A design method and computational architecture for generating and evolving building designs

Patrick Hubert Theodoor Janssen

School of Design
The Hong Kong Polytechnic University

A thesis submitted in partial fulfilment
of the requirements for the
Degree of Doctor of Philosophy

October 2004

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

_____ (Signed)

_____ (Name of student)

Abstract

The aim of this thesis is to contribute to the development of a practical evolutionary design approach — incorporating both design methods and software systems — that would allow a design team to evolve designs that they find surprising and challenging. This thesis has developed an overall framework that supports such an evolutionary design approach.

Genetic and evolutionary algorithms and software systems attempt to harness the power displayed by natural evolution. These algorithms and systems have been successfully employed during the design process in a number of different design fields. However, they have been limited to tackling a very narrow range of well-defined engineering problems. Typically, the evolutionary system is used to optimise certain parameters within a predefined parametric design. Due to a number of fundamental problems, the evolutionary approach has had limited success in evolving the overall configuration and organization of complex designs. This thesis investigates and proposes how these problems can be overcome for building design.

The primary problem to be overcome is generating designs that incorporate an appropriate level of *variability*, which is referred to as the *variability problem*. This affects both the performance of the evolutionary system and predictability of the types of designs that are produced. In an ideal system, performance is high and predictability is low. However, due to the variability problem, this is difficult to achieve. If design variability is very restricted, then performance may be high but predictability will also be high. If design variability is very unrestricted, then predictability may be low but performance will also be low. In order to evolve surprising and challenging designs, a system is required that both performs well and evolves unpredictable designs.

The proposed generative evolutionary design framework allows the design team to restrict design variability by specifying the character of designs to be evolved. This approach is based on the notion of a design entity that captures the essential and identifiable character of a family of designs. This design entity is called a *design schema*. The design team encodes the design schema as a set of rules and representations that can be used by the evolutionary system. The system can then be used to evolve designs that embody the encoded character.

The framework consists of two parts: a design method and a computational architecture. The design method consists of two phases: a generalization phase to develop and encode the design schema, and a specialization phase to evolve a specific design by using the encoded schema. In the first phase, the design team develops the schema with a type of design project in mind. However, the specific project does not yet need to be known. In the second phase, the schema is applied to a specific project and designs are evolved and adapted to the context and constraints of the project. One key advantage of this design method

is that the encoded design scheme can be re-applied to many different projects.

Two key requirements for the design method are that it should be conservative and synergetic. It should be conservative in that it should only deviate from existing design methods and processes where absolutely necessary. In practice, many designers follow a design process similar to the schema based process - a personal architectural character is cultivated during a lifetime of work and adapted for particular projects. This makes it easier for design teams to adopt the proposed method. The second key requirement is for a synergetic design method. It should be synergetic in that the contrasting abilities of the design team and the computational system should be exploited in a way that is mutually beneficial. The design team focuses on the creative and subjective task of developing and encoding the design schema, and the computational system is used for the repetitive and objective task of evolving alternative design models.

The second part of the framework is the computational architecture. This architecture specifies a system that can be used to run the evolutionary process. Its two key requirements are scalability and customizability. The architecture should be scalable in that the performance of the evolutionary system should not degrade unacceptably when used to evolve large and complex designs. Scalability is achieved by using a parallel computational model that reduces execution time, in combination with a decentralised control structure that improves the robustness of the system. The architecture should be customisable in that it should allow the design team to change and replace the evolutionary rules and representations. Customizability is achieved by breaking the system down into two parts: a generic core and a set of specialised components. The generic core does not need any modification by the design team and can be reused within any project. The specialised components, on the other hand, have to be specified by the design team. These components include a set of routines that encapsulate the rules and representations that constitute the encoded schema.

The feasibility of the proposed generative evolutionary design framework is supported by a demonstration of the process of encoding the design schema. A design schema is introduced that defines the character of a family of design. A crucial aspect of encoding such a schema is the creation of a set of rules and representations for generating alternative design models with an appropriate level of variability. The demonstration focuses on these generative rules and representations. A generative process is developed that can generate a variety of three-dimensional models of buildings that differ in overall organization and configuration but that share the schema character. This generative process is used to define generative rules and representations that are implemented as a set of Java programs. These programs are then used to generate a population of three-dimensional models of building design, thereby allowing the character and variability of the designs to be verified. The feasibility of the encoding process is successfully demonstrated.

Publications and Conferences

Patrick Janssen, John Frazer and Ming-Xi Tang (2005). Generative Evolutionary Design: A system for generating and evolving three-dimensional building models. Submitted to *The Third International Conference on Innovation in Architecture, Engineering and Construction (AEC 2005).*

Patrick Janssen, John Frazer and Ming-Xi Tang (2005). A design method and computational architecture for generating and evolving building designs. Submitted to *The Tenth Conference on Computer Aided Architectural Design Research in Asia, (CAADRIA 2005).*

Patrick Janssen, John Frazer and Ming-Xi Tang. Evolutionary Design Exploration Systems. In *Proceedings of the World IT in Construction Conference (INCITE2004)*, Langkawi, Malaysia, 18-21 February 2004, pages 79–84.

John Frazer and Patrick Janssen (2003). Generative and Evolutionary Models for Design. In *Communication and Cognition: Organic Aesthetics and Generative Methods in Architectural Design*, 36(3 & 4):187–215.

Patrick Janssen, John Frazer and Ming-Xi Tang (2003). Evolution Aided Architectural Design: An Internet based evolutionary design system. In *E-Activities in Design and Design Education - Proceedings of the 9th EuropIA International Conference (EIA 2003)*, Istanbul Technical University, Turkey, 8–10 October 2003, pages 163–172.

Patrick Janssen, John Frazer and Ming-Xi Tang(2003). Evolution Aided Architectural Design: A method of designing sustainable buildings. In *Sustainable Environment: Quality Urban Living - Proceedings of The Third China Urban Housing Conference*, The Chinese University of Hong Kong, 3–5 July 2003, pages 425–432.

John Frazer, Xiyu Liu, Ming-Xi Tang and Patrick Janssen (2002). Generative and evolutionary techniques for building envelope design. In *Proceedings of the 5th Generative Art Conference (GA2002)*, Politecnico di Milano University, Italy, 11–13 December 2002, pages 3.1–3.15

Patrick Janssen, John Frazer and Ming-Xi Tang (2002). Evolutionary Design Systems and Generative Processes. In *The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies*, March/April 2002, 16(2):119–128.

Patrick Janssen, John Frazer and Ming-Xi Tang (2001). Generative and Evolutionary Processes in Design. In *On Growth and Form: The Engineering of Nature, ACSA East Central Regional Conference*, University

of Waterloo, Canada, October 5–7, 2001.

Patrick Janssen, John Frazer and Ming-Xi Tang (2001). Generating-predicting soup: A conceptual framework for a design environment. In *Proceedings of the Sixth Conference on Computer Aided Architectural Design Research in Asia, (CAADRIA 2001)* University of Sydney, Australia, 19–21 April 2001, pages 137–148.

Patrick Janssen, John Frazer and Ming-Xi Tang (2000). Evolutionary Design Systems: A Conceptual Framework for the Creation of Generative Processes. In *Proceedings of the 5th Design Decision Support Systems in Architecture and Urban Planning (DDSS 2000)*, Nijkerk, The Netherlands, August 22–25, 2000, pages 190–200.

Cristiano Ceccato and Patrick Janssen (2000). GORBI: Autonomous Intelligent Agents Using Distributed Object-Oriented Graphics. In *Proceedings of the 18th Education and Research in Computer Aided Architectural Design in Europe Conference (eCAADe 2000)*, Weimar, Germany, 22–24 June 2000, pages 297–300.

Patrick Janssen (1999). An Embryonic Growth Process for Spatial Morphology. Masters Thesis, Department of Cognitive Science and Intelligent Computing, Westminster University, London, UK.

Acknowledgments

Special thanks go to my supervisor, Professor John Frazer and my co-supervisor Dr. Tang MingXi for their continuous encouragement, support, and guidance.

This Ph.D. was undertaken in the Design Technology Research Centre (DTRC), School of Design, Hong Kong Polytechnic University. As such, it was part of a more general DTRC research initiative. Generative and evolutionary design was an active focus area, with a number of Ph.D. projects being undertaken. I would especially like to thank two fellow Ph.D. students, Chan Kwai Hung and Sun Jian, for their generous input.

The ideas developed in this Ph.D. first germinated in 1995 while I was completing my Diploma in Architecture at the Architectural Association, in John and Julia Frazer's Diploma Unit 11. During this time, I worked closely with Manit Rastogi. These ideas were also further developed while working with Gianni Botsford on a number of commercial projects.

Finally, I would like to thank my wife, Jacqueline Elfick, for the stimulating discussions throughout this research, and for her general support and patience in my endeavour to undertake this project.

Contents

List of Figures

Part I

Overview

Part one consists of an introduction chapter that gives an overview of this thesis. First, the evolutionary design approach is introduced and certain key problems are identified. The main research objectives are then defined and an outline is given of the research proposition. Finally, methodological issues are discussed.

Chapter 1

Introduction

Contents

1.1 Overview of problem

1.1.1 Overview of evolutionary design

The evolutionary process in nature is an extraordinarily impressive design system. Natural designs far exceed human designs in terms of complexity, performance and efficiency. Evolutionary design systems aim to automate a part of the design process by using natural evolution as a model.

Rather than analyzing one or even several design alternatives, evolutionary systems consider whole populations of alternatives. This parallel approach of the evolutionary process has proved to be well suited to design processes that are typically divergent and exploratory. The field of evolutionary design has benefited from a large amount of interest and research (Frazer, 1995b; Bentley, 1999a; Bentley and Corne, 2002).

The aim behind creating such systems is not to duplicate or mimic an existing conventional design process. Rather, the aim is to create an alternative design approach that allows designers to work in ways that were previously not possible (Frazer and Connor, 1979).

Evolution in nature

In nature, the behaviour of individual organisms results in the population as a whole evolving and adapting to the environment. Each individual organism exists simultaneously in two related forms: as a *genotype* and as a *phenotype*. A genotype consists of a set of genes (the organisms DNA), where a gene can be thought of as a piece of information that can trigger a certain developmental process to unfold. The phenotype is the fully developed organism.

The life-cycle of an organism may be broken down into three interrelated and overlapping steps: *reproduction*, *development* and *survival*. Together, these steps result in the continuous cyclical process of life and death that drives the evolution and adaptation of the population as whole. The three evolution steps may be described as follows:

- The reproduction step involves creating new child genotypes from existing parent genotypes. For example, in sexual reproduction new genotypes may be created by crossover and mutation. Crossover interchanges sections of the genes from the parents. Mutations change particular genes as a result of copying errors during the process of creating the new genotype.

- The developmental step involves creating a complete organism — the phenotype — from a new genotype. Under the appropriate environmental conditions, the genes in the genotype will trigger a set of developmental processes, which will result in the gradual growth of an organism. In some cases, this growth process takes place in a highly controlled environment such as the mother's womb. In other cases, the growth process is exposed to the exterior environment, such as a plant growing from a seed.

- The survival step involves large populations of organisms competing — either individually or in groups — for limited resources in the environment. Organisms that are successful in this competition for resources tend to survive longer, and those that survive longer are likely to reproduce more often than organisms that die early. As a result, their genes become more common in the population. Survival is commonly characterised as a process of *natural selection*, whereby the environment kills certain organisms, thereby 'selecting' the remaining organisms for reproduction.

For each individual organism, these steps occur more or less sequentially: first an organism is conceived; then the organism grows from a seed or egg into the fully developed organism; and finally, if it survives, it may have a chance of reproducing.

Evolutionary algorithms

Universal Darwinism (Dawkins, 1983) suggests that the process of evolution can emerge regardless of the medium, be it biological, compu-

tational, cognitive, or through some other form. Within the computational medium, evolutionary algorithms have been highly successful in a many domains and for a wide variety of problem types. Mitchell (1999, p. 15-16) lists optimization, automatic programming, machine learning, economics, immune systems, ecology, population genetics, evolution and learning, and social systems as domains where evolutionary algorithms have been successfully applied. Beasley (2000) gives an overview of their application in planning, design, simulation and identification, control and classification.

Evolutionary algorithms are in some way analogous to the process of natural evolution. Such algorithms create a cyclical process in which a population of individuals is continuously manipulated to ensure that the population gradually evolves and adapts as a whole. Each individual in the population may represent a variety of entities, including a set of parameters in an equation, a solution to a problem, or a design that fulfils certain requirements.

These algorithms tend to include representations and steps that mirror (in highly simplified form) their natural counterparts. The genotype representation is a highly encoded version of the entity being evolved and the phenotype representation is a decoded version of the genotype. The reproduction step creates new genotypes; the development step transforms genotypes into phenotypes; and the survival step allows some phenotypes to survive.

Each evolution step requires a set of rules and representations. The representations specify a data-structure to be used to represent a particular aspect of an individual, such as the genotype or phenotype. The rules specify how one representation may be transformed into another representation. For example, the reproduction rules may specify how to transform two parent genotypes into one child genotype, and developmental rules may specify how to transform a genotype into a phenotype.

Evolutionary algorithm: An algorithm loosely based on the neo-Darwinian model of evolution through natural selection. A population of individuals is maintained and an iterative process applies a number of evolution steps that create, transform, and delete individuals in the population. Individuals are rated for their effectiveness, and on the basis of these evaluations, new individuals are created using 'genetic operators' such as crossover and mutation. The process is continued through a number of generations with the aim of improving the population as a whole.

Evolutionary algorithms differ from natural evolution — and from one another — in the type of evolution steps that they use. Evolutionary algorithms have an additional step: they use an evaluation process to assess the performance of each individual in the population. In nature, the process of evaluation is implicitly part of the survival step. The

evaluation of an organism remains positive as long as the organism does not die. With evolutionary algorithms, evaluation must be performed explicitly. This evaluation must ascertain how well the individual performs with respect to one or more objectives. For each objective, an evaluation score is calculated and stored as part of the individual. Another additional step that is commonly used is the selection step. These steps will be discussed in more detail in chapter 4.

Evolutionary algorithms differ significantly from the natural model in numerous other ways. Two key differences are the *evolution mode* and the *control structure*:

- The *evolution mode* refers to how the evolution steps process individuals in the population. In nature, the evolution steps are applied in parallel. At any moment in time, some organisms may be in the process of being born, others may be in the process of living, and yet others may be in the process of dying. This natural evolutionary process may be described as an *asynchronous* evolution mode, in that the life-cycles of the individuals in the population are not synchronised. Most evolutionary algorithms use a *synchronous* evolution mode. In this case, individuals in the population are processes in a synchronised manner, with each evolution step being applied to the whole population in turn. The synchronous application of all the evolution steps to the individuals in the population is described as a *generation*.

- The *control structure* refers to how the evolution steps are controlled. In nature, the evolutionary process is an emergent phenomenon that arises as a result of the behaviour of individual organisms. The evolution steps are applied locally and independently from one another. This is referred to as a *decentralised* control structure. Most evolutionary algorithms uses a *centralised* control structure, whereby the application of the evolution steps to individuals in the population is centrally orchestrated.

Evolution mode: The way in which the evolution steps process individuals in the population. The two main evolution modes are the *synchronous* mode and the *asynchronous* mode. With the synchronous mode, the evolutionary process stops and waits for the processing of all individuals by one evolution step to be completed - before proceeding onto the next evolution step. With the asynchronous evolution mode, evolution steps process individuals or small groups in the population as soon as they become available.

Control structure: The way in which evolution steps are controlled. Two main control structures are *centralised* control and the *decentralised* control. With centralised control, a cyclical process invokes and applies evolution steps to individuals in the population. With decentralised control, the evolution steps are autonomous processes that manipulate individuals in the population.

A wide range of evolutionary algorithms exist that use a synchronised evolution mode in combination with a centralised control structure. The four main types are *genetic algorithms* (Holland, 1975; Goldberg, 1989; Jong, 1993; Whitley, 1994), *evolution strategies* (Rechenberg, 1973; Bäck, 1996), *evolutionary programming* (Fogel, 1963, 1995), and *genetic programming* (Koza, 1992). Of these, genetic algorithms are the best known and most widely used. These algorithms will be discussed in more detail in chapter 4 (see section 4.3 on page 91).

Evolutionary design

In the design domain, evolutionary algorithms are used to create populations of alternative designs (Frazer, 1995b; Bentley, 1999d,c, 2000b; Bentley and Corne, 2002; Dasgupta and Michalewicz, 1997). The individuals being manipulated by the algorithm represent possible designs. Each design has a genotype representation and a phenotype representation. The genotype representation encodes information that can be used to create a model of the design, while the phenotype representation is the actual design model. This design model may be evaluated with respect to certain design objectives, which may require information about the environment in which the design is to be implemented and used.

Evolutionary design: A design approach that relies on evolutionary software systems to aid in the process of designing. Such a system employs evolutionary algorithms to evolve whole populations of design alternatives. The software may be used to evolve complete designs or parts of designs.

Two types of evolutionary design may be broadly identified: *parametric evolutionary design*[1] and *generative evolutionary design*. Parametric evolutionary design is usually used late in the design process and focuses on the optimization of design solutions to well-defined design problems. An existing design is defined and parts that require improvement are parameterised[2]. The evolutionary system evolves the parameter values.

[1] Some researchers refer of parametric evolutionary design as *evolutionary design optimization*.

[2] A parameter is a value that is assigned to a variable.

Such systems are generally described as convergent search systems that search the parameter space for an optimal or satisficing set of parameter values. Examples of parametric evolutionary design include (Rasheed, 1998; Rasheed and Davison, 1999; Dasgupta and Michalewicz, 1997; Caldas, 2001, 2002).

Parametric evolutionary design: A design approach that uses an evolutionary system to search for optimal or satisficing design solutions to well defined design problems. The overall design is predefined and those parts thought to require improvement are parameterised. This results in a parametric model into which values for parameters can be inserted to create alternative design solutions. The evolutionary system uses a set of fitness functions or objective functions to evolve an optimal or satisficing set of parameters.

Generative evolutionary design, on the other hand, may be used early on in the design process and focuses on the discovery of surprising or challenging design alternatives for ill-defined design tasks. A generative process is created that uses information in the genotype to generate alternative design models. The evolutionary system will tend to evolve a divergent set of alternative designs, with convergence on a single design often being undesirable or even impossible. Such systems are sometimes described as divergent systems or exploration systems. Examples of generative evolutionary design systems include (Frazer, 1990, 1992; Graham et al., 1993; Frazer, 1995b,c; Frazer et al., 1995b; Bentley, 1996; Rosenman, 1996b,a; Baron et al., 1997, 1999; Coates et al., 1999; Frazer et al., 2000; Funes and Pollack, 1999; Rosenman and Gero, 1999; Sun et al., 1999; Rosenman, 2000; Sun, 2001; von Buelow, 2002; Jackson, 2002).

Generative evolutionary design: A design approach that uses an evolutionary system to evolve surprising or challenging design alternatives, for ill-defined design tasks that embody multiple and conflicting objectives. Some kind of growth process is used to generate design alternatives that vary significantly from one another. The system then relies on either human judgement or evaluation algorithms to evolve a population of design alternatives.

With regard to the types of designs produced, the main difference between these two approaches relates to the variability of designs. With the parametric approach, design variability is low. Since the designs are all based on the same parametric model, the designs will all have the same overall organization and configuration. With the generative approach, the variability in designs can potentially be much greater.

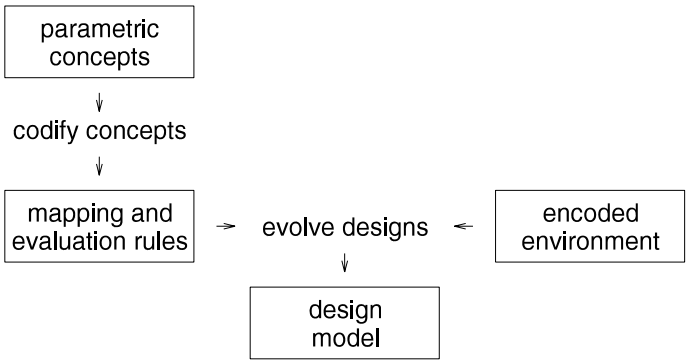

Figure 1.1: General parametric evolutionary method.

The difference in design variability is related to how the developmental step is implemented. With the parametric approach, the developmental step is fairly straightforward and is based on a parametric model of the design. The parameters are encoded in the genotype and a design model is produced by applying the parameter values to the parametric model. This is generally referred to as a *mapping* process. In the case of generative evolutionary design, the developmental step expands a compact genotype into a complex phenotype using a non-linear generative procedure. For example, the genotype may still contain a set of parameter values, but rather than being applied to a static model, they are applied to a set of growth rules that that generate design models. This type of process is referred to as a *generative* process.

For both approaches, a similar design method can be identified that involves two main stages: *codifying concepts* and *evolving designs*. With the parametric approach, the first stages involves codifying a set of mapping rules and a set of evaluation rules based upon the parametric model. With the generative approach, the first stage involves codifying generative rules and evaluation rules based on a set of generative concepts. For both the parametric and generative approaches, the second stage involves evolving alternative designs using on these rules. In addition, the evolutionary process may require information about the design environment, which must be encoded in an appropriate format. In general, this information is usually only used by the evaluation process. Some researchers have also proposed that in the case of the generative evolutionary design, the generative process may develop designs in response to the environment. Figure 1.1 and figure 1.2 on the following page show the main stages of the parametric and generative approaches.

Of these two approaches, the parametric approach is the more common and well developed. However, the generative approach is potentially much more powerful. A number of experimental generative evolutionary systems have been developed in several different design domains. This research will focus on the generative evolutionary design approach.

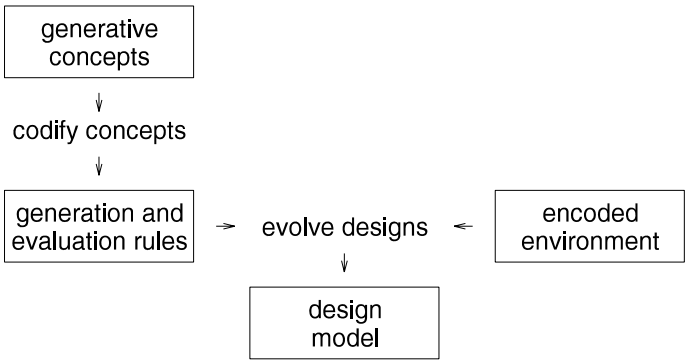

Figure 1.2: General generative evolutionary method.

1.1.2 Problem identification

Architecture is a design domain where the application of evolutionary design could be highly beneficial. In the past, the parametric approach has been successfully used to fine-tune a single aspect of a design at a detailed design stage. However, being able to evolve the overall design of a building early on in the design process would result in a much wider range of potential benefits. For this to be possible, the design models being evolved would need to vary significantly from one another, and as a result the generative approach would need to be used.

Primary problem

A number of experimental generative evolutionary design systems have been developed and these systems have had some success. These systems use a variety of generative techniques to produce designs, such as *shape grammars*, *L-systems*, and *cellular automata*. These techniques will be discussed in more detail in chapter 3.

Examples of generative evolutionary design systems include the following:

- Baron et al. (1997, 1999) have explored systems that evolve forms consisting of aggregations of elementary particles, called *voxels*. A grid of voxels is defined and the genotype then encodes that state of each voxel in the grid.

- Rosenman (1996b,a, 2000); Rosenman and Gero (1999) have developed a number of systems for evolving two-dimensional orthogonal plans for buildings. Plans are generated using a set of shape grammar growth rules that make small modifications to an existing plan. The genotype encodes the selection and application of rules.

- von Buelow (2002) have developed an evolutionary system for evolving structural trusses. The topology of the truss is defined by a connectivity matrix, and the geometry is define by the positions

of the joints. The genotype encodes both the topology matrix and the joint positions.

- Shea (1997, 2001, 2004) has developed as system for optimising space frame structures. (The system developed by Shea does not actually use an evolutionary algorithm. Instead, an optimization technique called *simulated annealing* is used.) The structures are generated using a set of shape grammar growth rules that add, replace and modify structural members.

- Coates et al. (1999); Jackson (2002) have experimented with evolutionary systems that use L-Systems to generate forms. The L-System is used to generate both two-dimensional and three-dimensional forms. The genotype encodes the rules used by the L-System.

- Bentley (1996, 1999b) has developed an evolutionary system for evolving solid object designs. Objects are represented as a set of non-overlapping solid primitives. The genotype encodes the position and shape of these primitives. The system uses a set of evaluation routines to evaluate designs.

