Diss. ETH No. 15035

Theory of Computation as a Vehicle for Teaching Fundamental Concepts of Computer Science

A dissertation submitted to

ETH ZURICH

for the degree of
Doctor of Sciences ETH Zürich

presented by

Raimond Reichert
Dipl. Informatik-Ing. ETH

born November 25, 1973, of German Nationality

accepted on the recommendation of

Prof. Dr. Jürg Nievergelt, examiner
Prof. Dr. Horst Müller, co-examiner
Dr. Werner Hartmann, co-examiner

2003

Acknowledgements

Many people have contributed to the development of the Kara environments, and I wish to express my gratitude to all of them.

Jürg Nievergelt and Werner Hartmann conceived the idea behind the Kara environments. They gave me the opportunity to realize the idea and continuously supported me during the development of the environments. Jürg Nievergelt kept a critical watch on Kara as it grew up, always trying to keep things as simple as possible, but not simpler. Werner Hartmann always reminded us of the users of Kara and helped to ensure that the environments are as user-friendly as possible.

Markus Brändle and Tobias Schlatter developed two offsprings of Kara. Together, they implemented the MultiKara environment, and Markus Brändle implemented the TuringKara environment.

Nicole Völki suggested using a pretty ladybug as the principal actor of the environment and designed the Kara ladybug. Reto Lamprecht later revamped the bug, and is also responsible for the user-friendly state machine diagram editor. They helped make Kara not only an academic but also a popular success.

Horst Müller provided us with his theorems about the capabilities of finite state machines working under various constraints. He also pointed us to fascinating topics related to finite state machines such as busy beavers and ants.

Remo Meier and Samuel Zürcher jointly developed the LegoKara environment to allow users to program a physical robot. They mastered the difficult challenge of building a robust Lego robot.

Horst Gierhardt has not only provided us with encouraging feedback but also with a wealth of teaching materials. He further helped popularize the Kara environments in Germany.

Last but not least, Myke Näf read this text with incredible care and helped improve it. As an 'outsider', he questioned many hidden assumptions.

Finally, many others have contributed to the Kara environments and supported me during the development: Beate Bernhard, Phillip Boksberger, Juta Bonan, Hans Fischer, Diana Hornung, Philip Meier, Marc Pilloud, Thomas Suter, Vincent Tscherter, Floris Tschurr, as well as all those I forgot to mention.

Abstract

In today's Information Society, knowledge of the fundamentals of information and communication technology (ICT) is a key qualification and must become part of general education. The concepts of formalization and programming are at the heart of computer science. Teaching these concepts as part of general education is a difficult challenge. Existing approaches are typically time-consuming, yet time is a scarce resource in school. There is a need for intuitive, user-friendly, high-quality educational software environments to support teaching and learning the basics of ICT within the limited time available.

Our approach uses the theory of computation as a vehicle for teaching selected fundamentals of computer science. The main contributions of this dissertation are the Kara programming environments described in this text. One goal of the environments is to allow users to write their first program successfully within an hour. The environments are based on finite state machines and offer:

1. An introduction to programming targeted at people with no prior programming experience (the Kara environment).

2. An introduction to the theory of computation based on two-dimensional Turing machines for students who study the theory of computation (the TuringKara environment).

3. An approach to teaching basic concepts of concurrency in an illustrative manner which can be used from elementary to advanced settings (MultiKara).

4. A smooth transition to programming in Java for people not familiar with Java or any similar programming language (JavaKara).

The widespread use of the Kara environments in many schools at different levels, and the highly positive feedback from teachers and students, have convinced us that our theory-based approach is successful and worth pursuing.

Zusammenfassung

In der heutigen Informationsgesellschaft sind Kenntnisse der Grundlagen von Informations- und Kommunikationstechnologien (ICT) eine Schlüsselqualifikation und müssen Teil der Allgemeinbildung werden. Zentrale Themen der Informatik sind die Konzepte der Formalisierung sowie des Programmierens. Das Unterrichten dieser Konzepte als Teil der Allgemeinbildung ist eine anspruchsvolle Herausforderung. Existierende Ansätze sind typischerweise zeitaufwendig, obwohl Zeit in Schulen eine knappe Resource ist. Es gibt einen Bedarf für intuitive, benutzerfreundliche, qualitativ hochstehende Lernumgebungen als Unterstützung für das Lehren und Lernen der Grundlagen von ICT im Rahmen der zur Verfügung stehenden Zeit.

Unser Ansatz verwendet die theoretische Informatik als Mittel zum Zweck, um ausgewählte Grundlagen der Informatik zu unterrichten. Die Hauptbeiträge dieser Dissertation sind die Kara-Lernumgebungen, die in diesem Text beschrieben sind. Ein Ziel der Umgebungen ist es, den Benutzern zu ermöglichen, innerhalb einer Stunde ihr erstes Programm erfolgreich schreiben können. Die Umgebungen basieren auf endlichen Automaten und umfassen:

1. Eine Einführung in die Programmierung, die sich an Leute ohne Programmiererfahrung richtet (die Kara-Umgebung).

2. Eine Einführung in die theoretische Informatik anhand von zweidimensionalen Turing Maschinen für Studierende, die sich mit der Theorie der Berechenbarkeit auseinander setzen (die TuringKara-Umgebung).

3. Einen Ansatz zum Unterrichten grundlegender Konzepte der nebenläufigen Programmierung in einer anschaulichen Art und Weise, der auf verschiedenen Stufen eingesetzt werden kann (MultiKara).

4. Einen reibungslosen Übergang zum Programmieren mit Java für Leute, die mit Java oder ähnlichen Sprachen nicht vertraut sind (JavaKara).

Die grosse Verbreitung der Kara-Umgebungen an vielen Schulen auf unterschiedlichen Stufen und das sehr positive Feedback von Lehrenden und Lernenden haben uns überzeugt, dass der Theorie-basierte Ansatz von Kara erfolgreich ist und dass es sich lohnt, diesen Ansatz weiter zu verfolgen.

Contents

Chapter 1

ICT: A Challenge for the Information Society

In today's Information Society, mastery of information and communication technologies (ICT) is a key qualification. In this chapter, we argue that education in computer science needs to become part of general education. More specifically, we further argue that the focus should be on the foundations of computer science. We propose that one core topic of such an education should be the idea of formalization. The goal is to teach the idea that in order to get computers to do anything, one needs to express the instructions in a formally defined system.

1.1 Information and Communication Technology as Part of Education

In the past 60 years, information and communication technologies have transformed the industrial society into the information society. This transformation has had an impact on all aspects of life, from the economy to social behavior to the sciences. The rapid evolution of the ICT also affects our expectations of education. Two kinds of forces influence the future of education:

ICT as a Subject. There is a strong demand for information and communication technology to become part of general education: to be learned, in appropriate steps, by practically everybody, throughout all stages of the educational life cycle, from elementary schooling to life long learning.

ICT as a Tool. There is also a strong demand for ICT to be used as a delivery tool to present information anytime, anywhere, and to enhance the learning process thanks to a quality of interaction unequaled by any previous educational technology.

We focus on ICT as a subject and on the need for ICT education to become part of education at different levels. For our purposes, we focus on formal education, from primary school to tertiary education. We view the process of education as resting on three pillars:

Literacy. High priority in public school is typically assigned to content that is relevant to today's job world. Folklore identifies "the 3 R's: reading, 'riting, 'rithmetic" as the main task of elementary school. These skills have become a necessary part of general education as a consequence of the industrial age. The OECD's definition of reading literacy for its PISA assessments reflects this [OEC02]: "Reading literacy is understanding, using, and reflecting on written texts, in order to achieve one's goals, to develop one's knowledge and potential, and to participate in society."

General Education. In addition to tangible, sellable skills, education has always included topics about which most adults later say "I have never used them since I left school". This category includes the majority of topics any student endured, namely most of those outside the profession chosen later. For example, general education includes studying a certain amount of literature, as well as learning how mathematical proofs work. The goal is to motivate students to develop intellectual curiosity.

Critical Thinking. Critical thinking encompasses the ability to understand, reflect, and discuss matters relevant to one's own life. For example, literacy ensures that we can read movie or book reviews; general education provides us with the relevant background knowledge. Critical thinking enables us to form our own opinion by a process of actively gathering, analyzing and evaluating information.

Each of these three components of education is highly relevant to an ICT education. Let us briefly argue the importance of each component.

Computer Literacy. Webster's New College Dictionary gives the following definition of computer literacy: "the ability to use a computer and its software to accomplish practical tasks". The ability to use computers is important because it is a prerequisite for entering the work force. This definition can be viewed as a common denominator of most definitions of computer literacy.

General Education in Computer Science. In the rapidly developing field of ICT, students need a deeper understanding of the potential of computer technology, its possible applications, and its limitations. Mere literacy cannot provide this kind of understanding. General education in computer science must teach the timeless foundations of computer science.

Critical Thinking. Because of the importance of ICT in our society, it is indispensable for students to be capable of following and understanding future developments of this technology.

ICT education calls for all three aspects, literacy, general education, and critical thinking. In the following, we focus on the content of computer science education as part of general education.

1.2 Computer Science as Part of General Education

There is a lot of disagreement about the content of computer science education as part of general eduction. Two of the main reasons for this disagreement are the rapid evolution of computer science education, and the debate on the nature of computer science itself. It is important to keep these two reasons in mind when arguing about the content of future computer science education.

The first difficulty in defining computer science education is the pace of evolution of the field. As the field of computer science evolved at a rapid pace in the last forty years, computer science education lagged behind. It is instructive to consider the evolution of the ACM's model curricula. Not only did the content definition evolve, the structure in which the curricula organizes its body of knowledge evolved as well. Curriculum 68 divided computer science into three large divisions: information structures and processes, information processing systems, and methodologies. These were further subdivided into a total of 17 topics such as programming languages, translators and interpreters, data processing and file management [ACH+68]. Its content reflected the technology of its time. Curriculum 91 divided computer science into nine subareas [Tuc91]. New topics included, for example, human-computer interaction and software engineering, reflecting the trend towards larger software with more emphasis on their user interfaces. The most recent draft of Curriculum 2001 discerns fourteen subareas, where the additional subareas include, for example, net-centric computing and social and professional issues, reflecting the growth of the Internet as well as the growing importance of social issues [AI02]. In summary, the development of the ACM curricula has been driven by technology. Hartmann and Nievergelt point out that the development of ICT education in general has been driven by technology instead of focussing on the long-lived fundamentals [HN02].

The Debate on the Nature of Computer Science: An Educational Challenge

The second difficulty in reaching agreement on the content of computer science education is the debate on the nature of the field. There is an abundance of conflicting definitions of computer science. One reason for this is given by Dijkstra who argues that information and communication technologies represent a revolution, not merely an evolution of prior knowledge or technology [Dij89], [Dij86]. He outlines two "radical novelties" which computers embody:

> **Conceptual Depth.** If one considers a town, many specialists are needed to build one, from town planners to architects to physicists. Dijkstra argues that, in marked contrast, a programmer "has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before".
>
> **Large-scale Digital Device.** While there were discrete machines before the computer, computers are the only large-scale digital devices. The discrete nature of computers bestows upon them the "uncomfortable property

that the smallest possible perturbations – i.e., changes of a single bit – can have the most drastic consequences".

With regards to the above two properties of computer science, Hartmanis takes a similar view [Har94]. On the conceptual depth of computer science, he quotes Knuth as saying that computer scientists "are individuals who can rapidly change levels of abstraction, simultaneously seeing things 'in the large' and 'in the small' ". As to the computer itself, he points out that not only is a computer a large-scale digital device, it is also a "universal device which can be instructed to perform any computation [...] but which [...] must be controlled with unprecedented precision".

The conceptual depth of computer science may explain why it encompasses both a mathematical facet and an engineering facet. There is disagreement over which facet is more important. The mathematical facet is favored and advocated, for example, by Dijkstra who dismisses software engineering altogether as a "doomed science" [Dij89]. The engineering facet is favored, for example, by Brooks in his talk *The Computer Scientist as a Toolsmith*, where he emphasizes that computer scientists engineer systems of "arbitrary complexity", and that a computer scientist as "a toolmaker succeeds as, and only as, the *users* of his tool succeed" [Bro96].

Others argue that computer science is not either a mathematical science or an engineering science, but both. For example, the report *Computing as a Discipline* by Denning et al stresses the fundamental importance of both facets, intending to "project an image of a technology-oriented discipline whose fundamentals are in mathematics and engineering" [DCG+89]. The report defines three paradigms, or cultural styles, by which computer scientists work (figure 1.1): theory (the mathematical roots of computer science), abstraction (the experimental sciences' aspects), and design (engineering). This reflects Dijkstra's observation on the conceptual depth of computer science: one paradigm is not enough to cover it all.

Paradigm	**Description in keywords**	**Examples**
Theory	Definitions and axioms, theorems, proofs, interpretation of results	complexity theory, architecture (logic), and programming languages (formal grammars and automata)
Abstraction	Data collection and hypothesis formation, modelling and prediction, design of an experiment, analysis of results	modelling potential algorithms, data structures, architectures (von Neumann)
Design	Requirements, specifications, design and implementation, testing and analysis	developing a system or a component of a system to solve a particular problem

Fig. 1.1: Denning et al's three-paradigm approach to computer science topics

Another view of computer science, which also stresses its multi-faceted nature, is given by Nievergelt [Nie95]. Computer scientists often divide complex problems into individual layers. Nievergelt applies this method to computer science itself and presents a layered tower of computer science consisting of the theory of computation, algorithmics, engineering, and applications (figure 1.2). The size of the individual layers hints at their relative emphasis, in a number of respects: recognition by society, economic potential, and number of jobs. The tower points out that many different kinds of activities take place under the name of computer science and illustrates the phrase "conceptual depth".

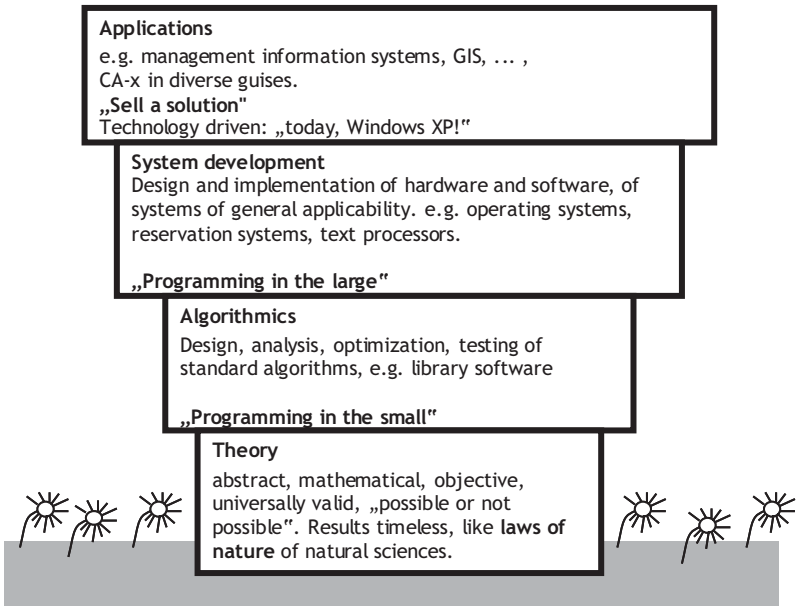

Fig. 1.2: Nievergelt's 'tower' of computer science

The Content of Computer Science Education: Focussing on Foundations

Computer science education as part of general education should concentrate on the long-lived fundamentals, because of the broad nature of the field. As an analogy, consider another engineering subject, civil engineering. Civil engineering and its relation to other sciences can be presented in a tower-like structure which is comparable to that of computer science: core physics, applied mechanical laws, engineering techniques, and applications. General education focusses on the foundation of civil engineering, on physics. It does not delve too deeply into the engineering aspects.

Hartmann and Nievergelt make the case that computer science education as part of general education, in analogy to long established subjects like physics, should focus more on the foundations of computer science [HN02]. Emphasizing timeless concepts helps avoid short-lived, technology-driven developments. It would be challenging and instructive to develop a whole computer science

education curriculum along the idea of concentrating on the foundations. But the design of such a curriculum is outside of our scope and must be the topic of a much broader discussion.

In the following, we concentrate on one core aspect of computer science: formalization in the form of programming. The emphasis is on the need to formalize instructions to a computer in a suitable, formally defined system. Our focus is not on programming per se, nor on a particular programming language.

1.3 Teaching Programming: From Intuition to Formal Specification

We argue that programming should be part of general education, not because of any direct utilitarian benefit, but in order to gain a personal experience as to what it means, and what it takes, to specify processes that evolve over time. Understanding this concept, and providing students with tools to gain hands-on experience, is important because modern society relegates an ever-growing number of every-day tasks to machines. These machines act as controllers that initiate actions based on their current state and on received inputs. The number of possible behaviors, of sequences of actions triggered by different environmental conditions, is usually so huge as to be impossible to enumerate. Yet, we aim to be sure that each and every possible behavior, of which only a tiny fraction will ever be played out, is 'correct' in some precise sense. The way to do that is to write a specification that captures the practical infinity of processes that may evolve over time, depending on received inputs. An algorithm is such a formal specification, and the concept of algorithm is surely among the most fundamental concepts required to understand computers.

In order to write programs, it is necessary to make the transition from an intuitive understanding of how an algorithm is supposed to work to a formal specification of the actual algorithm. This idea is one of the core ideas of computer science and therefore part of any computer science education. Knuth defines computer science as the study of algorithms where an "algorithm is a precisely-defined sequence of rules telling how to produce specified output information from given input information in a finite number of steps" [Knu74]. He outlines what he calls the educational side-effects of studying algorithms:

> It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not *really* understand something until he can teach it to a *computer*, i.e., express it as an algorithm. [...] The attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.

The fact that formalization and programming are at the heart of computer science is also reflected by the ACM Curriculum 91. It lists 12 "recurring principles" of computer science [Tuc91], one of which is the principle of "conceptual and formal models". It is defined as "various ways of formalizing, characterizing, visualizing and thinking about an idea or problem. Examples include formal models in logic, switching theory and the theory of computation, programming language paradigms based upon formal models [...]". The curriculum also argues that programming education plays a central role in computer science education:

> Programming occurs in all nine subject areas in the discipline of computing. It is part of the design process, it is used to implement the models that occur in the abstraction process, and it even occurs sometimes in the process of proving a theoretical result. [...] programming is an extension of the basic communication skills that students and professionals normally use in day-to-day communication.

1.4 Approach and Contributions of Thesis

The concepts of formalization and programming are at the heart of computer science. Teaching these concepts as part of general education is a difficult challenge, and existing approaches are unsatisfactory. There is a need for intuitive, user-friendly, high-quality educational software environments to support teaching and learning.

We claim that the theory of computation is an excellent vehicle for teaching the fundamentals of computer science. Our approach stresses the transition from an intuitive understanding of a problem solution to its formal description. We propose to use finite state machines as the model of programming in educational programming software. Chapter 2 describes this approach and the rationale for it in detail.

To support our claim, we have developed the Kara environments. These environments are educational software laboratories for teaching some fundamentals of computer science. Figure 1.3 shows an overview diagram of the environments.

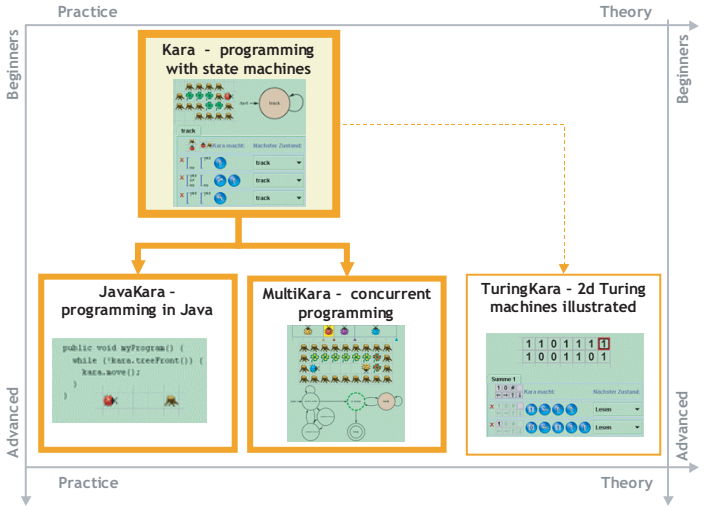

Fig. 1.3: The Kara environments

The Kara environments are the main contribution of this thesis. They address the following topics of computer science education:

Kara An environment for introductory programming based on finite state machines, described in Chapter 3. This is the basic entry-level environment, focussing on sequential programming concepts. There is a long tradition of so-called mini-environments for teaching programming which is discussed in Appendix A.

TuringKara An introduction to the theory of computation based on two-dimensional Turing machines, described in Chapter 4. The two-dimensional model is used because it greatly simplifies solutions in comparison to one-dimensional tape Turing machines.

MultiKara An approach to teach some basic concepts of concurrency in an exploratory software laboratory, described in Chapter 5. We introduce a well-defined classification of concurrency primitives and a new synchronization primitive called the "meeting room".

JavaKara An environment which offers a smooth transition to Java programming, described in Chapter 6. It introduces the basic concepts of imperative programming such as control structures and primitive data types.

Sample programming examples are presented for each environment. These examples are selected to illustrate important ideas that can be taught when using the Kara environments in class. All examples, consisting of problems and their solutions, are integrated into the environments themselves. They are therefore only presented in broad terms in this text.

User interfaces play a decisive role in educational software. Some important considerations in the development of the Kara environments are discussed in Appendix B. The architecture and design of the software itself are described in Appendix C.

Our experience, summarized in Chapter 7, has convinced us that the theory-based approach is worth pursuing. This conviction is strongly supported by the fact that many schools use the Kara environments at different levels.

Chapter 2

Theory of Computation as a Vehicle for Teaching Programming

This chapter presents our approach to teaching introductory programming as part of general education. We pursue a theory-based approach, using a model of programming which is rooted in the theory of computation. A good way to teach programming as an educational exercise, free from utilitarian constraints, is to use toy environments. These environments are designed to illustrate selected concepts in the *simplest possible setting*. The fundamental concepts of programming may be intellectually demanding, but they are not complex in the sense of requiring mastery of lots of details.

2.1 Teaching Programming based on the Theory of Computation

To teach the concept that computers are formal systems, we must choose a formal system within which we can express algorithms in a clear and concise manner. There are many possibilities – each programming language represents such a system. In the past, professional programming languages have typically served as the vehicle of choice for teaching programming, mostly in one of two approaches.

The first approach to teaching programming is to introduce languages such as Java or C# in a step-by-step fashion, where each step covers one construct of the language. Most introductory programming textbooks proceed in this manner. However, there are three objections to this approach. First, these languages are made for the professional programmer and are not tailored to the needs of introductory programming. They are complex and typically object-oriented, posing an additional challenge for the teacher: whereas object-orientation is relevant to "programming in the large", it is a non-trivial additional hurdle for beginners. Second, this approach emphasizes the structure and syntax of a particular language instead of the design of solutions. Third, the tools used in development are professional programming environments with project-management

and debugging facilities. They are complicated to handle and not suited for an introduction to programming.

The second approach to teaching programming is to use a *mini-language* to provide a more functionally oriented introduction to programming. Mini-environments provide students with practical, hands-on experience in environments tailored to their capabilities and needs. The problem with using mini-languages based on real-world languages is that they are inherently complex. It is unfortunate that much of the inherent conceptual complexity of the underlying programming language is carried over to the resulting mini-languages. Another problem is that real-world languages evolve constantly, that new languages are created every few years. Mini-environments based on real-world languages must therefore evolve as well.

The theory of computation offers a way out of the problems of complexity and constant evolution of the above approaches to teaching programming. We propose to use computational models as the models of programming in mini-environments. These models have properties which are highly desirable in introductory programming models. In education, we should strive for utmost conceptual simplicity. Theoretical models of computation are designed to be as simple as possible. They contain only a small number of semantically simple operations. For real life objectives, this may pose a formidable stumbling block and make many tasks difficult beyond practicality. However, in an educational setting, the problems to be solved can be chosen so as to be easily solved within the formal system. Hartmann and Nievergelt argue that the concepts of the theory of computation are not to be viewed only as an exotic specialty, but rather as an educational tool whose potential in computer science education is not yet fully exploited [HN02].

2.2 Requirements of Educational Programming Environments

Mini-environments have a long tradition, with prominent exponents such as the Turtle Geometry of Logo, and Karel the Robot. These and other mini-environments are described in more detail in Appendix A. These environments should satisfy two basic requirements: they should have small languages with simple semantics, and the user should program some type of virtual actor so that the actions of the programs can be easily and intuitively visualized. For the Kara environments, we consider the following extended list of requirements:

Content. The content of educational software should be selected based on the notion of Bruner's *Fundamental Ideas* to guarantee that essential and long-lived topics are taught [Bru60]. On the one hand, this requirement stems from our view of computer science education as part of general education. On the other hand, we emphasize long-lived concepts because software development is highly cost-intensive. Educational software should be designed with the goal of longevity in mind, so that it will not be necessary to update its content every other year.

Scope. The software should be broad in scope with respect to the range of problems students can tackle, from simple get-to-know problems to

tough-to-solve problems. Ideally, these problems cover all levels of Bloom's taxonomy of the cognitive domain in order to provide for intellectual challenges on a number of different levels [Blo56].

Time. In general education, learners must divide their time between many different subjects. Educational software should therefore allow its users to study the underlying content in relatively little time.

Interactivity. The software should allow for what Papert calls constructionism. Students learn best when they are in the active roles of designers and constructors [Pap80]. He argues that this "happens especially felicitously in a context in which the learner is consciously engaged in constructing a public entity, whether it is a sand castle on the beach or a theory of the universe." Software to support this idea must offer a high level of interactivity to allow users to create their own artifacts, and to let them watch the system react.

