes use of node internal counters and timers. It consists of four states: *start, neighbor, fade*, *wait*. All the nodes start in the *start* state and hopcount equal to 255. Whenever the pressure sensor is pressed the next state is *fade*, the node becomes a source(hopcount is 0), and the LED intensity is set as a function of the hopcount(highest). While, if a node detects the presence of a neighbor with hopcount less than *radius* the next state is *neighbor*. This ensures the light pattern to span to within the radius range(Figure 5.3b). The nodes in the *neighbor* state wait for *waitInterval* determining the delay by which the light pulse propagates. When this interval expires the node is in state *fade* and updates its hopcount. A node stays in the *fade* state for a fixed *fadeInterval*, after which the *wait* state is entered. In the fade state the hopcount is increased at a certain round *rate* updating accordingly the LED intensity. This gives the pulsating circle effect.

In conclusion, we show that our framework and the choice of using state chart model for specifying algorithms is able to generate non trivial algorithms and light patterns on protoDeck.

Chapter 6

Experimental Results

In this chapter we investigate via experiments the suitability of the previously described framework for the purpose of interactive design by means of spatial computing. Moreover, this chapter provides the necessary information to answer our research questions. To do that we perform several experiments. We first show interactive applications that use spatial computing primitives(Section 6.1). In Section 6.2 we benchmark *SpatialeLua* memory consumption and performance with respect to the spatial applications and neighborhood configuration. In Section 6.4 we discuss the possible issues that exist when shifting from simulation to the hardware platform. In the end, we discuss our findings and the suitability of our framework.

6.1 Spatial Applications

For designing interactive environments consisting of a space filled with a network of embedded devices, we strongly believe that using spatial computing is a viable solution. We show example applications which demonstrate the use of spatial, time, and spatial-time primitives for interactive design. We use two version for each application: state chart version and a manually implemented version. In the following, we present the *Firefly*, *Gradient* and *Spotlight* application which respectively are used to show time, spatial, and time-spatial capabilities of our platform. Finally, we compare the two versions of the applications.

6.1.1 Time primitive - Firefly

In order to prove the temporal behavior of our platform we implement a slightly modified Firefly algorithm of the *Netlogo Firefly Algorithm [50]*(NFA). The algorithm enables internal clock synchronization in a distributed network of nodes with only local communication. Each node maintains a time period(*period*) in which it glows for a certain time interval (*glowTime*). Syn-

chronization is achieved by each node by resetting its internal clock (*clock*) to the end of the glowing interval whenever it detects that at least one neighbor is glowing. This keeps going until the nodes are synchronized. The NFA is not suitable for nodes with low connectivity as it is in our case (example line topology in Figure 6.1) [48].

Figure 6.1: Problematic node topology for NFA

In fact, it happens to node b to reset continuously when node a and c flash in non-overlapping periods. To avoid that behavior we use a modified version of the NFA which uses a threshold(*listenTime*) in which a node listens for flashing neighbors. When this interval expires a node is able to finish its period and start its glow interval.

6.1.2 Spatial primitive - Gradient

To test the spatial behavior we use an application that builds a hopcount based gradient across the network. A pressure sensor on the node triggers the gradient creation. Initially, every node has an infinite hopcount value. As soon as a node's pressure sensor is activated, it sets its hopcount to zero. As a consequence, a node that detects a node in the neighborhood with a lower hopcount than its own will set its hopcount to the smallest hopcount in the neighborhood incremented by one. In that way, a distance measure from a source is established. The gradient primitive is also a great building block for interactive applications since it detects the presence of people triggering a sensor. This might trigger every sort of action or effects in an interactive environment. For example, it might show the direction to a certain source by means of a gradient of light intensity on the protoDeck floor.

6.1.3 Time-Spatial primitive - Spotlight

Applications for interactive environment usually require both time and spatial constructs. A perfect example is the previously presented application *Spotlight* which makes use of time and spatial constructs to create a spotlight like pattern (Section 5.2).

6.1.4 State Chart Generated vs Manual Version

As mentioned before, IDS produces executable code having a state chart structure for Netlogo, Lua, and Proto. We consider the Lua generated code that runs on *SpatialeLua VM*. In Figure 6.2 and Table 6.1 we show the comparison between state chart and manual version of the applications in terms

of its script size. We use the script size as an indicator for the application complexity. We can easily see that the script size of the IDS generated applications is much larger than the manual implementations. The reason is that a state structure is less compact than a manually implemented algorithm. However, as we will see later(Section 6.2) a smaller script size is not synonym for faster execution time since it highly depends on branching. In fact, state charts are characterized by a highly branched structure: only the code of a specific state is executed at each round. As a consequence, the expected linearity between script size and execution time is dropped.

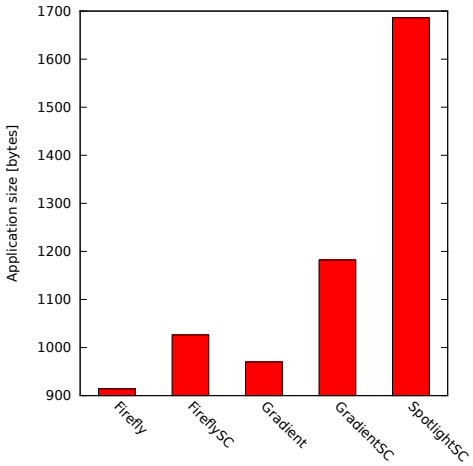


Figure 6.2: Script size comparison for Lua applications

Table 6.1: Application script size

Application	**Size [bytes]**
Firefly	914
FireflySC	1026
Gradient	970
GradientSC	1182
SpotlightSC	1686

6.2 Benchmarks

In order to answer the research question of whether it is possible to use spatial computing in off-the-shelf virtual machines we perform benchmark tests for memory consumption and performance on our embedded platform. In this section, we present and discuss the benchmark results. First, we show the memory consumption in case of both statically and dynamically allocated memory. Afterwards, we present a performance analysis in terms of execution time. For our tests we use the aforementioned spatial applications(state chart version and manual version). We compare both memory consumption and performance with the Proto platform of the *Snowdrop* project [48]. Since the latter is tailored for spatial computing application, we decide to use it as the reference platform.

6.2.1 Memory Consumption

The LPC1769 board comes with 64 kBytes of non-consecutive RAM consisting of two chunks of 32 kBytes. For *SpatialeLua* we use both chunks. We ran the same experiments several times for more reliable results. This Section is organized as follows. We first show the statically and dynamically allocated memory. Afterwards, we investigate the dynamic memory consumption of each application as a function of the neighborhood size and/or number of state variables. Finally, we compare the results with the Proto platform and discuss the implications for our framework.

Statically Allocated Memory

Figure 6.3 shows the breakdown of the statically allocated memory. The total amount of statically allocated memory is 4708 bytes of which 3220 bytes for the .bss and 1488 bytes for the .data segment.