- Frazer (1992); Graham et al. (1993); Frazer (1995b,c); Frazer et al. (1995b) have developed a variety of evolutionary systems that evolve three-dimensional forms. Forms are generated using cellular automata rules. The genotype encodes the cellular automata rules.

Many of these systems are capable of generating complex forms that vary in overall organization and configuration. However, none of these system are capable of evolving three-dimensional forms that resemble buildings.

The fundamental problem with the forms generated by these systems relates to design variability. The generative and evolutionary process is not restricted and constrained in a way that ensures that building designs are produced. So far, it has been emphasised that the variability cannot be overly restricted. But, equally important is that variability should not be completely unrestricted. When the output is highly unrestricted, the process of evaluating the models may become complex, or even impossible.

The evaluation step must perform a relative assessment of the design models in the population at any one time. This assumes that all design models can be meaningfully compared to one another. When the variability of design models is highly unrestricted, three main problems related to evaluation can be identified:

- First, when variability is unrestricted, the majority of models generated will be chaotic forms that cannot be interpreted, either by

the designer or the computer, as a design [3]. One option is for the evolutionary system to identify the models that can be interpreted as designs and to discard any chaotic models. This becomes a major task in itself. In many cases, the proportion of actual designs in the background noise of chaotic forms is so small that it becomes impossible for the evolutionary process to take hold.

- Second, the comparison of designs employing different architectural concepts and languages requires subjective judgements to be made. Whether one design is better than another becomes a matter of personal taste, and as a result such judgements cannot be performed by the computer. One option is to allow the designer to interact with the evolutionary system so that they may selectively 'kill' any designs that do not reflect their personal design ideas and beliefs. This will become an onerous task for the designer since only a small proportion of the designs will reflect their ideas and beliefs.

- Third, the analysis and simulation of designs that vary in unrestricted ways is problematic. Typically, analysis and simulation programs require designs to be specified as complex representations that use high-level semantic concepts to describe a design. For example, representations typically include spaces, walls and floors organised in precise arrangements. Conversely, generative programs that generate unrestricted variability describe designs using basic low-level geometric primitives. The task of inferring high-level semantic constructs from low-level geometric primitives is complex, if not impossible.

Despite these problems, some researchers have actively pursued the development of evolutionary systems capable of evolving designs that vary in highly unrestricted ways. The main motivation behind this highly generic approach is the desire to avoid restrictive processes that may exclude the best designs. For example, concerning generative evolutionary design systems, Bentley (1999b, p. 42) writes: "phenotype representations are typically quite general, capable of representing vast numbers of alternative morphologies (this is in contrast to representations for optimization, which can only define variations of a single form)." He later states that this approach "overcomes potential limitations of 'conventional wisdom' and 'design fixation' by evolving forms without the use of knowledge of existing designs or design components." Although this argument cannot be ignored, the problems identified earlier in relation to the evaluation step must also be considered. Developing a highly generic

[3]The distinction between forms that are designs, and those that are not — between *designs* and *chaotic forms* — is not based on the quality of designs, but is instead a much coarser distinction between those forms that may be understood to be a design — whether good or bad — and those that, due to their random and chaotic nature, simply defy any kind of understanding.

representation that does not exclude the best designs is of little use, if it results in the evolutionary process being severely hindered.

The main problem is finding a compromise between an evolutionary design system that is overly restrictive but performs well, and one that is highly unrestricted but performs poorly. This problem is referred to as the *variability problem*.

Other related problems

The variability problem suggest that the variability of designs should be restricted in some way. This results in an evolutionary system that is limited to evolving certain types of designs. This restriction leads to a second, related problem, which has been called the *style problem* (Bentley, 1999b). The style problem relates to the re-usability of the system. In particular, when the variability of designs is restricted in some way, this may result in all designs having a particular 'style', which in turn will result in an evolutionary system with low re-usability since only a small number of designers are likely to approve of the style.

If re-usability is low, each design team will be required to develop their own evolutionary system that incorporates restrictions on design variability that they approve of. This situation is clearly undesirable because most design teams do not have the resources, including time and expertise, to develop such systems. An approach must be found that both maximises the re-usability of the evolutionary system and allows the variability of designs to be restricted.

1.2 Overview of research

1.2.1 Research objectives

The overall goal of this research is to contribute to the development of a practical generative evolutionary design approach that would allow the design team to evolve the overall configuration and organization of buildings.

Primary research objective

In order to achieve the overall goal set out above, the primary objective is to develop a framework for the application of the evolutionary design approach that allows the variability problem to be overcome. This framework is referred to as the *generative evolutionary design framework*.

The framework consists of two parts: a design method and a computational architecture.

- The design method broadly defines a design procedure for using generative evolutionary design. One of the tasks defined by the design method is the task of evolving alternative designs. (Other tasks will be discussed below.)

- The computational architecture specifies the structure and organization of the software and hardware components for a generative evolutionary design system. Such a system would be used for the task of evolving alternative designs.

Design method: A semi-formalised design process that explicitly prescribes a way of designing a type of product. The process is structured as a set of tasks to be carried out by the designer or design team, possibly in some specific order. A design method is a conjecture of a potentially useful design process. It is useful to the extent that its application will lead to products that embody certain design qualities that are seen to be beneficial or desirable.

Computational architecture: An implementation plan of how significant software and hardware components of a computer system are structured and organised. This includes the functions and interactions of different components. Communication protocols and data formats may also be defined.

Other related objectives

In order for the generative evolutionary design framework to be effective, both the design method and the computational architecture need to fulfil certain key requirements. The design method should fulfil two requirements:

- The design method should be *conservative* in that it should, wherever possible, conform to existing design processes used by designers in practice. It should only deviate from existing design processes when it is essential to the success of the evolutionary approach. This conservative approach minimises the changes that are necessary on the part of the design team, and thereby renders it more acceptable.

- The design method should be *synergetic* in that the contrasting abilities of the design team and the computational system should be exploited in a way that is mutually beneficial. Through correlated action, the human designers and the computational system should be able to achieve results that would otherwise not be possible if they were applied in isolation. Such a collaboration requires a design method where the strengths and weaknesses of the human designers complement the strengths and weaknesses of the computational system.

The computational architecture should also fulfil two requirements:

- The architecture should be *scalable*, allowing for the evolution of large complex designs without performance being adversely affected. In most cases, the developmental and evaluation steps are likely to be the most computationally demanding. In order to avoid these steps becoming a bottleneck, some form of parallelization will need to be considered.
- The architecture should be *customisable*. Since generative evolutionary design is a relatively new research field, researchers are experimenting widely. Different researchers are likely to implement the evolution steps in a number of different ways, and the architecture should make the customization of the rules and representations used by these steps as easy as possible.

1.2.2 Research proposition

The core concept upon which the generative evolutionary design framework is based is the notion of a design entity that captures the essential and identifiable character of a family of designs. This conceptualization is defined as a *design schema*.

Design schema

The design schema focuses on a designers body of work, or their oeuvre. An intrinsic aspect of a design schema is that it is specific to one designer or design team. Related to this, is the idea that they embody a formative potential. Design schemas are seen as synthetic rather than analytic and may be used to generate a range of designs that all embody the character of the schema. Designs that embody this character are described as being *members* of the schema.

> **Design schema:** A design conceptualization that captures the essential and identifiable character of a varied family of designs by one designer or design team. It encompasses those characteristics common to all members of the family, possibly including issues of aesthetics, space, structure, materials and construction. Although members of the family of designs share these characteristics, they may differ considerably from one another in overall organization and configuration. Design schemas are seen as formative design generators; their intention is synthetic rather than analytic.

When a design schema is codified in a form that can be used by an evolutionary system, it provides a way of overcoming the variability problem. The encoded schema allows designs to be generated that differ widely in overall organization and configuration while at the same time

precluding chaotic designs. The task of codifying the design schema involves creating a set of rules and representations that define the evolution steps. Each evolution step is characterised as a process that transforms input data into output data. The rules define the transformations and the representations define the formats for the input and output data.

Design environment and niche environment

The environment for a design is defined as encompassing both design constraints and design context. Examples of design constraints may include the budget, the number of spaces, floor areas, performance targets and so forth. The design context may include site dimensions, site orientation, neighbouring structure, seasonal weather variations, and so forth.

> **Design environment:** The constraints and context for a particular design. The constraints describe the requirements that the building must fulfil and may include factors such as budget, spatial requirements, and performance targets. The context describes the building site and may include site conditions, neighbouring conditions and weather conditions. The design environment covers all those conditions that influence the success or failure of the design, but that are not part of the design itself.

The design schema is not specific to one design. When a design schema is created, the actual design environment may not yet be known. However, the design schema cannot be created devoid of any reference to the environment. Instead, the design schema may be developed with a certain *type* of environment in mind, referred to as the *niche environment.*

> **Niche environment:** A type or category of design environment, encompassing a range of possible constraints and a range of possible contexts. If a specific design environment falls in such a environmental niche set, this design environment is described as *matching* or *falling within* the environmental niche.

The design schema concept therefore entails a distinction between two different types of environment. The design schema is adapted to an niche environment, whereas the design model is adapted to the design environment.

The proposed design method

The design method is the first part of the framework. It breaks the design process down into two sequential phases. In the first phase, a design schema is developed that may be used to evolve designs for a

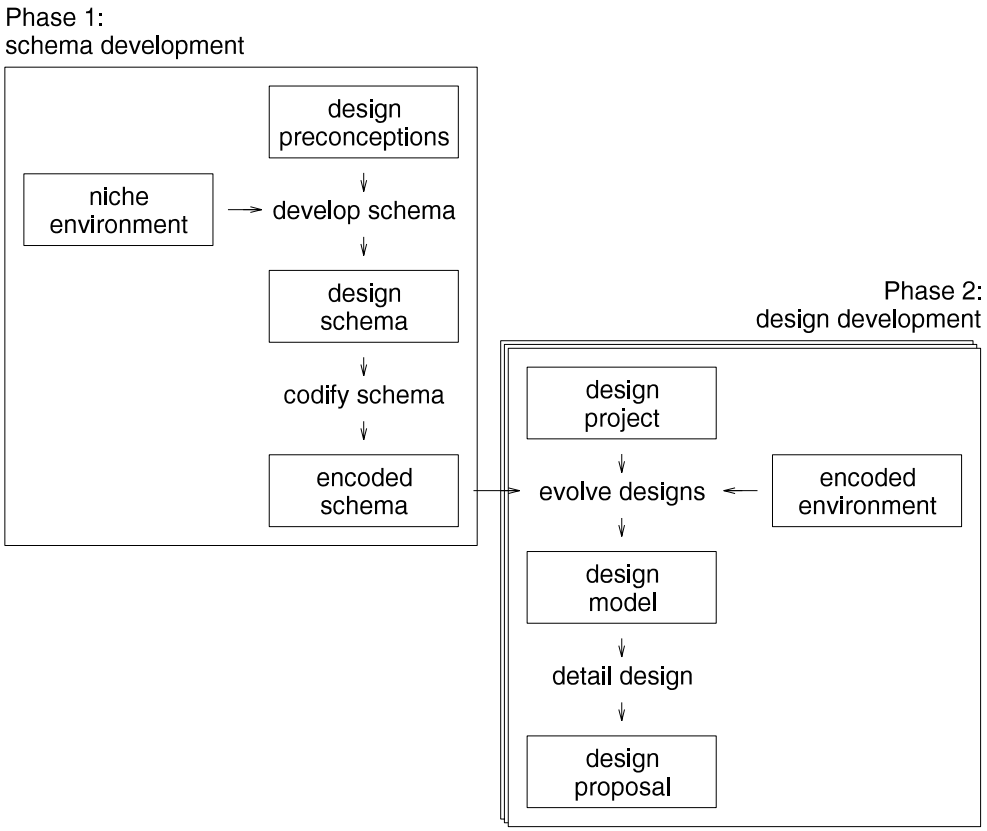

Figure 1.3: The schema-based evolutionary design method.

range of different projects. In the second phase, the schema developed in the first phase is applied to a specific project and a detailed design proposal is developed. Figure 1.3 shows the overall structure of the design method. The first phase may be viewed as a *generalization process*, and the second phase as a *specialization* process.

Each of the two phases is further broken down into two main stages:

- In the *schema development phase*, the design team develops a new design schema that may be used in a range of different projects. The two main stages are creating the design schema and encoding the design schema. The first stage is a form of conceptual design, where the design team develops an integrated set of design ideas adapted to a niche environment. The second stage involves codifying the schema in a format that is compatible with the evolutionary system.

- In the *design development phase*, the design team develops a detailed design for a specific design project. The two main stages are evolving a set of alternative design models and developing one of the designs into a detailed proposal. The first stage requires a generative evolutionary design system for which a computational architecture has been developed. This system uses the encoded

schema to evolve and adapt designs in response to the encoded design environment. The second stage involves developing a detailed design proposal as in a conventional design process.

The schema development phase creates a design schema that is manually adapted to an environmental niche by design team, whereas the design development phase creates a design proposal that is automatically adapted to the design environment by the evolutionary system. The encoded schema can be used in any project whose design environment falls in the niche environment for which the schema was designed.

The design method is conservative in that the overall structure of the design method is similar to a conventional design process commonly used by designers in practice. Numerous studies have shown that designers do not come to a design project with an empty mind but have a set of strongly held preconceptions — including general philosophical beliefs, cultural values, and specific design ideas — that they repeatedly apply in different design projects. Such a design process is seen to share significant similarities with the proposed design method.

The design method is synergetic in that it specifies a process where the design team can focus on those tasks that are predominantly creative and subjective, and where the computational system can be applied to those tasks that are predominantly repetitive and objective. Developing and encoding the design schema is seen as a creative and subjective task, while generating and evaluating alternative designs is seen as primarily a repetitive and objective task.

The proposed computational architecture

The computational architecture is the second part of the framework. The architecture provides an implementation plan for a generative evolutionary design system. Such a system is required in the design evolution stage of the proposed method discussed above. The system uses a set of rules and representations that constitute the encoded schema.

Figure 1.4 on the facing page shows the most significant components of the computational architecture. A single population is manipulated by seven steps: an initialization and a termination step, four evolution steps and a visualization step. The initialization and termination steps are used to initialise and terminate the evolutionary process. The four evolution steps consist of reproduction step, a development, an evaluation step and a survival step. Each of these steps extracts a small number of individuals from the population, processes these individuals, and either inserts the resulting individuals back into the population or — in the case of the survival step — deletes a number of individuals in the population. The visualization step allows design models in the population to be visualised.

The architecture specifies a parallel implementation using a standard client-server model in a networked computing environment. The server manages the population of designs and performs the reproduction and

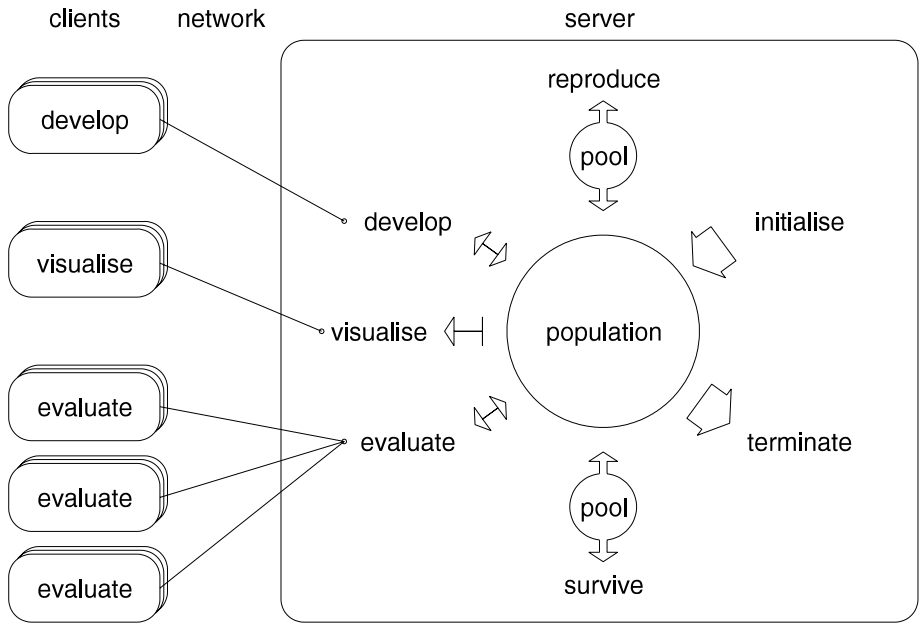

Figure 1.4: Computational architecture for generative evolutionary design system.

survival steps, while multiple client computers perform the developmental and evaluation steps. The architecture supports scalability in two ways:

- First, an asynchronous parallel evolutionary process is used that reduces the execution time and is highly effective in situations where the development and evaluation steps are costly. This is especially pertinent for evaluation clients, since the simulation or analysis of the performance of a building design can be time consuming.
- Second, a decentralised control structure is used with a client-server model that results in a *flexible and robust architecture*. The architecture is flexible in that client computers can easily be added and removed from the evolutionary process. The architecture is robust in that it can cope with failure of client systems in a graceful manner.

In order to support a high level of customizability, the architecture breaks down the evolutionary system into two main parts: the generic core and a set of specialised components. The generic core can be re-used in any design project and does not require any modification. In order to function, the generic core must invoke the services of the specialised components. These components are completely customisable and must be defined by the design team. Three types of specialised components may be defined by the design team: *routines*, *data-files* and *applications*.

- Routines encapsulate the rules and representations used by the evolutionary system. The design team must create a set of such routines that together constitute the encoded schema. The encoded

schema can be changed without requiring any changes to be made to the generic core.

- Data-files encapsulate information about the design environment. These data-files constitute the encoded environment. The data-files can be changed and replaced independently of both the generic core and the routines that define the encoded schema.

- Applications are existing software applications whose functionality the design team may require, in particular for modelling, visualising and evaluating design models. The architecture supports the integration of such applications in the evolutionary system. The networked client-server models allows these applications to be executed by clients running any operating system, as required by the applications.

1.2.3 Significance and potential benefits

The schema based evolutionary design approach has a number of potential benefits over existing design processes. Three potential benefits are highlighted: the first focuses on the ability to evolve designs for buildings that consume less energy; the second focuses on encouraging experimentation and innovation in the design process; the third focuses on allowing the client to gain more control over the design process.

Low-energy architecture

Sustainability has recently become an important issue in many domains, including building design. Sustainable development ensures that the needs of the present are met without compromising the ability of future generations to meet their own needs (WCED, 1990, p. 8). In order to make development in a particular environment (either Earth as a whole or some region of it) sustainable, the impact of humans on that environment must be kept in certain limits (sometimes described as the 'carrying capacity' of that environment). The environmental impact can be described by the following formula (Sylvan and Bennett, 1994, p. 47):

$$\text{Environmental Impact} = \text{Population} \times \text{Consumption} \times \text{Technology}$$

To reduce the environmental impact, the size of the human population must be reduced, the consumption per member of the population must be reduced, or improvements in technology must be made that reduce the impact of consumption.

For the designers of buildings, one of the main ways of reducing the environmental impact is by reducing the consumption element in the formula. In particular, building designers can design buildings that are well adapted to their environment, thereby reducing the energy consumed in their operation and maintenance. This has been highlighted by Maver

and Petric (2003), who write: "Central to the concept of sustainability, at least in the climatic region of northern Europe, is the issue of energy consumption. Leaving aside the energy embodied in the production and transportation of the materials for building, the energy expended in maintaining the building stock at acceptable levels of thermal comfort accounts for more than half of the total energy budget."

Maver and Petric (2003) then asks: "Better design; but how?" This question highlights the fact that adapting a design of a building to its environment is not straightforward. The ability of the human designer to foresee the future consequences of the numerous and interrelated design decisions made during the design process is limited, particularly when considering design decisions made early on in the design process. The complexity of the design task is typically so great that the designer will need to rely on the rule of thumb and their gut feeling when it comes to making early design decisions. The consequences of these decisions will only become clear later in the design process when it is likely to already be too late to explore alternative avenues. Even if further time and resources are available, only a small number of alternatives could ever be feasibly explored.

The schema based design approach uses an evolutionary system rather than human designer to explore these different alternatives. Such a system is inherently parallel in its mode of operation and is able to explore large numbers of possible alternative designs. The evolutionary process will ensure that the population of design models will gradually adapt to the design environment in which they are being generated and evaluated. (This approach has been explored by Caldas (2001); Caldas et al. (2003); Caldas and Norford (2004) for parametric evolutionary design.) By creating designs that are appropriately adapted to the environment, the schema based design approach may help to reduce the consumption per member of the population.

Experimentation and innovation

The design of a building is generally a one-off endeavour. Due to this one off nature of building design, the costs of any project-specific experimentation and innovation must all be borne by a single client. Frazer (1995b) writes: "Construction remains labour-intensive: it has never made the transition to a capital intensive industry with adequate research and development capabilities. It has been left largely to individual architects to take the risk of performing experimental and innovative prototyping in an uncoordinated and romantic or heroic manner. The ensuing (inevitable) failures have been catastrophic for both society and the architectural profession."

The schema based design approach provides an alternative focal point for experimentation and innovation that is not project specific. The design schema can be developed over a large number of projects. The costs of any experimentation and innovation expended on developing

the design schema may be shared between multiple projects. This is particularly relevant to design schemas that predefine a particular type of structural and constructional approach, including defining specific types of components and types of materials. In such cases, experimentation and innovation may involve building and testing numerous prototypes that explore these structural and constructional possibilities.

This approach to experimentation and innovation is similar to the approach of mass-customization of consumer products. Mass customization allows "the same large number of customers can be reached as in mass markets of the industrial economy, and simultaneously they can be treated individually as in the customised markets of pre-industrial economies" (Davis, 1987, p. 169).

Client role

From a client perspective, the design process may sometimes be perceived as somewhat opaque and mysterious. The schema design approach may provide a model of client interaction that empowers the client to experiment with the trade-off's that commonly emerge when developing a design.

This client interaction model focuses on the second phase the design development phase. This phase consists of model evolution and detailed design. During model evolution, the client guidance would centre on complementing the largely quantitative predictions performed by the evolutionary system with qualitative human judgements. During detailed design, the client guidance might centre on making choices concerning details and materials to be used, taking into account the cost implications.

An important aspect of this guidance is that it may require little professional and technical knowledge. The design team together with the technical software consultants would develop the design schemas, the standard details and configure the applications. The client could choose (or purchase) a design schema from their favourite designer and develop a design proposal with limited assistance from the design team. For example, as is common when purchasing a new car, the client might select from a predefined list the preferred types of accessories and interior finishes to be applied to the skeletal model (with cost implications being display back to the client).

During the 1970's, the ABACUS unit at Strathclyde University followed a similar approach, where programs were developed that allowed the end users to create their own designs. In one experiment, teachers were able to use specially developed software to create designs for nursery schools that were within budget and performed well at a technical level (Aish, 1977). Frazer et al. (1980); Frazer (1982, 1995b) has also developed a number of tangible interface system that allow prospective building users to explore different design solutions.

Lawson (1997, p. 292) summarises this approach as "using the com-

puter to de-skill parts of the process to the point where it could be conducted by non-experts. This illustrates one of the generic possibilities offered to us by computer systems. They can sometimes be used to reduce the expertise needed to carry out the task to the level of the novice. In this case the skills needed to draw out a design were reduced to assembling simple shapes on a computer screen allowing nursery school teachers to produce designs which could be compared with those produced by experienced and skilled architects."

1.2.4 Research methodology

In both the natural and cultural sciences, methodological pluralism[4] has led to the development of a vast variety of research methods. The majority of these can be described as descriptive research methods in that they focus on describing the way the world *is*. This research is concerned with contributing to the development of an evolutionary design approach that does not yet exist. This research is inherently prescriptive and as a result, research methods must be used that support this approach.

Many attempts have been made to develop prescriptive research methods[5]. Such research methods are common in design, engineering (Simon, 1981; Cross, 1993; Warfield, 1994; Cross, 2000) and computer science. All these disciplines are predominantly prescriptive. However, for this research, the methods developed in these disciplines are seen to be of limited value, either because they do not directly address the issues involved developing a computational system or because they tend to focus only on the technical implementation aspects of the computational system.

A research area with more appropriate types of research methods is the relatively new field of Information Systems [6]. Despite the fact that design systems might fall outside what is commonly understood to be an Information System, many of the research methods developed in this field are in fact suitable to this investigation.

Research frameworks

Information Systems researchers have proposed a number of research frameworks that include both descriptive and prescriptive research methods (Iivari, 1987; Nunamaker and Chen, 1990; Nunamaker et al., 1991;

[4]Methodological pluralism is the belief that there is not one correct research method, but that there are many different methods for different purposes. (Morgan, 1980; Polkinghorne, 1983; Hirschheim, 1985).