Usability and Visualization. Users need time to get to know the user interface of educational software. Since the user interface is not the subject of study in itself, it should be as self-explanatory as possible, minimizing the period of adjustment. It is further desirable to visualize the content to make it easier to understand. In educational programming environments, visualizing the execution of a program is particulary important. It helps students visually track what their programs are doing.

Nintendo-like Software. A further requirement of educational software, in particular of educational programming environments, is given by Guzdial and Soloway in *Teaching the Nintendo Generation to Program* [GS02]. The article argues that the high drop-out rates in computer science courses are due to educators having an outdated view of computing and students. Whereas "Hello World" got students excited when computers were still text based, today's "Nintendo generation" grows up in multimedia environments. Educational software needs to correspond to these multimedia environments and to the students' every day use of computers. Doing so, it extrinsically motivates learners, boosting students' exploration of the content underlying the software.

Ideally, mini-environments satisfy all of the above criteria. In particular, we believe that good educational software profits much from a content-selection based on the theory of computation. The main reason for this belief is that a good theory captures the essence of a field of study. It minimizes complex structures while maximizing the range of its applications. For example, there are several programming paradigms (e.g. imperative, object-oriented, logical, functional, distributed) and thousands of programming languages. In a computational sense, the Turing-machine model subsumes all these approaches, yet it is extremely simple. The Church-Turing hypothesis states that every computer-solvable problem can be solved by a Turing-machine – although the solution may be quite complex.

The Nintendo requirement may seem to contradict our theory-based approach to educational software. This is not the case, as the saying that "nothing is more practical than a good theory" applies to our approach. We do not imply

that we strive to teach the theory itself. On the contrary, we use the theory as a means to an end, to teach core issues on the basis of accessible, convincing examples.

2.3 The Kara Programming Model

The *Kara programming model* uses the theory of computation as a vehicle for teaching introductory programming and is based on the model of finite state machines. Using state machines to control toy robots was proposed by Nievergelt [Nie99]. The idea is to have a programming model which brings together theory and practice in a novel way. Using finite state machines has important advantages:

Every day analogies. The concept of finite state machines can be explained to students in terms of every day devices such as light switches, ticket machines, VCRs, or traffic lights.

Natural way to control a robot. A finite state machine basically reacts to inputs, the reaction depending on the current state. This is a natural way to specify the behavior of a robot which has only local information available. If the robot lives in a world which evolves over time, and if it may modify the state of the world itself, then this model is very powerful. It is used in experiments in artificial intelligence (see, for example, Braitenberg [Bra86], Brooks [Bro91]).

Visual programming. Finite state machines have a standard graphical representation, basically circles connected by arrows. They can easily be constructed by purely visual programming. This is a practical advantage for beginners who are often befuddled by textual syntax.

The concept of finite state machines is also interesting in its own right, though this may not be of immediate concern to programming novices. Every computer science student studies them, typically in courses on the theory of computation and in courses on logic design. They also play an important role in modelling the behavior of software systems. The UML standard includes state charts for this purpose [OMG02].

Each Kara environment is designed to illustrate some particular aspect of programming, so the precise definition of the world/actor and programming model depends on the individual environment and is described in detail in the respective chapters. Here, we only describe the basic idea of the environments in broad terms:

World and Actor. The actor is a ladybug in a simple, grid-like world (figure 2.1, left). The bug has sensors such as tree in front?, on leaf?, and it has commands like move, turn left, or pick up leaf.

Finite State Machine. The ladybug is controlled by a finite state machine. Figure 2.1 (right) shows a sample state machine which makes the bug walk in a circuit bordered by trees. For each transition in the state machine, a Boolean expression with the sensors of the actor as variables defines the conditions under which the transition is chosen. For example,

Fig. 2.1: Finite state machine to control circuit walk of ladybug

in state walk, if there is a tree to the left, and a tree in front, but there is no tree to the right, then the third transition will be chosen and the ladybug will turn right and step ahead.

The Kara programming model bridges the gap between the theory of computation and the more practical aspects of introductory programming. Figure 2.2 contrasts some typical properties of the semantics of computational models with the corresponding semantic properties of programming languages such as C++ or Java. The Kara programming model preserves the desirable properties of computational models in a programming model suitable for a mini-environment.

Computational Models	**"Programming Languages"**
♢ concise, non-redundant	♢ redundant (i.e., multiple ways to achieve a goal)
♢ small, self-contained	♢ large (i.e., language plus class libraries)
♢ special or general purpose	♢ general purpose
♢ abstract (e.g., manipulates symbols)	♢ concrete (e.g., manipulates text, screen etc.)
Kara Programming Model	
♢ concise, non-redundant ♢ small, self-contained ♢ special-purpose ♢ concrete ♢ easy-to-use	

Fig. 2.2: Computational Models vs. Programming Languages

The idea of using finite state machines in introductory programming courses has also been proposed by Parnas in *Teaching Programming as Engineering* [Par96]. It is interesting to observe that even though the context of his arguments is different from ours – he argues the case for the education of software engineers – they still reflect our arguments very well. He sketches the mathematical content of a first course on programming for software engineers where the first topic is finite state machines:

> It is essential that students see computers as purely mechanical devices, capable of mathematical description. [...] Students are also taught how to design finite state machines to perform simple tasks. [...] The emphasis is on understanding how the machines function and on designing them.

Parnas does not use finite state machines in the context of mini-environments for introductory programming, but the rationale he gives for using them is similar to ours. He also comments succinctly on the gap between theory of computation and programming:

> Instead of reacting to rapid change by focusing on fundamentals, programming books and courses try to keep up with the latest developments. [...] The few books that claim to focus on fundamentals are highly theoretical. [...] students are given theories that, while they don't go out of date, do not seem relevant to the task of programming.

The Kara programming model tries to address this situation by bridging the gap between theory and practice. We are convinced that students will learn to focus on problem-solving skills rather than on a particular tool in the form of some programming language. We are also convinced that the Kara model is concrete enough for students not to get lost in the theory of computation itself. Rather, it enables them to use the theory as a means to an end.

Chapter 3

Kara: Introduction to Programming for Beginners

The Kara environment serves as a vehicle to introduce the basics of sequential programming to novices (see [RNH00], [HNR01], [RNH01]). Kara shows that basic ideas of programming can be learned in a playful, enjoyable environment. Fundamental ideas such as correctness proofs and Boolean logic can be taught in an illustrative manner.

3.1 The Kara Environment

In the Kara environment, the ladybug lives alone in a flat grid world. The number of types of objects in the world, as well as the number of sensors and commands of the actor is kept to a minimum. Figure 3.1 shows (left) the main window with the world editor, the commands to control the bug, and the buttons to control program execution; (right) the visual program editor. In the example shown, the ladybug has to follow a trail of leaves and eat them up. The program is a one-state solution to this problem.

There are only three types of objects in the world: unmovable tree trunks, mushrooms which the bug can push around, and leaves, of which it can lay down or pick up any number. The bug itself is placed on either a free square or a square occupied by a leaf, facing one of four possible directions. Sensors inform it about its immediate surroundings:


Is there a tree on the square in front?


Is there a tree on the square to the right?


Is there a tree on the square to the left?


Is there a leaf underneath?


Is there a mushroom in front?

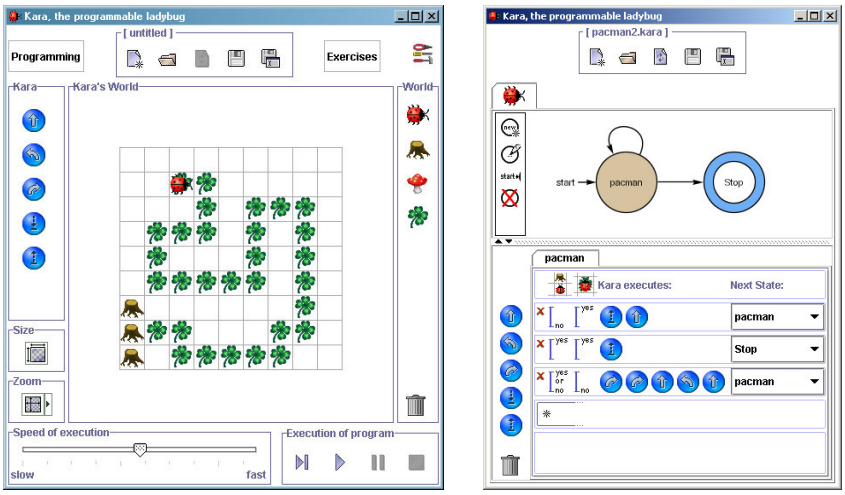

Fig. 3.1: The Kara environment

The bug actor can execute five primitive actions: advance one square in the current direction; turn right or left by 90° on the current square; put down or pick up a leaf. The world is a torus. When the bug steps off the world, it will reenter from the opposite side.

A number of exercise examples are integrated in the environment. An exercise consists of a problem statement, sample test worlds, and solutions (figure 3.2). Solutions have a description, and users can load the solution program into the program editor. Some exercises have a test case module which runs a user's program against a number of automated test cases and checks whether they satisfy the specification of the solution of the problem.

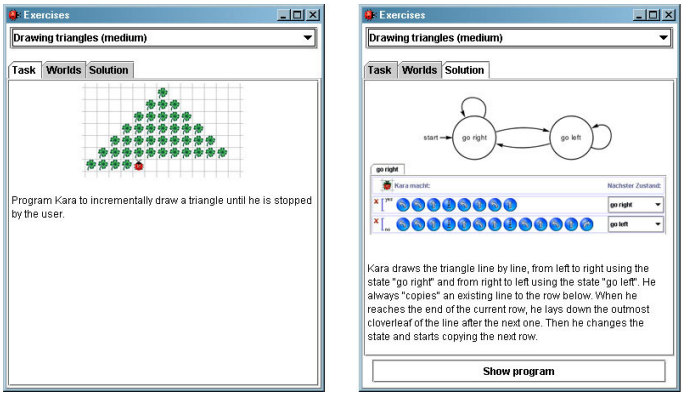

Fig. 3.2: Exercises in Kara

3.2 Examples

The following examples of finite state machines illustrate what can be taught with Kara. Each example was selected to make a particular point. Simple "getting-started" tasks are omitted.

Tracking a Wall of Trees: A One-State Example with Invariant

This example illustrates in a graphical manner how invariants can be used to ensure the correctness of a program. Consider a wall of trees of arbitrary shape, with arbitrary appendages, as shown in figure 3.3. The bug must be programmed to cycle endlessly around the perimeter of the wall, hugging it with its right side.

Fig. 3.3: Wall of trees, and program for "follow wall"

For the sake of simplicity, let us assume that program execution starts in a position satisfying the precondition shown in figure 3.4. At least one of the trees shown must be present.

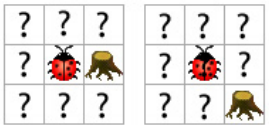

Fig. 3.4: Predicate "wall to the right (behind)"

The program has only one state track (figure 3.3). It maintains the precondition in each transition, making the precondition an invariant of state track. If there is no tree to the bug's right side, it turns right and steps forward. If there is a tree to its right, it steps ahead or turns left, depending on whether there is a tree in front of it. The only tricky case is when there is a tree in front. The bug may not turn left and then step ahead in a single transition, because it does not know whether there is tree to its left side or not.

The program is simple: it checks all possible configurations of the relevant sensors and acts accordingly. The finite state machine model is well suited to specifying the necessary logic of this solution in a concise yet readable manner. Also, the importance of invariants in reasoning about the correctness of algorithms can be shown in an illustrative setting.

Slalom Tours: A Simple Two-States Program

This example shows how two almost identical states complement each other. This is a principle found in many state machine examples. The bug is to walk a slalom around the trees in figure 3.5. The basic algorithm is simple: start with a left 'turn' around a tree of 180°, do a right turn of 180°, then do a left turn ... until reaching the right-most tree, where a turn of 360° is necessary.

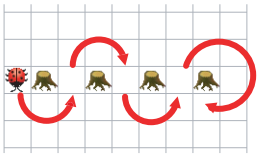

Fig. 3.5: Slalom between trees

The state machine illustrates how states can be used to simulate a Boolean flag. Between the trees, the program must either do a left turn or a right turn. The program in figure 3.6 uses the two states left and right to distinguish these two cases.

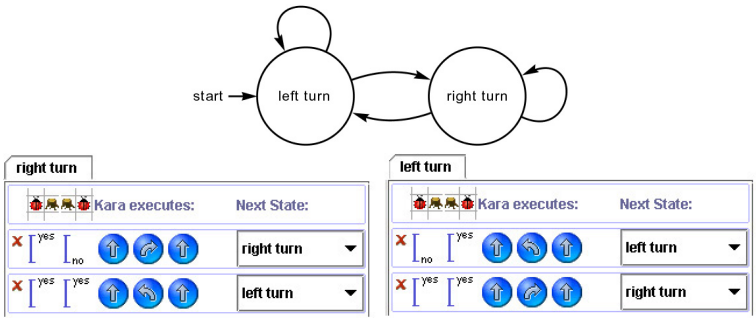

Fig. 3.6: The slalom program

Painting Binary Pascal Triangles: Using States as Memory

This example shows how states can be used as the memory in a calculation. Figure 3.7 shows a leaf pattern which represents the integers modulo 2 of the Pascal triangle, with its apex at the top left corner of the grid. An even number is represented as an empty square, an odd number as a square with a leaf on it. Creating the Pascal triangle pattern is quite challenging for students.

Figure 3.7 also shows the basic algorithm of the Pascal triangle construction. The ladybug paints the triangle row by row, walking from left to right. Upon reaching a tree, it walks back to the left, advancing to the next row.

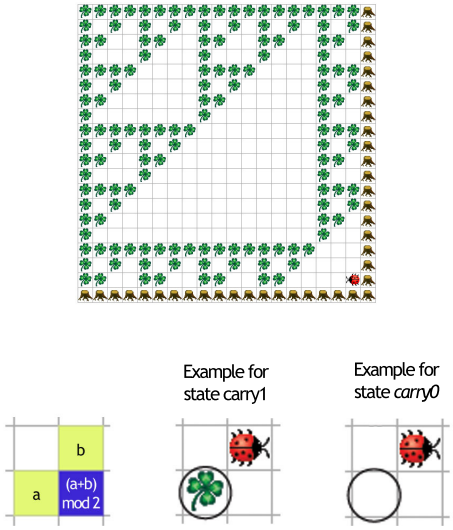

Fig. 3.7: The Pascal triangle and the calculation of its squares

The values of the squares are determined by the bug using its sensor on leaf and its current state. The sensor determines whether the bug is on a square with a leaf or not (square b in figure 3.7). The current state is used to remember whether there is a leaf on square a. In state carry0, the bug knows that square a represents an even number, and in state carry1, the bug knows that square a represents an odd number. The combination of sensor input and current state determines whether the bug will put a leaf on the square to its right, transiting to state carry1, or whether it will simply step ahead, transiting to state carry0. Figure 3.8 shows the program with its three states.

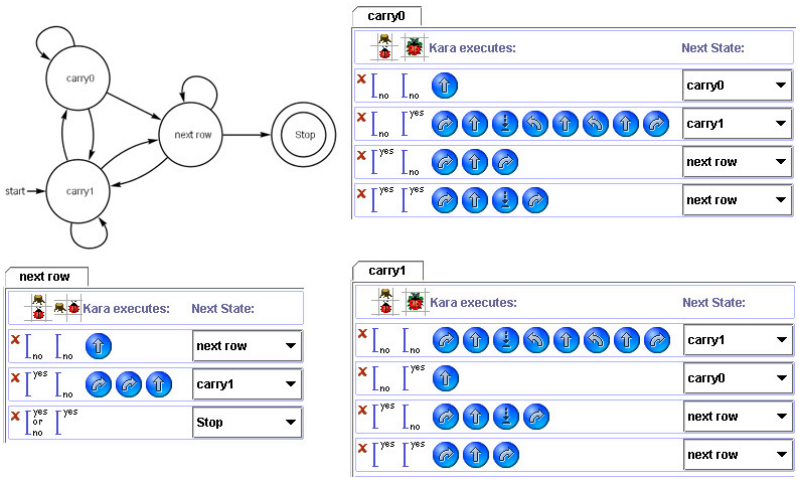

Fig. 3.8: Program for binary Pascal Triangles

Kara and the Automatic Ant: Simple State Machine, Complex Behavior

Kara can also be used to study the mysteriously complex behavior of simple Turing machines. Christopher Langton introduced the so-called automatic ant to illustrate this behavior (see, for example [Gal98]). An ant works on an infinite two-dimensional array, and follows a simple algorithm. If it is on a white square, it changes its color to black and moves to the square to its right. If it is on a black square, it changes its color to white and moves to the square to its left. Figure 3.9 shows a Kara program simulating an ant, where white is interpreted as an empty square, and black as a square with a leaf on it.

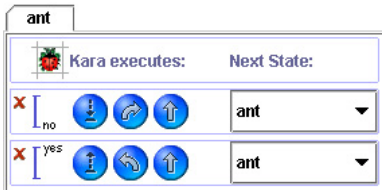

Fig. 3.9: Program for Langton's Ant

In an initially empty world, after about 10'000 state transitions with no easily recognizable pattern, the ant starts building a structure called highway by its discoverer James Propp. Figure 3.10 (left) shows a highway in Kara's world. It seems that in an world of unbounded size, an ant always starts building highways after a while, regardless of the initial configuration of the world. This conjecture remains unproven. L. A. Bunimovch and S. E. Troubetskoy proved that the path of an ant in an array of unbounded size is always unlimited, that is, the path of the ant will leave any bounded area.

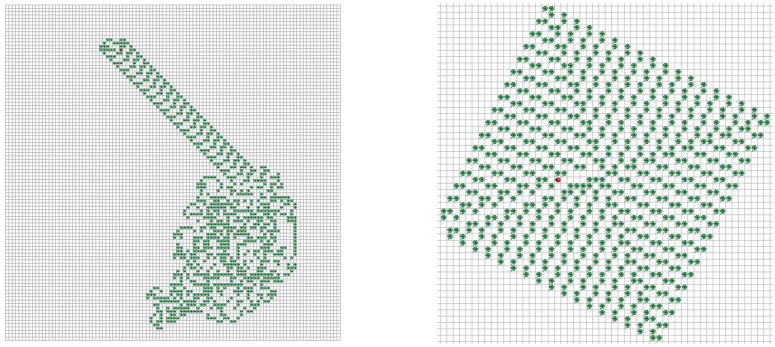

Fig. 3.10: Left: World of Langton's ant. Right: A modified ant's world.

Slight variations in the ant's algorithm result in drastic changes of the created pattern. For example, Horst Müller has modified the ant program so that the bug makes two steps ahead after a left turn, instead of one step (private communication). The resulting pattern is highly regular in contrast to the pattern of the original ant (figure 3.10, right). The ant example shows how

fragile algorithms can be and that small local changes can have drastic global consequences.

Models of Computation: Assumptions and Their Implications

Using finite state machines to control the ladybug might seem to impose severe restrictions regarding the range of tasks the bug can achieve. This leads to one of the core questions of computer science: what can be computed within different models of computation? The Kara environment can be applied to the investigation of this question. The "finite state machine" illustrates different computing models and shows how important it is to precisely define all the details. Subtle differences with respect to what the bug can do in its world change its power of computation. In this theoretical discussion we consider an idealized version of the bug living in a world of infinite size in which only leaves may exist.

"Read Only"-World – Finite State Machine Model

In this model, the ladybug is not allowed to modify the world, it may only move about. Under this assumption, its programs are finite state machines per definition. In an empty world, an automaton will soon either stop or fall into a periodic pattern. As a consequence, walks such as the infinite spiral in figure 3.11 are impossible. To distinguish the different, increasing lengths of the boundaries of the spiral, a state machine would need an unbounded number of states.

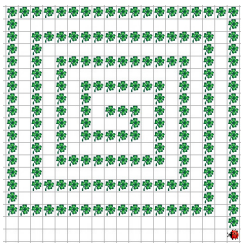

Fig. 3.11: Impossible spiral walk in an empty world

If the world is not empty, the bug can solve more interesting problems. In particular, one can consider pattern matching. Figure 3.12 shows how Kara can test a given world area for the occurrence of the pattern "empty square, square with leaf, empty square".

Pattern matching leads to the theory of regular languages, to the question which languages can be recognized by a state machine and which cannot. Figure 3.13 (top) shows an element of the regular "leaf language" containing all drawings of the structure "empty square, square with leaf", repeated an arbitrary number of times. A simple program with two states can decide whether a given drawing is an element of this language or not. However, deciding whether a drawing consists of any number of empty squares followed by the same number of squares with leaves exceeds the capabilities of a finite state machine (figure

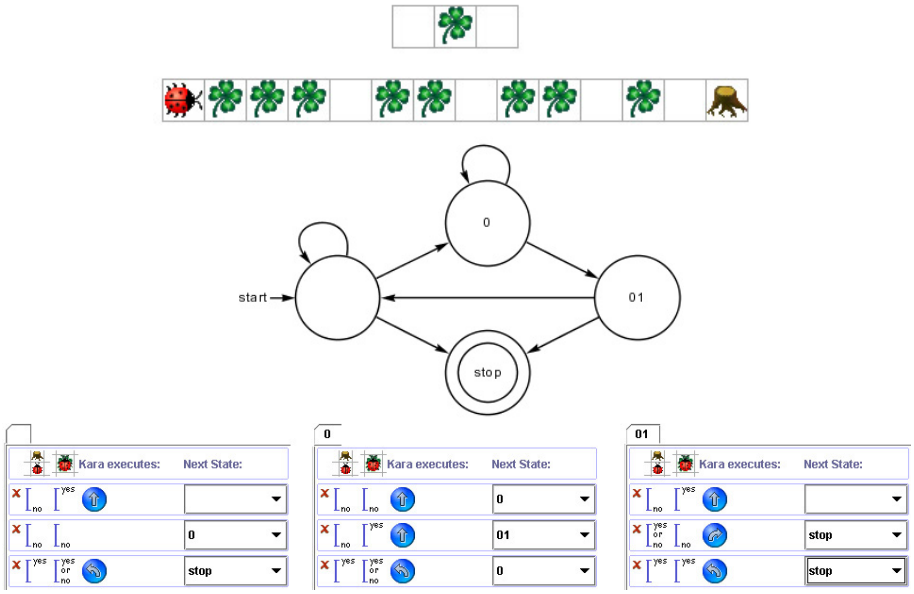

Fig. 3.12: Matching a simple pattern (world and program)

3.13, bottom). As in the spiral walk problem, the state machine would need to distinguish the possible numbers of squares in its state space, which is not possible in a world of arbitrary, unlimited size with a bounded number of states.

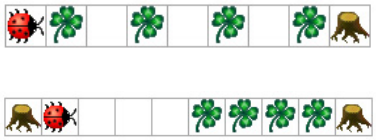

Fig. 3.13: Top: A pattern recognizable by a state machine (i.e., $(01)^n$). Bottom: A pattern not recognizable by a state machine (i.e., 0^n1^n).

"Read-Write"-World – Turing Machine

If the ladybug may lay down and pick up leaves at arbitrary places, it can use the world as a "read/write" memory of unbounded size. Under this assumption, its programs become the most powerful computing model in the hierarchy of automata – Turing machines which can compute anything algorithmically computable (given suitable encoding of input and output). As an example, figure 3.14 shows how a simple four-state machine (not shown) decides whether a drawing is an element of the non-regular language $\{0^n1^n\}$. The program proceeds by using one square as a marker, and moving leaves from right to left. The drawing is an element of the language if and only if the bug is on the marker at the end of program execution.

Fig. 3.14: Is a drawing an element of the non-regular language $\{0^n1^n\}$?

Exhaustive walks in arbitrary labyrinths: The Power of State Machines

Kara can also be used to show how difficult it can be to determine whether a problem can be solved under certain assumptions. When the ladybug is restricted by a wall of trees to an enclosed part of the world, it is not obvious which problems can be solved and which are unsolvable. The fact that its programs are Turing machines when working in a world of unbounded size is of no consequence when it is limited by a wall of trees. As an example, consider the labyrinth shown in figure 3.15. The bug is to visit each square in this labyrinth and put a leaf on it.

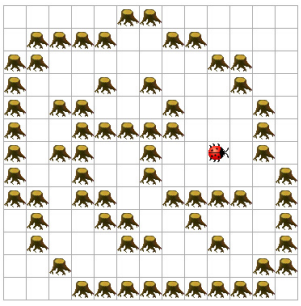

Fig. 3.15: A labyrinth for an exhaustive walk

A finite state machine can be implemented to tackle this problem using a backtracking algorithm. The proof of existence and the construction procedure for such an automaton was given by Horst Müller in his 1977 paper *A One-Symbol Printing Automaton Escaping from Every Labyrinth* [Mue77]. Actually constructing the automaton would require tedious work as it consists of 7968 states. In contrast to the standard definition of finite state machines, the Kara model allows an arbitrary number of commands for each transition. Horst Müller estimates that this would bring down the number of states needed to somewhere around several hundred states. To show how the algorithm works, he has implemented it with JavaKara, simulating the state machine in Java. The program has roughly 1500 lines of code and illustrates the complexity of

working with the limited capabilities of the finite state machine model.

The basic idea of Horst Müller's algorithm is that if the bug were able to distinguish four directional arrows on the squares of the world, a standard backtracking algorithm could be implemented. Arrow squares are simulated in Kara's world by considering "macro squares" consisting of four regular squares (figure 3.16). A clever interpretation of the possible combinations of empty squares, squares with trees, and squares with leaves within such a macro square yields an encoding of the four arrows. Parsing a macro square and distinguishing the arrows is rather tedious, as the bug does not have a "bird's eye" view of the world.