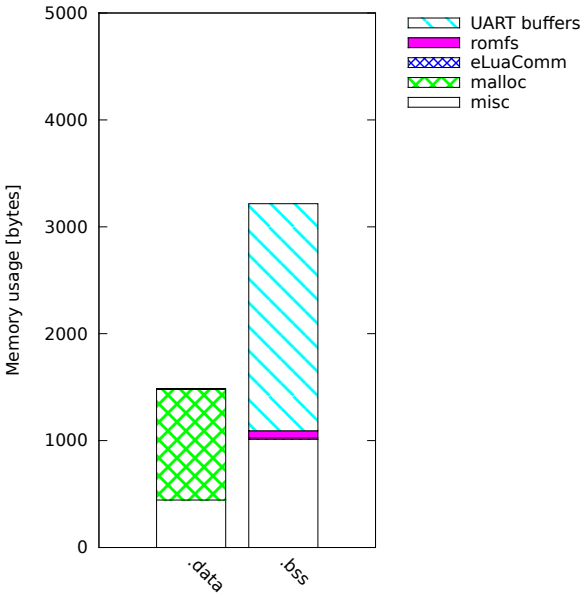


Figure 6.3: Static memory breakdown

The .bss is a data segment that contains all the uninitialized variables that are declared at the file level. In our case, it mainly contains both the transmission and the receiving buffer for each of the four UART ports(2048 bytes). Obviously, the reduction of the buffer size(256 bytes) for each UART could significantly decrease the amount of allocated memory. The .bss segment also contains the ROM filesystem(*romfs*) tables required by eLua(64 bytes). The *romfs* is a table which is used to load the modules after startup

of the VM. The remaining of the segment contains variables and data structures of eLua modules.

The .data segment contains all the global and static variables that are explicitly initialized with a value. In our case, most part of the .data segment is occupied by newlib's malloc implementation.

Dynamically Allocated Memory

In Figure 6.4 we see the maximum dynamic memory allocation breakdown for each application.

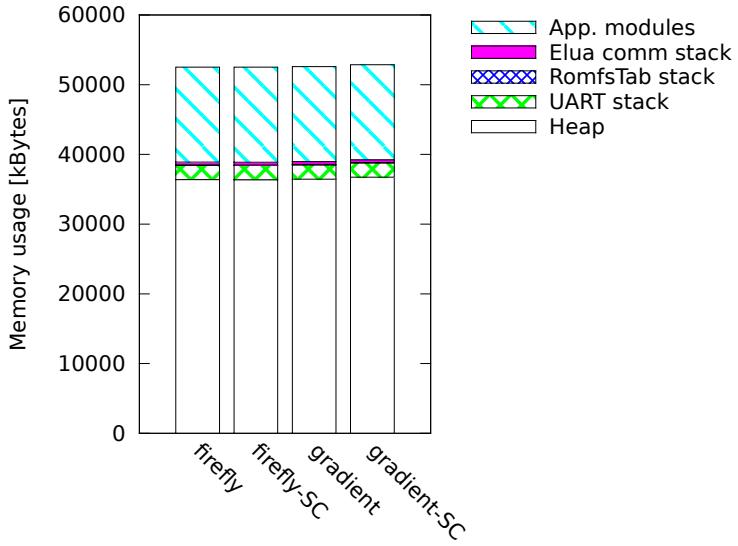
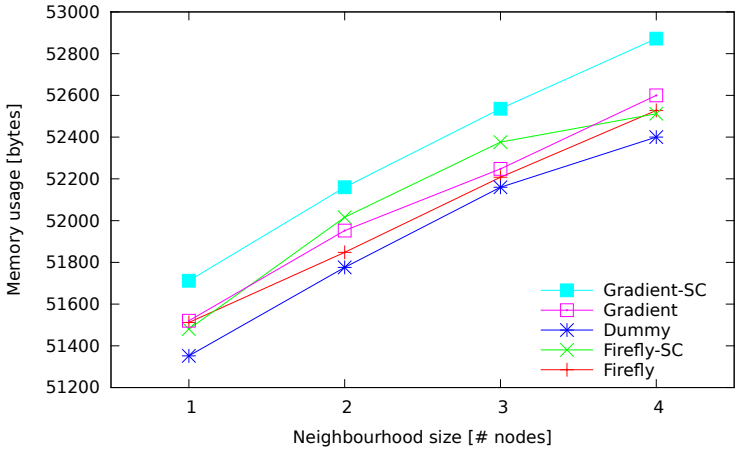


Figure 6.4: Dynamic memory consumption breakdown

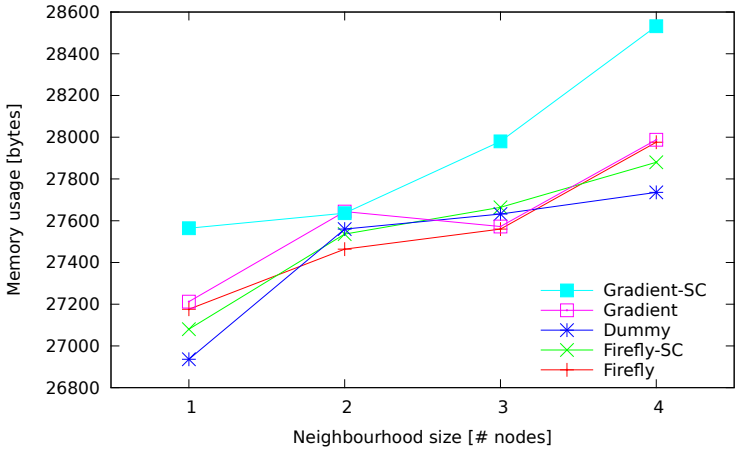
We see that all the application have approximately the same memory consumption. The only difference is the heap usage. That is as expected since each application uses different functions of the *"spatialib"* and therefore it account for more or less memory usage. However, the heap usage is similar and that is mostly due to the fact that the heap is used by the Lua interpreter and the *SpatialeLua* modules' data structures which do not differ from one application to the other. The application modules' byte-code consumes a substantial quantity of RAM (13632bytes). This further expands into the heap where the necessary data structure are allocated. In the following we show the influence of the neighborhood size and the number of exchanged state variables on the memory consumption. We show the overall memory usage allocated by malloc and in addition we show the memory usage of Lua. The overall memory consumption is used as indicator for the maximum memory needed for the application to run on the embedded platform.

Memory Consumption as function of the Neighborhood Size

In Figure 6.5 we show the memory consumption for the aforementioned applications. As expected, the dynamically allocated memory has a clear increasing linear trend: the memory allocation increases with the increase of the neighborhood size. That is more clear in Figure 6.5a where the overall memory consumption shows a linear trend for each application. We notice that the state chart versions require more memory compared to its manual version.



(a) System dynamic memory usage



(b) Lua dynamic memory usage

Figure 6.5: Dynamic memory usage with respect to the neighborhood size

A less clear behavior is observed in Figure 6.5b. In fact, Lua's memory consumption has still an increasing trend with the increase of the number

of neighbors but the slope for each step is irregular. The measurements are taken in the same runs but nevertheless they differ in terms of regularity. A possible explanation might reside in the garbage collection strategy of Lua which might take into account dead objects that have not been collected when the measurement was taken. However, the overall dynamic memory consumption is the most indicative and it shows that, increasing the neighborhood size by one, the memory consumption increases by a value that ranges from 300 to 400 bytes. In the future with a possible support for radio communication medium these findings might be indicative. In fact, this result signals that an increase in the neighborhood size is possible to a certain extent since the platform's memory is limited to 64 kBytes.

Memory Consumption with respect to the number of State Variables

Another aspect we investigate is the dynamic memory consumption as function of the number of state variables for each node. An increase in the number of state variables implies an increased packet payload and more information to be stored for each neighbor.