[5]Prescriptive research methods are described by a variety of terms including *systems development*, *artefact building*, and *engineering type research*.

[6]A common definition of the aim of Information Systems research is "the effective design, delivery, use and impact of information technologies in organizations and society" (Keen, 1987). The field is also known by a number of other names such as *Management Information Systems*, *Information Management* and *Informatics*. These names are largely synonymous; see (Davis, 2000)

Iivari, 1991; March and Smith, 1995; Järvinen, 1996; Iivari et al., 1998; Järvinen, 1999, 2000).

Typically, such research frameworks describes a long term research activity that will encompass a variety of individual research projects, each focusing on a particular type of research. Such frameworks identify specific research methods that are applicable during different stages of research, some of which are prescriptive and some of which are descriptive.

The frameworks differ in how the break down the research activity. Generally they will include three key stages (in some cases broken down into smaller stages): *conceptualization*, *implementation* and *evaluation*.

- The conceptualization stage includes the development of user methods, theoretical models and system architectures. Generally, this stage does not encompass implementation although some experiments and simulations may be performed to verify and demonstrate the feasibility of the system. (For example, see *experimentation*, as defined in Nunamaker et al. (1991).)

- The implementation stage includes both the development of prototype 'proof-of-concept' systems as well as the development of fully articulated production systems aimed at the target users. Based on the architecture developed in the previous stage, a detailed specification for the implementation of the system will need to be developed.

- The evaluation stages includes performance testing, case studies, field studies, and so forth. If the system being evaluated is a production system, a realistic evaluation in the social and organizational context for which it was designed may be carried out.

A research project will tend to focus on one stage of the research and will use the methods appropriate to that stage. The evaluation stage tends to use descriptive research methods, whereas the first two stages — conceptualization and implementation — typically rely on prescriptive research methods.

Individual research projects are assumed to link together in complex ways, thereby resulting in the long term research activity that covers the various research stages. The outcome of some projects may require a return to some earlier stage, while in other cases the output may become the foundations for research in the next stage.

For example, one project may develop and implement a particular prototype. If unexpected difficulties and constraints are encountered during implementation, the next research project may need to return to the conceptualization stage to modify the methods, models and architectures. On the other hand if the implementation project was successful, the next project may attempt to test and evaluate this prototype according to a set of benchmarks and criteria. Nunamaker et al. (1991)

write that "development is an evolutionary process. Experiences gained from developing the system usually lead to further development of the system, or even the discovery of a new theory to explain newly observed phenomena."

Only projects that focus on the latter stages are able to realistically evaluate a system. The output from individual projects at earlier stages in the research activity need to be judged in some other way. Various researchers have argued (March and Smith, 1995; Järvinen, 1999, 2000) that the output from any project — including those focusing on the development and implementation stages — should be judged based on its value or utility to a community of users. They suggest that this measure of usefulness should take into account the existing context and environment in which the output is being put to use. An output that is original in some way, is deemed to be research, provided that it has a use that is seen to be of some importance. The research contribution lies in the novelty of the output and in the persuasiveness of the claims that it is effective. Actual performance evaluation is not required at this stage.

Systems development research process

Nunamaker et al. (1991) see systems development as a research strategy that fits comfortably into the category of applied science, belonging to the engineering, developmental and formulative types of research. The development of a method or system is viewed as "a perfectly acceptable piece of evidence (an artefact) in support of a 'proof', where proof is taken to be any convincing argument in support of a worthwhile hypothesis. System development could be thought of as a 'proof-by-demonstration'."

They describe a general research process through which a particular concept will progress. This overall process may encompass a number of individual research projects. Nunamaker et al. (1991) write that "a concept with wide-ranging applicability will go through a research life-cycle of the form: concept - development - impact." Imagination and creativity in the concept stage leads to experimental design and implementation research in the developmental stage, which may eventually lead to research into user productivity and acceptance in the impact stage.

The research process consists of five key research stages:

- The construction of a conceptual framework. Researchers should justify the significance of the research question pursued. An ideal research problem is one that is new, creative, and important in the field. If the framework proposes new methods, techniques or designs, researchers may elect to develop a demonstration that validates the proposed methods, techniques or designs.

- The development of a system architecture. Researchers should put the system components into perspective, specify the system functionalities, and define the structural relationship and dynamic

interactions among system components. The system architecture provides a roadmap for the system building process.

- The analysis and design of the system. Researchers should consider and evaluate alternative approaches to implementation. A detailed specification to be used as a blueprint for the implementation of the system needs to be developed. Such a blueprint would need to determine data-structures, databases and knowledge bases must be determined.

- The building of the (prototype) system. Researchers should demonstrate the feasibility of the design and the usability of the functionalities by implementing a system. The prototype may also be further developed into a product that can be transferred into an organization. The building process can provide insights that may be helpful in redesigning the system.

- The observation and evaluation of the system. Researchers should test the performance and usability of the system, as well as observe its impact on individuals, groups and organizations. The test results should be interpreted and evaluated based on the conceptual framework and the requirements of the system defined in earlier stages.

Figure 1.5 on the next page shows the overall process. A particular research project is likely to focus on one stage, with the output from one research project providing the foundations for the next one.

Scope of this research project

This research project focuses on the first three stages of the research process defined by Nunamaker et al. (1991). For the first stage, a design method is developed; for the second stage, a general system architecture is developed; and for the third stage a detailed system architecture is developed. The stages involving the implementation and evaluation of the system are not included in this research.

The long term goal is to develop a user-friendly system with a comprehensive interface targeted at designers and architects. In order to achieve this goal, the core evolutionary engine must first be implemented, tested and evaluated. Such a process would no doubt result in a variety of modifications and improvements to the core system.

The main outputs developed by this research are as follows:

- The design method for evolutionary design. This method provides a design procedure for the *design team* to use the evolutionary approach.

- The computational architecture for an evolutionary design system. This architecture provides a plan for *researchers* to implement the evolutionary system.

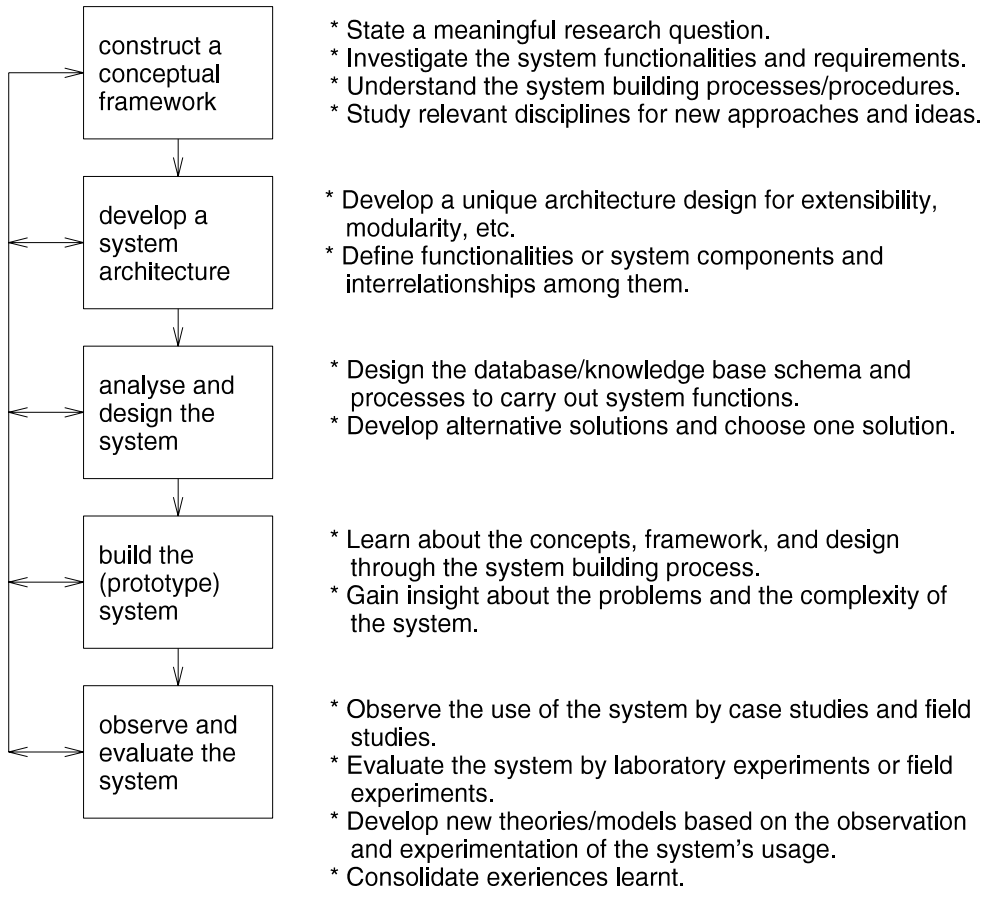

Figure 1.5: Systems development research process, proposed by Nunamaker et al. (1991). (Diagram is redrawn from (Nunamaker et al., 1991).)

Both the design method and the architecture are based on a number of previous models, methods and systems. By comparative analysis, this research highlights various aspects that are novel. In order to judge the value or utility of these novel aspects, different users need to be considered. The eventual target users of an evolutionary system are a design team that includes people with advanced computational skills. The design method is directed at these users. The computational architecture is not directed towards designers, but towards fellow researchers and experimenters in the field of design computation, especially the field of generative design.

The design method and the architecture should be judged based on their value to their respective communities of users. In both cases, this value is supported by a demonstration created to verify the feasibility of design schema concept. This concept is seen as the core concept upon which both the design method and the computational architecture are based.

For the demonstration, an example of a design schema is first described. This design schema defines a family of building designs that share the same overall character. A generative process capable of pro-

ducing designs that embody this character is described. The process of encoding the design schema using routines, data-files and applications is described. Three of the routines — the initialization, developmental and visualization routines — are implemented and a variety of designs are generated.

In addition to the demonstration, various arguments are made in this thesis to support the design method and the computational architecture. The key requirements identified earlier also emphasise value and utility. For the computational architecture, the requirement for customizability ensures that researchers are able to experiment with a wide variety of evolutionary rules and representations, and the scalability requirement ensures that researchers will be able to evolve large complex building forms. For the design method, the conservative requirement relates to minimising the changes the design team need to instigate, while synergetic requirement relates to harmonising the working relationship between the design team and the computational system.

Finally, three broad potential benefits of the evolutionary design approach have been identified: increased levels of adaptation to the environment; increased levels of experimentation and innovation; and increased levels of client participation in the design process. These potential benefits further emphasise the value of the overall approach.

1.3 Overview of thesis

This thesis is divided into four parts. Part one consists of this introduction.

Part two reviews work related to this research, and consists of four chapters: chapter 2 gives an overview of research into the design process; chapter 3 gives an overview of various generative techniques; chapter 4 gives an overview of evolutionary computation; and, chapter 5 describes a number of evolutionary design systems.

Part three presents the main proposition made by this thesis, and consists of three chapters: chapter 6 describes the evolutionary design method developed in this research; chapter 7 describes the computational architecture developed in this thesis; and, chapter 8 demonstrates the process of encoding a design schema.

Part four presents conclusions and future work.

Part II

Review of related work

Part two consists of four chapters that discuss the main areas of research upon which the generative evolutionary design framework is based.

- Chapter 2 discusses the design process, and in particular methods and theories related to this process. The importance and necessity of design preconceptions in the design process are emphasised.
- Chapter 3 introduces a variety of techniques to generate three dimensional form. These techniques may be used to create the rules and representations required by the developmental step.
- Chapter 4 reviews the field of evolutionary computation. The material covered is not specific to the design domain. Two computational architectures are identified, and the most common algorithms are discussed. The rules and representations used by these algorithms are described.
- Chapter 5 describes a number of evolutionary design systems. Five different systems are discussed, of which two are parametric and three are generative.

Chapter 2

Design process

Contents

2.1 Introduction

This chapter introduces a variety of design methods and theories. It consists of four main sections:

- In section 2.2, design methods are introduced. The overall approach and goal of creating such methods is discussed. Although the design methods of the 1960's aimed to provide a single rational and objective procedure for designing, later methods allow for subjectivity and ambiguity.

- In section 2.3, the role of the computer in the design process is discussed. Since the 1960's, computers have gradually become more prominent, but they have also become more subservient to the traditional design process. Recently, a number of researchers have suggested that the computer is once again being used to support more unconventional design processes.

- In section 2.4, the role of the designer in the design process is discussed. Research that emphasises the subjective and ambiguous nature of design is discussed. In order to tackle any moderately complex design task, the designer must introduce personalised and idiosyncratic preconceptions. Two types of preconceptions are discussed: a general design stance and more specific design ideas.

- In section 2.5, research relating to how preconceived design ideas are used in the design process is discussed. A number of different theoretical frameworks are introduced.

2.2 Design methods

The process of designing has been described as an "imaginative jump from present facts to future possibilities" (Page, 1966, quoted in Jones (1970)). Throughout history, design methods have been invented with the intention of rationalising and formalising this jump.

2.2.1 Design methods movement

Designing has often been characterised as a problem solving endeavour where the problem consists of a list of constraints and requirements and the solution fulfils these requirements. This approach culminated in the design methods movement.

The term 'design method' is perhaps most closely associated with attempts, after World War II and during the 1960s, to create a *design science* (Cross, 2001). In particular, the *design methods movement* hoped to create design methods that were based on science, technology and rationalism. The design methods movement received much attention from the design research community. It was part of a more general quest to develop a *design science*.

Cross (2001) writes: "The concern to develop a design science thus led to attempts to formulate *the* design method — a coherent, rationalised method, as the 'scientific method' was supposed to be... So we might conclude that *design science* refers to an explicitly organised, rational, and wholly systematic approach to design; not just the utilization of scientific knowledge of artefacts, but design in some sense as a scientific activity in itself." The quest for a design science was a major motivation in defining design as a problem solving endeavour. In this sense, the quest

for a design science may be thought of as being the broad framework which subsumes the design methods movement.

By framing design as a problem-solving enterprise, it became natural to talk of problem definitions, sub-problems, parameters, variables, optimal solutions, and so forth. The two key proponents of the design methods movement were Christopher Alexander and John Christopher Jones.

First generation design methods

In the 1980 conference entitled *Design : Science : Method*, Broadbent (1981) describes the design method developed by Alexander:

> "Christopher Alexander... had developed a Design Method which consisted of breaking the problem down into small components, sorting these by computer into those which interacted with each other and 'solving' the problems of each group by drawing a diagram, a piece of design geometry by which the conflicts were resolved between the components in each group.
>
> Initially, the 'bits' were 'Misfit Variables' - small statements concerning the problem... He had had further thoughts by 1967 about the nature of the diagrams by which small planning 'conflicts' in the building could be resolved, such diagrams were now called *The Atoms of Environmental Structure*.
>
> The 'Atoms' could be drawn because they resolved simple conflicts of, say, people's 'tendencies' to walk into a building from a certain direction. These, for Alexander, were matters of 'fact', which had nothing to do with feeling or opinion. The designer's task was to 'resolve' any 'conflicts' in such 'tendencies' by his 'Atoms' of geometry" (references omitted).

These early methods, developed during the 1960's, were criticised for being "mechanistic, over-simplistic, joyless, or as denying the essentially creative nature of design" (Jacques, 1981), and soon became discredited. By the early 1970s both Alexander and Jones had abandoned the methods that they had pioneered a decade earlier. Broadbent (1981) identifies the 1967 Symposium on *Design Methods in Architecture* as the key turning point, with the symposium being organised to include specific confrontation between those "representing a mechanistic, quantified view of design" and those "concerned, above all, with the 'human-ness' of human beings".

Second generation design methods

As a result of these criticisms, a second generation (Rittel, 1973) of methods emerged that differed from the earlier methods in two important re-

spects: first, the idea of optimal design solutions was rejected in favour of the notion of satisficing design solutions (introduced by Simon (1981)); and second, the idea of the omnipotent designer was rejected in favour of the notion of user participation. Cross (1993) writes: "The first generation (of the 1960s) was based on the application of systematic, rational, 'scientific' methods. The second generation (of the early 1970s) moved from attempts to optimise and from the omnipotence of the designer (especially for 'wicked problems'), towards recognition of satisfactory or appropriate solution-types... and an 'argumentative', participatory process in which designers are partners with the problem 'owners' (clients, customers, users, the community)."

Despite the shift in approach, the second generation methods still framed design as a problem solving endeavour and remained in the broader design science framework. Although the approach had become less simplistic and deterministic, designing nevertheless remained a question of discovering a satisficing solution to the design problem using rational means. Whereas first generation methods supposed that the omnipotent architect was best placed to make such discoveries, the second generation methods proposed the community as better placed to make such discoveries. Broadbent (1981) writes: "No one was to be an 'expert designer'. Designers had made a mess of things, and if they were to exist in the future, they should be mere technicians, converting to physical form what the *community* said it wanted."

As a result many of the second generation methods were similar to their first generation counterparts. For example, Alexander became disillusioned with the computer (Alexander, 1971) — seeing the precision required by the computer as being incompatible with creativity — and subsequently modified his method to exclude the computer. The modified method was called *A Pattern Language* (Alexander et al., 1977), the core of which consisted of 253 rules, called Patterns. Alexander defines a Pattern as a "three-part rule, which expresses a relation between a certain context, a problem, and a solution" (Alexander et al., 1979). The designer — who might actually be the client or the user rather than the architect — would apply the Patterns to the design problem in a structured way to build up a new design solution. Broadbent (1981) described Alexander's modified method (Alexander et al., 1977, 1979) as follows: "Having been transmuted once into 'Atoms' his 'Misfit Variables' then became 'Patterns' of specific problems. Then they became parts of *A Pattern Language* (1977)... Each Pattern consists of a general statement — very like one of the old Misfit Variables — supported by photographs and text. Most Patterns are summarised in the form of a diagram, very like one of the old Atoms."

2.2.2 Designing as a subjective process

Much of the research into design methods during the 1960s hoped to create a design process that would somehow exclude all forms of subjec-

tivity, thereby disallowing any personal creativity and intuition. The idea was that if the analysis of the 'problem' was sufficiently thorough, the 'solution' would emerge directly from the analysis. Furthermore, a objective procedure was sought that prescribed exactly how such a thorough analysis might be performed.

In the 1970s, subjectivity was once again readmitted into the design process. However, it was the users rather than the designers who were to have a subjective input, and once again with this input 'optimal' designs were to be created. Broadbent (1988) and Lawson (1972) have outlined these developments and highlighted a series of fundamental flaws and weaknesses.

In the 1980s, the rationalistic approach to design was almost completely rejected, with many designers instead focusing on symbolic form. Jonas (1997) describes this as a kind of liberation: "They presented themselves as sort of egocentric artists, which in fact was also a step of liberation and emancipation from the burden of the great but unachievable aims and claims (to work for a better society) in the era of functionalism".

Inherent in the changing attitudes towards design methods over the decades was a split between the rational techniques and more intuitive techniques. Jones (1974) writes: "They all wanted a complete recipe... Many people wanted this and perhaps all students wanted it all the time. But I feel one should resist any such thing if one's to continue living... I found a great split had developed between intuition and rationality, reason."

Today, notions of creating 'optimal' designs are no longer realistic. In *Design in Architecture*, Broadbent (1988, p. 463-464) writes "Gone are the days when architects believed that, given sufficient analysis, they could design the perfectly 'functional' building. They could never, in any case because... there really is no such thing." Throughout his book, he emphasises that it is a mistake to try and establish a once-and-for-all sequence which, when applied, would lead automatically to 'optimum' solutions.

Design methods are therefore understood to describe a much broader category of methods, and may include methods that do not assume that a rational approach to design is preferable. Some methods use random and accidental techniques to generate new designs, with some designers even resorting to 'random drawings' (Cross, 1999). For example, McLachlan and Coyne (2001) discusses the role of accident in the design process, and describes a design method used by Coop Himmelblau: "Himmelb(l)au developed plan forms from a metamorphosed image of their own faces, drawn over and over until the eyes became spaces in the city, and the stripes on their shirts became lines of solid mass that gradually evolved into built forms." This is not to say that the methods developed by the design methods movement are excluded, but rather that these methods are seen as just one of many types.

2.3 The role of the computer

2.3.1 The changing role of the computer

The use of computers in design has a short history; their use was first seriously considered in the 1960's, with the design methods movement. During these early days, researchers has ambitious hopes for the role of the computer might play in design. Computer programs were envisaged that aimed to support design approaches that were new and challenging. Since then, computers have gradually become more prominent in all aspects of life, including design. However, when the mechanistic approach to design became discredited, the computer also seems to have been rejected. Like Alexander, many felt that computers were somehow incompatible with creativity and humanity (Alexander, 1971). In describing Alexander's new state of mind, Broadbent (1981) writes: "The application of *any* method, particularly and computer methods, required a precision of approach which simply destroyed the frame of mind from which creative design could emerge. The act of designing required a tranquillity which simply could not be achieved with the computer."

Although computers have played an increasingly prominent role in design, the nature of this role has changed. Whereas early computer systems supported new ways of working, later systems were totally subservient to the conventional design process. These new systems focused on reimplementing manual tasks in digital form, with minimal disruption to the design process as it existed. The computer no longer had a role in the creative design process and was relegated to the support of manual chores. A typical example is a drafting software that simply recreates the drawing board methodology on a computer.

Recently, the role of the computer in the design process has once again become ambitious. Today, a whole variety of tools and applications exist to support design techniques that would have been inconceivable without computers.

2.3.2 Computers as design support medium

Schmitt (1999) discusses the future of computer-aided design, and writes:

> "Drafting is not the main purpose of computers in architecture any more. The computer is constantly changing its role and appearance, and becoming faster and more powerful in supporting designers every year. It was once useful to compare the computer to an electronic pencil or to a sophisticated typewriter. At the end of the twentieth century, this view would be a dangerous misrepresentation of the machine, as it has moved into new areas of support. It can act as a medium, and is in some cases, already a partner."(Schmitt, 1999)

Schmitt (1999) discusses two distinct roles for the computer: the computer as a *mere tool* (Frazer, 1991) and the computer as a *design support medium*. The computer as a mere tool must "prove itself in eliminating previously human activities with less cost and higher quality." The computer as a design support medium "is an interactive counterpart, not necessarily an intelligent being, but something that has knowledge and capabilities to offer in the area that we are interested in." Whereas the computer as a mere tool tends to emulate existing office instruments, the computer as a design support medium "simulates new design instruments, unthinkable without the computer. These instruments can turn into self-generating and self-referential systems."

As examples of computers as mere tools, Schmitt lists "word processors, when seen as replacing typists; spreadsheets, when seen as replacing calculators; CAD programs, when seen as replacing electronic pencils; office automation systems, when seen as a collection of desktop activities; rendering programs, when only seen as a way to impress clients." For computers as a medium, Schmitt discusses a number of recent technologies under the following headings: 'The Internet as an information source for architects', 'Data Bases - Building Memories', 'Drawing and Modelling', 'Simulation', 'Virtual reality', 'CSCW: a new kind of team work', 'The Virtual Design Studio (VDS): Multiplying Time', 'Engineering Data Management Systems (EDMS)' and 'Facility Management'. In each case, the use of these technologies in the design process is discussed.

Schmitt is highly critical of the role of computers as mere tools in the design process. He argues that to design highly complex artefacts with computer programs that simulate an electronic pencil makes little sense. Instead, he urges designers to make the best use of the powerful communication and simulation technologies that are now available. He writes:

> "If architects want to keep and to improve their role in the building process in the future, they need to employ the computer more effectively. This could include its use as a design support medium that assists designers in areas where they do not have sufficient knowledge or competence themselves. The most obvious application for the computer as a medium is interactive simulation. More advanced applications are computer supported methods and agents." (Schmitt, 1999)

Paradigms of computer-aided design

The idea of a machine as a partner is also proposed by Mitchell (1994). Mitchell (1994) has identified three different paradigms, which he describes as *designing as problem-solving*, *designing as knowledge-based activity*, and *designing as social activity*.

- Designing as problem-solving: This paradigm first emerged in the 1960's with Herbert Simon's book. The problem is formulated by

specifying a domain of possible solutions, a test which can be applied to distinguish acceptable candidate solutions for unacceptable ones, and the resources available for solving the problem. The problem is solved by searching for solutions. Many narrowly specialised design sub-problems can be solved in this way. One practical approach is to divide the labour between the human and the computer, which has led to the typical CAD systems and the use of the computer as a 'mere tool'.