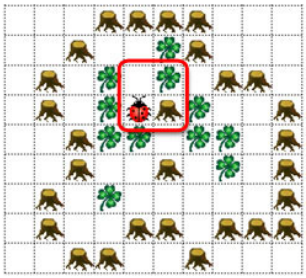

Fig. 3.16: Four squares constitute a macro square

3.3 Conclusions

The theory-based approach of Kara, which emphasizes a simple, yet powerful model of programming, yields an educational tool which can be applied with a broad range of audiences. Our experience, as discussed in Chapter 7, shows that it can be used for introductory programming in elementary and secondary schools, and it can be used in courses on the theory of computation with computer science students. The visual and intuitive nature of Kara illustrates otherwise abstract concepts such as proofs of correctness, invariants, or computational models in graphic examples. There is no need to learn a complex real-world programming language in order to study such fundamentals of programming.

Chapter 4

TuringKara: Two-Dimensional Turing Machines

The concept of Turing machines is an important topic in computer science education. The fact that with respect to the power of computability Turing machines and PCs are equivalent is an important point to be made. The intention of TuringKara is to support teaching this topic by providing an educational environment in which students can experiment with Turing machines in an easily accessible manner.

4.1 The Rationale for TuringKara's Two-Dimensional 'Sheet'

In the theory of computation, models of computation are of prime interest. The concept of computational models arose from the question "What can be computed by an algorithm, and what cannot?". The various formal models of computation, such as production systems, Turing machines, recursive functions, and the lambda calculus capture the intuitive concept of "computation by the application of precise rules". The standard universal models of computation were designed to be conceptually simple. It usually comes as a surprise to novices that the set of primitives of a universal computing machine can be simple, as long as these machines possess the two essential ingredients of unbounded memory and unbounded time.

Turing machines are a universal model of computation which play an important role in the study of computability. The standard Turing machine works on a single tape of unbounded length. Turing argued that with regards to computability, he could restrict the external memory of his model to a simple, one-dimensional tape [Tur37]:

> Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character

> of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e., on a tape divided into squares.

The definition of Turing machines is easy to understand, and their execution can easily be visualized. However, many examples of Turing machines are artificially selected because they solve problems which have simple solutions. Problems of interest, such as the addition of two numbers, are quite complex on standard Turing machines which work on one-dimensional tapes.

For its educational purpose, the TuringKara environment uses Turing machines which work on a two-dimensional sheet. Even though the sheet is irrelevant with regards to computability, it greatly simplifies the solutions of many problems. Note that in TuringKara, the world is not of unbounded size; it has a user-defined size. For most problems, this is not a restriction, as one can set the world size to suit the needs of the problem.

As an example, the task shown in figure 4.1 is to add a column of binary integers. These numbers are aligned vertically, as one would do on paper. A four-state Turing machine implements the addition and is quite simple. Solving the same problem on a one-dimensional tape is significantly more complicated, as the read/write head of the tape would have to do far more movements.

4.2 The TuringKara Environment

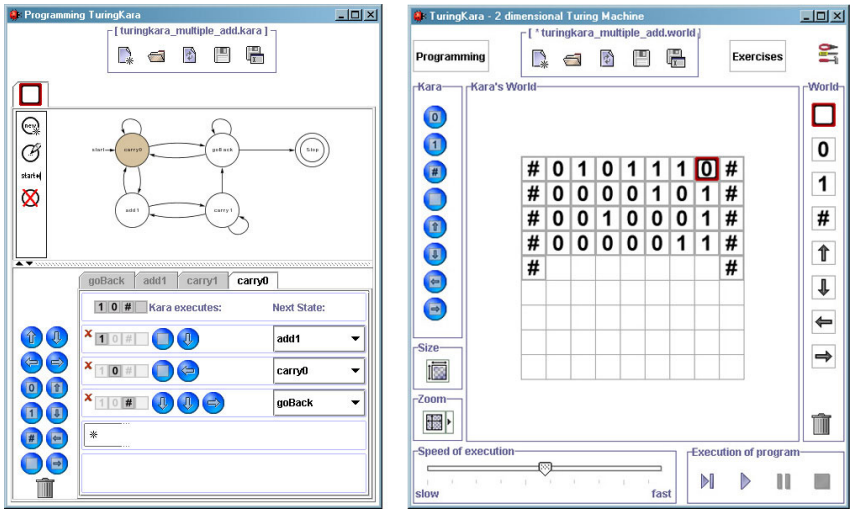

Fig. 4.1: The TuringKara environment

Figure 4.1 shows the user interface of the TuringKara environment which was implemented jointly with Markus Brändle. An effort was made to apply as few changes as necessary to the user interface of the Kara environment. To users, TuringKara should feel instantly familiar.

The read/write head, visualized as a red rectangle, can be moved up, down, right, and left, and has no sense of direction. The world objects are symbols of the alphabet 0, 1, #, □ (empty square), ←, →, ↑, ↓. The read/write head can write any symbol on any square in the world, regardless of its current symbol. The symbols do not have any inherent semantic for the read/write head and can be used for any purpose. They were chosen for reasons of convenience, as the arrows provide a means of placing markers in the world. For example, figure 4.2 shows a labyrinth whose borders are marked by the # symbol. The task is to visit each square within the labyrinth and mark it with the symbol 0. The figure shows how the program uses the arrows as markers to remember from which direction the read/write head entered a square.

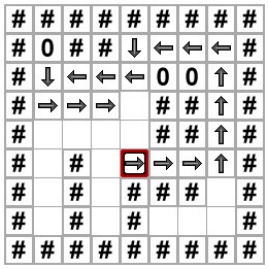

Fig. 4.2: Backtracking using arrows as path markers

The program editor is almost identical to the program editor of Kara. There is only one difference: the programmer can associate a set of symbols with each transition. For example, in the transition table shown in figure 4.3, the first transition will be chosen if the read/write head is on *any* of the following symbols: $0, \#, \leftarrow, \rightarrow, \uparrow, \downarrow$. If it is on an empty square, the second transition will be chosen; if it is on a 1, an error will occur.

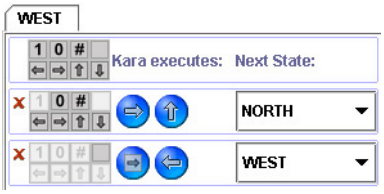

Fig. 4.3: A state transition table in TuringKara

The definition of TuringKara machines differs from the definition of standard Turing machines. In a standard Turing machine, a transition specifies exactly one symbol to be written and one move command of the read/write head to be executed. In TuringKara machines, any number of write and move commands can be associated with a transition. TuringKara machines are equivalent to standard Turing machines, and the conversion is straight-forward. The advantage of the TuringKara model is that it significantly decreases the number of states needed for most Turing machines.

4.3 Examples

The many examples for TuringKara include typical one-dimensional problems such as deciding whether an input string is part of some language or not, for example, whether an input string is a palindrome or not. However, problems exploiting the two-dimensional world are more interesting. The following examples illustrate the power of the two-dimensional world.

Multiplication of Binary Integers

In elementary school, every pupil learns how to multiply two numbers. This example shows how one can 'teach' multiplication to a Turing machine and the degree of precision required to do so. We have implemented the 'Gipsy' multiplication algorithm instead of the standard algorithm taught in elementary schools to simplify the resulting Turing machine. This algorithm is particularly easy to implement with binary representations of numbers: one multiplicand is divided by two, while the second multiplicand is multiplied by two. Multiplication and division by two can be done with simple left and right shifts, respectively. If the first multiplicand is odd, the second is added to the current result. When the first multiplicand becomes 1, the algorithm terminates.
This multiplication algorithm is relatively easy to implement with an nine-state TuringKara machine. Figure 4.4 shows how the multiplication of $5 * 4$ proceeds in the world. The two multiplicands are in the first line, separated by #, and the result is on the second line, initially 0. Dividing and multiplying the multiplicands in each step takes place on the first row only. When the second multiplicand must be added to the current result, the two-dimensional world is used to achieve a simple addition algorithm, with only little movement of the read/write head.

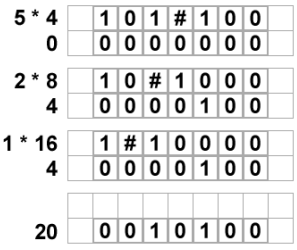

Fig. 4.4: Multiplying $5 * 4$ with TuringKara

Two-Dimensional Universal Turing Machine to Simulate Standard Turing Machine

The concept of a universal Turing machine (UTM) is very important in the theory of computation. It explains the idea of programmable machines by showing how one machine can be programmed to simulate an arbitrary machine. A UTM takes as its input another Turing machine, both its program and its tape, and simulates its execution. Markus Brändle has implemented a universal Turing machine in TuringKara which has 41 states. It is capable of simulating

any Turing machine which satisfies the following constraints: it works on a one-dimensional tape, and its alphabet only contains the symbols 0, 1, #, □. Even though the TuringKara UTM is complex, its simulation algorithm is quite straight-forward and fascinating to watch. It exploits the two-dimensional world to encode the Turing machine to be simulated in a visually intuitive fashion.

In the following, we show how a Turing machine must be encoded in the world before it can be simulated by the TuringKara UTM. Figure 4.5 shows how a single transition of the Turing machine to be simulated is encoded within the world. a, b are elements of its alphabet, i.e., 0, 1, #, □; m is the move of the read/write head to be made, i.e., $m \in \{\leftarrow, \square(stay), \rightarrow\}$.

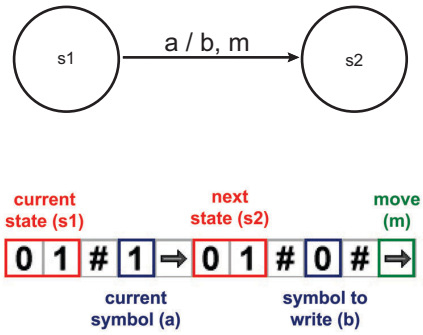

Fig. 4.5: A transition and its encoding for the TuringKara UTM

As an example, we encode a string-inverting Turing machine for simulation by the UTM. Figure 4.6 shows how a string inverter can be implemented as a TuringKara machine. The machine has a single state inverter with three transitions. It replaces 0 by 1 and 1 by 0 until it reaches the # symbol.

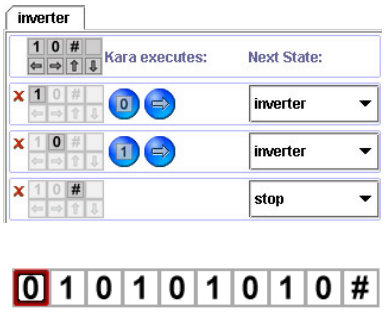

Fig. 4.6: The inverter Turing machine to be simulated

For the TuringKara UTM, each transition of the Turing machine to be simulated must be encoded in the world, as well as its start state and its tape. Figure 4.7 shows the encoding of the string-inverter. Its three transitions are encoded as described above, its start state is specified at the top of the world, and its initial tape is given at the bottom of the world. Some # symbols and empty lines are used by the UTM as markers and memory space.

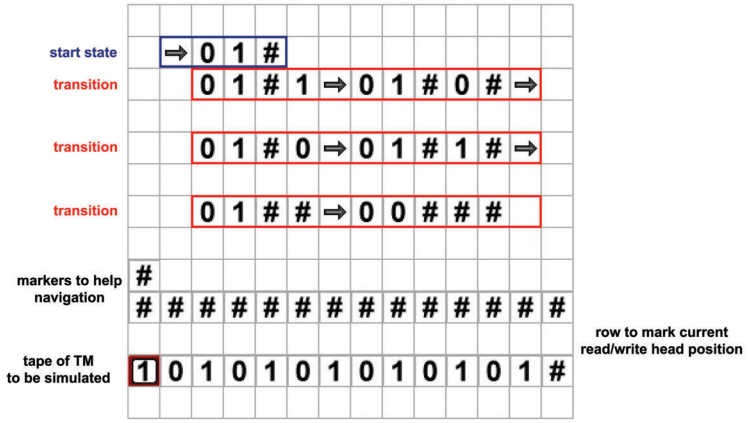

Fig. 4.7: The string-inverter encoded for the TuringKara UTM

Non-Computable: Two-dimensional Busy Beavers

This example addresses the core question of what is computable, or decidable, with a Turing machine. This question is of particular interest in the theory of computation. The standard example of an undecidable problem is the halting problem: a Turing machine cannot decide whether another Turing machine will terminate or not.

In 1962, Tibor Rado invented an instructive non-computable function. The function $\Sigma(n)$ is defined as the maximum number of marks a Turing machine with n states produces on an initially empty tape before termination. Turing machines with n states which produce $\Sigma(n)$ marks are called "busy beavers". The function $\Sigma(n)$ is well-defined but non-computable. In *The Turing Omnibus*, Dewdney mentions a result by M. W. Green from 1964 which illustrates what non-computable means in the case of the busy beaver function [Dew89]: $\Sigma(n)$ grows faster than any computable function. Formally, the following holds for any computable function $f(n)$, for infinitely many values of n:

$$f(\Sigma(n)) < \Sigma(n+1)$$

The four-state busy beaver produces 13 marks on the one-dimensional tape. Heiner Marxen has shown that the five-state busy beaver produces at least 4'098, and the six-state beaver at least 1.29×10^{865} marks [MB90]. Dewdney explains why busy beavers are so hard to find [Dew89]:

> One reason for the enormous difficulty of the busy beaver problem lies in the ability of relatively small Turing machines to encode

> profound unproved mathematical conjectures such as Fermat's last 'theorem' or the Goldbach conjecture (every even number is the sum of two primes). Knowledge of whether such machines halt is tantamount to proving or disproving such a conjecture.

Though Fermat's last theorem has been proven by now, the busy beaver function is provably non-computable. TuringKara can be used to simulate known busy beavers for one-dimensional tapes, at least those which produce only a relatively small number of marks. It can also be used to investigate busy beavers on its two-dimensional world. How many marks can a three-state TuringKara machine produce before terminating? We conducted a small contest, with the following constraints: only the symbols 1 and □ (i.e. blank square) could be used, and transitions must conform to the standard Turing machine definition. That is, each transition has exactly one write command followed optionally by a move command. Figure 4.8 shows the TuringKara machine we assume to be a three-state TuringKara busy beaver.

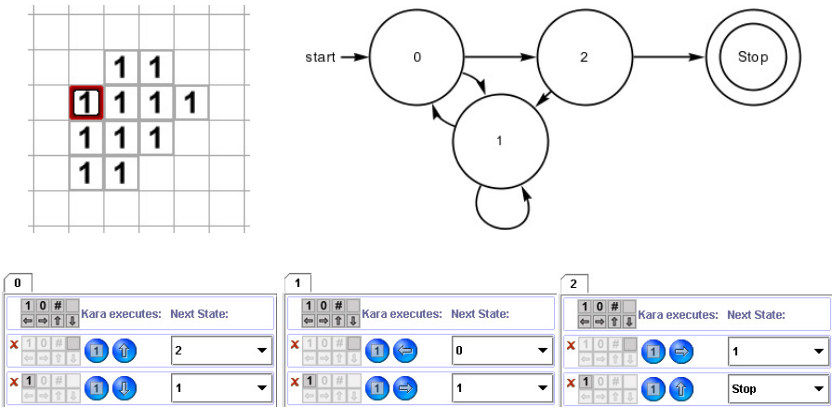

Fig. 4.8: TuringKara three-state busy beaver candidate and its world

4.4 Related Work

There are many Turing machine simulations available on the web as Java-applets. Most of these applets fall into one of two categories. The first category consists of applets in which users create Turing machines in a textual way, by entering the transitions manually. This is tedious, error-prone, and not motivating for students as it forces them to concentrate on using the correct notation instead of focussing on solving a problem. The second category consists of applets which offer a graphical representation of a given Turing machine. However, in many of these applets, users cannot create or modify the machine, but merely watch its execution. There is no real interaction and students are not involved in solving problems. There are also some Turing machine environments which do not fall into either category. For example, in Barwise and Etchemendy's *Turing's World*, Turing machines are created and edited graphically [BE00]. However, Turing's World is commercial Macintosh software, severely restricting its application in public schools.

4.5 Conclusions

TuringKara 'inherits' all advantages of the Kara environment: A visual and intuitive representation of problems and their solutions, and a broad range of challenging problems. It seems that two-dimensional Turing machines have not been used as an educational means. Yet the two-dimensional world of TuringKara greatly simplifies many Turing machines, making it possible for students to implement solutions of interesting problems with a reasonable effort. Our experience shows that TuringKara is a motivating approach to introduce students to the theory of computation.

Chapter 5

MultiKara: Introduction to Concurrent Programming

Concurrent programming is part of every computer science education at university level. It is a complex and mathematically difficult subject of growing importance. The intention of MultiKara is to support the intuition of students about concurrency by providing an educational software laboratory in which they can study concurrency problems and mechanisms to solve them.

5.1 The Challenges of Concurrent Programming

Concurrent programming originally was the realm of specialists building time-sharing operating systems or real-time device controllers. In the last 30 years, concurrent programming has become part of the mainstream of programming. When graphical user interfaces gave rise to event-driven programming, multi-threaded programming was no longer an option, but became a necessity. Multi-threading makes it possible to maintain the interactivity of the user interface while slower processes such as printing go on in the background. Programming languages were adapted to incorporate concurrency primitives. The importance of concurrent programming will continue to grow as it provides the basis for distributed programming. Future systems are going to make increasing use of distributed services through the Internet.

The challenge in teaching concurrent programming is that it differs in fundamental ways from sequential programming. One of the biggest challenges programmers face is the non-determinism introduced by the unpredictable interleaving of different threads of execution. For programmers, this has two consequences:

Program Analysis. Many programmers analyze their sequential programs by mentally simulating its dynamic behavior at run-time. This is difficult even for sequential programs, as there is a huge number of possible paths of execution through any non-trivial program. With concurrent programs, a pseudo-dynamic analysis is impractical. Interleaving leads to an explosion of the number of possible execution paths through

the different threads. It therefore becomes imperative to analyze the static properties of concurrent programs.

Testing and Debugging. Testing can reveal the presence of errors in sequential programs but cannot prove their absence. Due to the nondeterminism of concurrent program execution, testing gives even less information about errors in a concurrent program. And even if a test reveals the presence of an error, concurrent programs are notoriously hard to debug because reproducing the error may be almost impossible.

The view that concurrent programming requires a different kind of mind-set than sequential programming is reflected by Ben-Ari and Kolikant [BAK99]. They argue that concurrent and distributed programming should be taught at the high school level. They observe that "the challenging nature of the subject ensures that students must learn how to use critical thinking rather than hacking" when writing concurrent programs. Students appreciated the value of correctness proofs as an essential part of concurrent programming.

A further challenge for programmers is the mismatch between theory and practice. In theory, there are well defined concurrency primitives, such as semaphores, monitors, condition variables, asynchronous message passing, or communicating sequential processes. In practice, concurrency mechanisms are highly language-dependent. In Java, for example, the mutual exclusion mechanism is based on monitors, but only in a rudimentary way. Additional packages such as those presented by Lea are needed to actually use the models as defined in theory [Lea99]. In C#, as an other example, shared-memory synchronization mechanisms are part of the standard libraries. And there are many other types of primitives in other languages, for example, the SCOOP model of the Eiffel language presented by Meyer [Mey97]. Dealing with all these different primitives is quite challenging for developers.

5.2 MultiKara: Educational Laboratory for High-Level Concurrency Mechanisms

MultiKara is intended to give students an impression of the problems of concurrent programming and of a number of mechanisms to solve them. Because concurrent programming at the lower level is a very technical subject, MultiKara focuses on high-level mechanisms. These mechanisms can be used to coordinate the four ladybugs in MultiKara. The environment was implemented by Markus Brändle [Bra02] and Tobias Schlatter [Sch02].

Of the typical topics in introductions to concurrency – such as nondeterminism, process creation and activation, synchronization, process termination, process scheduling, and interprocess communication – MultiKara addresses issues of nondeterminism, scheduling, and concurrency mechanisms for mutual exclusion and synchronization.

MultiKara supports four concurrency mechanisms which fall into two categories: *inclusive* and *exclusive*. The inclusive mechanisms are used to temporally synchronize processes. The exclusive mechanisms achieve mutual exclusion of processes. The mechanisms of both categories can be applied either within the *state space* of the program or within the data structure of the *world*. The following table shows the four mechanisms supported by the MultiKara environment:

	inclusive	*exclusive*
world space	**meeting room**	**monitor**
state space	**barrier**	**critical section**

These concurrency mechanisms are what we call *anonymous*, for the sake of simplicity. That is, they are independent of the identity of the actors. There is no way, for example, to have mutual exclusion only between a subset of actors, or to define a barrier which only a subset must enter to release it. Also, there is no message passing mechanism in MultiKara because message passing schemes are not anonymous.

Of the four concurrency mechanisms in MultiKara, *monitor*, *barrier* und *critical section* are standard mechanisms of concurrent programming found in textbooks on the subject such as *Principles of Concurrent and Distributed Programming* [BA90]. The *monitor* guarantees mutual exclusion of the ladybug actors on user-defined elements of the world data structure. A state marked as a *critical section* is protected by binary semaphores so that at most one process is in a critical section state at run-time. The *barrier* is a special case of the rendezvous mechanism in which no data is exchanged; all actors must reach a barrier state before execution will continue. The term *meeting room* was coined to illustrate the underlying synchronization mechanism: The meeting begins only when all participants are present. The meeting room mechanism was introduced to illustrate the concept of barrier within the world of the actors.

It is interesting to note that there is an effort to apply aspect-oriented programming to separate concurrency-related code from the code it is working on. The idea is that a programmer need not worry about concurrency. In the case that some code must be used in concurrent settings, aspects would be used to add the necessary code. However, it is unclear whether this approach will be successful, and whether such a clear separation of concerns is possible with regards to concurrency.

5.3 The MultiKara Environment

Figure 5.1 shows the user interface of the MultiKara environment. An effort was made to apply as few changes as necessary to Kara's user interface. To users, the new environment should feel instantly familiar even though its user interface is more complex.

The program editor provides a separate state machine editor for each ladybug. At run-time, the state machines are executed independently, their threads of execution managed by the scheduler. The only way the programs can interact is by using the world and the objects within or via the built-in concurrency mechanisms.

Explicit Time Model and Scheduling

The programs of the individual actors are executed by interleaving their threads of execution rather than by simulating parallel execution. Time is divided into discrete intervals. During a single interval, one state transition is executed completely: first, all used sensors are tested and the appropriate transition is selected, and second, all its commands are executed. The fact that a transition

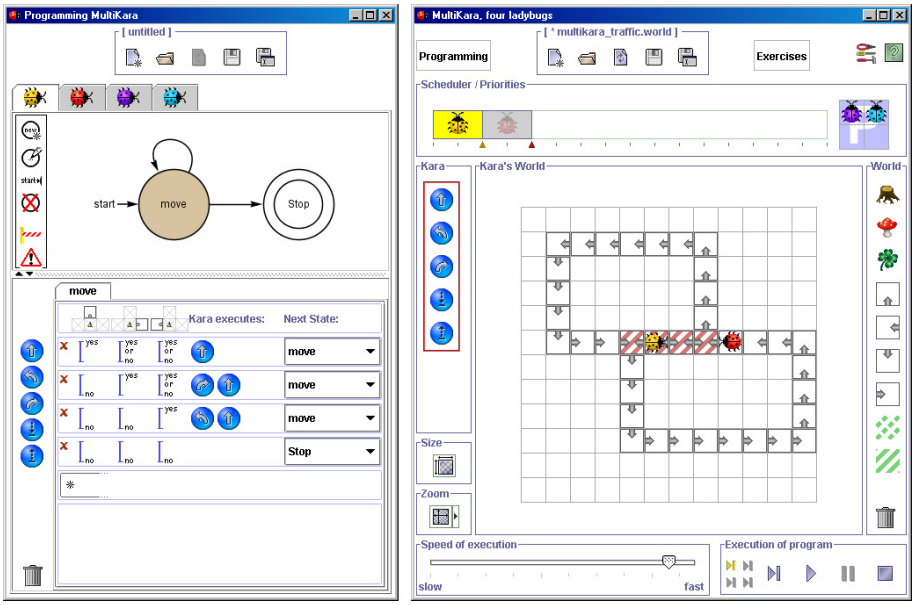

Fig. 5.1: The MultiKara environment

is atomic makes collision avoidance easy. Without atomic transitions, even a simple step-ahead command would be a problem. The situation shown in figure 5.2 illustrates this. Imagine one ladybug testing if the square in front is empty right before being interrupted. While it is asleep, the other ladybug steps forward and occupies the square. On its next turn, the first bug steps forward and collides with the other one.

Fig. 5.2: Two ladybugs which do not see each other

In the MultiKara "Scheduler / Priorities" panel, users assign priorities to the processes of the bugs, with a total of 16 time intervals (figure 5.3). If users do not want the program of a ladybug to be executed, they can drag&drop it onto the parking lot in the scheduler panel.

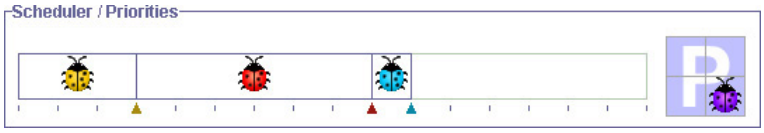

Fig. 5.3: Scheduler / Priorities adjustment panel

When users start program execution, the scheduler creates a new process for each active ladybug. These processes are initially put into execution state *ready to run*. Figure 5.4 shows the life cycle diagram of processes in MultiKara.

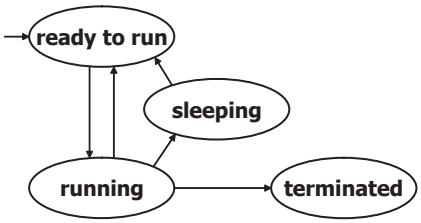

Fig. 5.4: Process life cycle

The scheduler determines which process will be the next to execute a state transition by choosing one of the processes in execution state *ready to run*. The probability of a process being selected is given by its relative priority among all *ready to run* processes. The process chosen by the scheduler is put into execution state *running*. After having executed its transition, the process is put either into execution state *ready to run* again, or into state *terminated* if its current state is the stop state. Program execution stops when all processes have reached the execution state *terminated*. Note that the scheduler can also schedule the processes in a round-robin manner, depending on the users' settings.