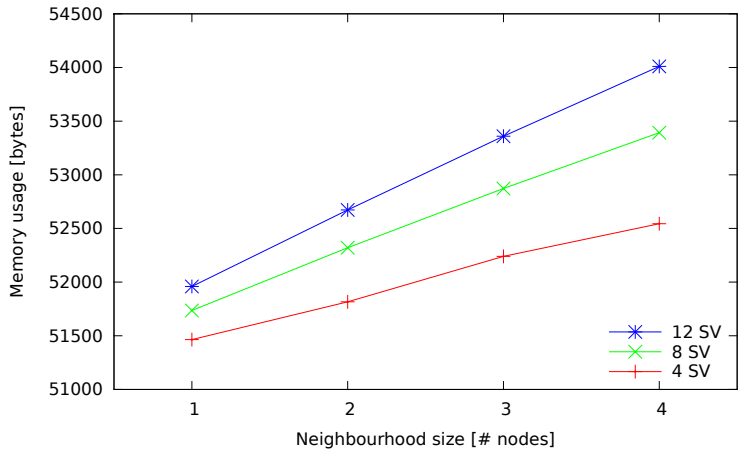


Figure 6.6: Overall dynamic memory consumption as a function of the neighborhood size and number of state variables.

We perform the measurements using the FireflySC application. As expected, increasing the number of state variables implies a greater memory consumption and a linear relation between memory consumption and neighborhood size(Figure 6.6). In Figure 6.7 we show the increase ratio of the memory consumption using as reference the one-node-neighborhood. For greater number of state variables, meaning a greater neighbor information table entry, corresponds a greater increase ratio. This is important in case of for future developments of the platform. The limitation to 64 kBytes of

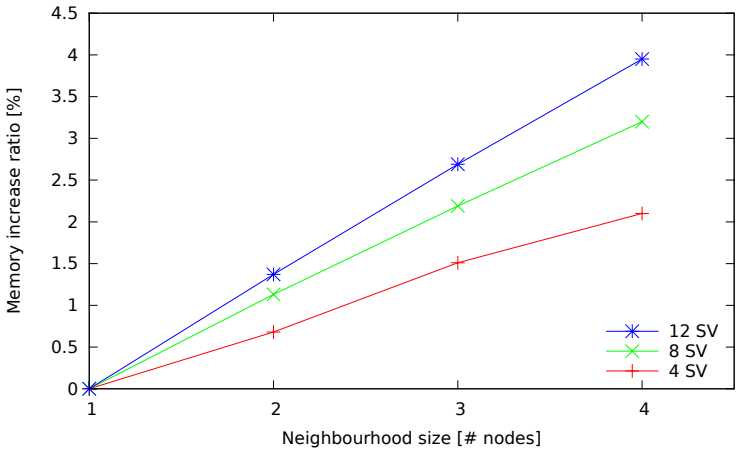


Figure 6.7: Dynamic memory consumption increase ratio.

RAM might become an issue when increasing both neighborhood size and the number of state variables.

Memory Consumption: eLua vs Proto

In this Section we compare *SpatialeLua*'s dynamic memory consumption to the Proto platform [48]. We compare the two platforms using the GradientSC and FireflySC applications. In Figure 6.8 we see that *SpatialeLua*'s memory consumption is one order of magnitude greater than Proto's in both cases. The exact values are shown in Table 6.2. The result is not surprising since elua is a general purpose virtual machine and the spatial capabilities are provided by modules which resides in RAM; while, Proto is a spatial computing tailored virtual machine providing spatial capabilities at the opcode level.

Platform	**GradientSC**	**FireflySC**
eLua	52872	52512
Proto	3920	4408

Table 6.2: Memory Consumption elua vs Proto.

However, it is interesting to see how the neighborhood size influences the memory usage. In Figure 6.9a we see the memory increase in bytes as a function of the neighborhood size for the Firefly application. The increase uses as reference the one-node-neighborhood. Both FireflySC and FireflySC in eLua consume more memory for all neighborhood sizes. On the other hand, if we compare the increase ratio of the memory consumption, using as reference always the one-node-neighborhood, we see that elua has lower increase ratios than Proto(Figure 6.9b). However, that does not tell the

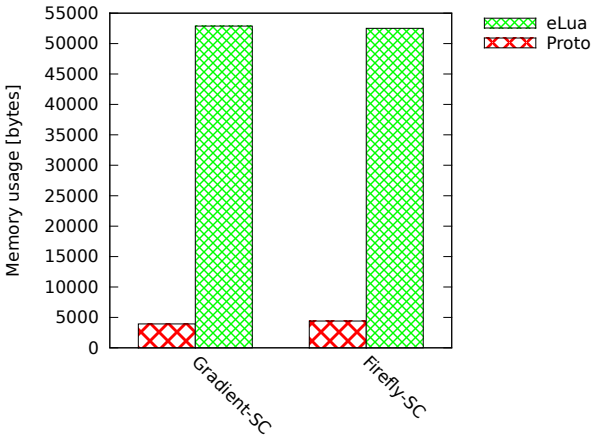


Figure 6.8: Dynamic memory usage comparison for Firefly and Gradient application

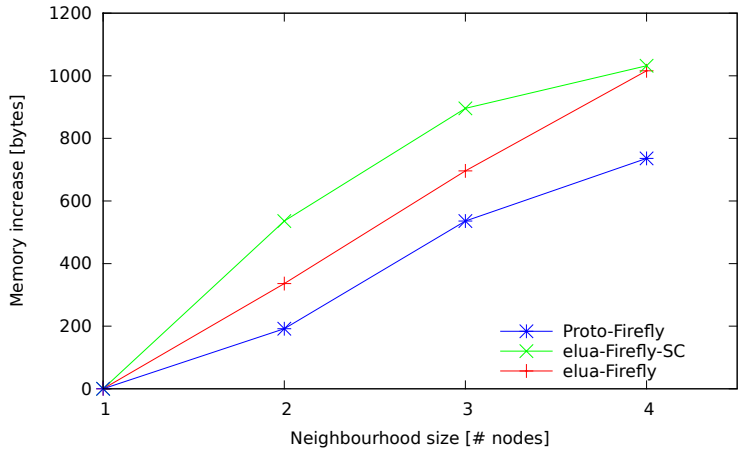
whole story since in quantitative terms Proto behaves much better. In fact, Proto has smaller increases in terms of bytes for each additional neighbor.

6.2.2 Performance

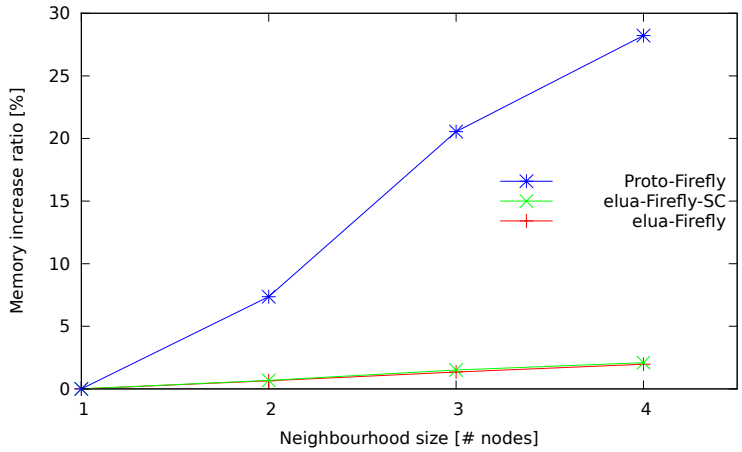
Interactivity is all about response time: interactivity needs a response time from both the human and system to be of the same order. In order for the system to be reactive to people's actions we have to guarantee a response time in order of the human perception time that is about $16.6ms$. In this Section, we perform execution time benchmarks to verify the platforms performance. We investigate how *SpatialeLua* performs with respect to application script size and neighborhood size. Moreover, we measure the execution time distribution for the various tasks performed in a VM round to provide the spatial computing capabilities such as communication, neighborhood discovery, export update, and neighborhood information retrieval. Afterwards, we compare our findings with the Proto platform.