- Designing as a knowledge-based activity: This paradigm emerged in the 1980's. A suitable formalism for the expression of design knowledge was first required, which was used to create knowledge bases that captured what successful designers knew. These knowledge bases were used to solve design problems by applying automated reasoning procedures to the facts of the specific design situation combined with the facts and rules contained in these knowledge bases. One approach for expressing design knowledge was to use predicate logic. Another was to develop shape grammars (see next chapter) that captured knowledge about how various physical components could be combined. Mitchell highlights that the fundamental weakness of such systems is that the knowledge base can never be complete and accurate. (See (Coyne et al., 1990).)

- Designing as a social activity: This is a more recent paradigm. There are multiple agents, some are human and some are software programs. Each agent has its own (not necessarily consistent, comparable, or compatible) knowledge base and problem-solving capabilities, and interacts over the network. These agents proceed by exchanging proposals, arguments and counter-proposals and counter-arguments, and they seek to form consensus. They import knowledge into the common pool, they construct some common intellectual ground, and they sometimes change each other's minds. There are conflicts, ambiguities, and misunderstandings that they must resolve. Key requirements are efficient networks and communication infrastructures.

Mitchell sees the last paradigm as the future of computer-aided design. In this paradigm, the computer is seen as a networked medium consisting of a variety of software agents that can automate labourious or difficult tasks.

2.4 The role of the designer

2.4.1 Problems and solutions

Today, the 'design problem' is widely regarded as ill-defined. (See, for example, Cross (1999) in which the views of a large number of 'expert

designers' are discussed.) An ill-defined problem is one in which the requirements do not provide sufficient information to enable a solution to be found. Such problems require additional information to be discovered, created and invented. Design problems, like most everyday problems, tend to be ill-defined in an extreme way in that the additional information is far greater that the information contained in the stated requirements. The process of discovering, creating and inventing the additional information is an essential part of the design process. Archer (1979) writes: "Some of the necessary further information may be discoverable simply by searching for it, some may be generatable by experiment, some may turn out to be statistically variable, some may be vague or unreliable, some may arise from capricious fortune or transitory preference and some may be actually unknowable. In addition, once known, some of the requirements may turn out to be incompatible with one another."

In 1967 Churchman (1967) related how Professor Horst Rittel proposed that a class of ill formulated complex social systems problems that involved many decision makers, and whose ramifications were thoroughly confusing, might be referred to as *wicked* problems. Design problems have since also become known as a *wicked problems* (Lawson, 1994, p. 2) . Churchman writes "The adjective 'wicked' is supposed to describe the mischievous and even evil quality of these problems, where proposed 'solutions' often turn out to be worse than the symptoms." The wickedness of the problem relates to the fact that the problem includes the task of discovering, creating and inventing the additional information.

Archer (1979) describes the relationship between problem and solution as follows: "The first thing to realise is that 'the problem' in a design problem, like any other ill-defined problem, is not that statement of requirements. Nor is 'the solution' the means ultimately arrived at to meet those requirements. 'The problem' is obscurity about the requirements, the practicability of envisagable provisions and/or misfit between the requirements and provisions. 'The solution' is a requirement/provision match that contains an acceptably small amount of residual misfit and obscurity."

Some theorists have altogether rejected the idea design is about problem solving . For example, Glanville (1998) writes: "Other aspects (e.g. solving a stated problem), although often understood as crucial, are not, I maintain, central to the study of the design act, no matter how important. Problem solving is its own discipline. I am happy to leave it to those interested." In a footnote he adds: "Some postulate primitive problem solving as a first venture towards design. History is as much a construction as any other account. I do not deny problem- solving and design coincide. But I insist design takes a space of its own."

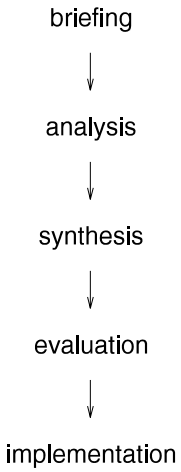

Figure 2.1: The stages of the typical 1960's design process.

2.4.2 Personal and idiosyncratic input

The typical 1960's design process

The typical 1960's design process consisted of five core stages: briefing, analysis, synthesis, evaluation, and implementation. Figure 2.1 show this typical process. Analysis, synthesis, and evaluation were the three core stages that related directly to the process of designing and were generally assumed to be applied multiple times at different scales. For example, Jones (1970, p. 63-69) uses this typical structure[1] to discuss the design process. Jones writes: "These can be described in simple words as 'breaking the problem into pieces', 'putting the pieces together in a new way', and 'testing to discover the consequences of putting the new arrangement into practice"'.

Jones acknowledges that the synthesis stage cannot rely purely on the analysis stage. Jones writes: "This is the stage when judgements and values, as well as technicalities, are combined in decisions that reflect the political, economic and operational realities of the design situation. Out of all this comes the general character, or pattern, of what is being designed, a pattern that is perceived as appropriate but cannot be proved to be right" (Jones, 1970, p. 66).

The reference to 'judgements and values' suggests that some additional kind of input is required from the designer. But it is not clear what kind of input this might be. The methods discussed by Jones indicate that this input focuses primarily on redefining the problem. However, in areas such as architecture, the input from the designer will be much more dramatic and substantial. Most designers will impose a complex set of personal beliefs, values and ideas onto the designs that they create.

[1] Jones (1970, p. 63-69) refers to analysis, synthesis and evaluation as divergence, transformation, and convergence.

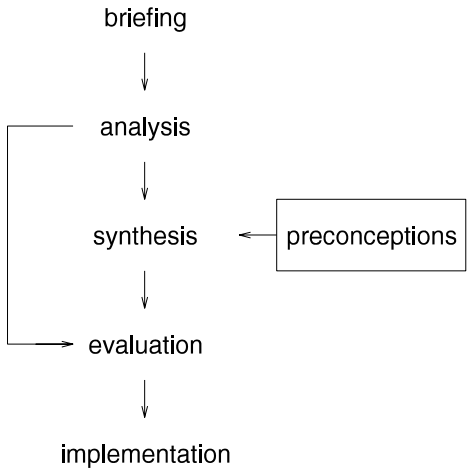

Figure 2.2: Broadbent's adaptation of the typical 1960's design process.

A modified design process

Broadbent (1988, p. 463-466) has developed a design model that includes such inputs as preconceptions. Broadbent writes: "The presumption behind much theorising of the '60s was that *somehow* the Process would generate the form. But it rarely happened like that for the obvious reason that most architects approach most of their designing with certain preconceptions concerning, not just the *partie* of the building type in question but also, specifically, the *style*."

He modifies the typical 1960's design process and inserts 'preconception' as an extra input (at right angles to the flow) into 'synthesis'. Broadbent writes: "Different architects, according to their 'paradigmatic' stance, will, given the same brief and even the same analysis come up with quite a different syntheses: Modern, Post-Modern or whatever... In other words *whatever* kind of analysis has been brought to bear that architect, conditioned by the 'paradigm' in which he works, will bring in *sideways* to the Process the kind of design he wanted to do anyway!" Figure 2.2 shows Broadbent's modified diagram.

The modified diagram would seem to be a more accurate depiction of the design process in fields such as architecture. Nevertheless, Broadbent's description of this process would suggest that the preconceptions are actually more important than any analysis. In such a case, the preconceptions would be the starting point for any design process, and these preconceptions would be adapted to fit the design task, with the design analysis coming in 'sideways'.

This dominance of the preconceptions of the designer is confirmed when a the work of most designer is analysed. In nearly all cases, the same set of beliefs and values may be identified in all the designs, with some designers repeatedly exploring the same specific design ideas. For example, in an interview, Frank Gehry describes his design process as follows:

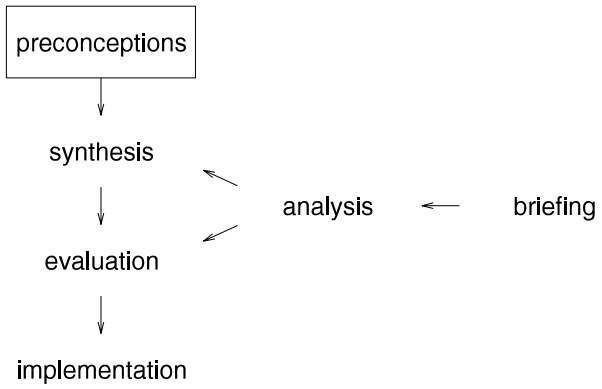

Figure 2.3: Broadbent's design process modified to account for the dominance of the designers preconceptions.

> "I was not conscious that it (the Bilbao Guggenheim) had something to do with what I did before until later because you know, I'm just looking at what I see. I tend to live in the present, and what I see is what I do. And what I do is react. Then I realise that I did it before. I think it is like that because you can't escape your own language. How many things can you really invent in your lifetime. You bring to the table certain things. What's exciting, you tweak them based on the context and the people: Kerns, Juan Ignacio, the Basques, their desire to use culture, to bring the city to the river" (Bruggen, 1997, p. 33).

In order to account for the dominance of the preconceptions, the typical 1960's design process may be further modified. Figure 2.3 shows the modified process.

2.4.3 Design preconceptions

When considering the nature of design preconceptions, two types of preconceptions can be identified: a general design stance and a specific set of design ideas.

- The design stance encompasses the broad beliefs and values that a designer holds. Designer may have the same design stance throughout their lifetime.
- The design ideas are related to the design stance, but are much more specific to particular types of projects and environments.

Design stance

Broadbent (1988, p. 463-466) describes general types of preconceptions as the designer's *paradigmatic stance*. Such a stance may have been

developed over a number of decades and may encompass broad philosophical beliefs, cultural values and perhaps some whimsical tendencies. A client's decision to employ a particular design team is likely to based — at least in part — on the paradigmatic design stance of the design team in question.

Lawson describes this design stance as a set of *guiding principles*.

> "The designer does not approach each design problem afresh with a *tabula rasa*, or a blank mind, as is implied by a considerable amount of the literature on design methods. Rather, designers have their own motivations, reasons for wanting to design, sets of beliefs, values, and attitudes. In particular, designers usually develop quite strong sets of views about the way design in their field should be practiced. This intellectual baggage is brought by the designer into each project, sometimes consciously and at other times rather less so... Whether they represent a collection of disjointed ideas, a coherent philosophy or even a complete theory of design, these ideas can be seen as a set of 'guiding principles'." (Lawson, 1997, p. 162)

Lawson describes how these guiding principles may differ in content from one designer to the next, and also how they are used in different ways. In terms of content, Lawson lists six types of constraints that play an important part in defining the guiding principles of a designer: client constraints, user constraints, practical constraints, radical constraints, formal constraints and symbolic constraints. The attitude of the designer towards these constraints, may to a large extent define their guiding principles. For example, classical architects such as Vitruvius, Renaissance architects such as Palladio and Alberti, and modernist architects such as Le Corbusier all developed a strong set of formal constraints, defined as a set of geometric and proportional rules.

Rowe (1987) has developed a similar concept, that he refers to as a designers *theoretical position*. The theoretical position consists of a set of general arguments and principles that may be either explicit or implicit. In some cases, the theoretical position is well defined, in other cases it is more implicit and vague. Most theoretical positions in some way attempt to address the question: "What is proper architecture?".

Rowe creates an analytical framework to describe and compare different types of theoretical position. The framework frames a theoretical position as a three stage argument that progresses from general statements to specific types of architecture. The three stages are labelled as *orientation*, *architectural devices*, and *production*. These stages are described as follows:

- *Orientation* covers the critical stance and larger purpose of the position.

- *Architectural devices* refers to architectonic elements and leitmotifs (for example, Le Corbusier's five points) that describe the position's production.

- *Production* describe a family of buildings identified by some label ('brutalist', for example).

According to Rowe, a theoretical position can be characterised as an argument that first sets out a broad orientation, this orientation will support certain architectural devices, and these devices will in turn lead to certain types of architectural production.

Rowe discusses a number of examples of theoretical positions. For example, he describes the *functionalist* position as follows (Rowe, 1987, p. 124). The orientation envisages architecture as a matter of efficiently accommodating the requirements in a manner consistent with material composition and construction. "Architecture must be made of the 'right stuff' and in the 'right way' without superfluous ornament or artifice". Under architectural devices, three devices are highlighted: the explicit expression of a building structure and process of fabrication, a spatial organization that grows directly from the program of uses, and a concern with standardization and systematic organization. The production is epitomised by the International Style "with its ubiquitous array of steel, concrete and glass commercial buildings, each consistent in basic format, regardless of location and resulting differences in cultural setting."

Design style

For a particular designer, the design stance is not thought of as a choice, in the sense that they may choose one stance or another. Rather, the stance is inherent in the way that they operate, and often constitutes a set of strongly held beliefs. The term *style* may not be appropriate to describe design stance. Style refers to a recognisable set of features of designs created during a particular historical period, whereas the design stance is a set of principles particular to one designer.

Lawson (1997, p. 163-166) highlights the fact that, although most architects have a clear design stance, rarely do they describe themselves as working in a particular *style*. Lawson writes: "Many architects today regard the style of architecture more as inventions of the critics than as sets of rules that they themselves follow."

In extreme cases, the design stance may become a *moral* stance, with the designer claiming their principles to be a set of universal truths, often related in some way to proportions and harmonies found in nature (Watkin, 1977).

2.5 Design ideas

2.5.1 The dominance of initial design ideas

As well as the overall design stance, designers will generally tackle a particular design task with certain design ideas. These ideas will be compatible with their design stance, but will be much more specific to the design task being considered. They reflect the design stance combined with a 'gut feeling' inspiration on how to approach the design task.

Primary generators

Drake (1979) conducted a series of interviews with British architects about their intentions when designing local authority housing. Through these interviews, Drake highlights how many architects (but not necessarily all architects) latch onto a relatively simple set of related concepts and ideas early on in the design process. Drake refers to this concept as a *primary generator*.

Concerning such primary generators, Drake emphasises three key points:

- They are developed early on in the design process, prior to a detailed analysis of the design problem.
- They are not created by a process of reasoning, but are instead an "article of faith on the part of the architect".
- They provide a framework that defines and directs the overall design approach. In particular, the primary generators structure the problem definition, rather than visa versa.

Drake concludes that early on in the design process, designers "fix on a particular objective, or small group of objectives, usually strongly valued and self imposed, for reasons that rest on their subjective judgement rather than being reached by a process of logic".

Lawson (1997, p. 188) describes a series of protocol studies[2] of design exercises supporting the conclusions reached by Drake (see Eastman (1970) and Agabani (1980)). Lawson (1997, p. 194) also emphasises the importance of these design ideas in the overall design process. In addition, Lawson stresses that the number of primary generators may be small. Lawson (1997, p. 194) writes: "Good design often seems to have only a few major dominating ideas which structure the scheme and around which the minor considerations are organised. Sometimes they can be reduced to only one main idea known to designers by many names but most often called the 'concept' or the 'parti' ".

[2]Protocol studies attempt to understand the personal cognitive design process by studying designers in the act of designing.

Enabling prejudices

Further evidence supporting the idea of the primary generator has also been collected by Rowe, using protocol studies and analysis of written sources. Three case studies of designers in action were analysed, and an attempt was made to reconstruct the sequence of steps, moves and other procedures used. In addition, further examples of the design process, taken from various written sources, were also analysed.

One of the key characteristics to emerge was the dominance of initial design ideas on the rest of the design process. Rowe (1987, p. 31) writes: "Initial design ideas appropriated from outside the immediate context of a specific problem are often highly influential in the making of design proposals. Quite often references are made to objects already in the domain of architecture. On other occasions, however, an analogy is made with objects and organizational concepts that are further afield and outside architecture". Rowe refers to these initial ideas and references as *enabling prejudices*.

Based on the case studies and written sources, Rowe emphasises two key points about such enabling prejudices:

- They are more important than the problem conditions and tend to be the driving force behind the whole design process.
- In many cases, they are not discarded at the end of a project, but instead become long-lasting themes explored through multiple projects.

The first point regarding the dominance of these design ideas is similar to the conclusions reached by Drake and Lawson. Rowe (1987, p. 32) highlights "the tenacity with which designers will cling to major design ideas and themes in the face of insurmountable odds. Often the concept the designer has in mind can only come to fruition if a large number of apparently countervailing conditions can be surmounted". One of the many examples that Rowe discusses is the account by Richard Rogers (as presented in Suckle (1980)) of the tension between the central design idea and the technical requirements for the Centre Pompidou in Paris. The central idea of an 'inside-out building' constructed from a prefabricated kit of parts resulted in a wide range of problems requiring the development of unorthodox methods of design, fabrication and construction.

To illustrate the second point, Rowe discusses the design ideas of John Johanson (once again, from Suckle (1980)). Johanson develops an analogy between architecture and electronic circuitry, with the circuits chassis representing the structural frame, the circuit components representing the functional enclosures, and the circuiting system representing channels for the circulation of people and mechanical systems. Johanson explores and applies this analogy in a number of projects. Thus, Rowe (1987, p. 31) writes: "Sometimes these analogies serve a designer's purpose for more than a single project and thus become incorporated as a central part of that individual's design thinking".

Design ideas as working methods

Frazer (1974); Frazer and Connor (1979); Frazer (2002) describe a general design methodology common to many design fields that develop design ideas through multiple projects.

Frazer conceptualises the design ideas as being embedded in the *working methods* of designers, and that it is these methods that "characterise their 'style' ". Furthermore, Frazer highlights how aspects of these methods are explicitly defined in many offices as standard details, templates, procedures, and so forth. He describes this methodology as both personal because it is particular to one designer, and generic because this designer will use it in multiple projects. Frazer writes:

> "It is common to find sets of standard details in architects' offices that serve to economise in time, ensure details are well tested, but also to ensure a consistency of detailing and to reinforce the house style. In many offices this extends to design procedures, approaches to organization and so forth. The same is true of industrial designers where again stylistic characteristics such as details, colour, preferred materials give economy, consistency, quality control and identifiable house style... The identifying characteristics often go through changes during the development of the designers, sometimes with abrupt changes as with Le Corbusier, but usually a continuous progression can be seen. The stylistic characteristics can continue with an office, studio or company, long after the death of the original designer."

Frazer makes two important points about design ideas:

- They are developed on a long-term basis through multiple projects.
- They are embedded in the practical working methods used by designers, encompassing procedures, tools and data.

2.5.2 Types of initial design ideas

The design ideas of a particular designer must be compatible with their overall design stance. However, beyond this requirement, the development of design ideas may be based on a variety of factors. Rowe (1987) and Lawson (1997) have both developed frameworks that define a number of different types of design idea. Rowe's framework conceptualises design ideas as a type of design heuristic, while Lawson's framework sees design ideas as resolving specific design constraints.

Design ideas as design heuristics

Rowe conceptualises these design ideas in the form of heuristics that will frame problem formulation and guide the search for design solutions.

Rowe (1987, p. 76) defines the term 'heuristic' broadly, highlighting that "the heuristics employed by designers may be quite subjective, having evolved from prior personal experience".

Rowe presents five classes of heuristics:

- *Anthropomorphic analogies* rely on physical actions of the human body (such as moving, sitting, standing, and so on) as a driving force.
- *Literal analogies* rely on existing form-giving configurations (which includes both iconic and canonic analogies, as defined by Broadbent Broadbent (1988)) as a driving force.
- *Environmental relations* relies on the concept of appropriate relations between the environment and the building (including currently significant issues of sustainability).
- *Typologies* rely on past solutions at a variety of scales (including complete building types, organizational templates, and prototype for parts of buildings) as a driving force.
- *Formal languages* are generalizations of other design ideas — in particular typologies and environmental relations — that consist of guiding structures or rules that manipulate formal design elements.

These heuristics constitute different types of analogy and reference that a designer may use to drive the design process forward. They are not mutually exclusive, and tend to be used in combination. In addition, they may be developed through repeated application, or they may be created for one project and subsequently discarded.

Design ideas as resolution of constraints

Researchers have also made a distinction between design ideas that result from specific aspects of the design environment, and design ideas that are more general and that may be applied to a range of different projects. For example, Rowe (1987, p. 2) describes two styles of designing: "Sometimes the unfolding of a design is strongly influenced by constraints derived from the initial setting of the problem, such as the context in which the building is to be built or its social purpose. On other occasions the process seems to be more determined by a designer's personal attitudes and prejudices towards such things as functional expression or modes of fabrication technology."

Lawson (1997, p. 189-202) conceptualises design ideas as constructs that are developed to resolve specific constraints. Such constraints may either be existing in the design problem, or may be self-imposed by the designer. Lawson argues that most designers will be highly selective in deciding which constraints to focus on, and during the early design stages, the number of constraints to be considered is likely to be small.

In order to investigate how initial design ideas are used in practice, Lawson (1997, p. 189-202) analyses the design presentations of three groups of students working on the same problem. Each group of students chose to focus on a different set of constraints — either self imposed or existing — and as a result, each group developed a fundamentally different set of design ideas.

Lawson identifies three main types of constraints:

- The existing constraints specified in the design requirements
- The existing constraints related to the design context, such as the site.
- The self-imposed constraints, embodied in their design stance.

These three types of constraints result in three types of design ideas that are fundamentally different from one another.

2.6 Summary

This chapter has discussed the design process, focusing mainly on methods and theories developed from the 1960's onwards. The main points are as follows:

- The methods and theories of the 1960's and 1970's suggested that the design process should be highly rational and objective, and in many cases it was assumed that one *correct* design method could be defined. Today, many researchers describe the design process as process of negotiation between multiple participants, involving people and systems interacting together and exchanging information and knowledge that tends to be incomplete, inconsistent and incompatible.
- The roles of the computer in the design process has changed. In the 1960's and 1970's, researchers proposed computer systems that supported completely new ways of working. In the 1980's and 1990's, computers in design became much more prominent, but they were generally used to support conventional working methods. More recently, the role of the computer is again becoming more ambitious, with computers being used to support non-conventional design process.
- When designers approach a design task, they have a set of design preconceptions that are personal and idiosyncratic. Designs task are typically ill-defined and ambiguous, and as a result such preconceptions are required to develop a design. Preconceptions are an important and necessary ingredient in the design process. Two types of preconceptions exist: a general design stance that encompasses the philosophical beliefs and cultural values of the designer,

and specific design ideas that are applicable in certain types of design projects.

- A number of researchers have explored the role of design ideas in the design process. Different types of design ideas have been identified. In particular, design ideas may be project specific, or they may be more generic and applicable in a variety of projects.

Chapter 3

Generative techniques

Contents

3.1 Introduction

This chapter consists of three sections, each introducing a different approach to creating programs that generate three-dimensional forms. These approaches may be used within the developmental step of an evolutionary system in order to create alternative design models. The first approach is the *parametric approach*, and is used by parametric evolutionary design systems. The second and third approaches — referred to as the *combinatorial* and *substitution* approaches — are more flexible. Generative

evolutionary design systems may use a combination of any of these three approaches in order to generate a variety of design models.

For each approach, two more specific techniques are identified.

- In section 3.2, the parametric approach is described, which involves generating forms by varying a number of parameters. Two techniques are introduced that differ in how the parameters affect the final form. With the *variational* technique, parameters are assigned to variables associated with a model of the form. Parametric modelling systems use this technique. With the *history based* technique, parameters are assigned to variables associated with a sequential procedure for creating a form.

- In section 3.3 the combinatorial approach is described, which involves generating forms by combining a predefined set elements. Two techniques are introduced that differ in how elements are combined. With the *algebra* technique, a set of element types are defined together with a set of operations for manipulating these elements. This technique is highly flexible, and is implemented within most standard CAD packages. With the *template* technique, an organizational template is defined into which elements can be inserted.

- In section 3.4, the substitution approach is described, which involves generating forms by starting with a seed form and repeatedly substituting parts of this form with new parts. Two techniques are described that differ in how the substitutions are performed. With the *grid based* technique, a grid is defined and substitutions are performed within this grid. Cellular automata are the best known example of this technique. With the *shape based* technique, substitutions are performed based on the geometry of the individual shapes. Fractals, shape grammars and L-systems are well known examples.

3.2 Parametric approach

3.2.1 Overview

Introduction

The parametric approach allows a variety of designs to be generated by varying constraints associated with either a model or a procedure. For example, a model may be created that defines certain dimensions as variable parameters, thereby allowing a variety of forms to be generated by varying those parameters. However, the parametric approach is not limited to varying simple dimensional parameters. The parametric approach is understood to cover what can be found in the literature under other headings such as relational modelling, variational design,

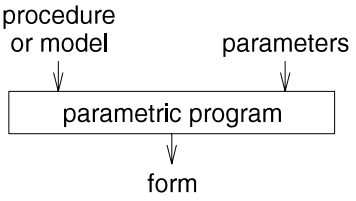

Figure 3.1: Main inputs and outputs for the parametric approach.

constraint-based design, and so forth. In the broader sense, the parametric approach can be thought of as varying constraints, where dimensional parameters are just one type of constraint. Monedero (2000) gives an overview of various types of parametric modelling approaches. He describes a constraint as follows: "A constraint is a relation that limits the behaviour of an entity or group of entities... Parallelism, perpendicularity, tangency, dimensionality are geometric constraints. But a model can also be based on formula like area = force/pressure. Constraints can also be specified as conditional relations...".