The execution of a transition can create a conflict. The example in figure 5.5 illustrates this. We assume that the current transition of the violet ladybug consists of making two steps ahead. However, it cannot enter the monitor squares in front of it because it is occupied by the bug on the right. Due to this conflict, the transition will not be executed.

Fig. 5.5: Violet ladybug cannot enter the yellow square

The scheduler first determines whether the execution of a transition creates a conflict or not. Transitions are atomic, so they are not actually executed if they create conflicts. However, to show users the conflict graphically, the transition is executed up to the command which creates the conflict. After this partial execution, a roll-back is done: (1) The effects of those commands of the transition which were already executed are undone, (2) the current state remains unchanged, (3) the process is put into execution state *sleeping*.

Users can also control the scheduler manually which can be helpful when testing and debugging programs. Figure 5.6 shows the program execution panel extended with four colored buttons. These buttons permit users to choose the ladybug process whose interpreter is to execute a transition.

Fig. 5.6: Program execution panel with manual scheduling buttons

World Objects

In MultiKara's world, there are bugs, leaves, trees, mushrooms, and direction signs. Each ladybug lays leaves in its own color. A sensor tests whether a leaf is its own or not. A bug cannot test a leaf for a specific color, as programs would have to be adapted for each bug individually. When bugs cooperatively solve a problem, the colored leaves give a visual impression of how much 'work' was done by each bug.

Direction signs can be put on any square to indicate from which directions a square may be entered. Squares without direction signs can be entered from anywhere. The idea is to provide a simple way for the bugs to follow user specified directional paths through the world.

Sensors

There are 8 types of world objects which can be detected by the sensors of the bugs: bugs, leaves, trees, mushrooms, and four kinds of directional signs. The bugs see the contents of four squares, the square they are on, the squares in front, to the left and to the right. Predefining all possible sensors would result in a huge sensor library. Therefore, only a few sensors are predefined. All other sensors have to be created by users using the sensor toolbox shown in figure 5.7.

5.4 Concurrency Mechanisms in MultiKara

In the following, we present the four concurrency mechanisms with examples to illustrate their application.

World: Meeting room

There is *one global meeting room* in the MultiKara environment which consists of all meeting room squares. This means that the bugs enter the meeting room whenever they enter any of the meeting room squares in the world. Note that the bugs cannot detect these squares. The global meeting 'room' can therefore consist of unconnected pieces of meeting room squares. The reason is that having more than one meeting room would complicate its semantics: which bug

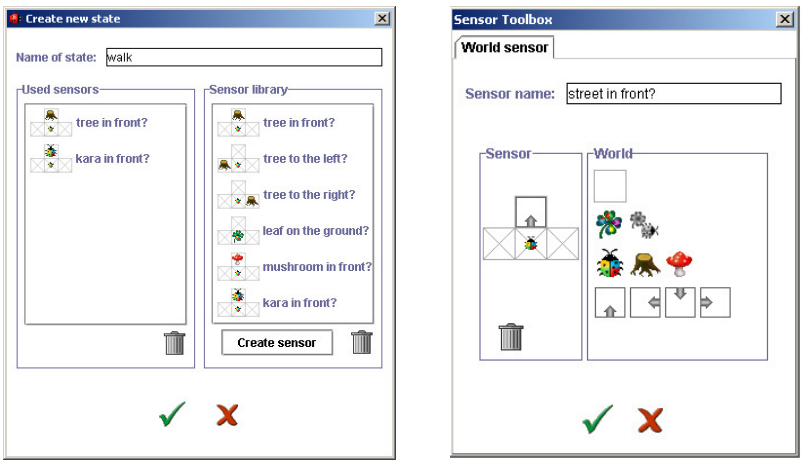

Fig. 5.7: State editor showing used sensors and sensor library (left) and sensor editor to create new sensors (right)

has to enter which room to complete a meeting? In order to keep the meeting room anonymous, that is, independent of the actors' identities, there is only one global meeting room.

The meeting room has two states, in and out. When the meeting room is in state in (represented by green colored world squares), actors entering the room are put to sleep. When all active actors have entered the room, the meeting room changes to state out (red colored world squares) and the scheduler wakes the sleeping processes. Only when all ladybugs have left the room, it will change to state in again. Note that if an actor attempts to re-enter the meeting room before all others have left it, it will be put to sleep until the room changes to state in.

As an example, figure 5.8 shows two bus lines with adjacent stations. The ladybugs simulating the buses are to wait on each other at the bus station so passengers can transfer from one bus to the other. The problem can be solved by marking the bus station squares as meeting room squares. The finite state machine makes the bugs track the streets, with no concern for the temporal synchronization with the other bug. In the case of randomized scheduling, it is not obvious how this problem could be solved without meeting rooms.

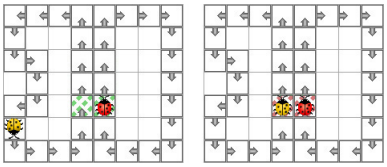

Fig. 5.8: Meeting Room in states in and in out, colored green and red

World: Monitor

There is *one global monitor* in the MultiKara environment which consists of all monitor squares. This means that the bugs enter the monitor whenever they enter any of the monitor room squares in the world. Note that the bugs cannot detect these squares. At most one bug may be in the monitor at any given time. The monitor has two states, free and busy. It is in state free (represented by green colored world squares) when no actor is in the monitor. When an actor enters the monitor, it changes to state busy (red colored world squares). If an actor attempts to enter a monitor world square while the monitor is busy, its process is put to sleep. When the actor currently in the monitor leaves it, the monitor changes back to state free, and all processes waiting on the monitor are woken up.

In the problem shown in figure 5.9, the bugs are to track their parts of the street. The monitor makes it possible to easily avoid collisions in the bi-directional parts of the streets. The critical parts are marked as monitors, and the state machine simply makes the bugs track the street. In the case of randomized scheduling, it is not obvious how this problem could be solved without the monitor mechanisms.

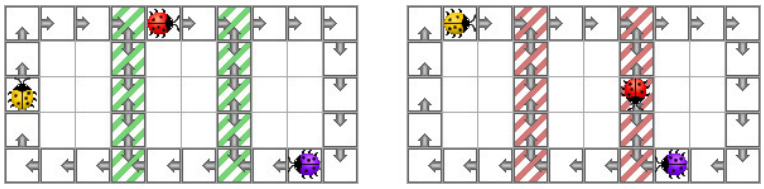

Fig. 5.9: Monitor traffic example (free and busy mode)

Note that it would be more efficient if there were more than one monitor in MultiKara. Each area of monitor squares could define one monitor, so that multiple monitors could be busy at the same time. In the above example, the two monitor areas are independent of each other and it would therefore be more efficient if they defined two monitors. However, we wanted all four concurrency mechanisms of MultiKara to be consistent. The monitor is the only mechanisms where multiple instances could be used anonymously; this is not possible with the meeting room, the barrier, or the critical section. Therefore, there is only one monitor even if the run-time of the programs increases somewhat.

Program: Barrier States

Barrier states synchronize the program execution of all active actors. They correspond to the meeting room squares in the world. There is *one global barrier* which is used by all barrier states. This means that the processes enter the barrier whenever they enter any of the barrier states. In the graphical editor, they are represented by thick dashed borders (figure 5.10). The barrier is activated when a barrier state becomes the current state of a state machine at run-time. The border of the barrier state is painted red, and the corresponding process is put to sleep. It will only be woken up when all processes have reached a barrier

state. At that moment, the barrier is deactivated, and all barrier states are again painted green.

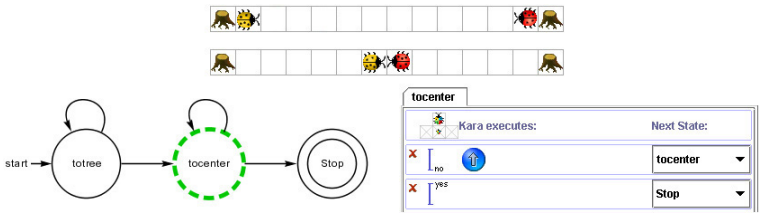

Fig. 5.10: Simple example for synchronized walks

As an example, the program in figure 5.10 makes two bugs find the center square between two trees. The two bugs walk from the trees towards each other, waiting for each other after each step. The waiting is achieved by a barrier state. This ensures that a ladybug with higher priority does not walk further than the other with lower priority.

Program: Critical Section States

Critical section states permit mutual exclusion within parts of a finite state machine. They correspond to the monitor squares in the world. There is *one critical section* which consists of all critical section states. This means that the processes enter the critical section whenever they enter any of the critical section states. At most one process can acquire the critical section. In the graphical editor, critical section states are represented by thick borders.

A process acquires the critical section when a critical section state becomes the current state of its state machine at run-time. When this happens, the critical section states of all other programs are painted red. When a state machine attempts to enter a busy critical section state, its process will be put to sleep until the critical section is released. This is the case when the state machine which currently owns the critical section enters a non-critical section state.

As an example, consider the game-like variant of the consumer/producer problem shown in figure 5.11. The yellow ladybug is the producer, endlessly trying to place lines of a random number of leaves in the world. The only constraint is that lines of leaves may not be adjacent to each other. The other bugs are the consumers and replace the yellow leaves with their own leaves. A bug must always consume a *whole* line of leaves at a time. The programs must avoid situations where two bugs start replacing leaves of the same line. Also, they should not start replacing before the producer is done with a line. Therefore, both the production and the consumption are critical sections of the programs.

Figure 5.12 shows the state machines for both the producer and consumer bugs. The producer walks about randomly, at times checking whether there is enough empty space to start a line of leaves that respects the space constraints. The consumers use critical section states to ensure that at most one of them is on a leaf. They only enter a square with a leaf on it when they make a transition

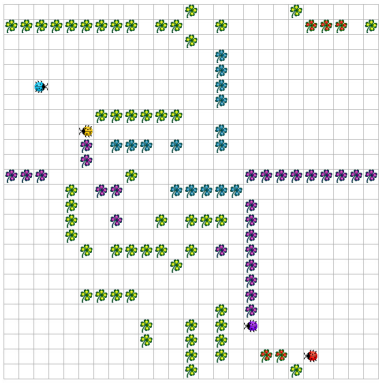

Fig. 5.11: Yellow bug produces lines of leaves which the others consume by replacing them with their own leaves

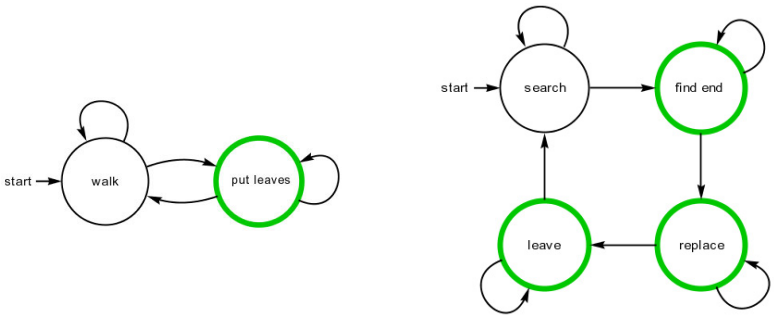

Fig. 5.12: Producer program (left) and consumer program (right)

into a critical section state. When they have replaced the leaves, they exit the leaf square and release the critical section.

5.5 Examples

Programming several bugs is quite challenging. Users must deal not only with collision avoidance, but also with synchronization. The inclusive concurrency mechanisms help solving problems which require solutions consisting of multiple phases. They enable the programmer to guarantee that all processes are in the same phase. The exclusive mechanisms make mutual exclusion possible, for example collision avoidance between the actors even when they are not facing each other.

Many problems which can be solved with a concurrency mechanism in the world (meeting room or monitor) can also be solved with the corresponding mechanism in the state space (barrier or critical section). In the case of the state space mechanisms, it may be necessary to use additional world objects to mark a square. The marked square serves as a sentinel, notifying the bugs that they should enter a barrier or critical section state. The programs using the

state mechanisms are more complex and less intuitive, since the concurrency logic is mixed with the actual algorithm.

Putting Down Leaves Between Two Trees: Multithreading without Concurrency Primitives

As a first example, we consider a problem whose solution does not require the use of a concurrency mechanism. The example illustrates how the bugs can cooperate using only their proximity sensors to avoid collision. It also shows the increase in size and complexity of programs for multiple actors in comparison to programs for one actor. The task in this example is to put a leaf on every square between two trees.

Figure 5.13 shows the initial world for a single ladybug. It has an arbitrary position between the trees and faces one of the trees. The corresponding program is straight-forward: The bug moves up to a tree, turns around and then puts down leaves while walking to the other tree.

Fig. 5.13: The ladybug must put a leaf on every square between the trees

With an arbitrary number of bugs, the solution is more complex. One wants to impose as few restrictions on the initial conditions as possible. We therefore assume that the bugs will be placed arbitrarily between the trees facing a tree. All bugs should have the same program which should work regardless of the number of bugs participating, their starting positions and their priorities. A solution is shown in figure 5.14.

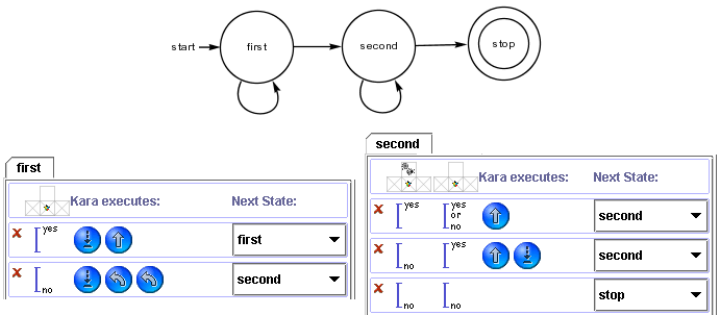

Fig. 5.14: A program for multiple ladybugs solving the problem in figure 5.13

Even though the program is only slightly more complex than its single bug counterpart, the argumentation for the correctness of the program is significantly more complex. There are more possible situations which can occur during execution, because leaves must be distinguished and collisions must be avoided.

In state first, the ladybug moves forward, putting down leaves, until the square in front is not empty. There are three possible reasons for the square ahead not to be empty: (1) There is a tree on the square, meaning the bug has

reached an end; (2) there is another bug on the square which will take care of the squares beyond; (3) there is a leaf on the square, which means that another bug will take care of the squares beyond. In all three cases, the bug turns around and changes to state second. In this state, the bug first walks over its own leaves, using the sensor which distinguishes its own leaves from leaves of the others. Then it continues walking and putting down leaves as long as the square in front is empty, stopping otherwise.

Filling a Rectangle: Cooperation Using Barrier States

This example illustrates how multiple actors can cooperatively solve a problem by using barrier states to synchronize their actions. One, two, three, or four ladybugs should fill a rectangle bounded by trees (figure 5.15). No assumptions are made on the number of participating bugs, on their initial positions within the rectangle, or on the direction they are facing.

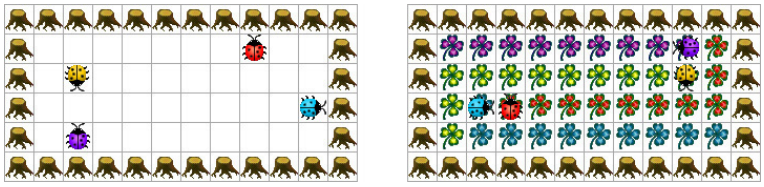

Fig. 5.15: Initially empty rectangle and filled rectangle

One solution to this problem is to have the bugs fill the rectangle in a spiral fashion. Figure 5.16 shows the state machine overview of this solution.

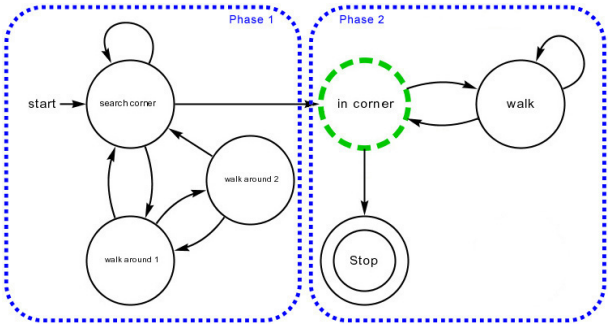

Fig. 5.16: Finite state machine of solution

The program is divided into two phases. In the first phase, each actor is to find a free corner of the rectangle (state search corner). The main difficulty is not to collide and to keep going when a corner is occupied by another actor (states walk around 1 and walk around 2). When an actor has found an empty corner, it uses the barrier to wait for the others to find their corners (barrier state in corner).

In the second phase, each actor draws an edge of the rectangle (state walk). After that, it turns right and uses the barrier (in state in corner) to wait for

the others. The second phase is repeated until the rectangle is filled. An actor stops when there is a leaf or another bug in front of it after waking up from the barrier. Note that it is therefore possible that an actor with high priority cuts off another actor with lower priority (figure 5.17).

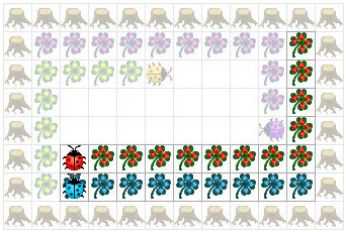

Fig. 5.17: The red actor cuts off the blue actor

This example illustrates the power of having a suitable concurrency primitive, i.e., the barrier. Horst Müller has solved this problem for the special case of two actors without using MultiKara's concurrency primitives [Mue02]. In contrast to the five-state machine using the barrier, his solution has more than 80 states, as his program implements a way for the two actors to synchronize using only the leaves in the world.

Stocking up Mushrooms: Implementing a 'Handshake' Protocol with Barrier States

Handshaking is a common procedure to coordinate two concurrent processes. First, one process does something while the other waits, then the other does something while the first process waits, and after that, both continue individually. As a MultiKara example, two ladybugs want to stock up mushrooms in a storage room. The yellow bug fetches the mushrooms and hands them over to the violet bug, which pushes them to their place in the storage room. Figure 5.18 shows the world before and after program execution.

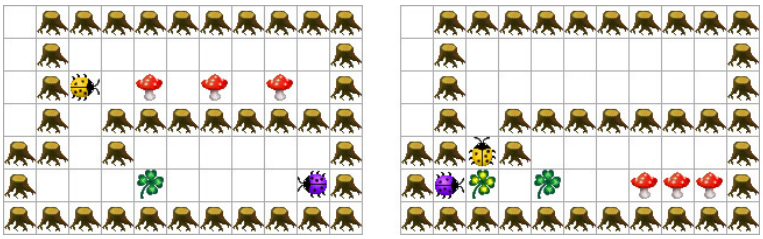

Fig. 5.18: Stocking up mushrooms

The main problem is to have the ladybugs wait for each other in the right spot when they hand over mushrooms. Two phases are needed to hand over the mushrooms, both synchronized via the barrier (figure 5.19). In the first phase, the yellow bug must wait until the violet bug is ready in the niche on the left. In the second phase, the violet bug must wait until the yellow bug has pushed

the mushroom downwards in front of it. Only then may the violet bug start pushing the mushroom to the right.

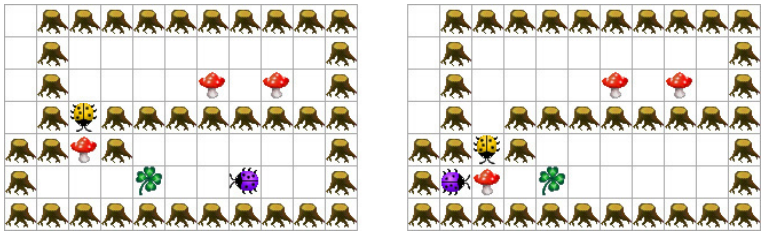

Fig. 5.19: The yellow ladybug waiting for the violet ladybug (left), and the violet ladybug waiting on the yellow ladybug (right)

Figure 5.20 shows the state machines of both actors. The programs are quite complex. The two phases for handing over mushrooms are implemented in the barrier states. Both programs enforce waiting in the first phase in state in_niche. Waiting in the second phase is accomplished by the states back and state wait_for_mushroom, respectively. Note that when the yellow bug does not find any more mushrooms to hand over, it uses the barrier in state put leaf, while the violet bug waits in in_niche. After waking up, the yellow bug puts a leaf on the square in front of it and terminates. The violet bug wakes up from the barrier in state wait_for_mushroom, notices the leaf in front of it, and terminates.

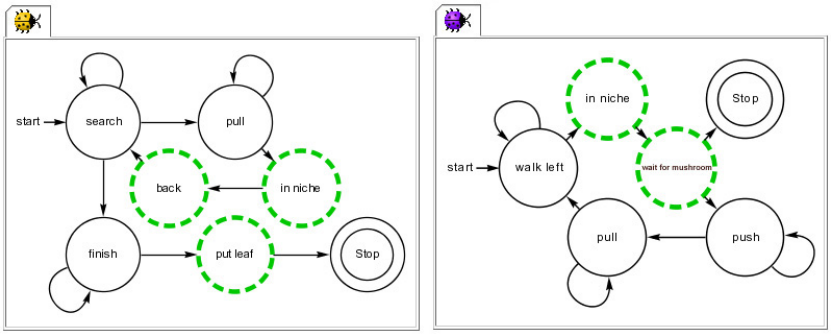

Fig. 5.20: Finite state machines for the mushroom problem

Deadlock and Starvation

Concurrent programming introduces dangers to the correctness of programs which do not exist in sequential programs, such as race conditions, deadlock, livelock, or starvation. Programmers must be aware of these dangers, their causes, and possible precautions. MultiKara illustrates some of these important concurrency-related problems.

In MultiKara, two kinds of deadlock are possible. The first kind may arise when the programmer uses more than one concurrency primitive in the same

program. A simple example illustrates this problem (figure 5.21). Both actors start in the meeting room which is therefore in state out. The yellow actor then leaves the meeting room, enters the monitor, makes one more step ahead, and tries to take another step forward. Now the two actors are deadlocked. The violet bug cannot leave the meeting room because the monitor is busy. The yellow bug cannot leave the monitor because the meeting room is still in state out. Both actors sleep, waiting for the other to release its mechanism. When the scheduler notices that all processes are sleeping, it aborts execution.

Fig. 5.21: A deadlock situation caused by multiple concurrency mechanisms

The second kind of deadlock arises in situations in which the actors block each other, but which cannot be detected by the scheduler. Such a situation can arise when the programmer is not careful enough in using the concurrency mechanisms. For example, in figure 5.22, the actor owning the monitor cannot leave it, because the exits of the tunnel are blocked. The others are sleeping, waiting on the monitor. The actor in the monitor will loop endlessly trying to find a way out of the tunnel. One way to avoid deadlock in this example would be to extend the monitor by one square on both sides of the tunnel. The bug in the tunnel could then escape and release the monitor.

Fig. 5.22: A deadlock situation caused by careless use of the monitor

There are two kinds of starvation situations which can occur in MultiKara. The first kind can be induced by randomized scheduling. Figure 5.23 shows an artificial example. The bugs cycle endlessly in a world of two rows, where the upper row consists of monitor squares. From left to right, the figure shows what can happen when the yellow bug enters the monitor first. If the other bugs are selected by the scheduler while the yellow bug holds the monitor, their processes are put to sleep. After the yellow bug has released the monitor, the scheduler wakes all processes sleeping on it. Starvation of a bug's process may occur if it is never selected by the scheduler after it wakes up, that is, if it is only selected when the monitor is already busy. The scheduler cannot detect this kind of starvation. However, the probability that such a scheduler-induced starvation occurs is small.

The other kind of starvation is induced by a bug. In figure 5.24, the yellow and red bug wait in the meeting room for the violet bug. However, if the violet bug only turns around on its current square and never enters the meeting room, then the two sleeping bugs will wait forever.

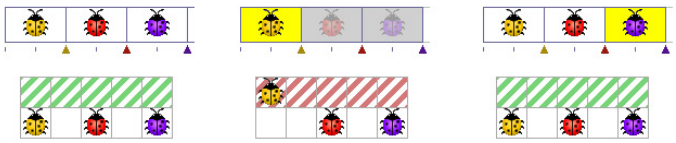

Fig. 5.23: Example of scheduler-induced starvation

Fig. 5.24: A starvation situation induced by the violet bug

5.6 Related Work

Though concurrent programming has moved into the mainstream, there is much less educational literature on concurrent than on sequential programming. Even in the most recent draft of the ACM Curriculum 2001, concurrent programming figures only as a small part of the operating systems course [AI02]. In many books which present an overview of computer science, it is more or less absent.

With regard to teaching sequential programming, mini-languages and mini-environments have a long tradition and represent a pedagogically sound approach. For concurrent programming, there are many different approaches, with no clear favorite. In the following, we present a number of different approaches to teaching concurrency, with no claim for completeness.

MultiLogo is a concurrent extension of Logo which lets the user control multiple turtles [Res90]. Its concurrency model is based on the concept of command queues. Each process has its queue of commands to be executed. Processes can send each other messages containing commands. These messages have two priority levels. The higher priority level means a sender process demands that his message command be inserted at the top of the queue of the receiver process. The lower priority level means it asks that the command be inserted at the end of the queue. There are two disadvantages of this approach. First, the command sending mechanism is quite complex. Second, and more important, the mechanism is quite different from most standard mechanisms of concurrency such as semaphores or monitors. The fact that a sender of a message can modify the command queue of the receiver makes reasoning about the invariants and about the correctness of MultiLogo programs impossible.