Execution Time as a function of the script size

In Figure 6.10 and Table 6.3 we see the relation between execution time and script size. It is clear that the script size cannot be used as an indicator of complexity and predictor for how execution time behaves with the increase of script size. As expected, Figure 6.10 confirms that due to branching the state chart versions of the application do not have a linear relation between execution time and script size.



(a) Increase rate of dynamic memory usage in eLua VM and Proto VM



(b) Increase ratio of memory consumption as a function of the neighborhood size

Figure 6.9: eLua vs Proto memory usage comparison

Execution Time as a function of the neighborhood size

In Figure 6.11 we see the relation between the execution time and the neighborhood size for each application. The influence of the neighborhood size to the execution time is almost negligible and in the order of microseconds. Another, aspect to notice is that the start chart version has not necessarily higher execution time as its counterpart version.

Updating of neighborhood information is a frequent operation in spatial applications. For example, determining the number of neighbors which have their LED on. We measure the execution time distribution for network information retrieval. Figure 6.12 shows the execution time distribu-

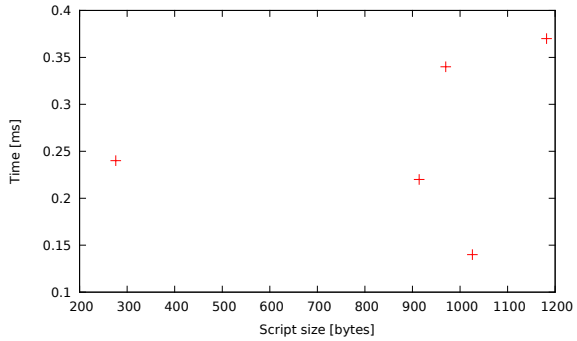


Figure 6.10: Execution time with respect to script size

Table 6.3: Application script size

Application	**Size [bytes]**
Dummy	276
Firefly	914
Gradient	970
FireflySC	1026
GradientSC	1182

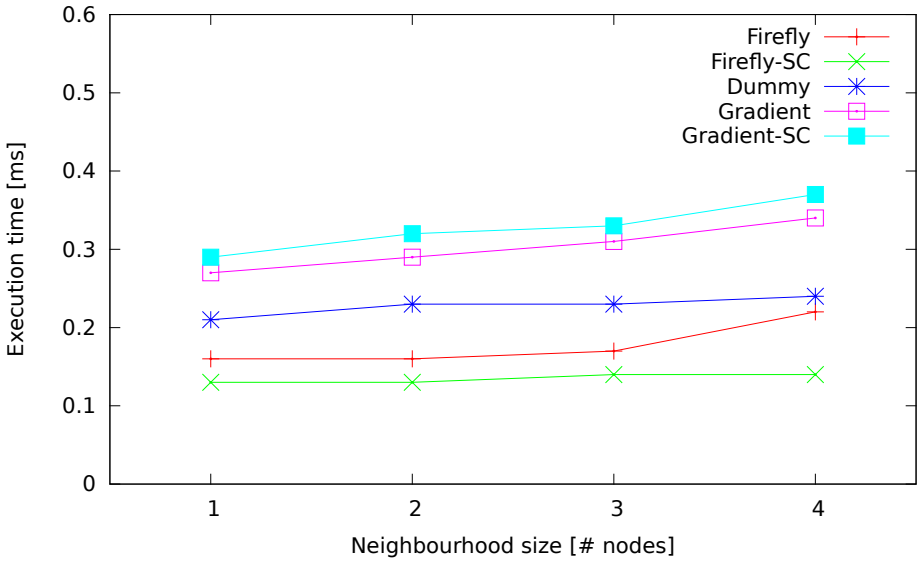


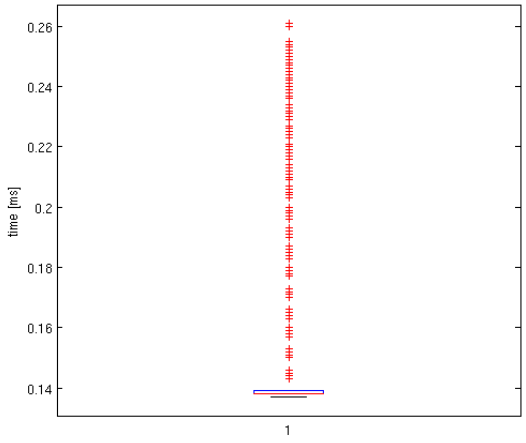
Figure 6.11: Execution time with respect to neighborhood size

tion for retrieving information from the neighborhood table data structure. We investigate the worst-case scenario which correspond to the four-node-neighborhood.

We see that the time is almost constant around $0.15ms$ which is good news. However, outliers are possible and these can be close to double the mean time.

SpatialeLua VM tasks' execution time analysis

In Figures 6.13, 6.14, 6.15 we see the execution time distribution for the three tasks performed at each virtual machine round. Each VM execution round performs communication(neighborhood information update), export-



Mean $[ms]$	**Std.Dev.** $[ms]$
0.150	0.031

Figure 6.12: Neighborhood information retrieval time distribution.

ing(inform neighbors of state variable changes), and beaconing. In the boxplots, we see that the execution times for each task are almost constant within a certain range. The histograms help giving a better view of the execution time distribution. A commonality is the presence of outliers which might overshoot by more than 50%. The probability of these outliers happening in the same round is remote but not impossible. This might affect the temporal correctness of the applications using timer or counters.

The last eventuality is investigated by studying the VM round execution time. Figure 6.16 shows the execution time distribution of the virtual machine round. The round execution time is not the straightforward sum of the tasks it consists of, since beaconing is performed periodically(every $32ms$) and the export task is only performed whenever a state variable has been altered. In Figure 6.16 we see that the range between the smallest and the greatest outlier goes from $0.375ms$ to $0.664ms$.

Figures 6.13, 6.14, 6.15 and Figure 6.16 show similar distributions. In Table 6.4 we summarize them by showing their mean and standard deviation. As we can see, the export task takes the highest amount of time on average. To notice, that the export and the beacon task have small standard deviation. Both tasks rely on the *eluaComm* library. On the other hand, the communication task comprises both the use of *eluaComm* and neighborhood information data structure management which is Lua based. We see that, in that case, the standard deviation is greater than in the other tasks. A reason might reside in the fluctuation in execution time due to Lua code interpretation. In addition, the fluctuations are most probably due to the fact that some phases of the Lua garbage collection stop the Lua program execution. This might explain the outliers in general.