Two parametric techniques

Monedero highlights two techniques to parametric modelling: the variational based parametric technique and the history based parametric technique.

- With the variational technique, a model of the form to be generated is first defined. The model incorporates a set of constraints defined as equations. The entire system of constraints for the design is solved simultaneously by a constraint solver. Unlike the history based technique, the variational technique generates a form without making reference to the sequence of modelling operations used to create the form.

- With the history based technique, a form is incrementally constructed by a form generating procedure. This procedure consists of a sequence of operations, with each operation requiring certain data values. The form can subsequently be modified by manipulating either the operations themselves or the data values used in a particular operation.

3.2.2 Variational based parametric technique

Creation of parametric model

With the variational approach, a model of the form to be generated must be predefined. The model may be a topological description of a complex form with a number of associated variables. A *parameter* is a variable to which other variables are related, and these other variables can be obtained by means of *parametric equations*. In order to generate a

form, the generative program requires a system capable of resolving these equations. Models may be irresolvable as a result of being either under constrained or over constrained. An under constrained model cannot be resolved because some additional parameter must still be specified. An over constrained model cannot be resolved because of the existence of some contradiction.

Concerning the variational approach, Monedero writes: "Parametric design based on variational geometry can recompute a design taking into account that actual situation, independently of the sequence that has been followed to reach this situation. The method relies on the description of parameters by means of equations and the availability of a system able to solve them."

Simple example

A simple example is given by Mitchell (1977, p. 40-43), when he models a rectangular room by defining three variables: *d1* represents the length, *d2* represents the width, and *d3* represents the height. One of the vertices of the room is defined as being fixed, and the other seven vertices can be calculated from the three variables. By substituting different values for the variables, different rooms can be generated.

More generally, this example can be described in terms of the degree of freedom (DOF) of the model. Monedero (2000) writes: "An object in a space, defined by the three co-ordinates of their N-vertices will have $3N$ DOF. To compute the new geometry, after any of these vertices have changed, $3N$ equations (with $3N$ variables) will have to be solved."The room has eight vertices and as a result may potentially have 24 DOF. The 24 equations can all be solved using the three variables described by Mitchell — *d1*, *d2*, and *d3*. The room model defined by Mitchell therefore has zero DOF which means that it is neither under-constrained nor over-constrained.

A serious drawback with this technique is that the model must, to a large extent, predefine the topology of the form, The forms generated are therefore very similar. In addition, as the complexity increases, the process of solving the equations becomes computationally expensive.

Yacht hull forms

As well as defining flat surfaces, the vertices can also define control points on curved surfaces. For example, Graham et al. (1993) (see also (Frazer, 1995b, p. 61)) used a variational parametric program in order to generate curved forms that represented yacht hulls. This was part of a larger system that used genetic algorithms in order to optimise the performance of racing yacht hulls. A model of the yacht hull was created in which the fairing of the curves of the hulls profile were defined by a set of control points. The variational parametric program generated alternative yacht hulls by varying these control points.

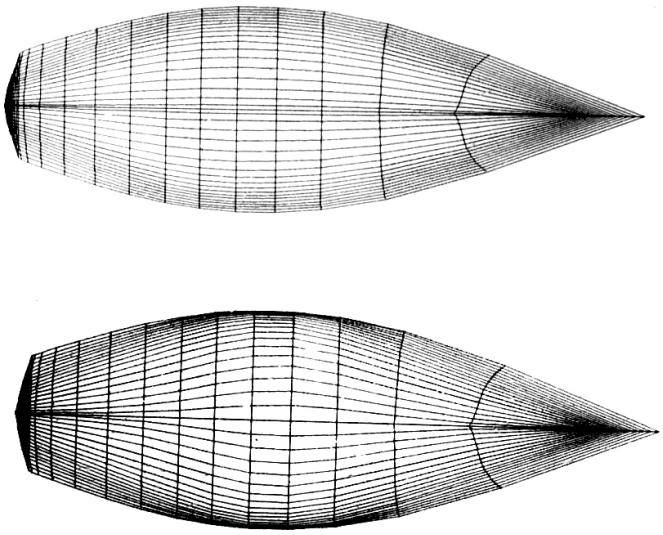

Figure 3.2: Optimization of yacht hull using a genetic algorithm. From Frazer (1995b, p. 61).

The examples of the room and the yacht are fairly simple. The variational parametric approach is capable of generating forms that are much more complex. In particular, this technique is dependent upon the equations being solvable, rather than on the complexity of the model.

Waterloo Station

Variational based parametric modelling techniques were used in the design of the Waterloo railway station in London Szalapaj (2000, p. 135). The arched roof of the train shed follows the curve of the railway, and increases in span down the length of the station. The roof is supported by a series of three-pin arches that change as the roof changes width along the curved tracks. A single parametric model of an arch was modelled, such that it encoded the underlying design rules for the whole family of arches. The complete roof model was then created by instantiating a series of these parametric arches, each with a different value for the span parameter.

Szalapaj (2000, p. 135) writes:

> "The parametric model can be extended from just the description of arches, through to the description of the connections between pairs of arches. This model can in turn be extended to the whole shed form, so that when any dimensional change is made, it is propagated through the whole model. Parametric expressions, therefore, allow users to change the values of key parameters, and to observe the propagation of changes on dependent expressions, and hence upon the de-

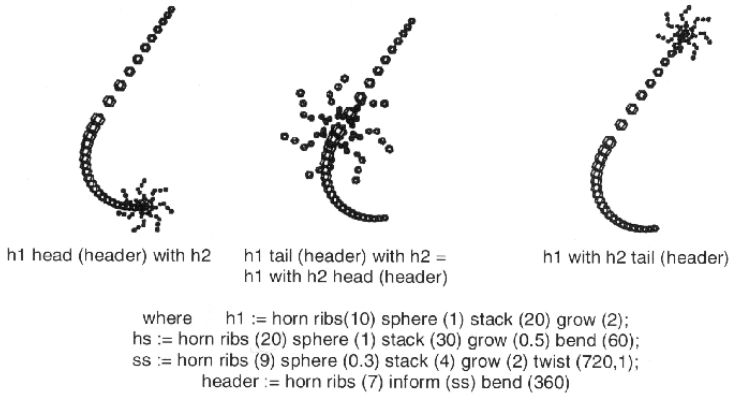

Figure 3.3: An example of a set of rules and the corresponding forms. From Todd and Latham (1999).

> pendent geometry. This is often referred to as *strategic manipulation*."

3.2.3 History based parametric technique

Creation of parametric procedure

Monedero describes the history based parametric technique as follows: "A graphically interactive parametric modeller allows the user to create a master model that can be used as a base to input parameters to the system and to request from the user the specification of constraints that will fix the model through a closed description of its components." In general, a form generating procedure is defined. A variety of forms can be generated by manipulating the constraints associated with this procedure.

The FormGrow program

An example of a history based parametric program is *FormGrow*, developed by Todd and Latham (1992, 1999), which artists can use to generate three-dimensional forms. *FormGrow* builds three-dimensional models of abstract organic forms using a set of growth rules. The basic growth process creates a compound form by duplicating a specified input form. When the input form is duplicated, it is subjected to a series of translations and transformations as specified by a list of rules. For example, the 'stack' rule will stack the input forms on top of each other, while the 'grow' rule will scale the input form each time it is duplicated. The input form can either be a primitive shape such as a sphere, or it can itself be a compound form. A simple script language allows the input forms and the translation and transformation rules to be specified.

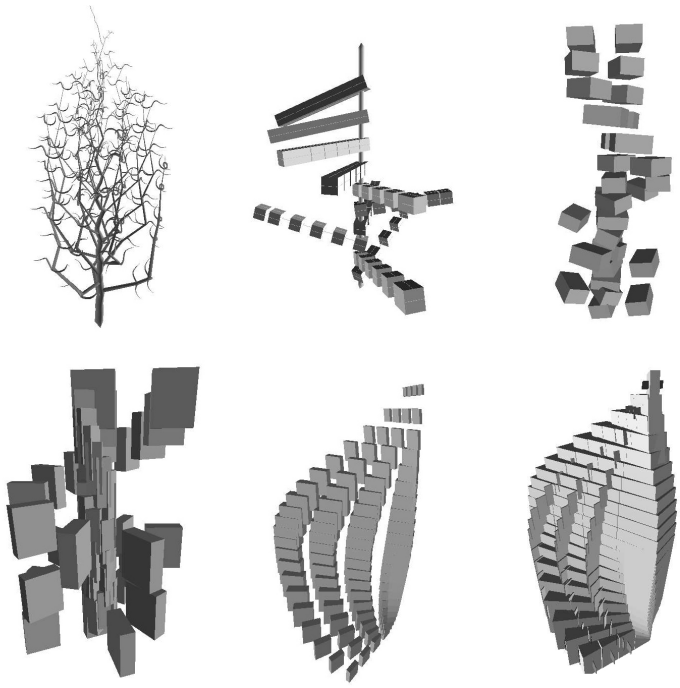

Figure 3.4: Some examples of forms generated using the Xfrog software.

The Xfrog program

Another history based parametric program is Xfrog (Lintermann and Deussen, 1999), a program that allows the easy generation realistic flowers, bushes and trees. More generally, the program can be used to generate a huge variety of hierarchical structures. In this case, a graphical interface allows the user to easily create a set of nodes and links that encapsulate the generative rules used to generate the structure. Lintermann and Deussen write: "The nodes of the graph are components that represent parts of a plant, and the edges denote creation dependencies... components encapsulate data and algorithms for generating plant elements. Generally, three categories exist: one group of components creates graphical objects like stems, twigs, leaves or geometrical primitives; the second multiplies other components; the third applies applies global modelling techniques." An example of the first type of component is a simple primitive like a cone. An example of the second component is the 'PhiBall' component which arranges child components on the surface of a sphere by the golden section algorithm. The parameters include the number of times to duplicate a child, the radius and the opening and closing angle of the spherical section, and the size of the children and their influence on the placement. An example of the third component is the 'Patch' component which specifies free-form deformations by moving control points. These three components can be combined and customised in order to encapsulate a set of rules that can generate a configuration of a few hundred cones.

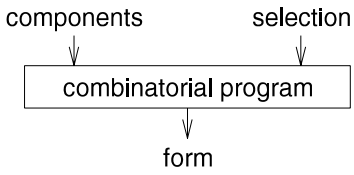

Figure 3.5: Main inputs and outputs for the combinatorial approach.

3.3 Combinatorial approach

3.3.1 Overview

Introduction

The combinatorial approach is perhaps the most general kind of approach for generating form. This approach creates forms by instantiating and assembling elements. The elements tend to be of a number of different types, but in some cases may all be instantiations of one type. For example, a combinatorial program may generate forms by defining a very small and generic type of element sometimes described as a voxel (a three-dimensional pixel). Forms can then be generated by assembling voxels.

Two combinatorial techniques

Two techniques to creating combinatorial programs can be identified: the algebra technique and the template technique. In both cases, the combinatorial program must incorporate two key parts: a set of element types, and a procedure for assembling elements. The algebra technique and the template technique differ in the procedure for assembling elements.

- With the algebra approach, an algebra is defined that, as well as including the element types, also includes a set of operations that position and combine elements in space. The algebra based combinatorial program can use these operations in order to create assemblies of elements.

- With the template technique, an organizational template is defined that consists of a structure into which elements can be inserted. The template imposes a certain organization on the elements. The template based combinatorial program generates forms by inserting alternative elements into the predefined template.

3.3.2 Algebra based combinatorial technique

CAD systems

With the algebra based combinatorial technique, a set of elements types and a set of operators must be defined. The elements and operators constitute what Mitchell (1990, p. 128-129) describes as an *algebra*. An

algebra consists of a set of element types available for instantiation, operators that transform these elements, and operators that combine these elements. For example, with a solid-modelling system, the set of element types consists of parametric boxes, cylinders, spheres, cones and prisms; the transformation operators include translation, rotation, reflection and scaling; and the combination operators are union, intersection and subtraction of solids. A combinatorial program could use this algebra as a basis for generating a wide variety of forms. Such a program might randomly instantiate boxes, and then randomly combine and transform these boxes.

An important aspect of the algebra based combinatorial approach is that the possible assemblies of elements are not restricted in any significant way. An assembly should be able to contain any set of elements in any order. As a result, the assembly process can freely generate assemblies without having to resort to any complex reasoning techniques. When the possible assemblies of elements are restricted to certain configurations, the set of element types, the operators and the restriction rules may be collectively described as a *grammar* (Mitchell, 1990, p. 131-133). A grammar is an algebra with an additional set of rules that describe which combinations of elements are valid and which are invalid. As a result, the set of forms that comply with a grammar are a subset of the set of forms that comply with an algebra. Such grammars are analogous to linguistic grammars. Mitchell writes: "Thus not every string of words in English is a sentence: only strings that comply with the rules of English grammar count as sentences".

Creating combinatorial programs that generate forms that comply with an algebra is fairly straightforward. This is due to the fact that the elements are independent from one another and as a result all possible combinations of elements are valid. However, generating forms that comply with a grammar are more complicated because the rules of the grammar need to be enforced. The template based combinatorial technique may be one way of enforcing these rules. The parametric approach and the substitutional approach, to be discussed later, may also be applicable.

3.3.3 Template based combinatorial technique

Creation of templates

With the template based combinatorial technique, a set of elements types and an organizational template must be defined. The template defines a set of place holders or locations in space into which elements can be inserted. Each location is associated with a list of possible element types that can be inserted into that location. A variety of forms can be generated by inserting elements into all the locations in the template.

The template may be highly generic. For example, the template may consist of some kind of three-dimensional cellular grid into which voxel

elements can be inserted. In such a case, the template is not specific to any particular type of form, but can be used to represent a wide variety of forms. In many cases, the template is highly specific thereby allowing elements of different types to be organised in a predefined manner. For example, the template may define an overall structure that consist of four or five locations into which elements need to be inserted. Instead of allowing all element types to be inserted into all locations, the possibilities may be restricted to ensure that valid combinations are produced.

Mitchell (1977) traces this technique back to 13th century Spanish scholar Ramon Lull, who developed a system for generating combinations of words. According to Mitchell, Lullian ideas exerted an important formative influence on the 17th century philosopher Leibniz, who wrote extensively on methods of invention and design by systematic generation of combinations.

Morphological method

An clear example of such an technique is the *morphological method* developed by Zwicky (1967, 1969) in the field of engineering design. The method is described by Mitchell (1977, p. 34-35) using the example of generating alternative designs for domestic windows. The design of the window is broken into five types of elements: frame, glazing, opening system, privacy system and sun protection system. For each type, a number of alternative elements are given. In order to generate a design for a window, one element for each element type is selected from the predefined lists of alternatives. For instance, one possible window design might consist of a steel frame, single pane glazing, vertical sash opening system, drapes for the privacy system, and exterior louvres for the sun protection system.

An important limitation of this technique is that the choice of element in one location should not affect the choice of element in another location. For instance, another possible window design might be generated by changing the vertical sash opening system to a pivoting opening system. This choice should not affect any of the other choices, although in this example this is clearly not the case. For instance, the pivoting opening system is likely to be obstructed by the exterior louvres.

Dimensional co-ordination

Developments in standardization and modular co-ordination of building components may also be thought of as using the template based combinatorial approach. Such templates are usually based on an orthogonal three-dimensional grid. By standardising the dimensions of building components, these components can be arranged and combined within this grid in a limited number of positions. A highly generalised and simplified version of such an approach was discussed by Bemis (1936), who has shown how building forms could be defined by combining four-inch cubes in an orthogonal grid.

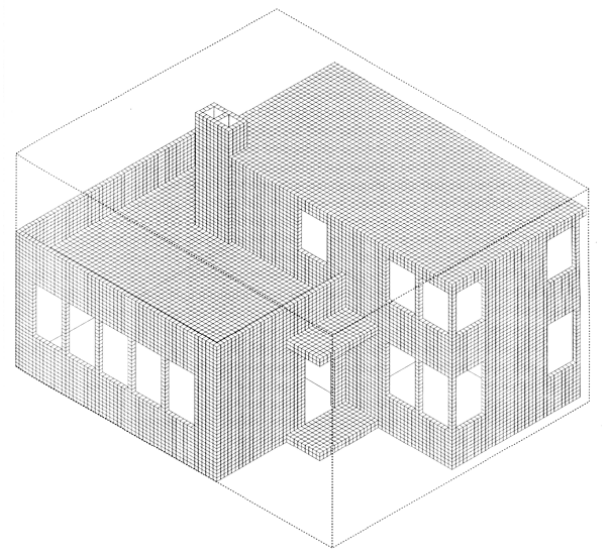

Figure 3.6: A house represented as a collection of four-inch cubes. From Mitchell (1990, p. 41).

3.4 Substitution approach

3.4.1 Overview

Introduction

With these programs, an initial form is provided, and this form is gradually manipulated by a set of rules that modify parts of the form. Time flows in discrete steps, and at every step a new form is created by applying a set of rules that manipulate parts of the current form. These rules are referred to as *transition rules* in that they define the transition from one time step to the next. The initial form is descried as the *seed form*. At each time step, the form to which the rules are to be applied is described as the *current form*. After a certain number of time steps, a *final form* is produced.

The transition rules are a type of *if-then* rule. In general, the rules specify that *if* a certain configuration occurs in the current form, *then* this configuration must be replaced by a new configuration. The *if* part of the rule is called the antecedent, and the *then* part of the rule is called the consequent. The rules specify two configurations: the antecedent configuration that must be found in the current form; and the consequent configuration that will replace the antecedent configuration if it is found. In order to ascertain whether a rule can be 'fired' or not, the current form will need to be searched for the antecedent configuration. If a match is found, the rule is fired. If no match is found, the rule simply does nothing.

The rules are iterative in the sense that they can be applied many times during the process of transforming the seed form into the final form. In many cases, the rules are also recursive in the sense that a

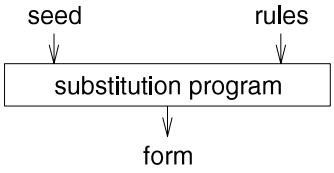

Figure 3.7: Main inputs and outputs for the substitution approach.

rule can be applied to its own output. Recursive rules[1]have a consequent configuration that contains one or more copies of the antecedent configuration.

Two substitution techniques

The process of searching the current form for an antecedent configuration requires each part of the form to be compared to the particular configuration. Exactly how this is done will depend on the type of substitution program. Two common techniques for substitution programs might be described as *grid based substitution technique* and *shape based substitution technique*. These techniques both define processes that generate form by applying transition rules in discrete time steps.

- Grid based substitution techniques predefine a cellular grid within which forms are defined as cellular patterns. The antecedent and consequent configurations are therefore also defined as patterns within the grid.

- Shape based substitution techniques, on the other hand, do not predefine any grid but allow the growth to take place in a continuous space by substituting existing shapes with new shapes.

The substitution approach allows for a number of ways of generating alternative forms. The most obvious approach is to either modify the seed form, transition rules, or the total number of time steps. Changing any of these will result in different forms being generated. Another possibility focuses on which transition rules to apply, with two possible approaches being described as the *indiscriminate approach* and the *selective approach*. The indiscriminate approach synchronously fires every rule that is applicable. For example, at some particular time step, three rules may be applicable to the current form. The indiscriminate approach fires all three rules. This will mean that all three antecedent configurations are deleted from the form and all three consequent configurations are added to the form. An alternative approach would be to selectively

[1]Note that this is different from the distinction between an iterative *process* and a recursive *process*. See Abelson et al. (1985, p. 29), who describe these processes in terms of their 'shapes'. Recursive processes build up a chain of deferred operations. When the end of the chain is reached, the operations are performed in reverse order. Iterative processes repeatedly perform one operation at a time, in sequence.

choose which rules to fire. For example, rather than firing all three rules, just one or two rules might be selected. Some substitution techniques only allow one rule to be fired at every time step.

3.4.2 Grid based substitution technique

Cellular automata

The best known grid based substitution program is the cellular automata. The simplest one-dimensional cellular automata consists of a line of cells of fixed length. Each cell can be in one of two states: on or off; white or black; alive or dead. Such a cellular automata generates patterns rather than forms. The seed pattern of the cellular automata may consist of any configuration of white cells. At every time step the state of all the cells are synchronously updated by indiscriminately firing all transition rules for which a match is found in the current pattern. The cellular automata reaches its final pattern either after a predefined number of time steps or when a desired pattern is produced. With a one-dimensional cellular automata, its behaviour through time can be captured by stacking the rows of cells created at each step on top of each other. In this way, a two-dimensional picture can be created, with time running down the vertical axis.

The transition rules determine the new state of a cell based on its current state and the current state of its immediate neighbours. Each rule defines an antecedent pattern and a consequent pattern. The antecedent pattern consists of the states of the centre cell and its two immediate neighbours to the left and to the right. The consequent pattern consists simply of the new state of the centre cell. So, for example, one rule might state that if the centre cell is black and the cells to left and right of the centre cell are both white, then in the next step the centre cell should remain black.

For this simple one-dimensional cellular automata there are a total of (2^3) eight possible antecedent patterns. As a result, exactly eight transition rules are required. Because each of these eight rules can have only (2^1) two possible consequent patterns, there are a total of (2^8) 256 possible sets of eight rules. [2].

[2]Wolfram (2002, p. 55-56) has studied the global patterns of all 256 possible sets of transition rules for the simple one dimensional automata. Most of these rules lead to global patterns of little interest; in many cases all the cells either become all white or all black, or in other cases a simple pattern persists. Of most interest are a few patterns that include structures that repeat but that also have a certain level of apparent randomness. In these cases, simple rules lead to levels of complexity that were in no way predictable from an analysis of the rules themselves. Based on these and other investigations, Wolfram (1983) developed a classification scheme that described four classes of cellular automata arranged in order of increasing complexity: class one lead to a uniform state; class two produce simple ordered structures that remain the same for ever; class three create patterns that are random in many respects but small scale structures such as triangles are always seen; and class four are somewhere between class two and class three in that they lead to complex patterns with dynamic local

This simple one-dimensional cellular automata can be extended in two key ways: the dimensionality of the grid may be increased or the number of states may be increased.

- The dimensionality of the grid can be increased, theoretically to any number. It should be noted that above three dimensions, it becomes difficult to visualise. As many of these programs rely on visual inspection of the output, the number of dimensions is usually limited to one, two or three. A two-dimensional cellular automata can be created on a square grid so that each cell has eight neighbours. This results in (2^9) 512 possible antecedent patterns, which in turn means that 512 rules are required, and this in turn means that there are an astronomical number (2^{512}) of possible sets of 512 rules. The 512 transition rules need not all be specified individually. Instead, more general rules can be specified.

- The number of possible states may be increased to any number. Rather than allowing for only two states represented as either a white or a black cell, cells can be allowed to take on a range of states. Often, additional states are displayed by using colour. For example, a one-dimensional cellular automata can be created where each cell is in one of three states: white, grey or black. In this case, there are a total of (3^3) 27. As a result, exactly 27 transition rules are required. As there are (3^1) three consequent pattern, this results in a total of (3^{27}) 7.6 trillion possible sets of 27 rules. This highlights another huge increase in complexity that results when the number of states is allowed to increase. When the dimensionality of the grid and the number of states are both increased, the resulting increase in complexity is far greater.

The first cellular automata was proposed by John von Neumann (with key suggestions coming from his friend Stanislaw Ulam) in 1952-3. Von Neumann was trying to develop an abstract model of self-reproduction in biology and he created a cellular configuration — consisting of 200,000 cells in total — that was able to reproduce itself when placed within a cellular automata with 29 possible states and a complicated set of transition rules. During the 1960s progressively simpler cellular automata capable of hosting self-reproducing structures were found[3]. Best known is John Conway, who in 1970 developed a set of transition rules for a two-state two-dimensional cellular automata that he called *The Game of Life* (Gardner, 1970, 1971). The Game of Life rules exhibit a range of complex behaviours and an immense amount of effort was spent finding

structures that interact in complicated ways.

[3]Wolfram (2002, p. 876) highlights an important difference between the approach implicit within von Neumann's work and the approach within later studies: von Neumann sought to produce complex behaviour (in this case, self-reproduction) by creating a complicated underlying system whereas later studies focused on producing complex behaviour by creating simple underlying systems.