Pascal-FC (functionally concurrent) by Burns and Davies is a Pascal variant designed to teach concurrency [BD88]. The language is based on Ben-Ari's extension of Pascal which introduces the `cobegin` ... `coend` construct and the semaphore variable type. Pascal-FC programs are compiled and then interpreted; the compiler and interpreter themselves are written in Pascal and therefore available on many platforms. The language supports a wide range of concurrency primitives such as semaphores, monitors, and synchronous message

passing. Pascal-FC offers a good introduction to concurrency for students who already know Pascal. However, the interpreter does not offer any visualization, making it hard to observe program execution at run-time.

The Ben-Ari concurrency interpreter (BACI) by Bynum and Camp is based on a language similar to Pascal-FC [BC96]. However, they stress that the idea is not to show how to use the different concurrency primitives, but rather how to implement them. Classic textbooks on concurrency, for example Ben-Ari's *Principles of Concurrent and Distributed Programming*, show how one primitive can be implemented in terms of the other [BA90]. Yet most programming environments do not let students experiment at this low level of implementation of primitives. The BACI addresses this situation and offers a way of experimenting at the implementation level.

Hartley shows how this approach can be realized in Java [Har98]. Java has only two built-in concurrency mechanisms. He shows how these primitives can be used to implement other primitives in Java. As with the BACI, this approach to teaching concurrency focusses on the technical aspects of implementing concurrency mechanisms and therefore requires a thorough understanding of those mechanisms as a prerequisite.

5.7 Conclusions

MultiKara provides an educational environment in which students can understand a number of basic principles of concurrent programming without having to deal with the complexity of real-world languages and environments. The finite state machines combined with the simple world of MultiKara form a good vehicle to illustrate concurrency mechanisms. They are visually intuitive and easy to understand, and they can be used to discuss concurrency related problems such as coordination, exclusion, deadlock, and starvation.

Chapter 6

JavaKara: A Smooth Transition From Kara to Java

An introduction to programming as part of general education based on the Kara environment teaches students about the fundamental ideas of programming. However, programming based on finite state machines does not convey an impression of what real-world programming in a modern programming language is like. The problem is that using a full-blown programming language in the classroom can become very time consuming even for small problems. In particular, if the problems involve graphics or user interfaces, their solutions are typically too complex for beginners. JavaKara addresses this problem by providing an easy-to-use mini-environment for introductory Java programming.

6.1 Educational Goals and Scope of JavaKara

The JavaKara environment (figure 6.1) offers a smooth transition from the finite state machines of Kara to the real-world language Java. The world of JavaKara is the same as that of Kara, but the finite state editor is replaced by a text editor. Students are already familiar with the world and the actor, so they can concentrate on learning the Java language and the handling of the Java compiler. The main benefit of JavaKara is that the 'output' of the programs is visual so students immediately see what their programs are doing.

The goal of JavaKara is to introduce students to the basics of the procedural programming model:

Decomposition into Methods. Students learn that decomposing a program into smaller sub-programs makes their programs more readable and helps avoid code duplication. Students must also learn the concept of scope of variables, parameter passing, and return values.

Order of Execution. Beginners must learn early on that the order of methods in the source code implies nothing about the order of execution at run-time.

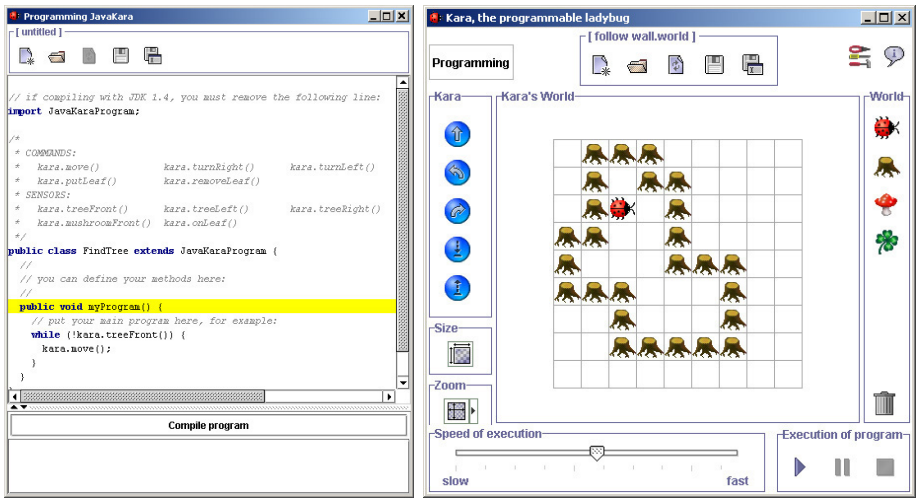

Fig. 6.1: The JavaKara environment

Branches and Loops. These two fundamental concepts appear in almost all JavaKara exercises illustrating if-branches and the different types of loops (for, while-do, do-while).

Boolean Logic. JavaKara yields many examples of Boolean expressions, for example, when the programmer needs to combine two or more pieces of information from the different sensors.

Basic Data Types, Arrays. Many JavaKara programs use basic data types like integers, booleans, strings, and one- or two-dimensional arrays.

This is about the scope which can reasonably be covered with JavaKara. Students learn a small subset of the Java language without having to deal with the classes of the Java class libraries.

JavaKara does not address the concepts of object-orientation for two reasons. First, we believe that beginners must master the underlying concepts of imperative programming as outlined above before studying object-orientation. Second, object oriented programming deals with programming "in the large", that is, with structuring a system into a collection of classes. We believe that these concepts are outside the scope of a mini-environment.

However, note that JavaKara does not place any restriction on what users can program. They have the complete Java language and all libraries at their disposal. However, it is not the goal of JavaKara to serve as a vehicle for object oriented programming. JavaKara is object-based in that programmers have access to objects. But usually, they use them without thinking about what it means that they are instances of some class. The underlying mechanisms are too complex for beginners who want to get started with Java.

Last but not least, note that Java was used as the language of choice simply out of practical considerations. All Kara environments are programmed in Java. It is therefore easy to integrate a user-written JavaKara program into the environment at run-time.

6.2 The JavaKara Environment

The world of JavaKara is identical to that of Kara. What has changed is the program editor. Whereas in Kara a graphical editor lets the user construct finite state machines, in JavaKara a text editor lets the user write Java code.

A code template in the program text editor helps users write their first programs. It is basically a black box with spots for users to "plug in" their code. The listing in figure 6.2 shows the template which defines a class FindTree.

```
import JavaKaraProgram;
/*
 * COMMANDS:
 *   kara.move()           kara.turnRight()      kara.turnLeft()
 *   kara.putLeaf()        kara.removeLeaf()
 * SENSORS:
 *   kara.treeFront()      kara.treeLeft()       kara.treeRight()
 *   kara.mushroomFront()  kara.onLeaf()
 */
public class FindTree extends JavaKaraProgram {

  // you can define your methods here

  public void myProgram() {
    // put your main program here, for example:
    while (!kara.treeFront()) {
      kara.move();
    }
  }
}
```

Fig. 6.2: The code template of JavaKara

The class of any program controlling the bug must inherit from the class JavaKaraProgram. This class is a small interface to the rest of the environment, so that users only sees what they need to see. The user's class inherits access to three different objects: kara, the interface to control the ladybug; world, to manipulate the world directly; and tools, which contains useful methods such as window-based string input and output. The most important object to the user is the kara object. Its command and query methods are listed in the comment at the top of the template.

Users must write a method called myProgram, which is the entry point from the JavaKara environment into their program. As an example, the myProgram method of the above template implements the "find tree" algorithm (see figure 6.2).

Users can define their own methods and instance variables within their classes. There is only one technical restriction which novice users will probably not encounter. If the class defines a constructor, it must be parameterless. The reason is that the system instantiates an object of the user's class, and it cannot know what values it should pass to the constructor.

Before users can execute a program, they must compile it. If there are any errors during compilation, the error messages are displayed below the program text editor. When users click on an error message, the corresponding line of code is highlighted in the editor window (figure 6.3).

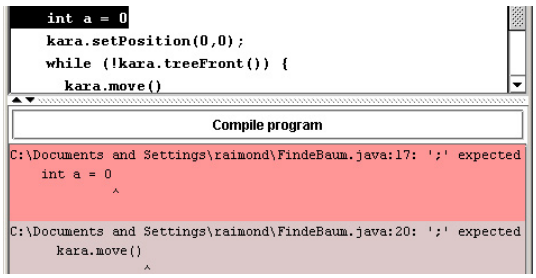

Fig. 6.3: Compiler error messages

6.3 Examples

Students can learn the basics of Java by working through a series of exercises of increasing difficulty. The code template of the program editor represents a "hello world" equivalent of JavaKara and makes the bug walk up to a tree. This example helps students get acquainted with the environment, in particular with the Java compiler. It serves as a starting point and can be enhanced to lay leaves or walk up to either a tree or a mushroom. In the following, we present three categories of JavaKara examples of increasing difficulty.

Controlling the Ladybug

The first category consists of exercises in which the programmer must control the actions of the bug. All the finite state machine problems of the basic Kara environment belong to this category. In JavaKara, these examples introduce students to the concepts of methods, control structures, and Boolean logic. For example, the following listing shows a JavaKara program which makes the bug track a wall of trees.

```
public class FollowTrees extends JavaKaraProgram {

  public void myProgram() {
    while (true) {
      if (kara.treeFront() && kara.treeRight()) {
        kara.turnLeft();
      }
      else if (!kara.treeFront()) {
        if (kara.treeRight()) {
          kara.move();
        }
        else {
          kara.turnRight();
          kara.move();
        }
      }
    }
  }

}
```

There are multiple ways to express the if-then-else structure of this program. For beginners, it might be interesting to compare different JavaKara programs

for the same problem. It might also be interesting to compare these programs with the corresponding Kara state machine.

A more advanced example is given in the listing in figure 6.4. The program uses variables to let the bug draw a spiral of a given size. An example of this type can be used to introduce integer variables and methods. The figure also shows the resulting spiral image.

```
public class Spiral extends JavaKaraProgram {

  void walk (int distance) {
    for (int i = 0; i < distance; i++) {
      kara.putLeaf();
      kara.move();
    }
  }

  public void myProgram() {
    final int MAX_LENGTH = 20;
    int d = 1;

    while (d < MAX_LENGTH) {
      walk (d);
      kara.turnRight();
      d ++;
    }
  }
}
```

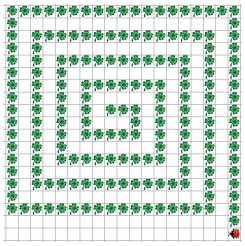

Fig. 6.4: Drawing a leaf spiral

Programming the World Array

Another category of exercises makes use of direct access to the world. The world is basically a two-dimensional array on which commands like world.setLeaf (int x, int y, boolean putLeaf) and queries such as world.isLeafAt (int x, int y) can be executed.

The examples of this category go significantly beyond the scope of problems of the basic Kara environment with its finite state machines. They introduce students to the concept of one- and two-dimensional arrays as a simple, compound data structure. For example, the following listing shows part of a program which draws a "black-and-white" Mandelbrot set image. Figure 6.5 shows the resulting image.

```
public void myProgram() {
  WIDTH = world.getSizeX();
  HEIGHT= world.getSizeY();
  world.clearAll();

  for (int row = 0; row < HEIGHT; row++) {
    for (int col = 0; col < WIDTH; col++) {
      double x = calcMandelX(col); // method not shown
      double y = calcMandelY(row); // method not shown
      int i = test(x, y);
      if (i == ITERATIONS) {
        world.setLeaf(col, row, true); // direct access to world
```

```
      }
    }
  }
}

int test (double x, double y) {
  double tmpX = 0, tmpY = 0;
  int i = 0;
  do {
    double tmp2X, tmp2Y;
    i++;
    tmp2X = tmpX*tmpX - tmpY*tmpY + x;
    tmp2Y = 2*tmpX*tmpY + y;
    tmpX  = tmp2X;
    tmpY  = tmp2Y;
  } while (((tmpX*tmpX + tmpY*tmpY) <= 4) && (i < ITERATIONS));
  return  i;
}
```

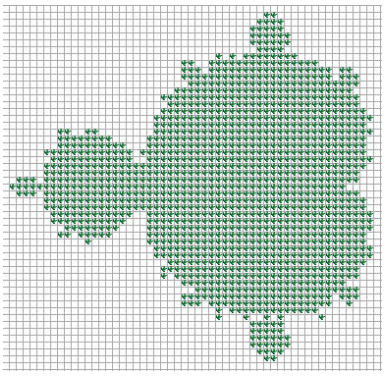

Fig. 6.5: The Mandelbrot set drawn with leaves

Algorithmically Demanding Problems

A third category of exercises consists of problems which can be described as "algorithmically demanding", at least for beginners. The idea of these exercises is to use JavaKara as a visualization tool which allows students to concentrate on the algorithm. Examples include finding the shortest path out of a labyrinth, solving the Queen's problem, or pattern recognition with neural networks.

As an example, we show the implementation of Lindenmayer systems with JavaKara. The biologist Aristid Lindenmayer used simple deterministic context-free grammars to describe the growth of plants. Assume we have a turtle graphics system where F denotes a step forward, and L and R denote left and right turns of a constant angle, say 90°. These three commands can be used as the alphabet for Lindenmayer grammars. A small grammar could consist of the single replacement rule F → FLFRFRFLF. If the initial string is F, it is replaced by FLFRFRFLF in the first iteration, and by FLFRFRFLFLFLFRFRFLFRFLFRFRF-LFRFLFRFRFLFLFLFRFRFLF in the second iteration, and so on. The resulting strings describe snow flake curves (figure 6.6).

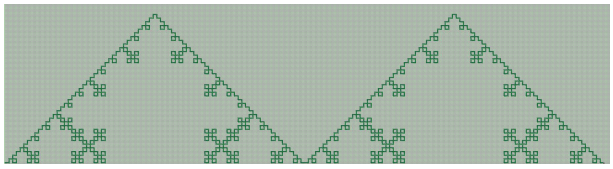

Fig. 6.6: A simple Lindenmayer system

The following listing shows an extract of an implementation of Lindenmayer systems with JavaKara. A simple find/replace rule specifies how the string is to be generated. The method interpret parses this string and moves the bug through the world accordingly. It also illustrates the concept of an interpreter in a graphical manner.

```
public class PatternGenerator extends JavaKaraProgram {

  /** Execute the F, L, R commands in string. */
  void interpret (String string, int stepLength) {
    for (int i = 0; i < string.length(); i++) {
      if (string.charAt(i) == 'F') {
        forward(stepLength);
      }
      else if (string.charAt(i) == 'L') {
        kara.turnLeft();
      }
      else if (string.charAt(i) == 'R') {
        kara.turnRight();
      }
    }
  }

  // rest of class not shown

}
```

Another example has its origin in the *Bundeswettbewerb Informatik*, a programming contest in Germany for pupils and students under the age of 21. One problem of the 2001/2002 contest was to program a mouse to find the shortest way from an arbitrary starting position to a piece of cheese within the labyrinth. Figure 6.7 shows an example of such a labyrinth for JavaKara, where the cheese is replaced by a leaf.

The original solution is available for download from the official web page of the contest [Bun02]. The solution is quite complex. First, it reads a description of the labyrinth from a text file and constructs a data structure to represent the labyrinth. Second, it determines the current position of the mouse in the labyrinth. The position of the mouse is not given in the map which describes the labyrinth. Rather, the programmer must derive its position by moving the mouse through the labyrinth until its path uniquely defines its current position. Third, the shortest path to the target position must be determined.

With JavaKara, the first step is not necessary, since the world is directly accessible. Further, it provides a natural visualization for this problem. The following listing shows part of a JavaKara solution of the problem which was implemented by Markus Brändle. The method uses a dynamic programming

Fig. 6.7: The ladybug must find the shortest path to the leaf

approach to find the shortest path: it determines all shortest paths backwards from the goal position through the labyrinth.

```
void determinePaths() {
  ArrayList squaresToConsider = new ArrayList();

  Point leafPosition = findLeaf();
  squaresToConsider.add(worldSquares[leafPosition.x][leafPosition.y]);

  while (squaresToConsider.size() > 0) {
    WorldSquare currentSquare = (WorldSquare) (squaresToConsider.get(0));
    for (int i = 0; i < 4; i++) {
      int posX = currentSquare.posX + deltas[i][0];
      int posY = currentSquare.posY + deltas[i][1];
      if (validCoordinates(posX, posY)) {
        if (!world.isTree(posX, posY) &&
            !(worldSquares[posX][posY].hasPrecedent)) {
          worldSquares[posX][posY].setPrecedent(deltas[i][2], currentSquare);
          squaresToConsider.add(worldSquares[posX][posY]);
        }
      }
    }
    squaresToConsider.remove(0);
  }
}
```

The Lindenmayer and shortest-path examples show the broad range of problems which can be solved in JavaKara. The focus is on the algorithms themselves, not on Java. For example, Lindenmayer systems can be used to discuss iteration vs. recursion, and there are many solutions to the shortest-path problem.

6.4 Related Work

JavaKara can be compared to introductory Java programming environments such as Bergin's KarelJRobot [Ber02] or IBM's RoboCode [IBM02] which are discussed in more detail in appendix A.

Note that the JavaKara environment does not attempt to be a full-blown Java programming environment aimed at beginners like the BlueJ environment by Kölling and Rosenberg [KR01]. BlueJ's objective is to teach object-oriented programming in an environment designed to illustrate concepts like inheritance or object instantiation to beginners.

6.5 Conclusions

The transition from the finite state machines of Kara to Java addresses the "mother-tongue problem" of teaching programming. For students, their first programming language becomes their mother-tongue of programming. If they encounter only one programming language during their education, they tend to equate programming with their particular mother-tongue programming language. For this reason, many curricula recommend introductory programming courses based on at least two languages. Kara and JavaKara are two different 'languages'. Students can experience what it means to implement programs in different languages. Its intuitive and visual user interface has made JavaKara very popular in many schools at different levels.

Chapter 7

Experience and Evaluation

Kara, TuringKara, MultiKara, and JavaKara are published on the web, along with teaching materials: a wide selection of exercises and their solutions, and slides for introductory presentations. The EducETH (www.educeth.ch) web server registers, on average, about 1'000 downloads of the applications per month. A growing number of institutions and people have web sites for their own courses which feature their material for the environments. Most notably, Horst Gierhardt of the Immanuel-Kant-Gymnasium, Heiligenhaus, Germany, has contributed much material to EducETH's Kara site.

From various feedback, we know that Kara is used extensively in Germany, Austria, and Switzerland. Kara is not only used at the high school level, but also in K6-8 grade schools, and even in introductory courses at universities. The feedbacks also show that if someone has a problem using Kara, the problem is typically related to Java or to the operating system, and can easily be solved. The following list gives a sample of the wide range of institutions in which the Kara environments are used in addition to the department of Computer Science at ETH Zurich:

Teacher education at universities. For example: the Humboldt university in Berlin, Germany; the university of Karlsruhe, Germany; the university of Potsdam, Germany; the university of Bielefeld, Germany; the university of Klagenfurt, Austria; the pedagogical academy of Steiermark, Austria.

Colleges of higher education. For example: the college of the state of Saarland, Germany; the college of higher education of the canton of Aargau, Switzerland; the college of higher education of Vorarlberg, Austria.

Secondary schools. For example: the Saint Franziskus secondary school of Kaiserslauten, Germany; the secondary school of Haren, Germany; the Ketteler secondary school of Hopsten, Germany; at the Tulla high school of Rastatt, Germany; at the vocational diploma school of Zug, Switzerland; at the high school of Luzern, Switzerland; at the German school in Barcelona, Spain.

Web references. The Kara web site is also linked by many educational web sites, for example: the education server of the state of Brandenburg,

Germany; the education server of the state of Bayern, Germany; the state institute of school in Bremen, Germany; the school council of the town of Stuttgart, Germany.

7.1 Kara: Introductions to Programming

Kara has been used at ETH Zurich for the past four years in several courses to teach the fundamentals of programming to high school students who never had any or only very little exposure to programming. The feedback we have received from these students is highly positive. They appreciate the fact that after a brief demonstration and about half an hour of getting to know the environment, they can start solving interesting problems. They also appreciate that within a few hours, they succeed in solving non-trivial problems. The feedback on the Kara environment from the surveys of the high school students can be summarized as following:

Motivating. Not surprisingly, Kara was most motivating for those students with the least prior experience in programming. All students found Kara either "very motivating" or 'motivating'.

Easy-to-use. All students found the handling of Kara's user interface either "very easy" or 'easy'.

The students' written comments show that Kara allowed them to focus on problem solving, on the logic and the correctness of their programs, without being distracted by the environment or by the textual syntax of a "real-world" programming language. They also appreciated the simplicity of the user-interface.

Teaching the Nintendo Generation to Program by Guzdial and Soloway may offer an explanation as to why students like the Kara environment [GS02]. Whereas "Hello World" got students excited when computers were still text-based, today's "Nintendo generation" grows up in multimedia environments. As the article notes, engaging students is critical for them to learn something well. And they will become engaged if the learning environments correspond more to their every day use of computers than to a text-based view of computers. Maybe this explains our observation that students often continued using the Kara environment after class. Its game-like appearance reminded them of computer games, and in combination with the simplicity of the programming model, they felt really involved.

Kara has also been used in the past four years in several courses to show teachers, often with little background in computer science, a way to teach the fundamentals of programming to their students. The feedback we have received from these users is also highly positive. The non-computer scientists among the teachers appreciate the fact that they succeed in solving non-trivial problems after being coached for just a few hours. More expert users are surprised to learn that the conceptually simple structure of programming using finite state machines raises fundamental questions of computer science.

Kara was also used in a course on theoretical computer science at ETH Zurich. The course used Kara to illustrate the concept of finite automata in a way which allowed the students to gain hands-on experience. They had to solve a number of simpler tasks as well as one of the more challenging tasks. They

could choose implementing either Bubble sort on bars of leaves or the binary Pascal triangle.

7.2 TuringKara in a Course on Theoretical Computer Science

The TuringKara environment was used in a course on theoretical computer science at ETH Zurich. Students must attend this course in the fourth semester of their computer science studies. The course covers topics such as models of computation like the Markov model, finite state machines, Turing machines, different types of languages, computability. The TuringKara environment provided hands-on experience in creating Turing machines. Students had to implement, as a simple exercise, binary addition. As a more complex and more challenging exercise, they had to implement binary multiplication. We also conducted a contest to find a three-state TuringKara busy beaver, an open problem which students eagerly tackled. The result is presented in the TuringKara chapter.

At the end of the semester, we conducted a survey with the 131 students who had attended the course. The survey addressed two questions. First, whether students thought their learning benefited from using the environments, and second, what they thought of the quality of the user interface of the environments. The latter part of the survey was based on the ideas presented by Hassenzahl et al in *Engineering Joy* [HBB01]. This article introduces questionnaires to determine the users' perception of hedonistic qualities of the user interface of a given software. The basic idea is to use so-called semantic differential questions to measure the hedonistic qualities. For example, is the user interface 'innovative' or is it 'conservative'? Is it 'impressive' or 'nondescript'?

The results of the survey show that practically all the students thought the TuringKara environment was useful with respect to learning about Turing machines; 86% gave it the highest or second-highest possible ranking (figure 7.1). A number of students explicitly said that TuringKara made it easier to "see what a Turing machine does" in comparison to static Turing machines on paper.

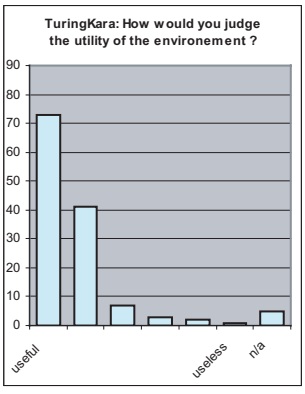

Fig. 7.1: Students' assessment of their learning benefit with TuringKara

As to the user interface, students judged the user interface of the TuringKara environment to be 'outstanding', 'clean', and 'interesting' (figure 7.2). However, it is interesting to note that, in comparison to the Kara environment which was also used and evaluated in the course, students judged TuringKara's user interface slightly less 'outstanding' and also slightly less 'clean' than Kara's user interface. The reason is probably that TuringKara is more complex and the world objects more abstract.

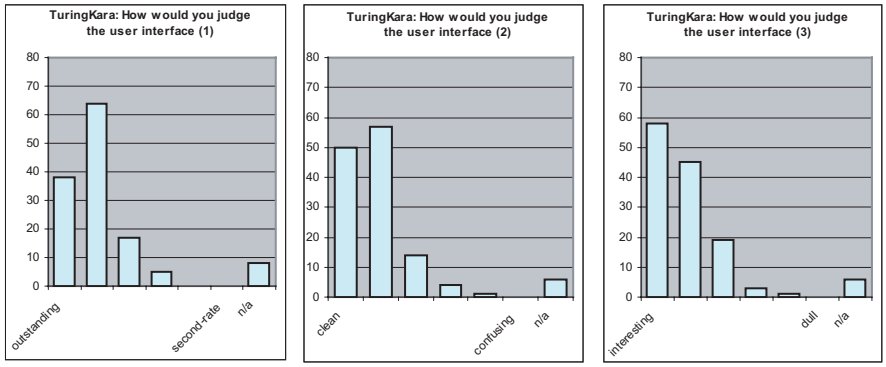

Fig. 7.2: Students' assessment of TuringKara's user interface

7.3 MultiKara Experience

As of yet, we have only very little experience with MultiKara, since it was added to the Kara application only in 2002.

7.4 JavaKara: Introductory Java Courses

The JavaKara environment is second in popularity to the Kara environment. Our own experience and the feedback we receive correspond to the feedback concerning Kara. Teachers appreciate that JavaKara gives them a comfortable way to teach Java to novices. One difference is that more technical problems are reported due to the fact that JavaKara requires the full Java software development kit to be installed. These problems are usually easily solved.

7.5 Lessons Learned

Our experience convinced us that the conceptually simple programming model of finite state machines is a convenient vehicle to introduce novices to programming. We believe that the simplicity of the model is the most important reason for the success of the Kara environments.