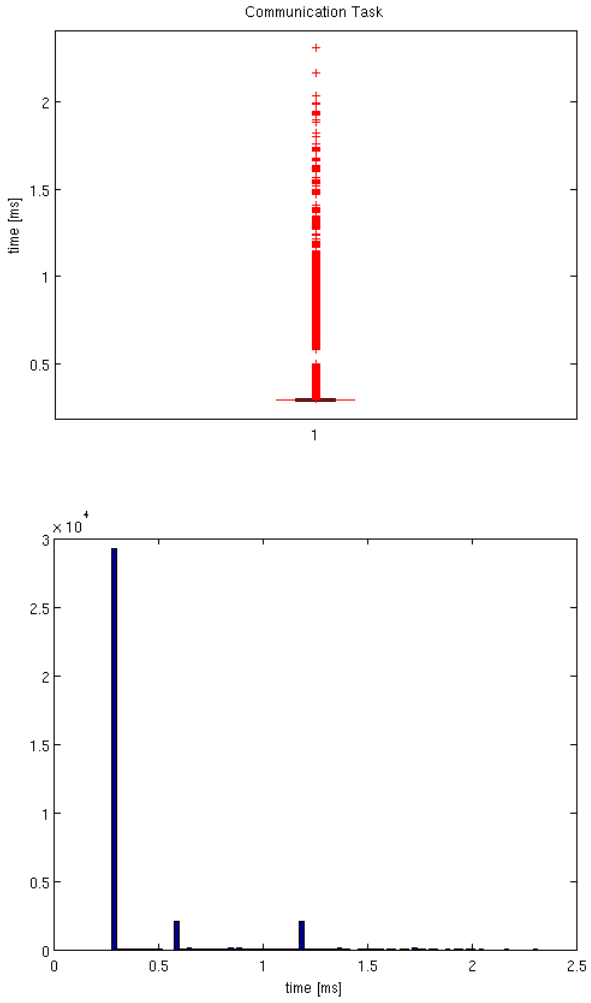


Figure 6.13: SpatialeLua's Communication task execution time distribution

Task	**Mean**$[ms]$	**Std.dev**$[ms]$
Communication	0.392	0.273
Export	1.134	0.043
Beacon	0.833	0.036
VM Round	0.581	0.340

Table 6.4: Summary execution time distribution

The round execution time is of particular importance for our purpose since it determines the response capabilities of our embedded platform when it has to satisfy the response time constraints for responsive interactive experiences. The measurements show that even though the greatest outlier

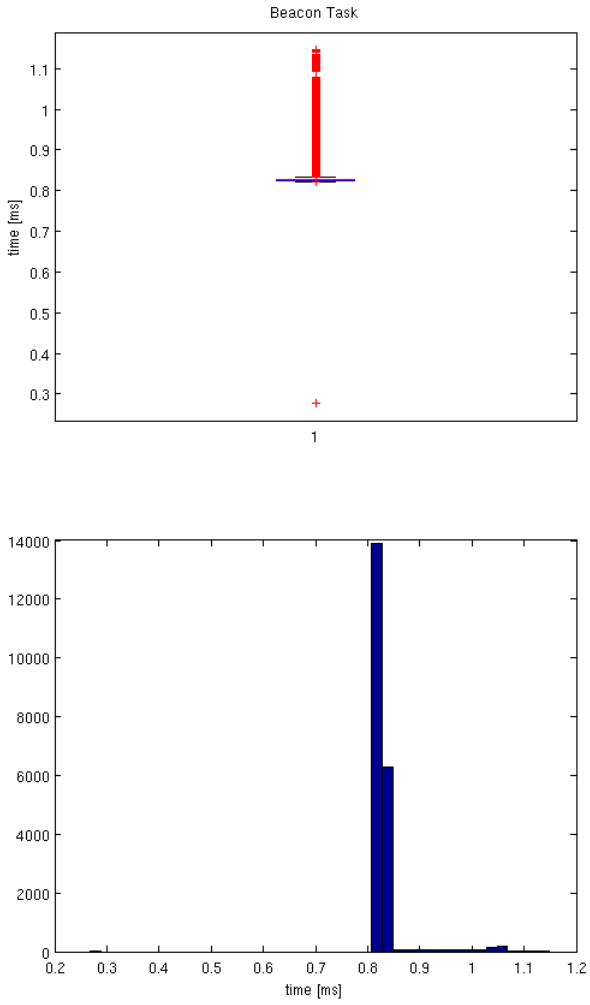


Figure 6.14: SpatialeLua's Beacon task execution time distribution

is about 3.6 ms for the VM round, *SpatialeLua* comfortably satisfies the 16.6ms requirement.

Performance: eLua vs Proto

In Figure 6.17 we compare the script and full loop(VM round) execution time of eLua and Proto [48]. The script is the Firefly application. In case of the script, elua performs better than Proto in both cases: elua's average and maximum execution time. However, that is not the case for the full loop(including all VM tasks) execution times. We see that Proto performs better than elua when we compare it to elua's worst execution time. That is not the case if we compare Proto with elua's mean execution time.

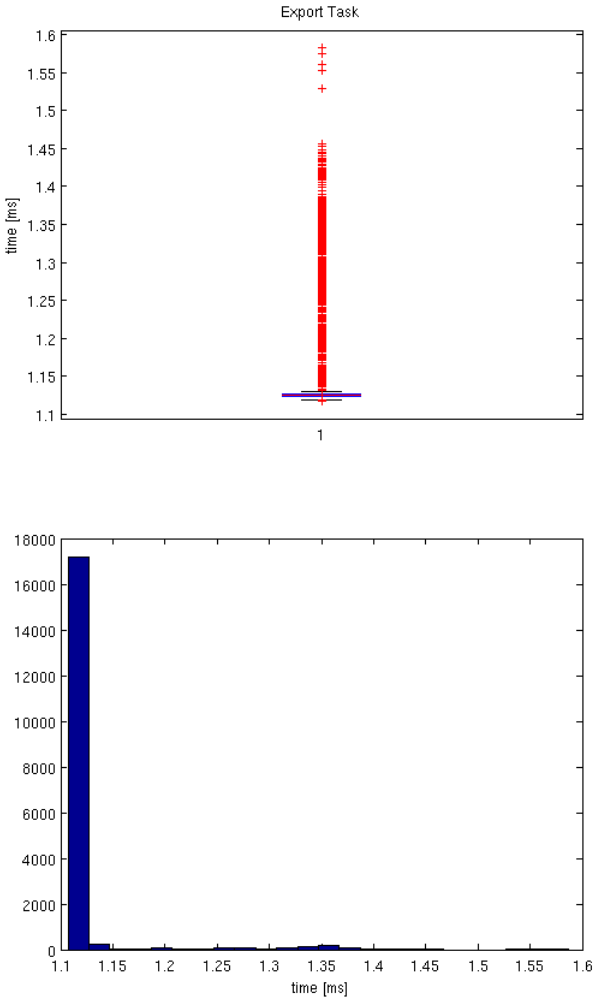


Figure 6.15: SpatialeLua's Export task execution time distribution

The difference in the full loop case resides mostly in the fact that the spatial capabilities of eLua are implemented in Lua, while Proto's spatial capabilities are at the op-code level. That implies better performance since the spatial capabilities are the key operations. Nevertheless, *SpatialeLua* shows acceptable performance for being a general purpose virtual machine adapted to spatial computing.