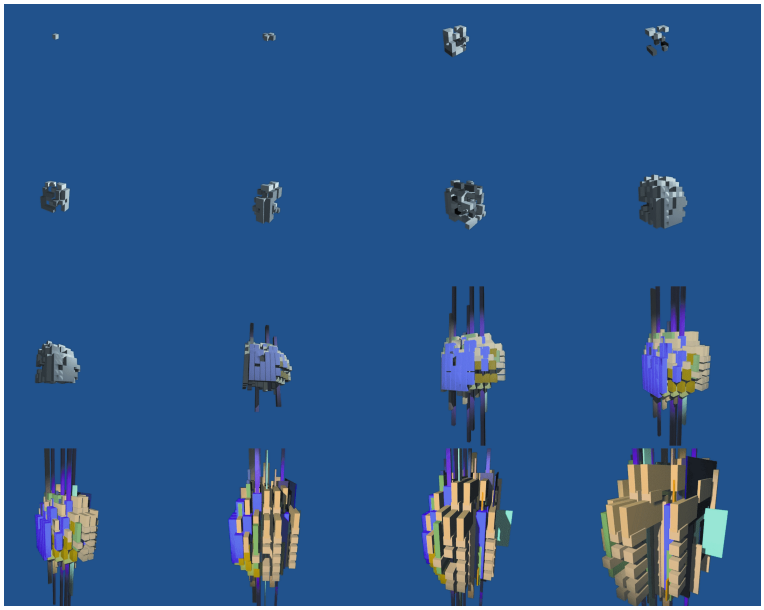

Figure 3.8: Generative sequence by Thomas Quijano and Manit Rastogi. From Frazer (1995b, p. 92-93).

initial conditions that lead to repetitive or other interesting behaviour. Eventually a configuration — described as a *breeder* — was found that was capable of self-reproduction.

Cellular structures

More recently, researchers in John and Julia Frazer's research unit (Frazer, 1995b) have created a variety of three dimensional cellular automata that generate a variety of structures. For example, in 1994, Quijano and Rastogi (Frazer, 1995b, p. 92-93) created a three dimensional cellular automata program on an orthogonal cubic grid. The cellular automata was initialised with a seed form that consisted of a single 'on' cell — a solid cube — in an invisible cubic grid. A set of three-dimensional transition rules were iteratively applied, resulting in a complex structure of three-dimensional cubes being produced. The antecedent pattern for these rules consisted of the state of the centre cell and the states of its six face neighbours; the consequent pattern consisted of the new state of the centre cell. For example, one rule might state that if the centre cell is 'off' and it has just one neighbour directly below it that is 'on', then in the next time step the centre cell should be switched 'on', thereby resulting in a new solid cube being inserted into the grid at that position.

Once a certain level of complexity was reached, the program departed from the restricted cubic formation by stretching and transforming the cubes into a variety of coloured orthogonal volumes. The transformation process of each cube was based on the configuration of its neighbouring cells. Each possible neighbourhood configuration (of which there was a

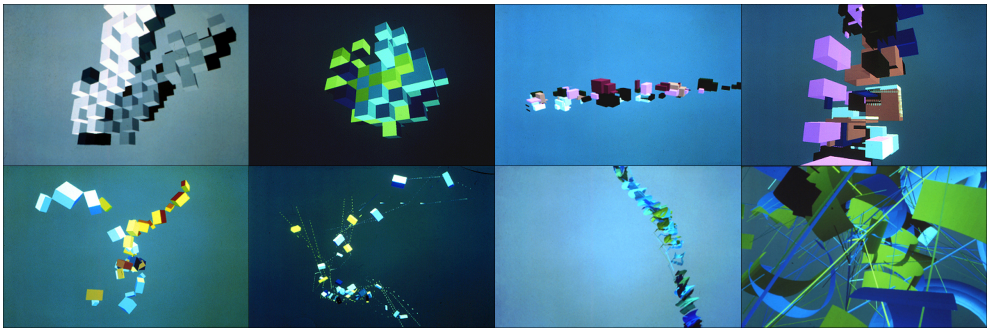

Figure 3.9: Generative sequence by Stefan Seemüller. From Frazer (1995b, p. 46-47).

total of $2^6 = 64$) resulted in one possible transformation. This resulted in an abstract form of overlapping of overlapping coloured volumes.

Other examples of similar generative programs that used three-dimensional cellular automata on a cubic grid include Seemüller (Frazer, 1995b, p. 46-47) and Nagasaka (Frazer, 1995b, p. 50-51).

3.4.3 Shape based substitution technique

Fractals

With the shape based substitution technique, the background space within which forms are generated is treated as continuous canvas rather than as a gridded cellular structure. The best known types of shape based substitution programs are two-dimensional fractals. As with grid based substitution programs, time flows in discrete steps, and at every step all transition rules for which a match is found in the current form are indiscriminately fired. In the case of fractal programs, the antecedent and consequent configurations of the transition rules do not specify gridded patterns. Instead, they consist of two-dimensional arrangements of shapes.

One of the first and best known geometrical fractals was the Koch curve, created by Helge von Koch in 1906. It consisted of only one transition rule that substituted a straight line by four connected lines. By repeatedly applying the same rule, the Koch curve is generated. It was Benoit Mandelbrot who coined the term 'fractal' and who initiated a whole new area of research with the publication in 1975 of his book *The Fractal Geometry of Nature* (Mandelbrot, 1975). The crucial insight that Mandelbrot presented in his book was that nested fractal structures were common throughout nature, as well as in mathematics. Since then, the general idea of the importance of fractals has become well established within science.

An important feature of these types of rules is that they do not specify absolute scale or orientation. The antecedent pattern does not specify the scale and the orientation of the pattern to be matched in the current

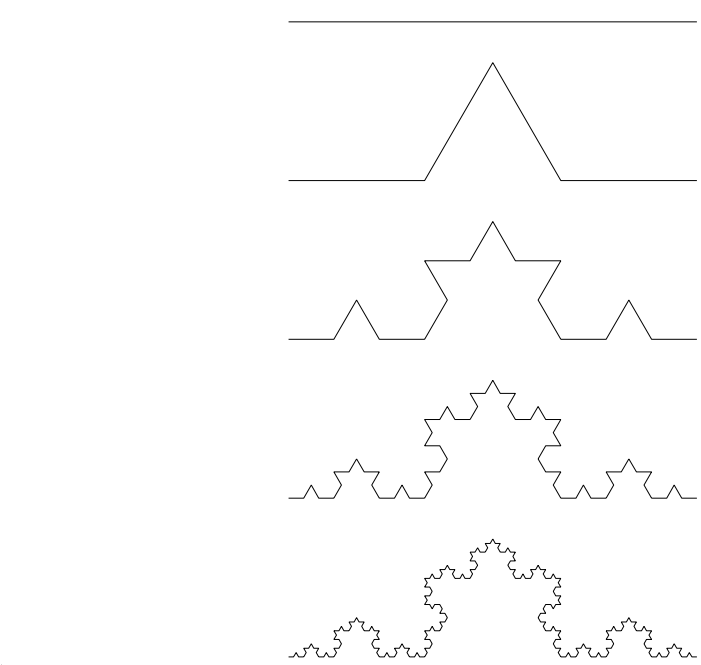

Figure 3.10: Generation of the Koch curve.

form; and the scale and orientation of the consequent pattern is relative to the pattern found in the current form. So, for instance, in the Koch example, the antecedent pattern does not specify the orientation or scale of the shape. It is this feature that allows a rule to be repeatedly applied to its own output.

In some cases this results in a conflict. This is because rules may be applied in more than one way. As a result, some fractal programs use additional markers in order to clarify how the rules should be applied. For example, a marker in the initial form might indicate the start point of a line. The antecedent pattern of the transition rule can also use a marker to ensure that the match is made using the correct rotation. The consequent pattern may also use markers indicating the start points of new lines.

Shape grammars

In most cases, fractal programs tend to be two-dimensional. As is the case with cellular automata, there is no reason why fractals programs cannot be developed that use three-dimensional substitution rules. In particular, programs known as *shape grammars* are two and three-dimensional shape based substitution programs that apply transition rules selectively rather than indiscriminately. Usually, rules are selected manually by the human user. Shape grammars were first proposed in 1971 by Stiny and Gips (1972) for the generation of configurations of shapes that represented abstract paintings and sculptures. This approach was later generalised (Gips, 1975; Stiny, 1975, 1980b). Knight (1994) gives a comprehensive overview of the theoretical foundations of shape grammars,

although computer programs are not discussed.

Shea (1997, 2001, 2002, 2004) has developed a generative structural design system — called *eiForm* — that uses shape grammar techniques to generate two-dimensional and three-dimensional space-frame structures. Structures consist of planar topology of structural members. (Three dimensional structures are essentially generated in two-dimensions and then projected onto a three-dimensional surface, such as a hemisphere or pyramid.) An initial design for a frame structure is defined, together with a set of rules that add, remove and modify structural members. The rules are developed by studying existing classes of design. For example, Shea developed a grammar for constructing traditional geodesic patterns. By iteratively apply these rules, different space-frame structures can be generated. This technique was combined with constraint satisfaction mechanisms, performance evaluation software and a simulated annealing optimization algorithm. The resulting system was used to develop a wide range of space frame structures. Figure 3.11 on the next page shows three space frame roof structures for an octagonal air plane hanger with walls that vary in height. The first design was optimised for pure efficiency, the second uses an aesthetic measure based on visual uniformity, and the third uses an aesthetic measure based on the golden proportion.

Shape grammars differ from fractals in two respects: the first difference involves the complexity of the process of matching rules to forms; the second difference involves the idea that one shape may be a *sub shape* of another shape.

- Within a shape grammar, shapes may be two or three-dimensional, and may include lines, planes or solids, although curves, curved surfaces and curved solids are generally excluded. In the three dimensional case, the antecedent and consequent arrangements of the transition rules consist of three dimensional arrangements of markers, lines, planes or solids. As a result, the process of verifying whether a particular rule matches the current form tends to be much more complex. Two arrangements of shapes are seen to be the same whenever one arrangement is a translation, rotation, reflection and/or scaled version of the other arrangement. Formally, these transformations are described as *euclidean transformations* or *affine transformations* (Knight, 1994, p. 44-47) (Flake, 1998, p. 94-98). For fractals in two dimensions, performing these transformations is feasible. In three-dimensions, the complexity of the transformations increases hugely.

- Sub-shapes can best be explained with an example. Consider a shape that consists of a two lines of the same length in a 'V' formation. Now consider a rule where the antecedent arrangement also consists of two lines in a 'V' formation, but in this case the lines are of different length. In the case of a fractal program, the rule could not be fired since the antecedent arrangement is not a

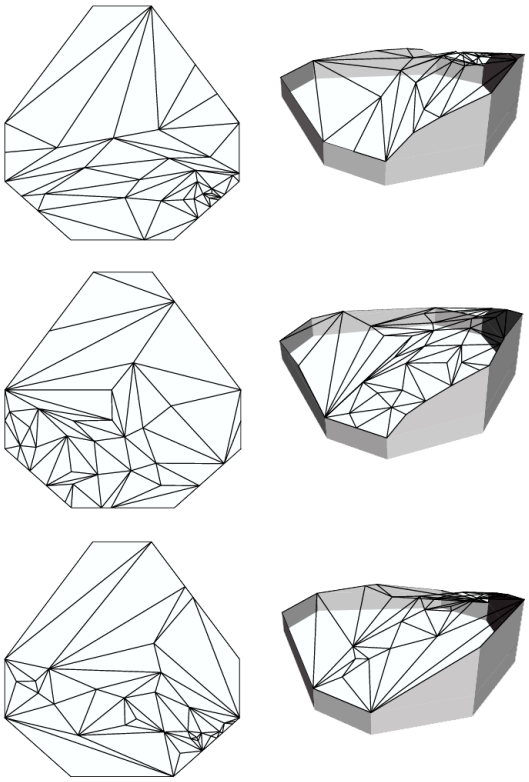

Figure 3.11: Three space-frame designs for an air plane hanger roof. From Shea (1997, p. 122–124).

> euclidean transformation of the current form. However, in the case of a shape grammar, the rule could be fired. Shape grammars introduce the idea of *sub shapes* (Stiny, 1980a). One arrangement of shapes is a sub shape within another arrangement of shapes when the former is completely contained within the latter. With shape grammars, a transition rule can be fired whenever the antecedent arrangement can be found in the current form, either as an identical arrangements of shapes or as a sub shape of some arrangement of shapes. Shape grammars have included this concept of sub shapes in order to allow for certain types of emergent behaviour (Stiny, 1994). However, this requirement further complicates the already complex task of matching rules to forms. As a result, shape grammars are often discussed in purely theoretical terms rather than as computer implementations.

Difficulties with creating implementations of shape grammars been highlighted by Knight (1999) and by Gips (1999). In response to a request by Gips for suggestions about computer implementations, Ulrich Flemming highlights the need for a robust implementation of a parameterised shape grammar interpreter that allows for the graphical defini-

tion of parameterised shape rules (which he describes as "a tricky, but intriguing proposition"). Flemming's students — Chien et al. (1998) — describe their frustration in attempting to create such an implementation, highlighting difficulties in determining parametric rule application. Gips asks: "Is the sub shape algorithm for shape grammars a solved (solvable) problem?"

L-Systems

Another well known example of a shape based substitution program are L-Systems. The growth process of almost all plants incorporates a repeated process of initiating new branches, thereby creating a branching pattern. In smaller plants such as ferns, these branching patterns can be very regular. In 1968, Lindenmayer (1968) invented a formalism, known as *L-systems*, that models plant growth. An L-system can be thought of as a fractal substitution program with certain connectivity constraints.

L-systems can be used to generate complex two-dimensional forms consisting of a series of straight line segments that resemble a variety of natural ferns and weeds (Prusinkiewicz and Lindenmayer, 1990). The line segments represent plant modules such as internodes, apices, leaves and branches. The line segments that the L-system manipulates fall into a number of different types, and these types are indicated in some way either by using colour or by having labels or markers associated with them. The types are then used within the antecedent and consequent patterns of the transition rules.

An important difference between L-systems and other fractal substitution programs is that L-Systems maintain the relationships between elements in the form. In a fractal substitution program, the connectivity between lines would not be maintained. In the case of an L-system, the insertion of an extra line into the centre of a structure forces the adjoining parts of the structure to re-position themselves in order to accommodate this extra line.

In order to simplify the process of maintaining the connectivity between line segments, the generation of the form tends to be broken down into two stages. In the first stage, the structure is represented in an abstract way as a string of symbols. The rules perform symbolic substitutions that expand the initial string into a more complex string. In the second stage, the completed symbolic string is then translated into a graphical form, with each symbol being mapped to a graphical operation.

L-Systems are often implemented as two-dimensional programs that represent the growing structure by using simple lines. Prusinkiewicz (1995) describes a number of three-dimensional L-systems that generate plants, flowers and trees. In addition to generating three-dimensional forms, L-systems have been further extended in a number of ways, including the development of stochastic L-systems and parametric L-systems. With stochastic L-systems, an element of random variation is introduced into the growth process. Transition rules are created that have the same

antecedent patterns. These rules will result in a conflict when the L-system finds that there is more than one rule that is applicable. In this situation, the L-system randomly chooses one rule from the set of possible rules. As a result, the forms that can be generated tend to be more natural. With parametric L-systems, the transition rules may include variable numeric parameters, thereby allowing the gradual phenomena of growth to be modelled. For example, the line labelled B may have a numeric parameter associated with it that defines its thickness. A transition rule may then specify that any line labelled B should be replaced by a new line that was 5% thicker, also labelled B. This would result in the line gradually becoming thicker with each time step, possibly simulating the thickening of a branch.

3.4.4 Context-free versus context-sensitive substitution approaches

The distinction context-free and context-sensitive is based on a distinction made by Chomsky (1956) in the field of linguistics, when he describes a number of different types of generative grammars. Generative grammars define a grammar of a language as a set of transition rules and sentences in the language that can be generated by applying these rules. The antecedent and consequent patterns consist of patterns of symbols in the language. Chomsky defined four kinds of generative grammar, classified according to the types of rules that are permitted: regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars.

The distinction between context-free grammars and context-sensitive grammars focuses on the antecedent pattern. Context free grammars have rules where the antecedent pattern can contain only one symbol, whereas context-sensitive grammars can have antecedent patterns with many symbols. As a result, the context-sensitive types of rules can take neighbouring symbols into account. This makes context sensitive grammars much more powerful.

Experiments by Wolfram

The importance of a context-sensitive generation processes has been highlighted by Wolfram in his research into complex systems. Wolfram (2002) describes a number of grid based substitution programs for generating complex one-dimensional, two-dimensional and three-dimensional forms. As well as cellular automata, Wolfram (2002) discusses two further types of program: mobile automata[4] and Turing machines[5] .

[4]Mobile automata are similar to cellular automata except that instead of updating all the cells in parallel, they have just a single 'active cell' that gets updated at each step.

[5]Turing machines are similar to mobile automata, except that the state of the cell and the colour of the cell are treated separately. They were invented by Alan Turing in

Wolfram has found that all these grid based substitution programs display certain similarities with respect to the complexity of the patterns that they create. In particular, he repeatedly highlights three points:

- First, complex patterns can be produced by all these systems by gradually increasing the complexity of the underlying system to a certain threshold.
- Second, the threshold of complexity required in the underlying system is surprisingly low.
- Third, increasing the complexity of the underlying system beyond this threshold does not yield patterns that are ultimately any more complex.

Context-sensitive L-Systems

A variety of context-sensitive L-Systems have been created in an attempt to model the flow of information within plants and between plants and the environment. Context-free L-Systems are sometimes described as *blind* (or as *0L-systems*), while context-sensitive L-Systems are sometimes described as *self-regulatory* (or as *1L-systems*) (Lindenmayer, 1968, 1982; Bell, 1986; Prusinkiewicz, 1995).

Context-free L-Systems use standard transition rules to substitute plant modules. They model a growth process that is controlled locally by each parent module, independent of the rest of the plant or the environment within which the plant is growing. Context-sensitive L-Systems, on the other hand, use transition rules where the antecedent pattern can specify neighbouring modules that must be present. Self-regulatory L-Systems model growth processes that are controlled by the parent module and by other neighbouring modules in the plant. For instance, the flow of nutrients or hormones in the growing plant can be modelled.

Despite the added level of complexity provided by context-sensitive L-Systems, they still leave out certain factors in the growth of plants that are important. In particular, parts of the plant can interact with each other without necessarily being neighbouring modules within the structure of the plant. For example, one branch of the plant may obstruct another branch from growing in a particular direction. Or the leaves at the top of the plant may shade branches further down, thereby limiting the amount of light that these lower branches can receive.

Systems that model these types of interactions are described as *sighted*. These systems use transition rules that have the ability to analyse the context into which the substitution is about to be made. They may

1936 to serve as idealised models for the basic processes of mathematical calculation. Wolfram (2002, p. 889) emphasises the fact that Turing focused on what complicated Turing machines could in principle do, rather than what very simple Turing machines can actually do. Wolfram cites Marvin Minsky as one of the first (1960s) to consider the simplest Turing machines capable of creating certain complex behaviours.

be thought of as context-sensitive systems that are more advanced than the self-regulatory L-Systems. Prusinkiewicz (1995) describes a series of approaches that use a variety of sighted mechanisms.

3.5 Summary

This chapter has discussed *parametric*, *combinatorial* and *substitution* approaches to generating form. These may be used within the developmental step of an evolutionary system in order to create a generative process. The main points are as follows:

- The parametric approach generates forms by parameterising either a model or a procedure. This approach is highly controlled, but the variability of the forms is limited.
- The combinatorial approach generates forms by combining components using either an algebra or a template. Using an algebra is very flexible, but the types of forms that are generated is difficult to control. Using a template allows for much more control, but variability is reduced.
- The substitution approach generates forms by repeatedly substituting components either by using a grid or by analyzing the shapes of the components. This approach allows for a small number of rules to be used to generate very complex forms. However, this approach is also unpredictable and difficult to control.

Other approaches and techniques also exist. A fourth approach that is not discussed here may be described as the *agent approach*: forms are generated by defining virtual agents that move, interact and collaboratively construct form in space. This approach is often inspired by insect colonies such as those built by ants and bees. These kinds of programs are also closely related to research in Artificial Life systems. Flake (1998, ch. 16) discusses a variety of agent based approaches.

Some researchers have also developed generative programs that are opaque; their inner workings are either difficult or impossible to discover. In such cases these programs are not discussed because they cannot provide any guidance to developing new generative programs. The numerous generative programs developed by Soddu (2002) fall into this category. Unfortunately, his published work does not explain how the programs work.

Chapter 4

Evolutionary computation

Contents

4.1 Introduction

This chapter provides a general introduction to evolutionary computation. Evolutionary design systems are not specifically discussed, but the more general issues covered in this chapter are nevertheless relevant to evolutionary design. The chapter consists of three main sections:

- In section 4.2, two general architectures for evolutionary systems are described: a synchronous architecture and an asynchronous architecture.
- In section 4.3, synchronous evolutionary algorithms are discussed in more detail. Genetic algorithms are described, and a number of other synchronous evolutionary algorithms are also introduced.

- In section 4.4, the rules and representations used by evolutionary systems to implement the evolution steps are described. These rules and representations may be used in both synchronous and asynchronous systems.

4.2 General architecture

4.2.1 Synchronous architecture

Synchronous versus asynchronous modes

Researchers have developed a variety of evolutionary algorithms that differ in many ways. Two important aspects are the *evolution mode*, which describes how the evolution steps process individuals in the population, and the *control structure*, which describes how the evolution steps are controlled. In chapter 1, a distinction was made between *synchronous* and *asynchronous* evolution modes, and between *centralised* and *decentralised* control structures.

The great majority of evolutionary algorithms, use a synchronous evolution mode in combination with a centralised control structure. A synchronous centralised architecture is identified that encompasses most of these algorithms. Such a general architecture is useful in that it allows the different evolutionary algorithms to be compared and contrasted. (It should be noted that this architecture does not necessarily represent the most efficient implementation. In practice many algorithms would not be implemented in this way.)

Overall structure

Figure 4.1 on the facing page shows the main components and interactions of the general synchronous evolutionary architecture. The general architecture is based on the description of the general evolutionary algorithm as defined by Bäck and Schwefel (1996); Back et al. (1997); Bäck (2000). However, their algorithm does not include the developmental step. The developmental step is included here because it is seen to be important as it allows a clear distinction to be made between the genotype and the phenotype.

In order to implement the synchronous evolution mode, the population is split into two parts, referred to as the *main population* and the *intermediate population*[1]. In addition to the three evolution steps in nature — reproduction, development, and survival — synchronous evolutionary algorithms also include an evaluation step and a selection step.

[1] At an abstract level, there is only one population. However, in order to simplify the implementation, the population is split into the existing generation and the new generation.

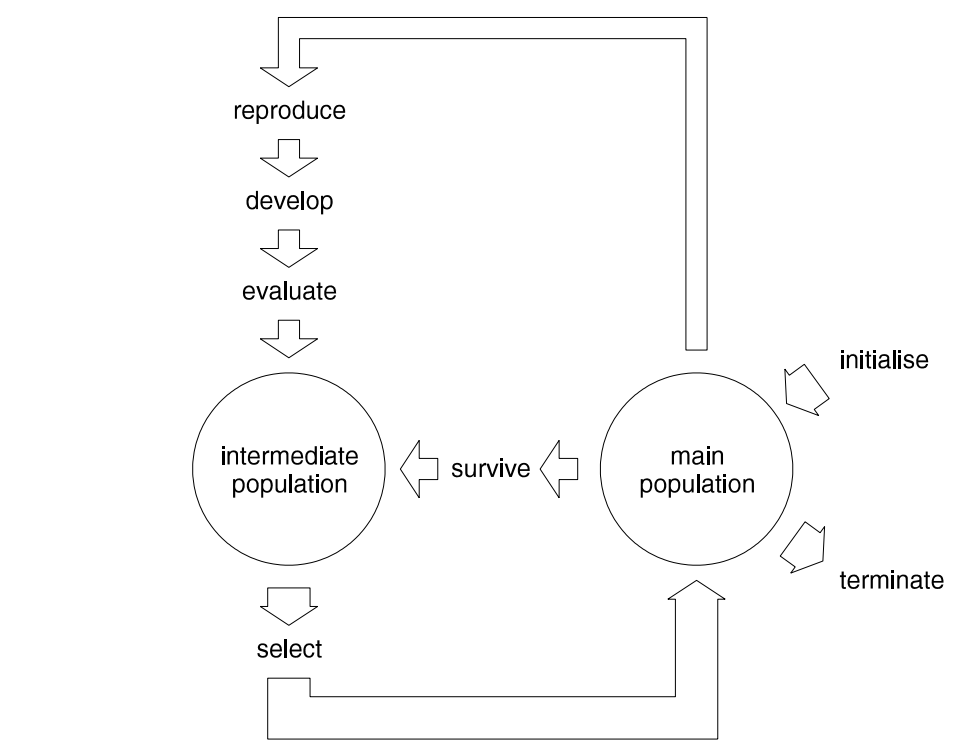

Figure 4.1: General evolutionary architecture for algorithms using the synchronous evolution mode.