Another important reason for the users' acceptance of the environments is their simple and intuitive user interface. Kara is used in class for just a few hours. It is therefore imperative that the user interface be as intuitive as possible.

From the received feedback, and from the comments of Adele Goldberg (private communication), we further learned that it is crucial for the Kara web site to clearly document the "learning paths" through the environments, as well as the different possible learning targets of the environments. We completely redesigned the original web site to achieve this goal.

Appendix A

Environments for Learning Programming

In this appendix, we summarize approaches on how to introduce novices to programming. Brusilovsky et al classify the various approaches in three categories [BKMT94]:

> **Incremental approach.** This is the classical and most widespread approach. The students are presented the constructs of the programming language to be learned step by step. Each construct enhances their abilities to solve incrementally more difficult problems.
>
> **Sub-language approach.** Another approach is to select a subset of a programming language, and to use this subset to teach particular concepts.
>
> **Mini-language approach.** Mini-languages are small programming languages designed solely for the purpose of programming education. Students learn programming by controlling the actions of, for example, a robot living in a virtual world.

In a later paper, Brusilovsky et al conclude that the mini-language approach is very effective, and that mini-languages can be applied to a wide range of target groups, from pupils in elementary schools to students at college level [BCH+97]. It is important to note that "the application of a mini-language is never the goal itself, but a method of mastering a set of notions and skills".

In their article *The black box inside the glass box: presenting computing concepts to novices*, Du Boulay et al discuss fundamental, desirable characteristics of programming languages and environments designed for educational purposes [dBOM99]. Summarizing both [BCH+97] and [dBOM99], the following criteria should be satisfied:

> **Decrease complexity.** Each program controls the actions of some machine. This machine is usually a PC, with its processor, main memory, and screen – a very complex machine. The first thing students have to acquire is a basic understanding of the machine to be programmed. Therefore, mini-environments should use a simpler target machine than the PC.

Hide complexity. The actions of the machine should be "black boxes", their inner workings hidden from the programmer. The semantics of the actions should be easy to grasp. The same holds for the current state of the machine. It may be complex internally, but its semantics should be simple enough to visualize.

Visualization. It is important to have a graphical display of the actions and of the current state of the machine. The programmer should perceive the machine as a "glass box" of which all aspects can be visualized. On top of that, the visualization of program execution helps track the evolution of the current state of the machine.

Small language. A language with a small set of constructs is a must for an introductory programming language. Students should be able to master the mini-language in its entirety in as little time as possible.

Easy-to-use programming environment. A mini-version of an integrated development environment should support writing syntax error free programs to help students focus on the semantics of their programs.

Interesting problem set. Problems which are abstract or hard to define distract the novice from learning how to solve problems algorithmically. Problems which can be easily, maybe even graphically explained are better suited to novices' needs.

The mini-language approach meets these requirements. The languages are typically used to program some active entity, an actor living in a simple world on the screen. Examples of actors are robots, turtles, or any kind of phantasy figure. Students learn programming by instructing the actor to solve given problems. Mini-languages and mini-worlds provide instant visualization: Students can literally observe what their programs are doing.

It is important for a good mini-environment to have a stimulating and motivating world and actor for which interesting and easily understandable problems can be formulated. A careful balance must be maintained between the complexity of the world and the possible problems for the actor. On the one hand, learning the ropes of the mini-environment should require as little time as possible. On the other hand, it must be possible to design challenging problems to show students that programming is an intellectually demanding process.

A.1 Logo and the Turtle Geometry

The mathematician, computer scientist, and psychologist Seymour Papert conducted renowned projects in the 1970's at the Massachusetts Institute of Technology with the goal of making children the 'builders' of their own intellectual 'buildings'. In particular, the goal was to enable children to discover geometric knowledge on their own. The computer was to serve as a powerful tool with which the children could formulate algorithms to create certain patterns and test these algorithms. To this end, the language Logo was created as a dialect of the LISP programming language.

Logo is a general-purpose language. To adapt it for the purpose of teaching geometry, the sub-language approach was used in combination with the idea

of the mini-language approach. The resulting "Turtle Geometry" involves programming a turtle, either a robotic one drawing on the ground, or a virtual one drawing on the screen.

Papert intended Turtle Geometry to be a mathematics learning environment for children. He observed that a child who has difficulties with mathematics is often labelled mathematically untalented. However, a child having trouble learning a foreign language like French is not labelled untalented for French. We know that if the child had grown up in France, it would have learned French without any conscious effort. Papert argued that in "Math Land", children would learn mathematics as naturally as their mother tongue. Logo and the Turtle Geometry were his vision of Math Land. Children should be enabled to "learn by making" which in Papert's view is more than learning by doing.

The turtle has a position and faces a direction, specified in degrees. These two properties fully describe the state of the turtle and can be easily visualized. The commands understood by the turtle are also simple. It can move forward, painting its trail, or turn at its current position. Logo's turtle is blind: it cannot obtain any information about the current state of its world. This restricts Logo programming to creating geometric patterns.

Logo supports recursion, which makes it possible to formulate algorithms concisely. As a general-purpose language, Logo also has variables, parameters, branches, repeat statements, blocks, and much more. Figure A.1 shows a program painting the Snow Flake Curve (screen shot created with StarLogo):

```
to snowflake :length
if :length < 10
[ move :length ]
[ snowflake :length/3
  left 60
  snowflake :length/3
  right 120
  snowflake :length/3
  left 60
  snowflake :length/3 ]
```

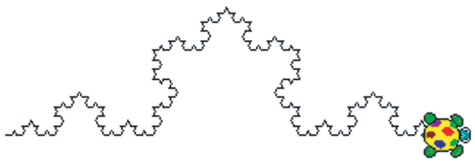

Fig. A.1: Snowflake curve

Logo is more powerful than a first glance reveals. It was Papert's intention to design "a computer language that would be suitable for children. This did not mean that it should be a 'toy' language. On the contrary, I wanted it to have the power of professional programming languages, but I also wanted it to have easy entry routes for nonmathematical beginners" ([Pap80], p. 210).

A.2 Karel, the Robot

Even though Logo was not designed to teach programming, it influenced the future development of mini-languages and mini-environments aimed at teaching programming. The first prominent mini-environment of this type was Karel the Robot, created by Richard Pattis in 1981 and described in his book *Karel the Robot – A Gentle Introduction to the Art of Programming* [Pat95]. Pattis places the emphasis on solving problems in a structured, well-planned fashion. He stresses the need to distinguish the different stages: precise definition of the

problem, planning the solution, implementing the solution, testing the program, and debugging. The robot is programmed in a mini-language based on Pascal which is specially designed to meet the needs of novices.

Karel lives in a simple world, which is basically a grid. It can only move from grid point to grid point, horizontally or vertically. Different types of objects can be in the world, walls between the grid points, beepers on the grid points. Contrary to Logo's turtle, Karel has sensors telling it whether there is wall in front, whether there is a beeper underneath, and what direction it is currently facing. Karel carries a bag which can be used to collect beepers. It understands commands like move, turnleft, pickbeeper, putbeeper.

An introduction to programming with Karel is typically structured by exercises of increasing complexity, which incrementally introduce the basic components of an imperative programming language: if branches, while loops, and procedures. The concept of invariants can be introduced in an intuitive, graphical manner. The following listing shows a sample program in which Karel endlessly tracks the wall of a room.

```
BEGINNING-OF-PROGRAM
  DEFINE-NEW-INSTRUCTION
    turnright AS BEGIN
    turnleft;
    turnleft;
    turnleft;
  END;
  DEFINE-NEW-INSTRUCTION wallfind AS BEGIN
    WHILE front-is-clear DO
      move;
  END;
  BEGINNING-OF-EXECUTION
    wallfind;
    WHILE front-is-blocked DO BEGIN
      turnright;
      WHILE front-is-clear DO BEGIN
        move;
        turnleft;
      END
    END
  END-OF-EXECUTION
END-OF-PROGRAM
```

To experienced programmers, Karel's language is simple. However, novices are confronted with two hurdles. First, they need to debug their program text before the compiler accepts it. This can be very frustrating for beginners trying to express themselves in a language which is still foreign to them. Second, and more importantly, Karel's language is procedural. There is an inherent complexity to procedures, the need to understand the concept of order of execution. The procedures are not written in the same order as they are executed. To comprehend what happens, one needs at least a rudimentary grasp of the concept of the procedure call stack. For novices, understanding this concept is a challenge; understanding recursive algorithms is even more challenging.

A.3 Successors of Karel the Robot

A brief history of the mini-language approach is given in [BCH+97]. It is interesting to note that in Russia, mini-environments were introduced in the early 1980's which were similar to but developed independently of Karel the robot. Since the first version of Karel the robot appeared, there have been numerous re-implementations.

To help beginners getting the structure of their programs right and avoiding unnecessary typos, the Karel Genie environment by Chandhok and Miller includes a structure editor (figure A.2; screen shot taken from [BCH+97]) [CM89]. The programmer writes programs using context-sensitive menus which offer the legal program structure elements for the current cursor context. Little has to be typed manually, keeping the number of syntax errors low, and thereby relieving the programmer of "syntax debugging". Karel Genie was part of a whole family of structure editors implemented on the Macintosh platform. They became quite popular in the US in the 1980's [MPMV94].

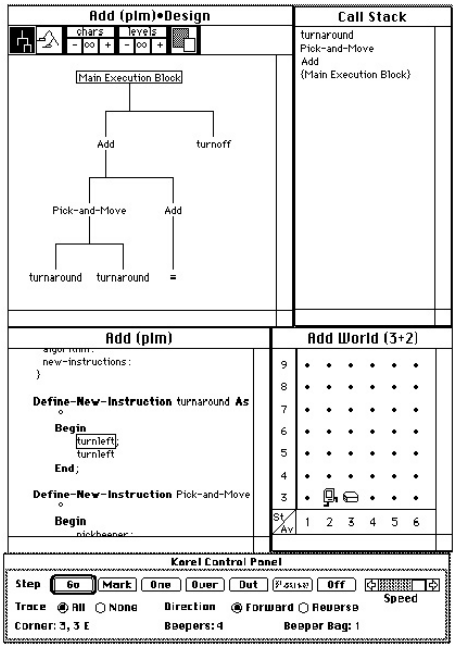

Fig. A.2: The Karel Genie environment

Later implementations of the Karel approach were based on other languages than Pascal. When object-oriented programming became popular, Bergin introduced Karel++, a robot programmed either in a language based on C++ [Ber97] or in a language based on Java [Ber02]. The emphasis is on teaching object-oriented concepts, in particular the subdivision of a system into classes. JKarelRobot by Buck and Stucki can be programmed either in the language of the original Karel, in a language based on Java syntax or in a language based on LISP syntax [BS01]. There are other mini-environments for introducing Java, for example, the hamster by Boles [Bol99]. However, the lifespan of these envi-

ronments is limited to the period of time during which their underlying language is popular.

Some mini-environments attempted to make the world of the robot more interesting by introducing 3d graphics. The idea seems to have been introduced first in 1992 by Hvorecky [Hvo92]. A more recent implementation is The Robot Karol by Freiberger and Krsko [FK02].

A.4 Game-like Environments Targeting Children

A number of more game-like programming mini-environments targeted at children have been built, though not all of them with the goal of teaching programming per se.

For example, KidSim alias the commercial StageCast Creator by Smith et al was developed with the ultimate goal of solving "the end-user programming problem" which is to empower non-computer scientists to program active agents [SCS94]. As a step towards this goal, KidSim addresses end-user programming in the context of agents living in micro-worlds. In contrast to Karel the robot and its successors, children using KidSim have more freedom in constructing their own worlds and their own actors. For example, they can paint an actor and define arbitrary attributes such as height, weight, or likes chocolate, and use these attributes in their programs. A KidSim program is specified in terms of graphical rewrite rules. Figure A.3 shows a simple game from the tutorial. The rule in the rule editor on the bottom right advances the agent if there is no obstacle in front of it.

The graphical rewrite rules are constructed using programming by demonstration mechanisms. One problem is that the order of the rules is crucial, as more than one rule might apply to a given situation. In KidSim, there is no concept of state with which the rules could be subdivided. This makes it hard to get the order of the rules right.

Kahn's ToonTalk environment has an ambitious goal [Kah96]. It attempts to be a self-teaching programming environment for children. It is based on the theoretical model of concurrent constraint programming. The concepts of this programming model tend to be difficult to grasp even for computer science students. In ToonTalk, programming is done using "accessible metaphors". Everything is graphically represented by a metaphor which should be easy to understand. For example, an asynchronous communication channel is represented by birds which have a common nest. When given a message, a bird flies to its nest and puts the message in a queue of messages already waiting to be collected from the nest. The idea is that children should be able to understand the metaphor without having to think about the underlying computing primitive. In the same fashion, there are metaphors for all computing concepts used.

In contrast to most other mini-environments, ToonTalk is explicitly based on the theoretical general-purpose programming model of concurrent programming. This may explain why ToonTalk is rather demanding when one wants to construct robot programs which achieve something meaningful.

The Alice environment by Dann et al is a "interactive animation" mini-

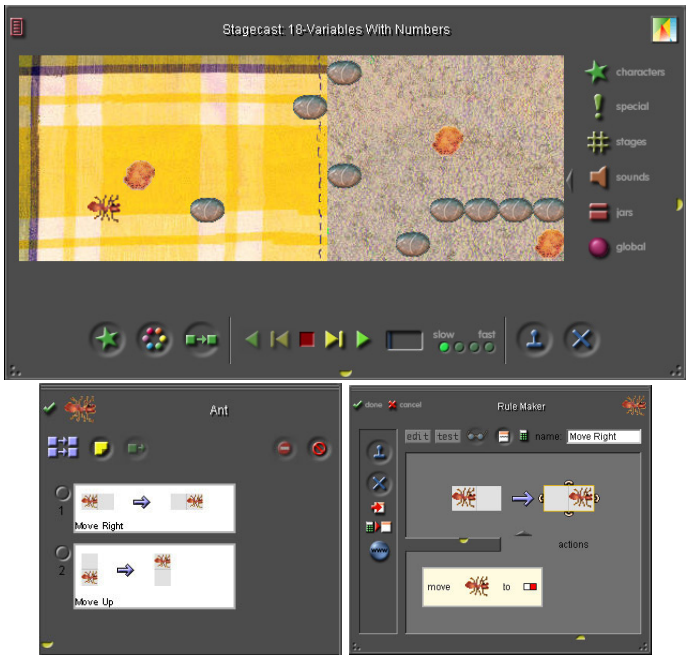

Fig. A.3: StageCast example. Top: world; bottom: all rules, and rule editor

environment based on 3d virtual worlds [DCP00]. Its main goal is to visualize concurrent program execution in such a manner that a debugger is unnecessary. The user constructs a 3d virtual world like the example in figure A.4. The animation sequences access elements of the scene via a tree structure representation. The animation sequences shown in the figure make the bunny face the helicopter and hop, and the helicopter blades spin concurrently. The sequences are coupled with events (not shown in the figure), that is, `HelicopterBladesSpin` is executed upon the start of the animation, and `BunnyHop` is executed when the user clicks the space bar.

In Alice, there is no concept of state, as the animation sequences are independent of each other. The only state in Alice is the state of the world itself. As there are no variables, an animation sequence always reacts in the same way, regardless of what other parts of the program are doing or what happened before.

A.5 Frameworks for Learning Programming

A popular approach to teaching programming is the usage of small frameworks. The idea is that the instructor gives students a framework structure which they extend. Typically, the framework contains code which visualizes many aspects of possible extensions. This approach is usually applied at an intermediate-advanced level, with the goal of teaching aspects of object-oriented programming like inheritance and related principles. Basically, this approach could be described as using the visualization of the world and actor of a mini-environment,

Fig. A.4: Alice example. Top: world scene,bottom: two animation sequences

but using a real-world programming language.

For example, the *Get A Life* framework by Pattis lets students build simulations of artificial life systems such as cellular automata or animal flocking behavior by extending the framework [Pat97]. The framework provides rich graphical visualizations.

Bergin's KarelJRobot implementation can also be viewed as a framework [Ber02]. Students do not work within a mini-environment. They extend the robot classes externally, using the classes as a framework on which to build their robot applications.

A more recent example of a framework is RoboCode [IBM02]. As figure A.5 shows, RoboCode is both a Java framework which the programmer can extend and a mini-environment, as it provides a simple integrated programming environment. The basic idea is slightly martial and game-oriented. Robots are programmed to fight each other on a battlefield initialized by the user.

The following listing shows a RoboCode program. Its main program rocks the robot back and forth, swinging its gun around. RoboCode is inherently multi-threaded, as any number of robots may be fighting each other. It also provides basic event handling mechanisms to allow the programmer to react to external events. In the example, the robot fires each time another robot crosses its gun sight.

```
package man;

import robocode.*;

public class MyFirstRobot extends Robot {
  public void run() {
    while (true) {
```

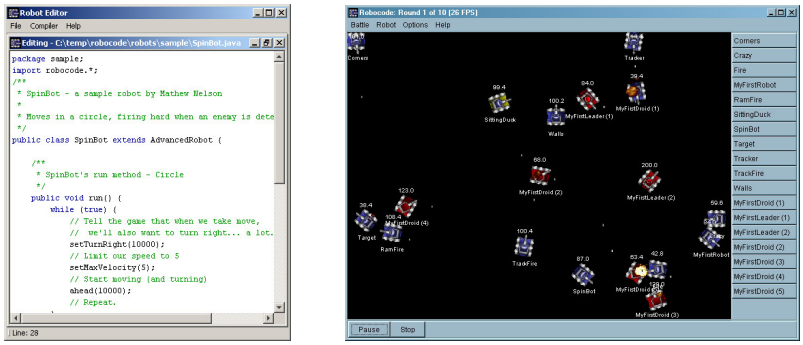

Fig. A.5: The RoboCode environment

```
      ahead(100);
      turnGunRight(360);
      back(100);
      turnGunRight(360);
    }
  }

  public void onScannedRobot(ScannedRobotEvent e) {
    fire(1);
  }

}
```

RoboCode is more complex than most of the mini-languages mentioned above, because the robots are programmed in Java and the programmer must deal with event handling.

A.6 Languages Designed for Education

Probably the first successful language designed for educational purposes was Basic. Kurtz recounts its history in vivid detail in [Kur78]. It is interesting to note that the rationale for Basic is basically the same as that for mini-environments: "While science students will learn computing naturally and well, the nonscience group produces most of the decision makers of business and government [...] This [...] begged the conclusion that nonscience students should be taught computing". Kurtz and Kemeny felt that merely lecturing about computing was not good enough, but that programming languages such as Fortran or Algol were too complex for their purposes. They developed Basic, putting ease-of-use first of all considerations. For example, to the programmer, there is no distinction between source code and executable even though Basic originally was a compiled language. Also, there is no distinction between integer and real numbers, because "to our potential audience the distinction between an integer number and a noninteger number would seem esoteric. A number is a number is a number." Basic quickly attracted a large number of users. It is still used today: in education, an often used implementation is TrueBasic; in industry, a favorite is Microsoft's VisualBasic.

In the early 1970's, Wirth developed the language Pascal as a concise, strongly typed language [Wir71]. Pascal was an ideal vehicle for the systematic programming approach favored by Wirth [Wir73]. He wanted to teach programming as a discipline in its own right, as the systematic construction and formalization of algorithms. Pascal was a huge success, both in education and in industry. There were numerous implementations and variants which are still in professional use today.

Even though to the professional computer scientist, Basic and Pascal are 'small' languages whose syntax can be captured in a small number of pages of EBNF, they are still general-purpose languages. For the introductory purposes of mini-environments such as Karel the robot, they were deemed to complex.

Another approach to teaching programming is the Kernel Language approach described by Van Roy and Haridi [RH02]. They criticize that programming is either taught as a craft or as a branch of mathematics. The craft approach does not emphasize the theoretical, scientific aspects of programming, whereas the mathematical approach is too formal and too restricted to be of practical use. They propose an approach which Reinfelds calls *Teaching of Programming with a Programmer's Theory of Programming* [Rei02]. This approach focuses not on a programming language, but on the underlying principles. The Kernel language captures the essence of many different programming paradigms. Using the Kernel language, one can study how, for example, object-orientation can be implemented using only a small set of fundamental concepts. The teaching of this approach is supported by the Mozart Programming System.

Implementing different programming paradigms using a Kernel language is an interesting approach to study these paradigms. As part of a computer scientist's education, this approach can be helpful to focus not on a specific programming language, but rather on the underlying principles. However, for introductory programming courses as part of general education, this approach is too technical, demanding a high degree of abstract thinking of the students.

Appendix B

User Interface Design of the Kara Environment

The Kara environment is an educational software, and as such its user interface was designed carefully to be as easy-to-use as possible. The interface of any educational software should not obfuscate the concepts to be learned. In Kara's case, it must be obvious to users that they are constructing worlds and finite state machines. Furthermore, the time to learn the handling of the software should be as short as possible, because users use it only for a brief time. This is particularly true for an environment like Kara which is designed to be used by students for just a few hours. Another goal of the user interface design is that students should enjoy working with the environment. Programming is often conceived as tedious by novices. The environment should not reinforce this view by being itself tedious, or even dull.

B.1 The World Editor Window

In terms of Tidwell's human-computer interaction pattern language *Common Ground*, Kara's world editor is a *WYSIWYG editor* [Tid02]. With the *Toolboxes* for controlling the robot and the world objects, this area of the main window constitutes a *Central Working Surface*. The user interface controls needed to edit the world are grouped together with the world the user is editing.

> **Drag&Drop.** *Localized Object Actions* group object actions together and spatially localize them close to the objects they work on. One example is drag&drop, which is an important user interface metaphor in the Kara environment. The user can drag&drop world objects, not only within the world, but also from the world object toolbox into the world. This is what users intuitively expect and try to do, so the user interface conforms to their expectations (figure B.1). The drag&drop metaphor is also used in the program editor.

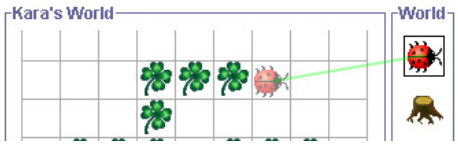

Fig. B.1: Dragging objects into the world

Visibility. The environment emphasizes the principle of visibility. Every action the user can take should be immediately visible, not hidden in a menu. However, some seldomly used features are available only via a popup context-menu by right clicking with the mouse (figure B.2). This is another example of *Localized Object Actions*.

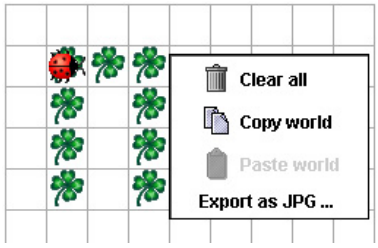

Fig. B.2: Popup context menu in world editor

The user interface literature warns against distinguishing novice and expert users of a software (see, for example, Raskin's *Humane Interface* [Ras00]). The context menu of the world editor is not an "expert mode" of the environment. Its features are invisible, yet available to all users. However, many users will not *need* these features. Therefore, they were removed from the visible working surface.

Error Messages. The *Important Message* pattern suggests making error messages visually obvious, maybe even using sound to notify the user. The Kara environment uses red color to distinguish between normal dialog windows and error message windows.

B.2 The Program Editor Window

The principles applied to the world editor were also applied to the program editor. There are two *Central Working Surface* instances, one for the state machine diagram editor, and one for the state transition table editor.

The diagram editor, originally implemented by Reto Lamprecht [HL01], has a *Toolbox* to create, edit, and delete states. The user can drag&drop to move states in the diagram, and to create transitions between states (figure B.3 top). The state transition table editor also has a *Toolbox* to its left, containing commands which can be inserted into the transitions via drag&drop (see figure B.3 bottom).

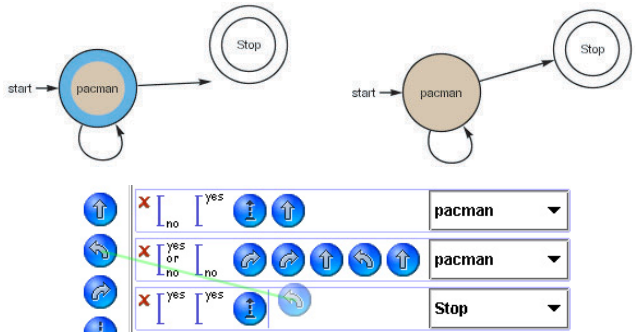

Fig. B.3: Drag&drop in the program editor

Both the state diagram editor and the state transition table editor have *Localized Object Actions* in the form of context menus. The menus let the user copy&paste states and transitions. Again, these features are not a must to work with the program editor, but optional short-cuts.

The program editor is entirely visual. The user only needs to type names for the states. With respect to the environment, the names are irrelevant and could be omitted. However, it makes programming easier for users because they can identify states by their names rather than by their position in the state diagram.

B.3 Executing Programs

Visualizing the execution of programs is done in the standard technique of algorithm animation. The program code and the data structures modified by it are both displayed to allow the user to trace execution (see, for example, [SDBP98]). This technique is applied in most mini-environments as well as professional debuggers. Figure B.4 shows the situation before and after a command was executed.

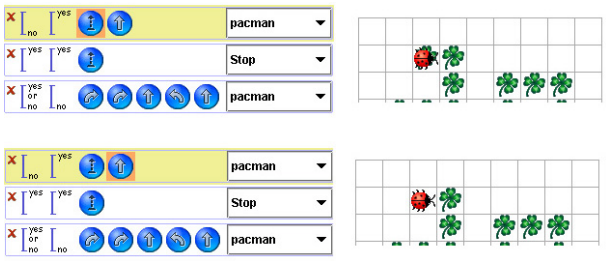

Fig. B.4: Executing a single step of a program

'Mode' is a concept often criticized by the user interface literature. The problem is that the program may react differently to the user's actions depending on the current mode. Users must remember in which mode they put the program. Nevertheless, the Kara environment has two modes. First, users can

put the interpreter in deterministic or non-deterministic mode in the preferences dialog. They can keep this dialog open so that the current mode stays visible.