6.3 ELua Optimizations

As described in Section 2.6.2 it is possible to extend elua with both C modules and Lua modules. During the development process the initial focus

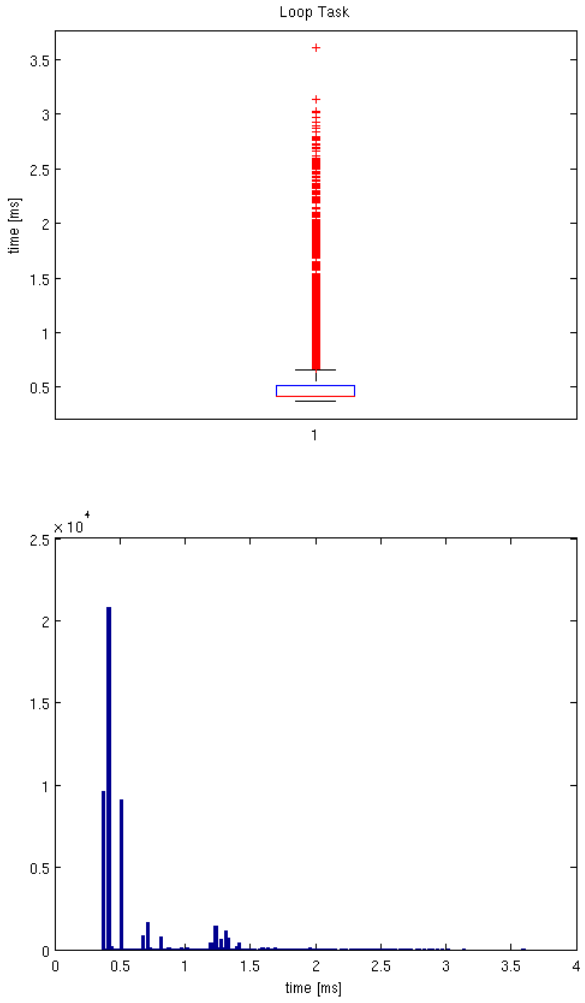


Figure 6.16: SpatialeLua VM round execution time distribution

was on getting a working version of the eLua VM with the necessary spatial computing capabilities. Hence, the focus was not on performance or memory usage. Partly, it was due to the lack of experience with Lua and the elua platform. That experience was achieved slowly during the development process and in ambitious collaborations such as the Phototropia project at ETH Zurich(see Section 5.1). In this Section we show how that experience influenced the final version and what trade-off was accepted when choosing whether to use C or Lua modules. Besides the language choice, we adjusted the Lua modules in order to fully exploit Lua's VM features [31]. In Table 6.5 we show the modules of the initial version of *SpatialeLua* and the added improvements.

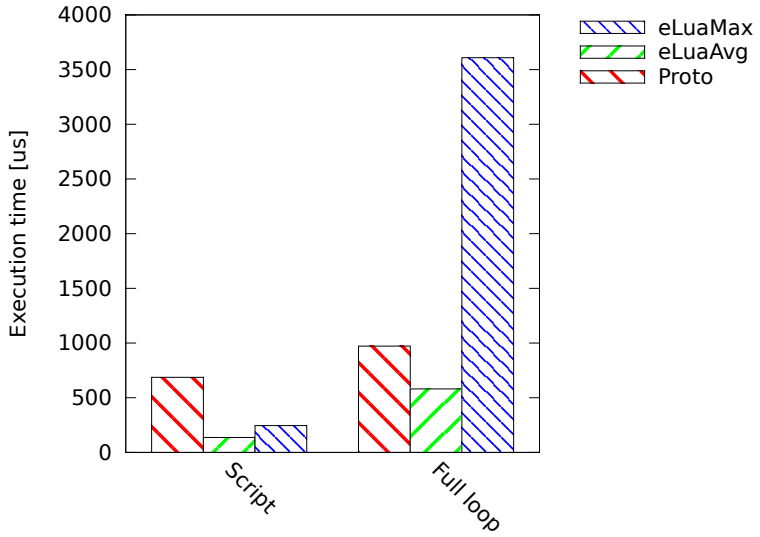


Figure 6.17: eLua vs Proto execution time comparison for script and application loop.

Module	Optimization	Description
ADC PWM Neighborhood Application	Lua optimization	Exploit the Register-based property of LUA: all variables and functions are declared as local(stored into register).
Packet	C module	Packet information and creation in C

Table 6.5: Summary execution time distribution

For most of the modules we decide to optimize the Lua code to better exploit the register-based property of the Lua VM. These register are tables(stored in the Lua stack) that behave like registers. Each function can use up to 250 registers. That means that the Lua pre-compiler can store all local variables and functions in "registers" that can be accessed very fast in Lua [31]. From the performed benchmarks we know that the Neighborhood module comprising communication and neighborhood information management is the most time and memory consuming part of our platform. Nevertheless, we decide to keep the implementation in Lua in favor of portability of the spatial capabilities. On the other hand, we converted the packet information and creation module into a C module. The main reason is that bitwise shifts are not possible in Lua and divisions take much more time. In addition, the packet module is used most frequently and an implementation in C appeared to be the most obvious choice.

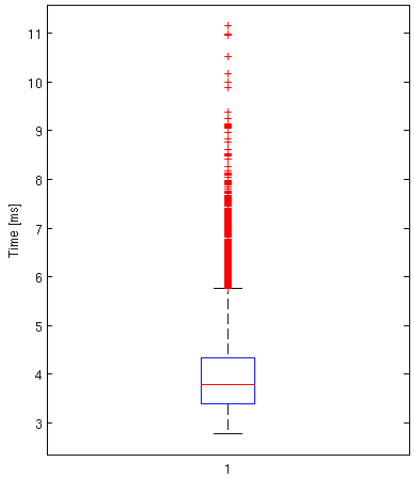


Figure 6.18: SpatialeLua loop execution time distribution without optimizations

In Figure 6.18 we see the VM loop execution time distribution of the initial not optimized version. The range spans from $2.77ms$ to $11.17ms$ that is to wide and unreliable for our purposes.

Version	**Mean**$[ms]$	**Std.dev**$[ms]$
Not opt.	4.063	0.916
Opt.	0.581	0.340

Table 6.6: Mean and standard deviation for optimized and not optimized elua.

In Table 6.6 we compare mean and standard deviation of the full loop for both version optimized and not optimized. With the performed optimization we are able to lower the mean execution time and standard deviation of *SpatialeLua* with benefits for time reliability for our time constraint applications. As we see in Figure 6.19 the memory usage also improves. Overall, we reduce the dynamic memory consumption by 4224 bytes.

These findings demonstrate that *SpatialeLua* can be further improved by converting Lua modules to C modules. If a more domain specific version would be necessary an area of further optimization would be represented by the *Neighborhood* module. The reason are two: first, it contains the greatest amount of table data structures which occupy much more memory than arrays in C; second, the communication protocol is performed in this module and optimizing it would imply a gain in performance. However,

Version	Lua [bytes]	Overall [bytes]
Not opt.	31252	56736
Opt.	27880	52512

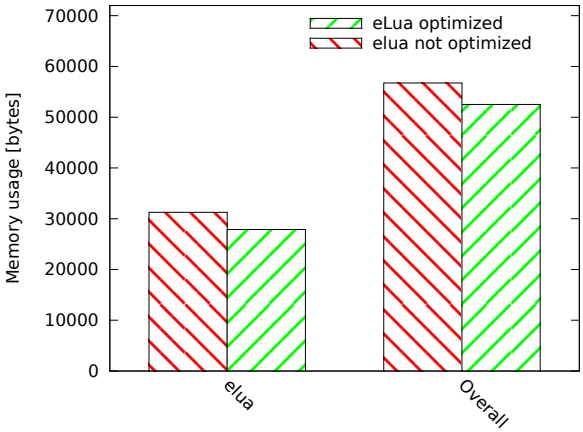


Figure 6.19: Dynamic memory usage comparison: elua with and without optimizations

it is up to the design requirements and hardware platform to indicate the direction to undertake.