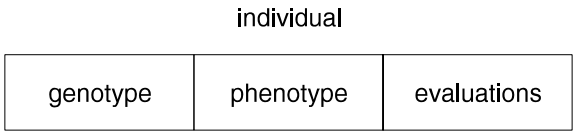

Figure 4.2: The three representations of an individual in an evolutionary algorithm.

The two populations continuously replace one another in a cyclical synchronous fashion. Each evolution step acts on the whole population. The survival steps copies a set of individuals from the main population to the intermediate population. The reproduction step selects parents from the main population and uses these to create a new set of individuals. These new individuals are then developed and evaluated and added to the intermediate population. The selection step then selects evaluated individuals in the intermediate population and copies them to a new main population.

An individual in the population is considered to consist of three parts: a genotype, a phenotype and a set of evaluation scores. Figure 4.2 shows the three main parts of an individual in an evolutionary algorithm.

- The survival step copies a set of existing individuals from the main population into the intermediate population. This allows individuals to survive through many generations. Typically, individuals are selected based on their fitness.

- The reproduction step creates a set of new individuals. Parents are usually selected randomly from the main population and genetic operators are then applied to the parent genotypes in order to create new genotypes.

- The development step creates a phenotype for each new genotype. In many cases, this process is a straightforward mapping process: genes in the genotype are mapped to a set of parameter values a parametric design model. In other cases, the developmental process may be much more complex, involving rule based growth processes.

- The evaluation step creates an evaluation score for each new phenotype. The evaluation scores are created by assessing the performance of the individual with respect to a particular objective. If more than one objective is specified, then this step will create multiple evaluation scores for each phenotype. The evaluated individuals are added to the intermediate population.

- The selection step selects a set of individuals in the intermediate population and copies them to a new main population. The evaluation scores are used to calculate a fitness value for each individual. The selection operator is defined in such a way that individuals with higher fitness are likely to be copied to the main population more often.

It should be noted that the survival and reproduction steps also include selection mechanisms. The survival step may select individuals from the main population for survival. The selection procedures will be discussed in the next section. The reproduction may select parents from the main population, though this is usually performed randomly.

Survival step

The survival step allows individuals to survive from one generation to the next. As a result, a population will include individuals created during different generations, referred to as *overlapping populations*. This means that parents will be able to compete with children. Jong (1975) uses the term *generation gap* to describe the size of the overlap: a small generation gap means that almost the whole population is replaced, while a large generation gap means that almost the whole population survives. The size of the generation gap differs depending on the algorithm being used; some algorithms rely heavily on survival while others do not allow any survival.

This step needs to select which individuals should be copied to the intermediate population, and which should be left behind to be discarded. Those that are being discarded may be thought of as being replaced. A variety of strategies for selecting individuals are used, which are referred

to as *replacement strategies*[2]. Four common replacement strategies are *worst*, *random*, *inverse proportional*, and *generational* (Alba and Troya, 1999).

- The worst replacement strategy deletes the worst individuals in the population.
- The random replacement strategy deletes random individuals in the population.
- The inverse proportional replacement strategy selects individuals to be deleted using the inverse of roulette wheel selection.
- The generational replacement strategy deletes all the individuals in the population. In this case, there is no generation gap.

The term 'replacement strategy' may suggest that new genotypes can only be added to the population if they replace existing members. However, this is not always the case. The number of new genotypes added to the intermediate population may be much larger than the number of individuals that were deleted. This results in an intermediate population that is larger than the main population. The selection step must then reduce this larger population back down to the size of the main population.

Synchronous evolution modes

The synchronous evolution mode encompasses three more specific types of modes that differ in terms of the size of the generation gap, and the replacement strategy used. Three common evolution modes are the *generational*, *elitist* and *steady-state* evolution mode:

- The generational evolution mode uses the generational replacement strategy. There is therefore no generation gap, and as a result the survival step can be omitted completely. The intermediate population will consist entirely of child genotypes.
- The elitist evolution mode (Jong, 1975) uses a small generation gap and commonly uses the worst replacement strategy. A small number of the fittest individuals are allowed to survive from one generation to the next. The replacement strategy usually consists of deterministically selecting the fittest individuals from the main population and copying them into the intermediate population. (The *worst* individuals are therefore left behind to be deleted.) The majority of genotypes in the intermediate population will still be created by the reproduction step.

[2]Alba and Troya (1999) uses the term *replacement policy*.

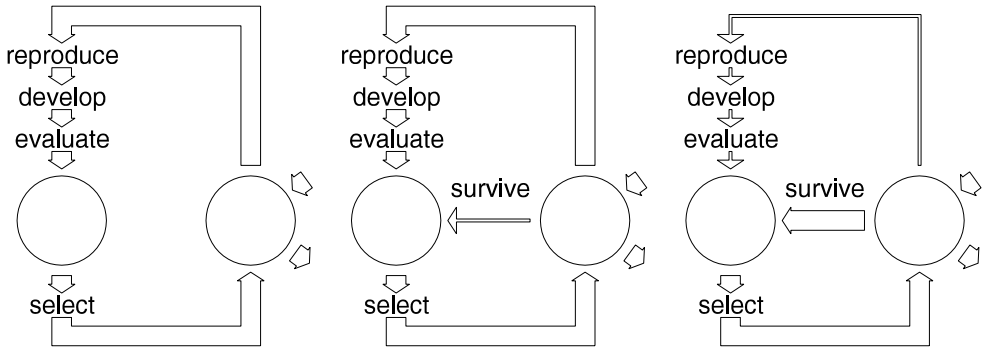

Figure 4.3: Comparing the generational, elitist and the steady-state evolution modes.

- The steady-state evolution mode (Whitey and Kauth, 1988; Syswerda, 1989) uses a large generation gap. Any of the replacement strategies discussed above may be used. The reproduction step only creates one or two genotypes every generation, with most of the individuals being allowed to survive. Those individuals that are not deleted are all copied into the intermediate population.

Figure 4.3 depicts the three modes, where the arrow width represents the number of individuals. Recently, a number of researchers have found that steady-state evolutionary algorithms have a much better performance that generational evolutionary algorithms (Whitley, 1989; Syswerda, 1991).

Types of parallelism

Evolutionary algorithms are usually implemented as serial processes on a single stand-alone computer. Due to their mode of operation they can easily be parallelised. Parallel evolutionary algorithms break down the evolutionary process into smaller sub-processes, which are then processed simultaneously using multiple processors. One of the main motivations for such parallel implementations was the desire to reduce the overall time to complete the task. If complex solutions are being evolved, the execution time of an evolutionary system can be extraordinarily long, with some researchers reporting times of up to one CPU year (Luke, 1998). In such cases, it is essential to reduce the speed of the overall evolutionary process by using some form of parallelism.

Researchers have also discovered that some forms of parallelization that fundamentally change the behaviour of the evolutionary process improve the quality of the solutions obtained. For example, some parallel evolutionary algorithms search different subspaces of the search space in parallel, thus making stagnation or premature convergence less likely. It should be noted though that such advantages are dependent on the restructuring of the evolutionary process, rather than the parallel implementation. The same advantages can be obtained by implementing the restructured evolutionary process in a sequential manner.

A number of researchers have proposed classifications and taxonomies for parallel evolutionary algorithms (Grefenstette, 1981; Cantu-Paz, 1997; Alba and Troya, 1999; Nowostawski and Poli, 1999). In general, there are two approaches to the parallelization of evolutionary algorithms (Grefenstette, 1981; Cantu-Paz and Goldberg, 1999): either a single global population is maintained and individuals are evaluated in parallel or multiple populations are evolved in parallel with some level of communication between them [3].

- With global parallelism, a single population of individuals is maintained and the individuals in the population are processed in parallel. These algorithms generally focus on parallelising the evaluation step, as this is usually the most computationally expensive step. Such algorithms are variously described as *global parallel evolutionary algorithms*, *master-slave parallel evolutionary algorithms*, or *panmitic parallel evolutionary algorithms*. In such a case, the master maintains a single population and runs the main evolutionary algorithm. When individuals need to be evaluated, they are sent to the slave processors and when evaluation is complete, the results are returned to the master. Two categories can be identified depending on how the population of individuals are processed (Cantu-Paz, 1997; Alba and Troya, 1999; Nowostawski and Poli, 1999): *synchronous global parallelism* and *asynchronous global parallelism*.

- With multiple population parallelism, the population is divided into multiple sub-populations, with each population being evolved in parallel by separate evolutionary processes that communicate with each other. Algorithms differ in the sizes of the sub-populations and the frequency and type of communication between the evolutionary processes. The two categories most commonly described in the literature are *coarse-grained* and *fine-grained* parallel evolutionary algorithms (Cantu-Paz, 1997; Alba and Troya, 1999; Nowostawski and Poli, 1999; Alba and Troya, 2001). The former break down the population into multiple large sub-populations or *demes* with infrequent communication. The latter break the population down into many small sub-populations with frequent communication.

Parallelism using multiple populations is complex, may require specialised hardware, and requires additional parameters affecting population sizes and information exchange to be set. Global parallelism using a single population is more straightforward, both theoretically and practically in terms of implementation. Synchronous and asynchronous[4] types of global parallelism will be discussed in more detail below.

[3] The approach of evolving multiple populations in parallel should not be confused with the two populations used in the general synchronous architecture. In the latter case, these two populations are mot being evolved in parallel. Rather, a single population has been split into two parts in order to simplify the implementation.

[4] In the literature, the terms *synchronous* and *asynchronous* are also used to de-

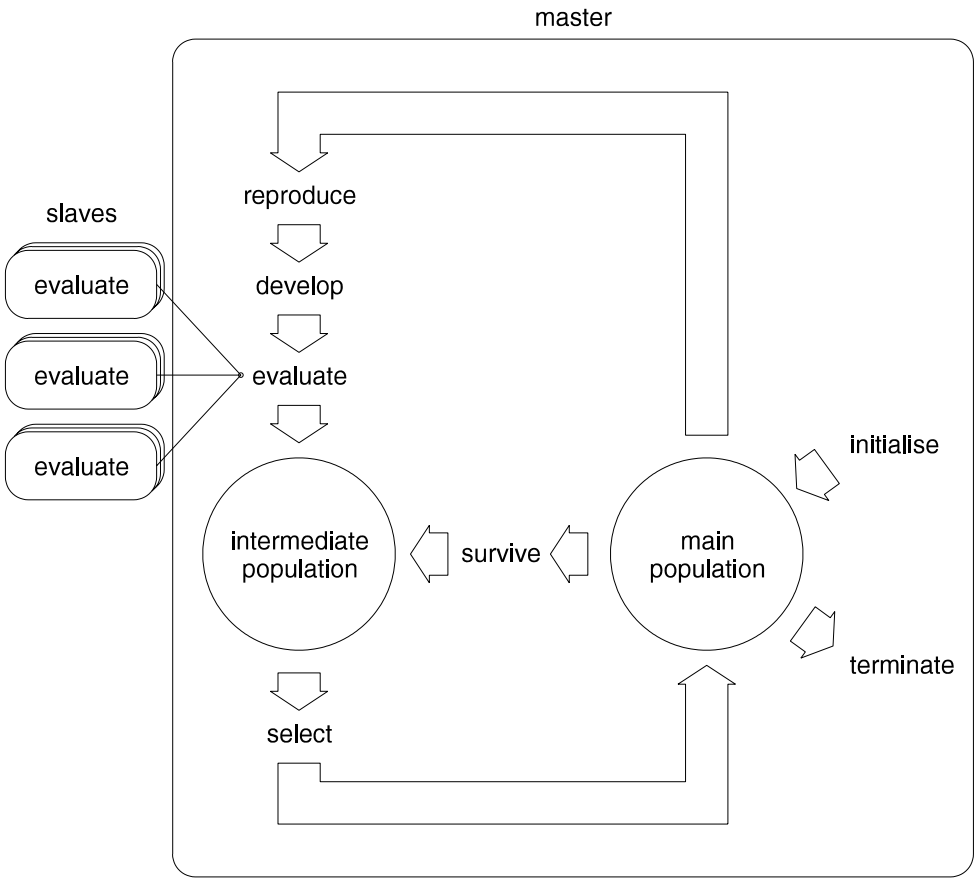

Figure 4.4: Synchronous global parallel architecture.

Synchronous global parallelism

With synchronous global parallelism, a master processor runs the main evolutionary process, and delegates the task of evaluating individuals to one or more slave processors. The evaluation step on the master processor assigns equal numbers of genotypes to each slave, the slave evaluates the genotypes and then returns the results. Figure 4.4 shows the master and slave processors in relation to the four evolution steps. One of the advantages of this method is that it is relatively easy to implement. This method could also be used to parallelise the other three evolution steps. The evaluation step is often emphasised in the literature because it is usually the most costly step.

The reduction in overall execution time that can be expected depends on two factors: the computation cost of the evaluation function, and the communication cost of transferring data between the master and the slaves.

- The computation cost is dependent on the complexity of the evalu-

scribe the exchange of information between sub-populations in multiple population parallelism. However, in this thesis, these terms are used only to refer to global parallelism.

ation process and on the number of genotypes that each slave must evaluate. The computation cost decreases as more slaves are added.

- The communication cost is dependent on the communication method and on the number of slaves, with cost increasing in proportion to the number of slaves.

Provided that the communication cost does not exceed the computation cost, the speed up will be significant. This approach is particularly suitable for cases where the evaluation step is expensive.

Cantu-Paz (1998) has analysed the execution time of synchronous global parallel evolutionary algorithms, and shown that there is an optimal number of processors that minimises the execution time.

With this approach, the parallel implementation has no effect on the fundamental behaviour of the algorithm, beyond reducing the overall execution time. The solutions that are evolved will be the same, regardless of whether sequential or parallel implementation is used.

4.2.2 Asynchronous architecture

Possible alternative architectures

The architecture described above uses a *synchronous* evolution mode with a centralised control structure. An alternative to this architecture is an *asynchronous* evolution mode, with either a centralised control structure or a decentralised control structure.

- For an asynchronous mode in combination with a *centralised* control structure, a review of the literature revealed that, although rare, systems using such an architecture have been developed. This architecture will therefore be discussed next.

- For an asynchronous mode in combination with a *decentralised* control structure, a review of the literature did not discover any systems using such an architecture[5]. Such an architecture will therefore not be discussed in this chapter. However, the generative evolutionary design system proposed in this research uses such an architecture, and this architecture will therefore be discussed at length in chapter 7.

Definitions of asynchronous evolution

Asynchronous evolutionary algorithms using a single population have been mentioned by a number of researchers when describing parallel global evolutionary architectures (Cantu-Paz, 1997, 1998; Alba and Troya,

[5]Various systems have been developed using a distributed peer-to-peer model (Chong and Langdon, 1999; Arenas et al., 2002). This is not the same as the decentralised model being considered here, and should be considered as another possible architecture.

1999; Nowostawski and Poli, 1999). For example, Cantu-Paz (1998) describes the difference between synchronous and asynchronous parallel genetic algorithms (GA) as follows:

> "If an algorithm stops and waits to receive the fitness values for all the population before proceeding into the next generation, then the global parallel GA is called *synchronous* and it has exactly the same properties as a simple GA, with a possibility of better performance being the only difference. However, it is also possible to implement an asynchronous global GA where the algorithm does not stop and wait for any slow processors, but it does not work exactly like a simple GA (it resembles a GA with a generation gap)."

Cantu-Paz (1998) does not elaborate on this resemblance[6].

Example of an asynchronous evolutionary algorithm

One example of an asynchronous evolutionary algorithm is provided by Rasheed and Davison (1999)[7]. The algorithm is introduced below, and will be discussed in more detail in chapter 5 (see section 5.2 on page 113).

The evolution steps are performed for many individuals in parallel using a master-slave model. The master processor creates new individuals one at a time, and sends these individuals to slave processors for evaluation until all slaves are occupied. When a slave completes the evaluation process, the master immediately adds the evaluated individual to the population irrespective of what the other slaves are doing. The master processor will then immediately create a new individual to be evaluated by the idle slave processor. As a result, the algorithm does not need to wait for each slave to complete its evaluation, but can apply the steps independently from one another as and when slaves become available.

[6]Nowostawski and Poli (1999), in their taxonomy of parallel genetic algorithms, define asynchronous global parallelism and steady-state parallelism as separate taxa. They write: "The difference lies in the selection operator. In an asynchronous master-slave algorithm, selection waits until a fraction of the population has been processed, while in steady-state GAs, selection does not wait, but operates on the existing population." However, this statement is thought to be misleading because steady-state algorithms typically delete the worst member in the population which requires all individuals in the population to be evaluated before the ranking can be performed. The algorithm would therefore have to process the whole population before proceeding to the next generation.

[7]It is not clear whether Rasheed and Davison (1999) describe their own algorithm as being asynchronous: they write "In the case of generational GAs, the global parallel GA is called *synchronous* if the master waits for the slave to finish evaluating an entire generation before generating any individuals of the following generation... If the master does not necessarily wait, the GA is called *asynchronous*... In this paper, however, we are interested in applying parallelism to a steady state GA." This would suggest that they see the synchronous-asynchronous terminology to be only applicable to evolutionary algorithms using a generational evolution mode.

Overall structure

From the brief comments relating to asynchronous global parallel algorithms and based on the evolutionary algorithm described by Rasheed and Davison (1999), a *general asynchronous evolutionary architecture* is inferred. This architecture is shown in figure 4.5 on the next page.

The asynchronous architecture uses an asynchronous evolution mode in combination with a centralised control structure. The evolutionary process therefore controls the evolution steps. However, these steps differ from their synchronous counterparts in the number of individuals that are processed. With the synchronous architecture, steps such as the selection and survival step typically process the whole population. For example, the survival step may need to delete the weakest individual in the population and will therefore need to process the whole population to find the weakest member. With asynchronous evolutionary algorithms, the reproduction, development and evaluation steps all process single individuals and the selection and survival steps process small groups of individuals. This allows these steps to be performed in an asynchronous manner, independently from one another.

Since individuals are reproduced and inserted back into the population one at a time, this asynchronous evolution mode is often referred to as a steady-state mode. However, this steady-state mode operates slightly differently from the synchronous steady-state evolution mode. With the synchronous version, only one individual can be processed at any one time. Only when this processing is complete can the evolutionary system start processing the next individual. With the asynchronous mode, large numbers of individuals may be processed at the same time. In some cases, the number of individuals being processed may actually be larger than the size of the population.

The five evolution steps may be described as follows:

- The selection step selects parents from the population. Selection may be performed randomly or some kind of fitness based selection rules may be used. The number of parents selected depends on how many are required in the reproduction step; usually two parents are required. Copies are made of the parent individuals and passed on to the reproduction step.

- The reproduction step creates a single new individual by applying the genetic operators to the parent genotype in order to create a new child genotype. This new individual is passed onto the developmental step.

- The developmental step creates a phenotype for the new individual, and then passes the individual onto the evaluation step.

- The evaluation step creates one or more evaluation scores for the new individual by assessing its performance with respect to a set

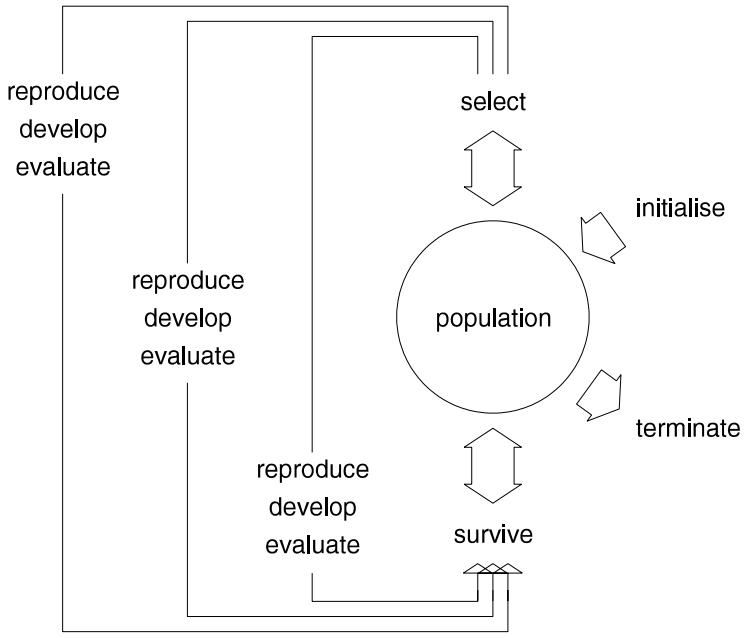

Figure 4.5: General evolutionary architecture for algorithms using the asynchronous evolution mode.

of objectives. The evaluated individuals are then passed onto the survival step.

- The survival step selects one existing individual from the population to be replaced by the new individual. Selection may be performed randomly or some kind of fitness based selection rules may be used.

Specific rules and representations used by the evolution steps are discussed below in section 4.4 on page 98.

Asynchronous global parallelism

One problem with synchronous global parallelism is that the whole evolutionary process has to wait until the slowest processor has completed evaluating the genotypes assigned to it. Only when all evaluations have been returned to the master processor can the selection step start creating a new main population. It has been suggested that one way to overcome this is to allow the evolutionary process to proceed even when all evaluations have not yet been completed. This is the main reason for implementing an asynchronous evolutionary algorithm. Figure 4.6 on the next page show the asynchronous global parallel architecture.

As with synchronous global parallelism, the reduction in overall execution time is dependent on the computational cost of the evaluation step, and the communication cost of transferring data between the master and the slaves.

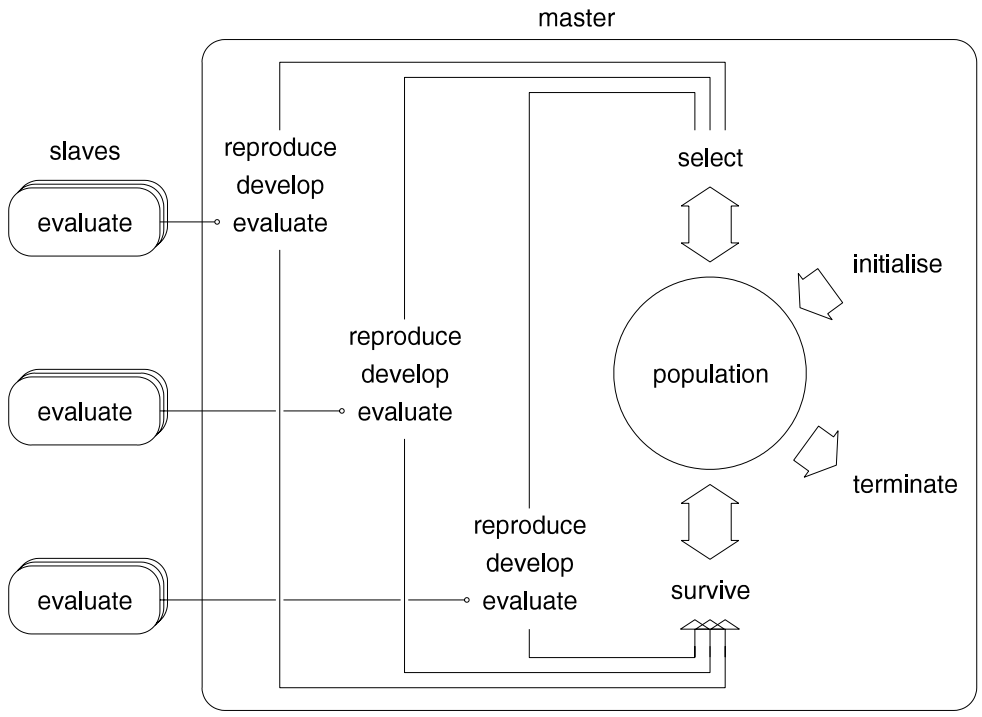

Figure 4.6: Asynchronous global parallel architecture.

4.3 Synchronous evolutionary algorithms

Synchronous evolutionary algorithms, being the most common, will now be discussed in more detail. Of the synchronous algorithms, genetic algorithms have been the most popular. Genetic algorithms can be traced back to the early 1950s when biologists used computers to simulate biological systems (Goldberg, 1989). However, in the strict interpretation, the *genetic algorithm* refers to the model introduced by Holland (1975) and his students at the University of Michigan. This model is referred to as the *canonical* genetic algorithm (Whitley, 1994). This algorithm was originally proposed as a general model of adaptive processes, but by far the largest application of the techniques is in the domain of optimization (Jong, 1993).