The second mode distinguishes editing from executing programs. While a program is executed, users are prohibited from modifying the program. If they attempt a modification, an error message explains that they must first stop execution. It would be possible to do without this mode. However, we felt users should follow the edit–execute cycle typical of most programming environments.

Appendix C

Architecture and Design of the Kara Environments

The main goal in designing the overall architecture of the Kara environments was to have an architectural framework which could accommodate all environments (Kara, TuringKara, MultiKara, and JavaKara). It became clear that the structure of the system should accommodate the following requirements:

World editor. The world editor must be able to manage varying sets of "world objects" with different semantics. For example, the TuringKara environment uses a set of world objects (read/write head, 0, 1, □, etc.) different from that of the ladybug environments (ladybug, tree, leaves, mushrooms).

Program editor. It must be possible to use at least two different program editors: a state machine editor for the environments based on the state machine programming paradigm, and a text editor for the JavaKara environment. The program editor must also be capable of supporting the creation of programs for multiple actors.

Program interpreter. It must be possible to use at least two different interpreters: a state machine interpreter and a Java 'interpreter'.

Scheduling and the Number of Actors. Some environments have a single actor which is controlled directly by an interpreter. Other environments have multiple actors and need a scheduler to manage the individual interpreters of these actors. The design must provide for both cases.

User interface. The common parts of the various user interfaces must be immediately recognizable to a user working with different environments.

With the goal of making the development of such a family of environments manageable, the following technical requirements had to be met:

Common code base. The basic idea behind the architecture of the applications is to maximize the common code base in order to make maintenance of all applications easier. In many cases, components of one environment are specialized adaptations of abstract, generalized common components.

Modular structure. It must be possible to easily add a new environment to the existing environments. This means that new environments are extensions of the existing environments in the object-oriented sense. For example, the MultiKara environment is built upon the Kara environment and reuses many of its components.

Another requirement, not related to the structure of the system, was that the Kara environments should be platform-independent and run on as many platforms as possible. For this reason, the software is written in Java. The environments have been successfully tested on Windows (95, 98, 2000, ME, XP), Linux, Solaris, and Mac OS X with a number of Java versions (Java 1.2, 1.3, 1.4).

C.1 Overall System Architecture

From the above requirements, the division of the application into three subsystems based on two major abstractions was derived. The first abstraction is that of an *editor*. Both the world editor and the two program editors are concrete implementations of this abstraction (figure C.1). The second abstraction is that of an *interpreter*.

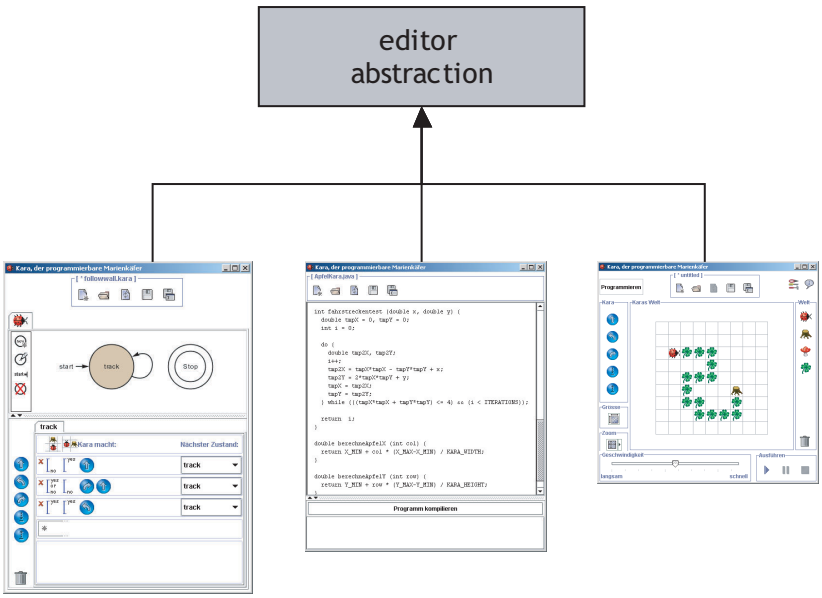

Fig. C.1: Implementations of the editor abstraction

The three subsystems are the world editor subsystem, the program editor subsystem, and the interpreter subsystem (figure C.2). For each subsystem, there are implementations for the different environments. The applications communicate with these subsystems through interfaces that are designed to be as small as possible in order to allow a maximal degree of freedom on both sides.

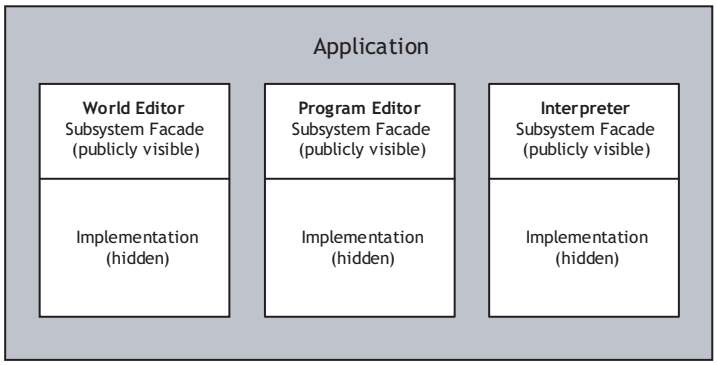

Fig. C.2: Application and its three subsystems

The environments comprise approximately 50 packages containing more than 380 classes, with a total of more than 72'000 lines of code. Figure C.3 shows some of the top level packages of the environments and their dependencies, without their subpackages. Note that the educeth term in the package names denotes the EducETH web server which hosts the applications. The kapps package (short for "kara applications") is the top level package for all environments.

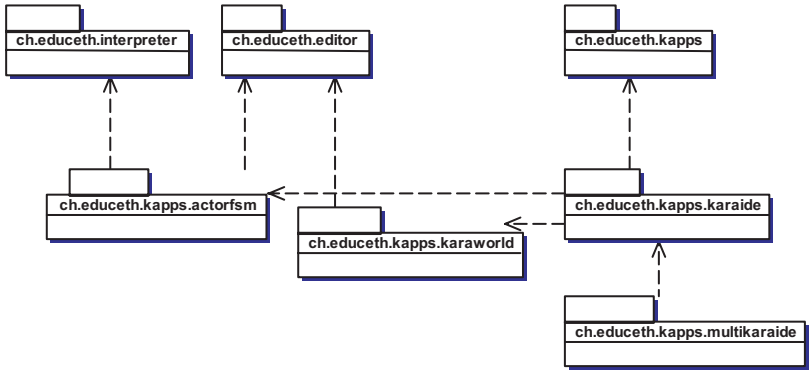

Fig. C.3: Top level packages of the environments (extract, without subpackages)

The following sections discuss the structure of the data models of the environments. The user-interface related classes are not discussed. For example, the karaworld package models the world of Kara and contains a subpackage editor which contains the graphical world editor. The editor and other user interface components are basically views on the data models, typically implemented as realizations of the classical *Model-View-Controller* design pattern.

The Three Subsystems of the Environments

The main entry point of the Kara environments is the class Application in the package kapps. The Application class sees the three subsystems through *facade interfaces*. Figure C.4 shows this overall structure.

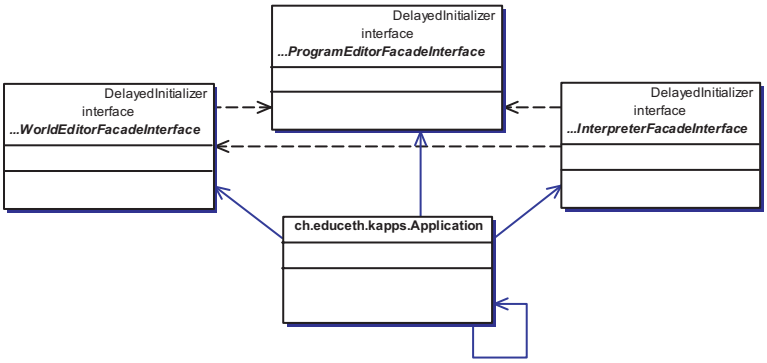

Fig. C.4: Overall system structure (details not shown)

Facade is one of the design patterns from the *Design Patterns* book [GHJV86]. Several of these design patters were applied in the development of the environments; any reference to a design pattern in the remainder of this chapter will refer to the definitions of this book.

The configuration file of the applications determines the classes of facade implementations of the subsystems to be used for the current environment. The following listing is an extract from the configuration file. It determines which facade implementations are to be used for the Kara and MultiKara environments.

```
<karamodel>
  <kara>
    <facades>
      <worldeditorfacade>
        ch.educeth.kapps.karaide.KaraWorldEditorFacade
      </worldeditorfacade>
      <programeditorfacade>
        ch.educeth.kapps.karaide.KaraProgramEditorFacade
      </programeditorfacade>
      <interpreterfacade>
        ch.educeth.kapps.karaide.KaraInterpreterFacade
      </interpreterfacade>
    </facades>
    <!-- .... -->
  </kara>
  <multikara>
    <facades>
      <worldeditorfacade>
        ch.educeth.kapps.multikaraide.MultiKaraWorldEditorFacade
      </worldeditorfacade>
      <programeditorfacade>
        ch.educeth.kapps.multikaraide.MultiKaraProgramEditorFacade
      </programeditorfacade>
      <interpreterfacade>
        ch.educeth.kapps.multikaraide.MultiKaraInterpreterFacade
```

```
      </interpreterfacade>
    </facades>
    <!-- ... -->
  </multikara>
  <!-- ... -->
</karamodel>
```

Defining the facades in the configuration file makes it easy to add a new environment to the existing application. The following listing shows how the Application class creates instances of the facades of the three subsystems, based on the above configuration extract. After creating the facade instances, they are connected according to the dependencies shown in figure C.4.

```
Class weFacadeClass = Class.forName(
  Configuration.getInstance().getString(
    Konstants.KARAMODEL_WORLDEDITORFACADE
  )
);
worldEditorFacade =
  (WorldEditorFacadeInterface)weFacadeClass.newInstance();

Class programEditorFacadeClass = Class.forName(
  Configuration.getInstance().getString(
    Konstants.KARAMODEL_PROGEDITORFACADE
  )
);
programEditorFacade =
  (ProgramEditorFacadeInterface)programEditorFacadeClass.newInstance();
programEditorFacade.setWorldEditorFacade(worldEditorFacade);

Class interpreterFacadeClass = Class.forName(
  Configuration.getInstance().getString(
    Konstants.KARAMODEL_INTERPRETERFACADE
  )
);
interpreterFacade =
  (InterpreterFacadeInterface)interpreterFacadeClass.newInstance();
interpreterFacade.setEditorFacades(worldEditorFacade, programEditorFacade);
```

The facade interfaces define what the application needs to know about a particular subsystem. As an example, figure C.5 shows the facade of the program editor subsystem in detail. The application instantiates an object of a class implementing the program editor facade interface. It queries the facade to obtain its user interface (method programEditorGui), among other things. Later, the interpreter facade will query the program editor facade to obtain the interpreter listeners of the program editor subsystem, if any (method interpreterListeners). The program editors use such a listener to block out user modifications to the program while it is executed.

Editor Abstraction

In each Kara environment, there is a program editor and a world editor. The EditorInterface in figure C.6 defines the abstract interface of a simple editor. An editor must be able to create a new, empty content (method newFile), to write its content to disk (method save), to read its content from disk (method load). It

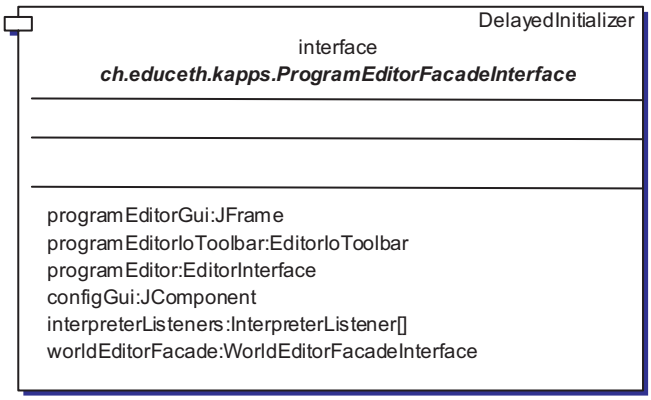

Fig. C.5: The facade of the program editor subsystem

must be possible to query the editor whether there have been any modifications since the last save, load or newFile command (method isModified).

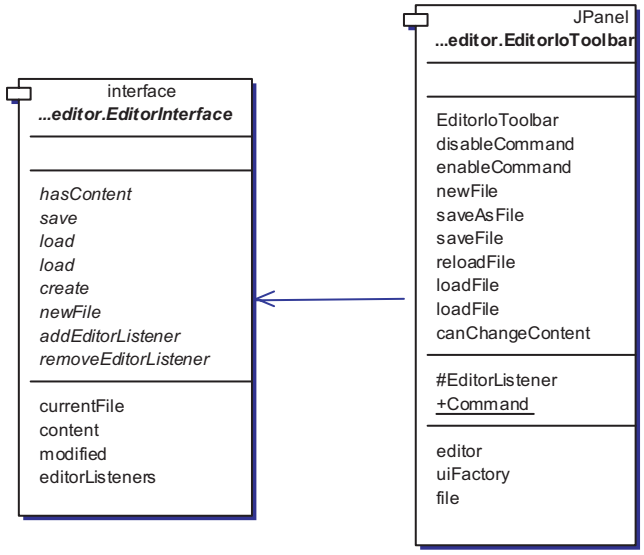

Fig. C.6: Editor interface

There are no menus in the environments, with the exception of local popup menus. The user basically controls the user interface through toolbars. The EditorIoToolbar implements a simple toolbar with buttons to create, load, reload, and save files. The editor interface and the toolbar provide a uniform interface to all editors of the environments. They are completely independent of the type of the content of the editor. To provide for this independence, an interface EditorToolbarUiFactoryInterface (not shown in figure C.6) encapsulates the creation of application-dependent user interface elements, according to the *Factory* design pattern.

Interpreter Abstraction

The state machine based environments have interpreters completely different from that of the JavaKara environment. Some of the state machine based environments need a single interpreter, some need a scheduler to manage multiple interpreters. These requirements led to the definition of a simple "general-purpose" interpreter abstraction. Figure C.7 shows the structure of the basic interpreter-related interfaces and classes.

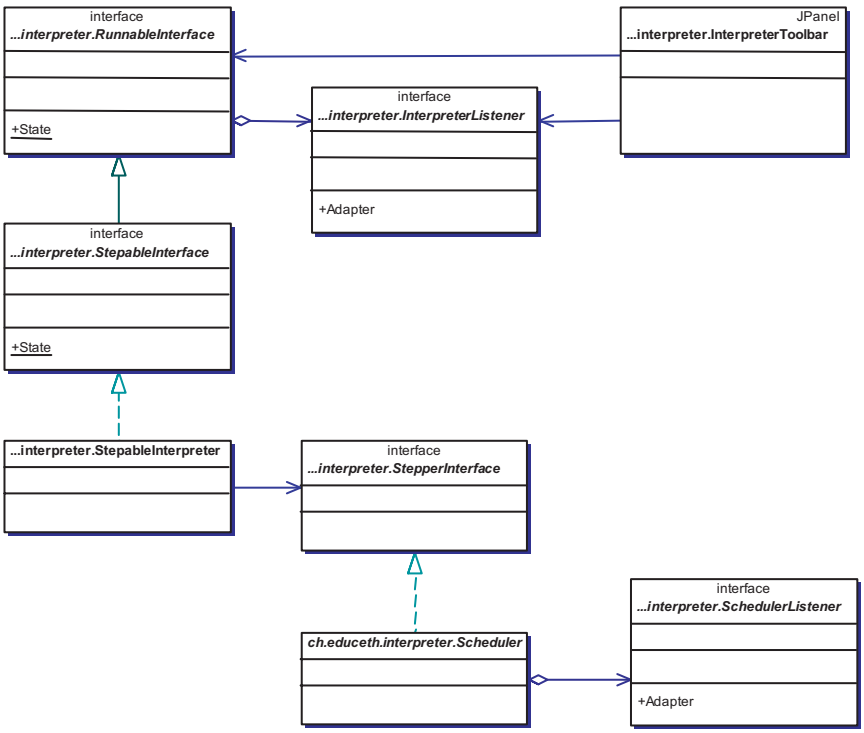

Fig. C.7: Interpreter-related interfaces and classes

The RunnableInterface defines an interpreter which is capable of running a program. It is assumed that after each event of interest (not shown in figure C.7), it will notify its InterpreterListener objects. This realizes the *Observer* design pattern so that interested parties can follow the progress of the interpreter. The StepableInterface defines an interpreter which is capable of executing a program step-by-step. The reason for the distinction between these two types of interpreters is that within the JavaKara environment, there is no debugger to execute a Java program stepwise.

The StepableInterpreter class is an implementation of the interface, managing the execution of a StepperInterface object in a separate thread. This interface basically defines a command method executeStep and a query method executionFinished. These two methods allow the interpreter to control the stepper.

A Scheduler is itself capable of step-by-step execution of a number of step-by-step capable interpreters. The SchedulerListener interface realizes the *Observer* design pattern to allow interested parties to track the actions of the scheduler.

The InterpreterToolbar implements a simple toolbar with buttons to start, step, pause, and stop execution of programs. The toolbar is completely independent of the actual interpreter instance controlled by the toolbar. To provide for this independence, an interface InterpreterToolbarUiFactoryInterface (not shown in figure C.7) encapsulates the creation of application-dependent user interface elements, again according to the *Factory* design pattern.

Finite State Machine Model

With the exception of the JavaKara environment, all environments revolve around finite state machines. The actorfsm package defines what a state machine is, with regard to the Kara context: a controller for some type of actor. Figure C.8 shows the overall structure of this package.

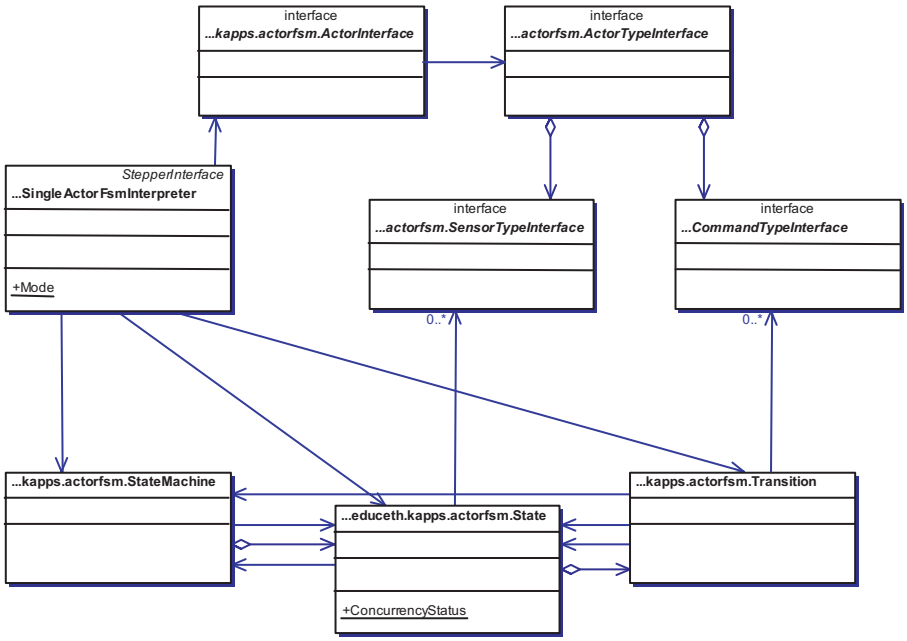

Fig. C.8: State machine model

A StateMachine consists of a number of State objects, which in turn consist of a number of Transition objects. A state machine is specified as a controller for a certain ActorTypeInterface implementation. An actor type defines the types of sensors (SensorTypeInterface) and the types of commands (CommandTypeInterface) of an ActorInterface object of this actor type.

For each state, the sensors relevant to it are specified. For each transition, the input values for these sensors are specified, as well as a list of commands. When the state machine interpreter tries to find a transition from the current state, these specified input values are compared with the actual sensor values of the actor. The following listing shows how the inputMatch method of the Transition class compares these values.

```
boolean inputMatch(boolean[] sensorInput,
```

```
SingleActorFsmInterpreter.Mode mode) {

  for (int i = 0; i < sensorInput.length; i++) {
    if (this.sensorInput[i] != SENSOR_NOTUSED) {
      int intSensorInput = sensorInput[i] ? SENSOR_TRUE : SENSOR_FALSE;
      if ((mode == SingleActorFsmInterpreter.Mode.AND) &&
          (intSensorInput != this.sensorInput[i])) {
        return false;
      }
      else if ((mode == SingleActorFsmInterpreter.Mode.OR) &&
                (intSensorInput == this.sensorInput[i])) {
        return true;
      }
    }
  }

  return (mode == SingleActorFsmInterpreter.Mode.AND);
}
```

The state machine interpreter SingleActorFsmInterpreter can be run in two modes. In AND mode, all sensor values must match the specified values, whereas in OR mode, one match suffices. Kara and MultiKara run in AND mode; TuringKara runs in OR mode.

The graphical state machine diagram editor is based on an editor by Reto Lamprecht [HL01]. The graphical transition table editor was modelled after an idea used in a similar editor by Thomas Suter [Sut01]. The drag&drop mechanism used throughout the environments is also an extension of his work.

World Model

A central data structure in all environments is the representation of the world. All environments use the same basic world model: a two-dimensional container of world objects. The difference between the environments is the types of objects the world can contain. Figure C.9 shows the world model related interfaces and classes.

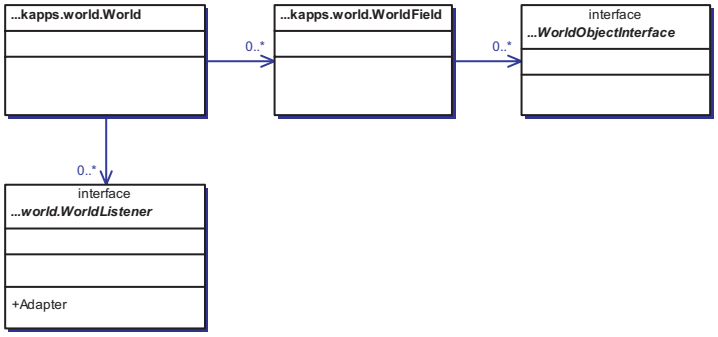

Fig. C.9: World data model

Class World represents a two-dimensional array of WorldField objects. A world square contains a stack of WorldObjectInterface objects. This interface defines the semantics of a world object. For example, method canCombineWith (WorldObjectInterface other) queries a world object whether it can be on the

same world square stack as the specified other object. Implementations of this interface are environment-dependent and therefore not part of the world model package, but part of the environment-specific packages.

C.2 Putting it All Together: The ide Packages

The above discussion showed parts common to many or all Kara environments, like the editor and interpreter abstraction and the abstract world editor. The following sections outline the packages which contain the actual "integrated development environments" for the Kara and MultiKara environment; the respective packages for TuringKara and JavaKara are comparable in structure.

The Kara "Integrated Development Environment"

The karaide package puts together all the pieces needed to create the Kara environment. It adds classes to customize some of the abstractions so that they can be used in the context of the Kara environment. Figure C.10 shows some of the data model classes of this package.

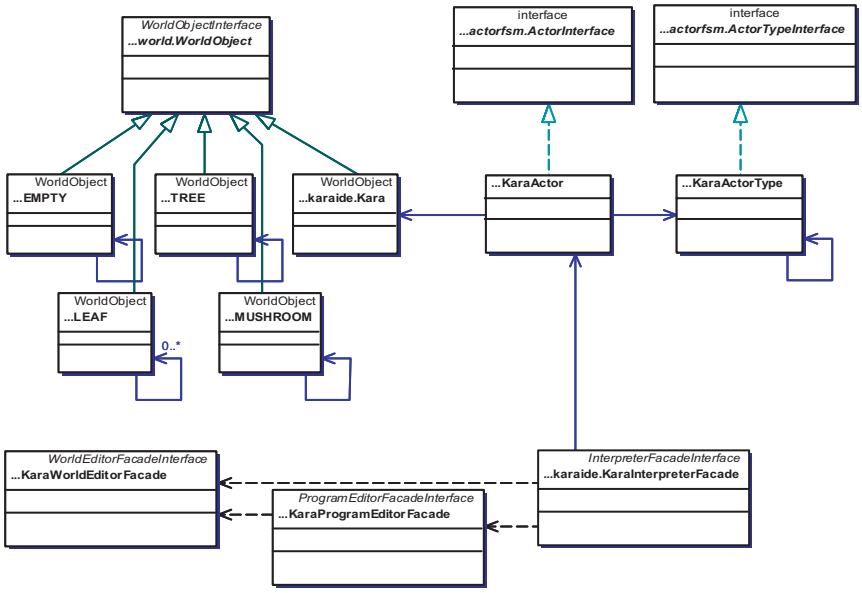

Fig. C.10: Package karaide structure

The classes EMPTY, LEAF, TREE, MUSHROOM, Kara extend the WorldObject class, which is an abstract implementation of the WorldObjectInterface. Instances of these classes are used as prototypes for the world editor. The KaraActor class implements the ActorInterface and is basically a realization of the *Adapter* design pattern. Such an adapter is needed because the state machine editor package is completely independent of Kara and its world.

The three facade classes implement the interfaces of the application facades, creating all objects needed by the Kara environment and hooking them up when the application starts.

The MultiKara "Integrated Development Environment"

The classes of the multikaraide package extend the classes of the karaide package and add some new classes. Figure C.10 shows the classes from both packages to illustrate the relationship between them.