6.4 Simulation vs. Real World

As mentioned before we provide DeckSim, a Netlogo based simulator which models protoDeck floor. DeckSim is used to test the the behavior of the distributed system and provide a visual feedback of the given design. In this Section, we show to which extent DeckSim is able to simulate the real world scenario. We use the Firefly, Gradient, and Spotlight application and generate the Netlogo and Lua code. In Table 6.7 we show the outcome of comparing simulation with real world.

Application	DeckSim → protoDeck	Differences
FireflySC	different	Time handling; no synchronization in protoDeck
GradientSC	identical	No differences at all
SpotlightSC	similar	Radius light limit works; decrease of circle does not happen

Table 6.7: Comparison between simulation and real world

As expected, the shift from simulation to real world is problematic. As we

see in Table 6.7 the timing differences of simulation and real world influence the visual feedback of the algorithm as well as the validity of the algorithm. In fact, both FireflySC and SpotlightSC include time and in both cases simulation and real world differ to some extent. Certainly, it depends on how accurate DeckSim models the target platform. However, DeckSim can be used in some cases as a preliminary algorithm verification tool.

6.5 Discussion

Our findings show that spatial computing is achievable with an off-the-shelf virtual machine. We are able to construct time, spatial, and time-spatial primitives for interactive applications. The benchmark results show that *SpatialeLua* is able to satisfy the memory and performance requirements. The memory consumption benchmarks emphasize the high memory usage of the eLua VM. However, it shows that for the current target application consisting of mesh-like network of interconnected embedded platforms everything works fine. In terms of performance *SpatialeLua* proves to be responsive enough for our purposes. The worst case loop execution time is about $3.5ms$ and it is way lower than the limit of $16.6ms$ set by the human visual perception. If we compare *SpatialeLua* to the Proto platform it is clear that Proto outperforms SpatialeLua in every performance related aspect.

Results at hand we are able to answer our research questions(Section 1.3):

- By providing evidence through spatial computing applications for creating interactive user experiences on protoDeck we show the suitability of spatial computing as a methodology for Interactive Design and Architecture. Moreover, we show that architects need methods to specify behaviors for large scale distributed systems(see user survey and Phototropia project). Possible limitations to spatial computing might be grasping the concept of locality and implicit communication by the students and architects/designers. This was experienced in the collaboration and interaction with non-IT specialists(user survey, GUI conceptualization, presentation in Zurich).

- The benchmark findings support the use of off-the-shelf virtual machines such as eLua for embedded platforms. The trade-off of such as choice is the effort to create language-specific spatial capabilities. We achieved that by implementing the necessary communication protocol, neighborhood management and spatial constructs modules. The trade-off is represented by high memory consumption and lower execution time performance. A possible limitation can be represented by an increase of the neighborhood size in case of radio medium support. In that case, different factors like the modules memory consumption and

the limit on the maximum neighborhood size might represent a limit for *SpatialeLua*. For that inconvenient the use of C modules instead of Lua modules might be taken into consideration. Worth to mention is the fact that we used the official release of the eLua VM(v0.8). In the 'bleeding edge' version the byte-code of all the eLua modules does not reside in RAM anymore and they are directly loaded from the flash memory. In our case this would cause a drastic decrease in dynamic memory usage of at least 13632 bytes. As benefits inherent to eLua we mention the readability of the Lua code, easy maintainability and extensibility of the VM, debugging tools, and a fast script execution. Another positive point for eLua and Lua is the huge community behind both projects. The development and user community is very active for both eLua VM and Lua.

- With the creation of IDS we aim to hide the underlying technological aspects and spatial computing concepts from end-users. For that purpose we provide a state chart for local agent behaviors as well as a CodeGenerator. The latter catalyses the specifications and generates the code for the embedded platform(eLua in our case). IDS targets protoDeck but it is designed to be extended with other target platforms. Besides the generation of the application code from the state chart, IDS generates the spatial library which specifies the data exchanged in the neighborhood and its interpretation. This offers the possibility to come up with its own spatial abstractions. The downside of this approach is the flexibility in itself since it gives rise to a more error prone scenario: syntax errors in the configuration files, buggy spatial abstraction algorithms etc. Another aspect is sensor calibration. The Phototropia experience uncovered the possible pitfalls when dealing with unreliable sensors which may require hands on the code or hardware. In these cases IDS fails and an engineer is still required. However, magic does not exist.

In conclusion, in Table 6.8 we show a spec comparison between eLua (*Spatialelua*) and Proto as platform for spatial computing.

First we mention aspects related to the VM typology such as VM type, programming language type, and how spatial computing is achieved. In Table 6.8 we rate(three stars highest) aspects such as memory usage, performance, development tools, community support, code readability, and portability. In terms of memory consumption and performance, Proto outperforms eLua for obvious reasons such as VM type and and spatial computing constructs. However, eLua provides great debugging tools which are inherent to Lua, while Proto does not provide any at all. Which in the end means more debugging time and low productivity. Another, factor contributing to a lower productivity is code readability. For programmers with

Aspect	**eLua**	**Proto**
VM type	general purpose	spatial computing
Programming language	procedural	functional
Spatial Computing	library	instruction set level
Memory usage	*	***
Performance	**	***
Devel. Tools	***	*
Community support	***	*
Code readability	***	*
Portability	**	*

Table 6.8: ELua vs Proto specs

a strong background in procedural languages(the majority), procedural programming languages such as Lua are more readable and understandable than functional languages such as Proto. Other aspects in favor of eLua is the huge and active web community support and the continuous development of eLua. This is not the case for Proto which is a stale project with a slow update rate. Finally, eLua has a port for 19 different embedded platforms which ease the work to bind the spatial computing features since the underlying drivers and services are ready out-of-the-box. On the contrary, Proto needs to be provided with the necessary drivers and communication library for a specific platform. In conclusion, we claim to be able to provide a means to non-IT specialists to exploit the spatial computing capabilities of an off-the-shelf virtual machine for the creation of applications for interactive environments. The choice of whether to chose Proto or eLua resides depends on the importance given to the pros and cons of the two platforms. If memory consumption is the key aspect than Proto is the best choice. If memory is not an issue than eLua is clearly the best choice for the aforementioned reasons.

Chapter 7

Conclusions

The purpose of this thesis was threefold: use spatial computing in interactive environment design, show that spatial computing is possible is possible in off-the-shelf virtual machines for embedded platforms, and show that it possible to abstract or hide the underlying technological aspects(spatial computing) from end-users. We do that, by providing a framework called Interactive Design Studio targeting non-IT specialists such as designers and architects.

IDS targets protoDeck. ProtoDeck is part of ProtoSpace 3.0, a state of the art multi-purpose facility designed for the development of non-standard, virtual, and interactive architecture at the Faculty of Architecture of the Delft University of Technology. The floor's tiles are equipped each with an LPCXpresso 1769 board from NXP. We provide a way to specify agent-level algorithms via state charts XML. In order to hide the technological aspects we implemented a CodeGenerator tool which can be tailored to a specific target platform by means of configuration files. The tool takes as input the SCXML and the platform specific configuration files and generates the platform specific code. In our case, we decide to use eLua VM which is a Lua based virtual machine for embedded platforms. In order to make spatial computing possible, we extend eLua by providing the necessary drivers and modules(eLuaComm and spatialib). We provide eLua(*SpatialeLua*) with a communication protocol, neighborhood management and discovery, and viral code dissemination. Worth to mention is the capability of defining custom spatial abstractions by providing the CodeGenerator module with the necessary configuration and library files.