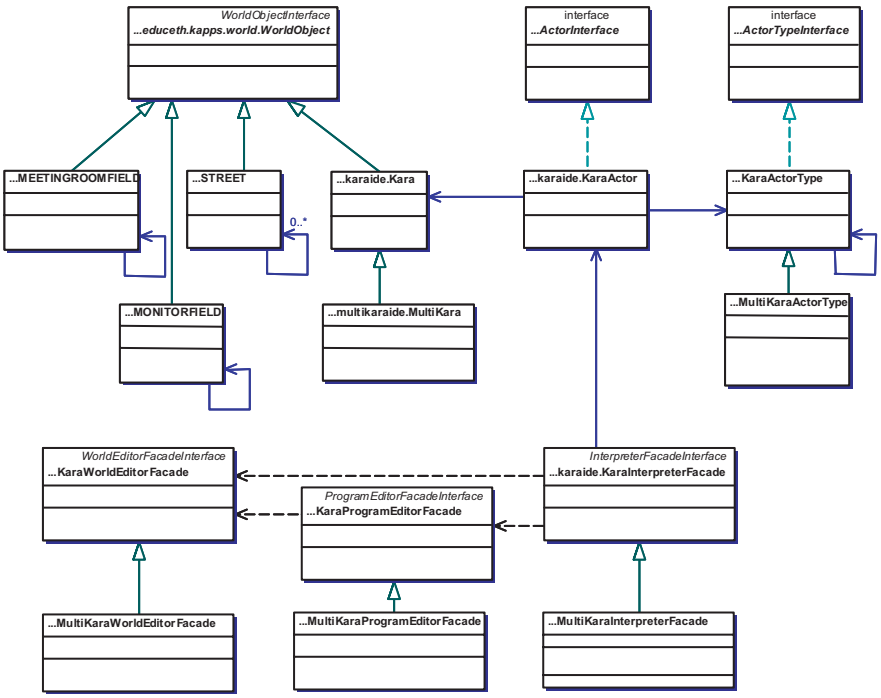

Fig. C.11: Package multikaraide structure

MultiKara adds three new types of world objects to the Kara environment: STREET, MONITORFIELD, MEETINGROOMFIELD. Class MultiKara extends class Kara; its move command makes sure the square in front can be entered with respect to a possible street sign on that square. And while both environments use the same actor implementation, KaraActor, MultiKara has a different actor type, MultiKaraActorType. This actor type is capable of dynamically adding sensors to itself, which is a feature that the KaraActorType does not have.

The three facade classes extend their counterparts of the karaide package, redefining some creator functions to create certain objects as instances of the classes of the multikaraide package.

C.3 Implementation Issues

The Kara environments are a software project of respectable size and a certain complexity, in particular since the project is not a single application, but rather a framework for a whole family of applications. Special care was taken on the implementation level to achieve reliable and maintainable code. The following sections outline some of the techniques applied during development.

Design by Contract

Design by Contract was introduced by Bertrand Meyer and is explained in detail in *Object Oriented Software Construction* [Mey97]. The idea is "viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations". Assertions express pre- and postconditions for methods as well as invariants for classes. The goal is to make software more reliable by providing a conceptual tool which can be used at all stages of development: analysis, design, implementation, and documentation.

The Java language does not have built-in facilities for design by contract. There exist a number of tools to add such facilities. iContract by Reto Kramer is non-commercial software [Kra98]. The assertions of iContract are written as expressions which are a superset of Java, compatible with a subset of the latest UML Object Constraint Language. Assertions are specified using the JavaDoc code documentation mechanism. For example, the following listing shows the assertions on a method removeItem of RowInterface, where a row is a container of items. The precondition states that the item to be removed must be an element of the row. The postconditions state that the item will not be found in the row after removal, and that the size of the row will have decreased by one.

```
/**
 * Remove <code>item</code>.
 *
 * @pre has(item)
 * @post !has(item)
 * @post size()@pre - 1 == size()
 */
void removeItem(ItemInterface item);
```

An iContract pre-compiler translates the assertions into instrumented Java source code. Compiling this code yields an instrumented executable. One drawback of the pre-compiler approach is that it is hard to use a pre-compiler without full integration into the development environment. Markus Brändle wrote a plug-in for Borland's JBuilder, the environment used for developing Kara. The plug-in integrates into the environment and makes it easy to compile and execute with the assertions enabled.

Test Cases

Test cases are an important development technique in any programming project of non-trivial size. Even though this is not a new idea, the importance of test cases was proclaimed anew in the recent past by Beck's *Extreme Programming* approach to project development and management [Bec99]. In particular, this approach emphasizes so called "unit tests", typically defined as a white-box test

that is concerned with a small part of implementation-oriented functionality. A collection of unit tests is assembled into test suites, where test suites can also contain test suites. This hierarchical nesting of test suites makes it possible to have test suites on different levels of an application, and to have one top level test suite which runs all test cases of the application.

In the development of the Kara software, we used the JUnit test framework by Beck and Gamma to implement test cases [BG02]. The basic idea is to make sure that if we change something, everything still works. For example, we want to make sure that all example programs execute without causing an exception. The following listing shows a code fragment which executes programs on a number of test worlds. The programs and worlds are chosen in such a way that the execution should terminate and succeed without causing exceptions.

```
 while (programs.hasMoreElements()) {
  String programName = (String)programs.nextElement();
  String[] worldNames  = (String[])testCases.get(programName);

  InputStream program = getClass().getResourceAsStream(programName);
  programEditor.load(program);
  for (int i = 0; i < worldNames.length; i++) {
    InputStream world = getClass().getResourceAsStream(worldNames[i]);
    worldEditor.load(world);

    interpreter.play();
    synchronized (interpreter) {
      interpreter.wait();
    }
  }
}
```

We implemented test cases on a number of different levels. There are test cases for testing individual methods of a class, for testing individual classes, for testing the collaboration of multiple classes, and for testing whole environments.

Implementation Principles

Two major principles of the designing class interfaces from [Mey97] were observed in the development of the Kara software:

Command-Query Separation. The idea is to view an object as a machine. A machine has a publicly visible interface and a private, internal state. The interface defines commands as methods which change the internal state of the object, but do not return a value. The interface also defines queries as methods which return information, but do not change the observable state of the object. This basically guarantees that "asking a question does not change the answer".

Operands and Options Principle. An operand argument to a method is an object on which the method will operate. An option argument specifies how the method should operate on the operands. For example, in a method call like console.printReal (5.4, "##.##"), 5.4 is the operand, and the string represents an option. The Operands and Options Principle states that the arguments of a method should only be operands, but not options. Options should be set using setter methods. The reason is

that operands are more stable during the evolution of software, whereas options are likely to evolve.

Efficiency and Memory

Even though interpretation of Java programs has become much faster with its newer versions, and even though the graphical user interface libraries have been optimized, the rendering of graphical components is still not very fast. Another problem is that Swing, the graphical user interface library of Java, has problems with memory leaks.

We therefore optimized the rendering of our own graphical components as much as possible, identifying performance bottlenecks with Borland's OptimizeIt [Bor02]. For example, we achieved a massive speedup of the rendering of the world view. At first, we used the standard Java graphics mechanism for scaling the world view at any arbitrary zoom factor. This made the width of a square a real number and rendering very expensive. We decided that only scale factors could be used such that the width of a square view is an integer. We also implemented the zooming ourselves to speed up things even more.

The problem of memory leaks has two issues. First, long-lived objects hold references to short-lived objects, which prevents the garbage collector from freeing the latter. A mechanism to solve this problem was presented by Reichert in [Rei99]. The basic idea is to use so called weak references from long-lived objects to short-lived objects, and not the usual direct, strong references. Second, some memory leaks are caused by bugs in the Swing library. They can usually be avoided if creation of Swing objects is avoided as much as possible. We therefore use pools to which Swing objects are added when they are no longer needed. When such an object needs to be created, the creator method first checks the pool to see if an object can be reused. A new object is created only if this is not the case. This keeps the number of newly created Swing objects to a minimum.

File Format for Input and Output

The file format for worlds and programs is XML. To create the XML representations the Java™ Architecture for XML Binding (JAXB) is used [Sun02]. The architecture provides an API to automate the mapping between XML documents and Java objects. DTDs (document type definitions) are used to define the XML structure of the files to be written. JAXB implicitly validates the XML structure according to the specified DTD. This makes it extremely easy to assure that a created XML structure is valid and that it can later be used to recreate the Java objects.

DTD for worlds

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT XmlWorld (XmlWallPoints, XmlObstaclePoints,
XmlPaintedfieldPoints, XmlKaraList,
XmlStreetList, XmlMeetingroomList*, XmlMonitorList*)>

<!ATTLIST XmlWorld sizex CDATA #REQUIRED>
<!ATTLIST XmlWorld sizey CDATA #REQUIRED>
<!ATTLIST XmlWorld version CDATA #REQUIRED>

<!ELEMENT XmlWallPoints (XmlPoint*)>
<!ELEMENT XmlObstaclePoints (XmlPoint*)>
<!ELEMENT XmlPaintedfieldPoints (XmlPoint*)>

<!ELEMENT XmlPoint EMPTY>
<!ATTLIST XmlPoint x CDATA #REQUIRED>
<!ATTLIST XmlPoint y CDATA #REQUIRED>
<!ATTLIST XmlPoint type CDATA #IMPLIED>


<!ELEMENT XmlKaraList (XmlKara*)>
<!ATTLIST XmlKaraList startingOrder CDATA #IMPLIED>
<!ATTLIST XmlKaraList priorities CDATA #IMPLIED>
<!ATTLIST XmlKaraList parkingOrder CDATA #IMPLIED>

<!ELEMENT XmlKara EMPTY>
<!ATTLIST XmlKara x CDATA #REQUIRED>
<!ATTLIST XmlKara y CDATA #REQUIRED>
<!ATTLIST XmlKara name CDATA #REQUIRED>
<!ATTLIST XmlKara direction CDATA #REQUIRED>


<!ELEMENT XmlStreetList (XmlStreet*)>
<!ELEMENT XmlStreet EMPTY>
<!ATTLIST XmlStreet x CDATA #REQUIRED>
<!ATTLIST XmlStreet y CDATA #REQUIRED>
<!ATTLIST XmlStreet type CDATA #REQUIRED>


<!ELEMENT XmlMeetingroomList (XmlMeetingroom*)>
<!ELEMENT XmlMeetingroom EMPTY>
<!ATTLIST XmlMeetingroom x CDATA #REQUIRED>
<!ATTLIST XmlMeetingroom y CDATA #REQUIRED>

<!ELEMENT XmlMonitorList (XmlMonitor*)>
<!ELEMENT XmlMonitor EMPTY>
<!ATTLIST XmlMonitor x CDATA #REQUIRED>
<!ATTLIST XmlMonitor y CDATA #REQUIRED>
```

DTD for programs

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT XmlStateMachines
  (XmlStateMachine*,XmlSensorDefinition*)>
<!ATTLIST XmlStateMachines version CDATA #REQUIRED>

<!ELEMENT XmlStateMachine (XmlState*,XmlTransition*)>
<!ATTLIST XmlStateMachine startState CDATA #IMPLIED>
<!ATTLIST XmlStateMachine actor CDATA #REQUIRED>

<!ELEMENT XmlState (XmlDescription, XmlSensors)>
<!ATTLIST XmlState name CDATA #REQUIRED>
<!ATTLIST XmlState finalstate CDATA #IMPLIED>
<!ATTLIST XmlState x CDATA #REQUIRED>
<!ATTLIST XmlState y CDATA #REQUIRED>
<!ATTLIST XmlState barrierstate CDATA #IMPLIED>
<!ATTLIST XmlState criticalsectionstate CDATA #IMPLIED>

<!ELEMENT XmlDescription (#PCDATA)>

<!ELEMENT XmlSensorDefinition EMPTY>
<!ATTLIST XmlSensorDefinition identifier CDATA #IMPLIED>
<!ATTLIST XmlSensorDefinition name CDATA #REQUIRED>
<!ATTLIST XmlSensorDefinition description CDATA #IMPLIED>
<!ATTLIST XmlSensorDefinition parameterString CDATA #IMPLIED>


<!ELEMENT XmlSensors (XmlSensor*)>

<!ELEMENT XmlSensor EMPTY>
<!ATTLIST XmlSensor name CDATA #REQUIRED>

<!ELEMENT XmlTransition (XmlSensorValues, XmlCommands)>
<!ATTLIST XmlTransition from CDATA #REQUIRED>
<!ATTLIST XmlTransition to CDATA #REQUIRED>

<!ELEMENT XmlSensorValues (XmlSensorValue*)>
<!ELEMENT XmlSensorValue EMPTY>
<!ATTLIST XmlSensorValue name CDATA #REQUIRED>
<!ATTLIST XmlSensorValue value CDATA #REQUIRED>

<!ELEMENT XmlCommands (XmlCommand*)>
<!ELEMENT XmlCommand EMPTY>
<!ATTLIST XmlCommand name CDATA #REQUIRED>
```

Bibliography

[ACH+68] Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T. E., Keenan, T. A., Kehl, W. B., McCluskey, E. J., Navarro, S. O., Rheinboldt, W. C., Schweppe, E. J., Viavant, W., and David M. Young, J. Curriculum 68: Recommendations for academic programs in computer science: a report of the ACM curriculum committee on computer science. *Communications of the ACM*, 11(3):151–197, 1968.

[AI02] ACM, and IEEE. *Curriculum 2001.* http://www.computer.org/education/cc2001/, October 2002.

[BA90] Ben-Ari, M. *Principles of Concurrent and Distributed Programming.* Prentice Hall, Essex, England, 1990.

[BAK99] Ben-Ari, M., and Kolikant, Y. B.-D. Thinking parallel: the process of learning concurrency. In *Proceedings of the 4th annual SIGCSE/SIGCUE on Innovation and technology in computer science education*, pages 13–16. ACM Press, 1999.

[BC96] Bynum, B., and Camp, T. After you, Alfonse: a mutual exclusion toolkit. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 170–174. ACM Press, 1996.

[BCH+97] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. Mini-languages: A way to learn programming principles. *Education and Information Technologies*, 2(1):65–83, 1997.

[BD88] Burns, A., and Davies, G. Pascal-FC: A language for teaching concurrent programming. *ACM SIGPLAN Notices*, 23(1):58–66, 1988.

[BE00] Barwise, J., and Etchemendy, J. *Language, Proof and Logic.* CSLI Publications, 2000. http://www-csli.stanford.edu/hp/ (October 2002).

[Bec99] Beck, K. *Extreme Programming Explained: Embracing Change.* Addison-Wesley, 1999.

[Ber97] Bergin, J. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming.* Wiley, New York, 1997.

[Ber02] Bergin, J. *Karel++ for Java.* http://csis.pace.edu/~bergin/KarelJava/, October 2002.

[BG02] Beck, K., and Gamma, E. *JUnit Testing Framework*. http://www.junit.org/, October 2002.

[BKMT94] Brusilovsky, P., Kouchnirenko, A., Miller, P., and Tomek, I. Teaching programming to novices: A review of approaches and tools. In *Educational Multimedia and Hypermedia. Proceedings of ED-MEDIA'94 – World Conference on Educational Multimedia and Hypermedia*, pages 103–110, 1994.

[Blo56] Bloom, B. *Taxonomy of Educational Objectives*. Longmans, London, 1956.

[Bol99] Boles, D. *Programmieren spielend gelernt. Mit dem Java-Hamster-Modell.* Teubner, 1999.

[Bor02] Borland Inc. *OptimizeIt*. http://www.borland.com/optimizeit/, October 2002.

[Bra86] Braitenberg, V. *Vehicles and Experiments in Synthetic Psychology*. M.I.T. Press, 1986.

[Bra02] Braendle, M. Multikara. Diploma thesis, Dept. of Computer Science, ETH Zürich, 2002.

[Bro91] Brooks, R. A. Intelligence without representation. *Artificial Intelligence*, 47:139–159, January 1991.

[Bro96] Brooks, F. P. The computer scientist as toolsmith II. *Communications of the ACM*, 39(3):61–68, 1996.

[Bru60] Bruner, J. S. *The Process of Education*. Harvard University Press, 1960.

[BS01] Buck, D., and Stucki, D. J. JKarelRobot: A case study in supporting levels of cognitive development in the computer science curriculum. In *Proceeding of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, volume 33, pages 16–20, February 21–25 2001.

[Bun02] Bundeswettbewerb Informatik. *Programming problems of the 20^{th} Bundeswettbewerb Informatik 2001/2002*. http://www.bwinf.de/download/201beispiellsg.pdf, October 2002.

[CM89] Chandhok, R. P., and Miller, P. L. The design and implementation of the Pascal GENIE. In *Proceedings of the seventeenth annual ACM conference on Computer science: Computing trends in the 1990's*, pages 374–379, 1989.

[dBOM99] du Boulay, B., O'Shea, T., and Monk, J. The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human-Computer Studies*, 51(2):265–277, 1999.

[DCG+89] Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[DCP00] Dann, W., Cooper, S., and Pausch, R. Making the connection: programming with animated small world. In *5th annual SIGCSE/SIGCUE conference on Innovation and technology in computer science education*, pages 41–44, 2000. See also http://www.alice.org/ (October 2002).

[Den89] Denning, P. J. A debate on teaching computing science. *Communications of the ACM*, 32(12):1397–1414, 1989.

[Dew89] Dewdney, A. K. *The Turing Omnibus: 61 Excursions in Computer Science*. Computer Science Press, 1989.

[Dij86] Dijkstra, E. W. On a cultural gap. *The Mathematical Intelligencer*, 8(1):48–52, 1986.

[Dij89] Dijkstra, E. W. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1398–1404, December 1989. Published within [Den89].

[FK02] Freiberger, U., and Krsko, O. *The Robot Karol*. http://www.schule.bayern.de/karol/, October 2002.

[Gal98] Gale, D. *Tracking the automatic ant and other mathematical explorations. A collection of mathematical entertainment columns from the Mathematical Intelligencer*. Springer, New York, 1998.

[GHJV86] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986.

[GS02] Guzdial, M., and Soloway, E. Teaching the nintendo generation to program. *Communications of the ACM*, 45(4):17–21, 2002.

[Har94] Hartmanis, J. Turing award lecture on computational complexity and the nature of computer science. *Communications of the ACM*, 37(10):37–43, 1994.

[Har98] Hartley, S. J. "Alfonse, your Java is ready!". In *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 247–251. ACM Press, 1998.

[HBB01] Hassenzahl, M., Beu, A., and Burmester, M. Engineering joy. *IEEE Software*, 18(1):70–76, January/February 2001.

[HL01] Häfeli, C., and Lamprecht, R. Interactive learning components for the study of finite automata. Diploma thesis, Dept. of Computer Science, ETH Zürich, 2001.

[HN02] Hartmann, W., and Nievergelt, J. Informatik und Bildung zwischen Wandel und Beständigkeit. *Informatik-Spektrum*, 25(6):465–476, December 2002.

[HNR01] Hartmann, W., Nievergelt, J., and Reichert, R. Kara, finite state machines, and the case for programming as part of general education. In *Symposia on Human-Centric Computing Languages and Environments*, pages 135–141. IEEE, September 2001.

[Hvo92] Hvorecky, J. Karel the Robot for PC. In *Proceedings of East-West Conference on Emerging Computer Technologies in Education*, pages 157–160, April 1992.

[HW01] Hoffman, D. M., and Weiss, D. M. *Software Fundamentals: Collected Papers by David L. Parnas.* Addison Wesley, 2001.

[IBM02] IBM. *RoboCode.* http://robocode.alphaworks.ibm.com/, October 2002.

[Kah96] Kahn, K. ToonTalk™ — an animated programming environment for children. *Journal of Visual Languages and Computing*, 7(2):197–217, 1996. See also http://www.toontalk.com/ (October 2002).

[Knu74] Knuth, D. E. Computer science and its relation to mathematics. *The American Mathematical Monthly*, 81(4):323–343, April 1974.

[KR01] Klling, M., and Rosenberg, J. Guidelines for teaching object orientation with Java. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33–36, 2001. See also http://www.bluej.org/ (October 2002).

[Kra98] Kramer, R. iContract—the Java Designs by Contract tool. In *Proceedings Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara/CA, USA*, pages 295–307. IEEE Computer Society Press, Los Alamitos, 1998. See also http://www.reliable-systems.com/ (October 2002).

[Kur78] Kurtz, T. E. Basic. In *The first ACM SIGPLAN conference on History of programming languages*, pages 103–118, 1978.

[Lea99] Lea, D. *Concurrent Programming in Java: Design principles and Patterns.* The Java Series. Addison Wesley, 2nd edition, 1999.

[MB90] Marxen, H., and Buntrock, J. Attacking the busy beaver 5. *Bulletin of the EATCS*, (40):271–251, February 1990.

[Mey97] Meyer, B. *Object-Oriented Software Construction.* Prentice Hall PTR, second edition, 1997.

[MPMV94] Miller, P., Pane, J., Meter, G., and Vorthmann, S. R. Evolution of novice programming environments: The structure editors of Carnegie Mellon University. *Interactive Learning Environments*, 4(2):140–158, 1994.

[Mue77] Mueller, H. A one-symbol printing automaton escaping from every labyrinth. *Computing*, 19:95–110, 1977.

[Mue02] Mueller, H. Parkettierung eines Rechtecks mit zwei Karas. Private Communication, June 2002.

[Nie95] Nievergelt, J. Welchen Wert haben theoretische Grundlagen für die Berufspraxis? Gedanken zum Fundament des Informatik-Turms. *Informatik-Spektrum*, 18(6):342–344, December 1995.

[Nie99] Nievergelt, J. "Roboter programmieren" – ein Kinderspiel. Bewegt sich auch etwas in der Allgemeinbildung? *Informatik-Spektrum*, 22(5), October 1999.

[OEC02] OECD. *PISA: The OECD Programme for International Student Assessment*. http://www.pisa.oecd.org/, October 2002.

[OMG02] OMG. *UML Standard*. http://www.omg.org/uml/, October 2002.

[Pap80] Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, NY, 1980.

[Par96] Parnas, D. L. *Teaching and Learning Formal Methods (editors C. N. Dean and M. G. Hinchey)*, chapter Teaching Programming as if it were Engineering, pages 43–55. Academic Press, 1996. Reprinted in [HW01].

[Pat95] Pattis, R. E. *Karel the Robot – A Gentle Introduction to the Art of Programming*. Wiley, New York, second edition, 1995.

[Pat97] Pattis, R. E. Teaching OOP in C++ using an artificial life framework. *ACM SIGCSE Bulletin*, 29(1):39–43, 1997.

[Ras00] Raskin, J. *Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Publishing, 2000.

[Rei99] Reichert, R. Interact with the garbage collector to avoid memory leaks. *JavaWorld (http://www.javaworld.com/)*, 1999.

[Rei02] Reinfelds, J. Teaching of programming with a programmer's theory of programming. In *IFIP Working Group 3.2 Working Conference "Informatics Curricula, Teaching Methods, and Best Practice"*. Kluwer Academic Publishers, July 2002.

[Res90] Resnick, M. Multilogo: A study of children and concurrent programming. *Interactive Learning Environments*, 1(3):153–170, 1990.

[RH02] Roy, P. V., and Haridi, S. Teaching programming broadly and deeply: The kernel language approach. In *IFIP Working Group 3.2 Working Conference "Informatics Curricula, Teaching Methods, and Best Practice"*. Kluwer Academic Publishers, July 2002.

[RNH00] Reichert, R., Nievergelt, J., and Hartmann, W. Ein spielerischer Einstieg in die Programmierung mit Java. *Informatik-Spektrum*, 23(5), October 2000.

[RNH01] Reichert, R., Nievergelt, J., and Hartmann, W. Programming in schools – why, and how? In Pellegrini, C., and Jacquesson, A., editors, *Enseigner l'informatique*, pages 143–152. Georg Editeur Verlag, 2001.

[Sch02] Schlatter, T. Cora – Concurrent Kara. Diploma thesis, Dept. of Computer Science, ETH Zürich, 2002.

[SCS94] Smith, D. C., Cypher, A., and Spohrer, J. KidSim: programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, July 1994. See also http://www.stagecast.com/ (October 2002).

[SDBP98] Stasko, J., Domingue, J., Brown, M. H., and Price, B. A., editors. *Software Visualization: Programming as a Multimedia Experience*. M.I.T. Press, February 1998.

[Sun02] Sun Inc. *JAXB: Java™ Architecture for XML Binding*. http://java.sun.com/xml/jaxb/, October 2002.

[Sut01] Suter, T. Interactive learning components for the study of context free grammers and languages. Diploma thesis, Dept. of Computer Science, ETH Zürich, 2001.

[Tid02] Tidwell, J. *A Pattern Language for Human-Computer Interface Design*. http://www.mit.edu/~jtidwell/common_ground.html, October 2002.

[Tuc91] Tucker, A. B. Computing curricula 1991. *Communications of the ACM*, 34(6):68–84, 1991.

[Tur37] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2(42):230–265, 1936–1937.

[Wir71] Wirth, N. The Programming Language Pascal. *Acta Informatica*, 1:35–63, 1971.

[Wir73] Wirth, N. *Systematic Programming: An Introduction*. Prentice-Hall, 1973.

Curriculum Vitae

Education

1986 – 1993 Gymnasium in Sarnen, OW

1993 Matura, Type B

1993 – 1999 Studies of Computer Science, ETH Zurich

1999 Master's degree in Computer Science (Dipl. Inf. Ing. ETH)

1997 – 1999 Studies of education sciences, ETH Zurich

1999 Didaktischer Ausweis ETH in Computer Science

2000 – 2003 Ph. D. studies in computer science, ETH Zurich
Advisor: Prof. Dr. Jürg Nievergelt
Co-referees: Prof. Dr. Horst Müller, Dr. Werner Hartmann

Awards

2000 Fritz-Kutter award of ETH Zurich for master thesis

2002 European Academic Software Award for Ph. D. project 'KaraToJava'