We evaluate IDS and *SpatialeLua* using spatial(Gradient), time(Firefly), and spatial-time(Spotlight) applications on protoDeck. By performing memory consumption and performance benchmarks we show that eLua VM is suitable as platform for spatial computing and interactive installations.

In conclusion, we state that IDS provides a all inclusive spatial computing tool-chain that enables architects/designer to design complex interactive user experiences by abstracting away from technological aspects.

7.1 Future Work

The IDS framework aims to provide all the necessary tools and features that in our opinion are necessary to render accessible spatial computing for non-IT specialists. Since it is the first version several aspects can be further improved.

The agent-level algorithm specification in the form of state charts has to be verified for more complex and diverse applications. Moreover, it is still an open question whether it is the best way to interface a future GUI with the CodeGeneration module.

The customizability of the CodeGenerator is major feature but can also become the Achilles heel. In fact, in the current state there is no validity control for the configuration and library files. This might endanger the correct functioning of the whole tool-chain. On the other hand, the choice of using the XML format facilitates the creation of a GUI -based validation tool.

The benchmarks showed that eLua consumes quite a lot of RAM which might limit the increase of the neighborhood size and the switch from wired to wireless network. In that case, further platform specific optimizations might help. Moreover, the choice of converting some Lua modules to C might be considered since Lua data structures use more memory.

Finally, a workshop for architecture students using the tool would help targeting the areas which have to be improved or rethought.

Bibliography

[1] Arduino2max. http://www.arduino.cc/playground/interfacing/MaxMSP.

[2] Firefly. http://www.fireflyexperiments.com/.

[3] Firmata. http://www.arduino.cc/playground/Interfacing/Firmata.

[4] Grasshopper. http://www.grasshopper3d.com/.

[5] Maxuino. http://www.maxuino.org/.

[6] Processing. http://processing.org/.

[7] Rhino3d. http://www.rhino3d.com/.

[8] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, May 2000.

[9] H. Abelson et al. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.

[10] M.P. Ashley-Rollman, S.C. Goldstein, P. Lee, T.C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 2794–2800, 2007.

[11] J. Bachrach and J. Beal. Building spatial computers. Technical report, Tech. report, MIT CSAIL, 2007.

[12] Tobias Baumgartner, Sándor P. Fekete, Tom Kamphans, Alexander Kröller, and Max Pagel. Hallway monitoring: distributed data processing with wireless sensor networks. In *Proceedings of the 4th international conference on Real-world wireless sensor networks*, REALWSN'10, pages 94–105, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21(2):10 – 19, march-april 2006.

[14] Jacob Beal and Jonathan Bachrach. Programming manifolds. In André DeHon, Jean-Louis Giavitto, and Frédric Gruau, editors, *Computing Media and Languages for Space-Oriented Computation*, number 06361 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[15] J. Beal, S.O. Dulman, K. Usbeck, M. Viroli, and N. Correll. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter Organizing the Aggregate: Languages for Spatial Computing, pages –. IGI Global, 2012.

[16] Tilde Bekker, Janienke Sturm, and Berry Eggen. Designing playful interactions for social interaction and physical play. *Personal and Ubiquitous Computing*, 14:385–396, 2010. 10.1007/s00779-009-0264-1.

[17] N. Biloria. Emergent technologies and design. *eCAADe 23*, pages 441–447, 2005.
[18] Bill Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann, first edition, March 2007.
[19] D. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.
[20] A. Crabtree, T. Hemmings, and T. Rodden. Pattern-based support for interactive design in domestic settings. In *DIS 2002 Proceedings*, pages 265–276. ACM, 2002.
[21] Max/Msp Cycling'74. http://cycling74.com/.
[22] Tobi Delbrck, Adrian M. Whatley, Rodney Douglas, Kynan Eng, Klaus Hepp, and Paul F.M.J. Verschure. A tactile luminous floor for an interactive autonomous space. *Robotics and Autonomous Systems*, 55(6):433 – 443, 2007.
[23] S. Dulman. *Robotics in Architecture*, chapter Practical Programming of Large-Scale Adaptive Systems. JapSam Books, 2012.
[24] eLua project website. http://www.eluaproject.net/.
[25] Tozer; David J. Fowler; Glenville C. E. Inter-bus system, 03 1988. Patent.
[26] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS*, volume 3560 of *LNCS*, pages 466–466. 2005.
[27] M.H. Haeusler. *Media facades: history, technology, content*. Avedition, 2009.
[28] JC Hubers. Collaborative design in protospace 3.0. *Changing roles; new roles, new challenges*, 2009.
[29] CCM (Caroline) Hummels, JP (Tom) Djajadiningrat, and CJ (Kees) Overbeeke. Knowing, doing and feeling : communicating with your digital products. 2001.
[30] Roberto Ierusalimschy. http://www.lua.org/.
[31] Roberto Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006.
[32] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 2–1–2–26, New York, NY, USA, 2007. ACM.
[33] B Knep. http://www.blep.com/healingPool/.
[34] Joanna Kulik, Wendi Heinzelman, and Hari Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wirel. Netw.*, 8(2/3):169–185, Mar. 2002.
[35] Wolfgang Lefevre, editor. *Picturing machines 1400-1700*, chapter The origins of early modern machine design, pages 53–84. The MIT Press, 2004.
[36] Lamport Leslie. Computation and state machines. http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html, April 2008.
[37] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.
[38] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18:15:1–15:56, '09.

[39] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW '03, pages 342–, Washington, DC, USA, 2003. IEEE Computer Society.

[40] Peter Marwedel and Peter Marwedel. Specifications and modeling. In *Embedded System Design*, Embedded Systems, pages 21–118. Springer Netherlands, 2011. 10.1007/978-94-007-0257-8_2.

[41] F Morbini. http://code.google.com/p/scxmlgui/.

[42] R. Nagpal. *Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics*. PhD thesis, Massachusetts Institute of Technology, 2002.

[43] Radhika Nagpal and Marco Mamei. Engineering amorphous computing systems. In Federico Bergenti, Marie-Pierre Gleizes, Franco Zambonelli, and Gerhard Weiss, editors, *Methodologies and Software Engineering for Agent Systems*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 303–320. Springer US, 2004. 10.1007/1-4020-8058-1_19.

[44] Radhika Nagpal, Howard Shrobe, and Jonathan Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *Proceedings of the 2nd international conference on Information processing in sensor networks*, IPSN'03, pages 333–348, Berlin, Heidelberg, 2003. Springer-Verlag.

[45] B. Quinn. *Textile Futures: Fashion, Design and Technology*. Berg Pub Ltd, 2010.

[46] P Schachtschabel and L Suijker. http://msc1.hyperbody.nl/index.php/.

[47] Phototropia A self-sufficient architectural vision. http://www.caad.arch.ethz.ch/blog/?p=2778.

[48] Karger Steffan. An embedded spatial computing platform for interactive environments. MSc thesis, TU Delft, July 2012. Snowdrop Project.

[49] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Trans. Auton. Adapt. Syst.*, 6(2):14:1–14:24, June 2011.

[50] U Wilensky. Netlogo fireflies model, 1997. http://ccl.northwestern.edu/netlogo/models/Fireflies. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[51] U. Wilensky. Netlogo, 1999. http://ccl.northwestern.edu/netlogo/.

[52] Franco Zambonelli and Marco Mamei. Spatial computing: An emerging paradigm for autonomic computing and communication. In Michael Smirnov, editor, *Autonomic Communication*, volume 3457 of *Lecture Notes in Computer Science*, pages 227–228. Springer Berlin / Heidelberg, 2005. 10.1007/11520184_4.