4.3.1 Canonical genetic algorithm

Users of genetic algorithms

Genetic algorithms have been applied to a wide variety of problems. In most cases, they are used to build problem specific software systems. Users with programming skills will typically build customised systems for specific problems, possibly using generic libraries of functions and subroutines. The development of ready-made systems for non-programmers is relatively rare, and if they do exist they tend to be focused on a narrow application domain. The genetic algorithm, and more generally

the evolutionary algorithm, tends to be seen as an adaptable concept for problem solving (Back et al., 1997).

Alba and Troya (1999) provide a comprehensive survey of the most important genetic algorithm implementations. This survey classifies implementations into three types: ready-made menu-driven implementations that are developed for a specific application domains; source code and libraries that allow users to implement a particular algorithm; and flexible programming toolkits that allow users to construct a number of different algorithms from predefined components. Implementations within the first classification are aimed at non-programmers, while the second and third classifications require the user to have extensive programming skills. Out of the 36 implementations surveyed, only four fell within the first classification.

The main reason that ready-made systems are rare is due to the need to embed problem specific knowledge within the system. The main area that contains such knowledge is the evaluation step. This includes the genotype length, the mapping process and the evaluation function or objective function. (The fitness function on the other hand is not problem specific; see section 4.3.1 on page 94.) The researcher that uses a genetic algorithm must define these aspects of the evaluation function in a problem specific manner.

In addition to the evaluation step, users of genetic algorithms also need to define the evolutionary parameters and create an initial population. The evolutionary parameters include the population size, the crossover probability and the mutation probability. The choice of these parameters will also be affected by problem specific knowledge. The initial population of genotypes may be generated by randomly generating binary strings. In some cases, researchers have found that the performance of the algorithm will be improved if the population is seeded with carefully design problem specific genotypes.

Search and search spaces

Genetic algorithms are often characterised as search systems for finding 'optimal solutions'. In most cases, a near optimal solution is provided and the task of improving the solution is then characterised as a search problem. This idea of searching among a selection of candidate solutions gives rise to the search space concept (Wright, 1932; Newell et al., 1967; Kanal and Cumar, 1988).

The search concept encompasses some notion of distance between candidate solutions. An algorithm for searching this space is a method for choosing which candidate solutions to test at each stage of the search process. In most cases the next candidate solution(s) to be tested will depend on the results of testing previous sequences; most useful algorithms assume that there will be some meaningful relationships between 'neighbouring' candidate solutions - those close together in the space. As a consequence of the search space concept, the idea of a fitness landscape

arises naturally. A fitness landscape is a representation of the space of all possible solutions along with their fitnesses. It is referred to as a landscape because the fitness values can in some cases be visualised as 'hills', 'peaks', 'valleys', and other features analogous to those found in physical landscapes. The task of the genetic algorithm is to home in on the highest peaks in this landscape.

A large number of search algorithms have been developed in order to try and search as efficiently and thoroughly as possible. Hill-climbing is one of the best known. This algorithm utilises the iterative improvement technique whereby, each iteration, a new solution is searched for in the neighbourhood of the current solution. This is known as a point-to-point search because at any one time, only a single point is being processed. However, such point-to-point search algorithms are susceptible to stagnation at local peaks and have difficulty searching rugged fitness landscapes. In order to overcome these drawbacks, numerous alternative algorithms were introduced. The most successful alternatives discarded the point-to-point search strategy in favour of a parallel strategy, whereby whole populations of solutions are considered at any one time. Genetic algorithms are based on the intrinsically parallel process of natural evolution. As such they have earned the reputation for being an efficient and robust type of parallel search algorithm suitable for certain types of complex problems (Paul Schwefel, 2000).

The individual in the population

Each member of the population is described as an *individual* and has two parts. The first part is a fixed-length binary string[8] that encodes the parameters for a solution, referred to as a *genotype* (Holland, 1975) or a *chromosome* (Schaffer, 1987). The binary string is mapped to a set of real values that are the parameters being optimised. The second part is a real number that represents the quality of the individual with respect to some predefined objective, referred to as the *fitness*. Figure 4.7 on the next page shows the two parts of the individual. Within the genotype string , the sub-string '1011' encodes one of the parameters in the solution.

For example, one common application of the genetic algorithm is function optimization, where the goal is to find a set of parameter values that maximise a function. In the case of a function with six variables, the mapping step must first break the binary string into six sub-strings, and then translate each sub-string into a real number. The six real values are then used to evaluate the function. This evaluation may either represent the fitness directly, or a comparative fitness value can be calculated by taking into consideration all other members in the population. If at some later stage a different function needs to be optimised, then the evaluation rules will need to be changed.

[8] A binary string is a list of '0's and '1's.

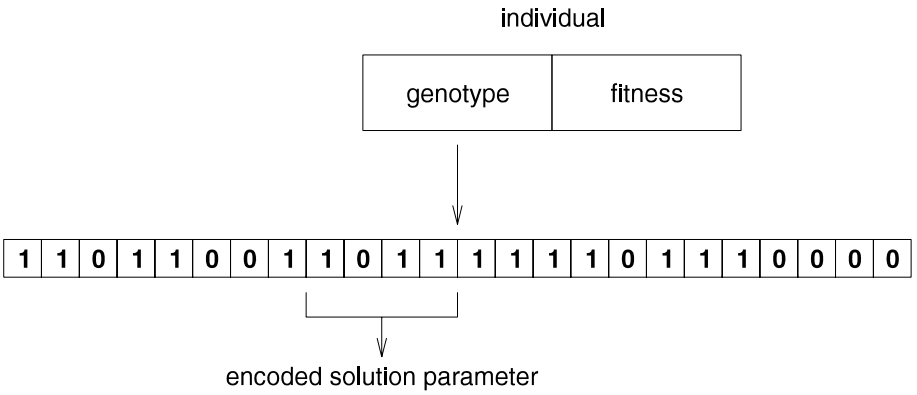

Figure 4.7: The representation of an individual, consisting of a binary string genotype and a real valued fitness.

Evolution steps

The synchronous evolutionary process has been described as consisting of five key steps: survival, reproduction, development, evaluation, and selection. With canonical genetic algorithms, the developmental and survival steps are not used. (The developmental step may be thought of as the process of mapping the binary string to a set of real values. However, most researchers in field of genetic algorithms conceptualise this as being part of the evaluation step. The survival step is ruled out since a generational replacement strategy is used.)

For the evaluation step, the terms 'evaluation' and 'fitness' are often used interchangeably. However, Whitley (1994) has highlighted a important distinction: "The *evaluation function*, or *objective function* provides a measure of performance with respect to a particular set of parameters. The *fitness function* transforms the measure of performance into an allocation of reproductive opportunities. The evaluation of a string representing a set of parameters is independent of the evaluation of any other strings. The fitness of that string, however, is always defined with respect to other members of the current population." The process of determining the fitness may be thought of as being part of either the evaluation step or the selection step.

The canonical genetic algorithm is usually described as consisting of three steps: reproduction, evaluation and selection.

- The survival step is not used because the generational replacement strategy deletes all individuals in the main population.
- Reproduction creates a new intermediate population. Pairs of parents are randomly selected from the main population and recombination and mutation are applied to the genotypes of these parents with fixed probabilities. The recombination operator breaks off the tail of each parent string at some random location, and then swaps these tails to create two new child strings. This is referred to as

one-point crossover. The mutation operator will change each bit in a genotype with a fixed probability. The probability of recombination is usually quite large (e.g. 60% or 70%), while the probability of mutation is usually low (e.g. 1%).

- The developmental step is not used.
- Evaluation involves computing the fitness of each genotype in the intermediate population. This consists of three processes: first each genotypes is broken down into a set of fixed-length binary segments which are decoded into integer values and then mapped to a set of real values; second, these values are then used by the *evaluation function* or *objective function* in order to obtain an evaluation for each individual; and third, the fitness for each individual is calculated using the *fitness function*, which typically consists of dividing each evaluation by the average evaluation for the population.
- Selection replaces the main population with a new population of genotypes. The genotypes currently in the main population are first discarded. New genotypes are then repeatedly selected from the intermediate population and copied into the main population. This selection mechanism ensures that genotypes with a higher fitness have a higher chance of being selected. This selection mechanism is described as *fitness proportionate selection* or *proportional selection*. Genotypes with high fitness are likely to be copied many times, whereas genotypes with low fitness may not be copied at all.

Exploration and exploitation

Holland (1975) characterises the process of adaptation of individuals during the evolutionary process as a balance between two opposing tendencies: *exploration* and *exploitation*. Exploration refers to the creation of diversity within the population. This is achieved primarily through the genetic operators: mutation introduces innovation, while recombination changes the context of existing genetic information. Exploitation refers to the reduction of diversity as a result of a selection process that favours individuals of higher fitness. This balance between exploration and exploitation is critical for the evolutionary process. Holland's original genetic algorithm was proposed as an "adaptive plan" for accomplishing a proper balance between exploration and exploitation in adaptive systems.

The level of exploitation within an evolutionary system is often characterised as *selection pressure*. If the selection pressure is high, the evolutionary process quickly converges on the fittest individuals. If the selection pressure is low, the evolutionary process becomes divergent. Excessively high selective pressure may result in the evolutionary process converging too quickly; this is described as *premature convergence*. Conversely, if the selection pressure is excessively low, then the evolutionary process may lose direction and start randomly drifting; this is known as

stagnation. Goldberg and Deb (1991) and Bäck (1994) have attempted to quantify the selective pressure of different selection approaches.

Many researchers have found that it useful to change the selection pressure as evolution progresses. At the start of the evolutionary process, low selective pressure is useful because it allows many new possibilities to be explored through a divergent search process. By gradually increasing the selective pressure the process becomes more convergent, focusing on the fitter individuals and exploiting their genetic material.

Building blocks hypothesis

In order to help explain how genetic algorithms work, Holland (1975) proposed the *building block hypothesis*. This hypothesis postulates that canonical genetic algorithms work by discovering, emphasising, and recombining good 'building blocks' of solutions in a highly parallel fashion. A building block is a small sub-section of the genotype. The idea is that good solutions tend to be made up of good building blocks.

Holland (1975) formalised this notion of building blocks in the *Schema Theorem*. This theorem shows that the canonical genetic algorithm (using binary representations and single point crossover) provides a near-optimal strategy for increasing the number of good building blocks.

The Schema Theorem has been the subject of much critical discussion. Many researchers argue that it fails to capture the full complexity of the genetic algorithm. (For an introduction to these issues, see Mitchell (1996, ch. 4)).

4.3.2 Other common synchronous algorithms

Evolution strategies

Evolution strategies were created by Rechenberg (1973) and further developed by Bäck (1996). The genotype representation consists of a fixed length list of real values, called a *real-valued vector*. The phenotype representation is specific to the problem application. Initially, evolution strategies had a population of just one parent and one child in each generation (Schwefel, 1965; Rechenberg, 1973). In the 1980s, researchers developed more advanced evolution strategies using populations with multiple parents and multiple children (Bäck, 1996).

- The survival step has two versions, referred to as (μ, λ) and $(\mu+\lambda)$, where μ is the number of new individuals created each generation, and λ is the size of the main population. In the first version, there is no survival and individuals cannot be copied from the main population to the intermediate population. The intermediate population is of size μ. In the second version, all individuals in the population survive. Each generation, all genotypes in the main population are copied to the intermediate population, and new genotypes are created and added to this population. The size of the intermediate

population is therefore $\mu + \lambda$. The size of μ is typically around seven times λ (Bäck, 1996).

- The reproduction step randomly selects parents from the main population and applies recombination operators and a mutation operator in order to generate new real valued vectors. The recombination operators may recombine the values within two parents, or they may use a different parent for every variable in the vector. The mutation operator replaces randomly selected values within the vector with new normal-distributed random values.

- The developmental step must create the phenotype from the real valued vector. The mapping from a binary string to real values is not required. However, the mapping from the real values to the more complex structure that represents the phenotype is still required.

- The evaluation step calculates a fitness value for each genotype in a problem specific manner.

- The selection step creates a new main population using deterministic selection (see section 4.4.3 on page 108).

An important further development of evolution strategies is their ability to evolve parameters controlling the evolutionary process. This is referred to as *self-adaptation*. Rudolph (2000) gives a brief overview of contemporary evolution strategies.

Evolutionary programming

Evolutionary programming was created by Lawrence Fogel (Fogel, 1963) and further developed by his son David Fogel (Fogel, 1995). As with evolution strategies, the genotype representation consists of a real-valued vector and the phenotype representation is problem specific.

- The survival step is similar to the process used by $(\mu+\lambda)$ evolution strategies, with all individuals in the population surviving. Each generation, all individuals in the main population are copied to the intermediate population, and new individuals are created and added to this population until its size has doubled.

- The reproduction step creates new genotypes from single parents using only mutation. Each genotype in the main population is used to create a new genotype by making a copy of the parent vector and mutating the real values within this vector. (Fogel (1995) has shown that mutation is able to simulate operators such as crossover.)

- The developmental step is similar to that of evolution strategies. The real valued vector must be mapped to the phenotype in a problem specific manner.

- The evaluation step uses the values in the real valued vector to calculate the fitness value for each individual.
- The selection step creates a new main population using either deterministic selection, tournament selection or proportional selection (see section 4.4.3 on page 108).

Genetic programming

Genetic programming was created by Koza (1992) as a way of evolving computer programs. In this case, each genotype is actually a computer program, usually written in symbolic languages such as LISP. There is therefore no distinction between the genotype and the phenotype.(It may be argued that the phenotype should be considered to be the behaviour of the program as it runs or the outcome, rather than the program itself (Bentley, 1999a, p. 56-57). For example, if the program produces some data, then this data may be thought of as being the phenotype.)

- No survival is allowed. Evolution proceeds in a synchronous generational manner similar to genetic algorithms.
- The reproduction step is similar to that of the genetic algorithm, except that specialised operators are used. Each computer program is represented as a tree, with the crossover and mutation operators creating new trees by modifying and rearranging the branches of the parent trees. The crossover operator switches a randomly chosen branch of one tree with a randomly chosen branch of another tree; the mutation operator deletes a randomly chosen branch and replaces it with a randomly created new branch. These operators are carefully designed to ensure that they do not create invalid or meaningless programs.
- The developmental step is not necessary as the genotype represents the computer program directly.
- The evaluation step evaluates the performance of the computer programs by running the programs and analyzing their behaviour and output.
- The selection step creates a new main population using a proportional selection (see section 4.4.3 on page 108).

4.4 Rules and representations

Synchronous and asynchronous evolutionary algorithms may use similar rules and representations in order to define the evolution steps. In general, these rules and representations may be seen as transformations that

take one or more individuals as input and produce one or more individuals as output. In the literature, the rules are also sometimes referred to as *operators*; for example, reproduction operators, and selection operators.

A wide variety of such rules and representations have been developed. This section will discuss some of these rules and representations in more detail.

4.4.1 Genotype representation

Complex representation

Genetic algorithms using binary string representation together with bit mutation and one-point crossover have the advantage of strong theoretical foundations (Goldberg, 1989, p. 40-41). Nevertheless, in practice, many researchers have found that the performance of the algorithms can be improved by using more complex types of rules and representations (Michalewicz, 1993, 1996). Michalewicz (1996, p. 3) advocates "the use of proper (possibly complex) data structures (for chromosome representation) together with an expanded set of genetic operators". He argues in favour of evolutionary algorithms that employ complex representations and rules. Michalewicz also provides many examples where such algorithms have been successfully used to solve complex problems that canonical genetic algorithms were unable to solve.

For problems with continuous parameters, a common trend is to use a real valued genotype representation directly, rather than a binary representation that is then mapped to real values (Janikow and Michalewicz, 1991; Radcliffe, 1991; Wright, 1991; Bäck, 1993; Eshelman and Schaffer, 1993). New types of reproduction rules have also been developed in order to perform mutation and recombination of such strings.

For problems with complex constraints, it is now generally accepted that performance of the genetic algorithm can be improved by creating new problem specific rules and representations for the reproduction and the selection steps (Wolpert and Macready, 1995; Michalewicz, 1996). In many cases, string based genotype representations are replaced by other more complex types of structures such as trees and graphs. For example, Koza writes: "Representation is a key issue in genetic algorithm work because the representation can severely limit the window by which the system observes its world... String-based representation schemes are difficult and unnatural for many problems and the need for more powerful representations has been recognised for some time" (Koza, 1990) (quoted in (Michalewicz, 1996, p. 4)). For such representations, reproduction rules need to be developed that are specific to these representations.

Strong and weak evolutionary algorithms

Evolutionary algorithms can range from being generic to highly specific to a particular problem. Michalewicz (1993, 1996) describes a hierarchy

of evolutionary algorithms capable of solving hard optimization problems. At the bottom of the hierarchy are evolutionary algorithms that are problem specific, while at the top of the hierarchy are programs that are the most generic. The set of problems to which an evolutionary algorithm can be applied is described as the *domain* for that program. The problem specific programs have a narrow domain and are described as *strong*, while the generic ones have a broad domain and are described as *weak*. Genetic algorithms are identified as the weakest programs and are at the top of the hierarchy. Below genetic algorithms, the evolutionary algorithms get progressively more problem specific "by using 'natural' representations and problem-sensitive 'genetic' operators" (Michalewicz, 1996, p. 289).

Michalewicz puts forth the hypothesis that the stronger program should generally perform better that the weaker program, based on "a number of experiments and on the simple intuition that problem-specific knowledge enhances an algorithm in terms of performance (time and precision) and at the same time narrows its applicability" (Michalewicz, 1996, p. 291).

The domain and the performance of an evolutionary algorithm are related to each other via the rules and representations used. A narrow domain implies problem specific rules and representations which in turn implies a good performance. A broad domain implies generic rules and representations which in turn implies a poor performance. The programmer of an evolutionary algorithm must decide on an appropriate balance between domain and performance. An additional factor that must be considered by the programmer is the time and effort required to develop the system. Michalewicz (1996, p. 303) writes: "The development of a stronger, high-performance system may take a long time if it involves extensive problem analysis to design specialised representation, operators, and performance enhancements".

Using a standard algorithm such as a genetic algorithm would clearly reduce the development time and effort. However, Michalewicz (1993) has shown that for most complex problems, the performance of these weak programs is poor when compared to stronger alternatives. Michalewicz (1993, p. 303) writes: "If one is solving a transportation problem with hard constraints (i.e. constraints which must be satisfied), there is little chance that some standard package would produce any feasible solution, or, if we start with a population of feasible solutions and force the system to maintain them, we may get no progress whatsoever — in such a case the system does not perform any better than a random search routine". As a result, a new system may need to be built from scratch and could require a considerable amount of time and effort on the part of the programmer.

4.4.2 Developmental step

Distinction between genotypes and phenotypes

With canonical genetic algorithms, the evaluation step first maps the genotype to a set of parameters, and then uses these parameters in order to calculate the genotypes fitness. The phenotype is embedded within the evaluation step, and as a result the existence of a phenotype is not emphasised. Nevertheless, many researchers have found it useful to make the genotype-phenotype distinction more explicit. Once this distinction has been made, two types of search space can be identified: the genotype representation defines a *genotype space* and the phenotype representation defines a *phenotype space*. A developmental mapping function is thought of as a function that maps points in the genotype space to points in the phenotype space. The genetic operators act on points in the genotype space, and the evaluation function acts on points in the phenotype space (Back et al., 1997).

If the types of solutions being evolved by a genetic algorithm are susceptible to parameterization, then a fairly simple mapping function can be created. However, with many real-world problems, the potential solutions are complex and cannot be parametrised in a straightforward manner. As a result a simple mapping function may no longer be feasible. Under such circumstances, researchers tend to follow one of two approaches (Michalewicz, 1996; Back et al., 1997):

- A special genotype representation may be created that is closer to the phenotype representation. This would allow a simpler type of mapping function to be used. The disadvantage of this approach is that special reproduction rules would have to be developed that are able to manipulate the genotype representation.

- A more complex developmental process may be created that transforms the genotype into the phenotype. Such a process would allow a standard genotype representation and standard reproduction rules to be used. The disadvantage of this approach is that the complex mapping procedure may introduce other problems that hinder the evolutionary process.

Michalewicz (1996) argues for the first approach. A genotype representation that is closer to the phenotype representation is described as being a 'natural' representation. He writes: "It seems that a 'natural' representation of a potential solution for a given problem plus a family of applicable 'genetic' operators might be quite useful in the approximation of solutions of many problems, and this nature-modelled approach... is a promising direction for problem solving in general." Back et al. (1997) also suggests that many researchers prefer "natural, problem-related representations".

Other researchers have found that, in some cases, the second approach is more appropriate. They have found more complex developmental pro-

cesses analogous to the developmental processes in nature may have some advantages. During the last decade, there has been an increase in research into such developmental processes — referred to as *gene expression* (Kargupta, 2003) — in genetic and evolutionary computation. O'Neill and Ryan (2000) list a number of areas, including genotype-phenotype distinction, genetic code evolution, distributed fitness evaluation, the role of introns, and degenerate genetic codes and neutral mutations.

The developmental process in nature

In nature, the genotype of a plant or animal cannot be directly evaluated. Instead, the genotype is developed into the phenotype — the fully developed organism — by a complex developmental process. Through the machinery of the cell, the DNA is copied into another long string called RNA. The RNA then leaves the cell nucleus to be read by a cellular device that brings amino acids corresponding to the coding sequence together to be linked in the proper order to make a polypeptide of perhaps several hundred amino acids. When finished, the polypeptide folds up into a complex shape to form a protein. In total, about 100,000 different proteins can be created in this way. The shape of the protein then gives it certain functions and properties; for example, certain shapes might allow it to bind to fit together with other proteins to form cell structures, while others might allow it to bind to chemicals and change the speed with which they react. The definitions for all these steps exist as physical properties of the various entities involved.

In order to create complex organisms with cells that differ from one another, the expression and repression of genes must be carefully orchestrated. Kauffman (1993, p. xvii) describes this process as follows: "Cell types differ because different subsets of genes are 'active' in different cell types... The expression of gene activity is controlled at a variety of levels, ranging from the gene itself to the ultimate protein product. It is this web of regulatory circuitry which orchestrates the genetic system into coherent order. That circuitry may comprise thousands of molecularly distinct interconnections".

The role of the environment in this process is often underestimated, and the role of the DNA is often overemphasised. The DNA is sometimes described as a 'blueprint' for an organism. However, this is misleading as it suggests that the DNA defines the complete design of the organism. Even the idea that the DNA consists of a set of 'instructions' is not accurate. The instructions are embedded within the environment as physico-chemical processes, and the genes in the DNA only provide the initial trigger for these processes to occur (Kauffman, 1993). The genes contained in the DNA can therefore be thought of as a set of triggers that initiate certain processes. These triggers will only have the desired outcome if the genes are in an environment that contains the suitable instructions.

Advantages of a developmental process

Angeline (1995) provides an overview of evolutionary systems that incorporate a developmental step. The developmental step is referred to as *morphogenic processes*. Angeline sees the inclusion of the developmental step as a way of extending the abilities of evolutionary algorithms to more readily construct large, complex, structures. In particular, two benefits of including a developmental step are highlighted:

- The genotype representation can be designed to ensure that the genetic operators are more likely to create viable offspring. This may increase the chances of evolving appropriate solutions.
- The genotype representation can be compressed into a compact form thereby reducing the size of the genotype's search space, which will reduce the time needed to evolve large structures. Uninteresting areas in the phenotype space may also be excluded.

Other researchers have proposed additional advantages. O'Neill and Ryan (2000) argue that the developmental step allows the genotype representation to incorporate a number of genetic techniques found in nature that may help the evolutionary process. They list techniques such as the use of *generalised encodings* that allow a wide variety of phenotypes to be developed using the same genotype representation, *degenerate encodings* that allow a range of different genotypes to result in the same phenotype, and *positional independence* where the position of particular genes within the genotype does not affect the developmental process.

Bentley (1999b) highlights the possibility of improved constraint handling. He argues that the development step can be designed to ensure that every possible genotype is mapped to a legal phenotype that abides by certain constraints.

Types of developmental processes

Angeline (1995) formally defines the development step as consisting of a function that transforms the genotype into the phenotype. Three types of developmental function are defined: *translative developmental functions*, *generative developmental functions* and *adaptive developmental functions*:

- Translative developmental functions typically involve a trivial or near trivial mapping that is used to encode the phenotype into appropriate form for the evolutionary process. Such mappings simply translate the genotype representation to the phenotype representation without much expansion. In most cases there is no interaction between the genes, with each phenotypic trait being specified by a single gene. Examples include binary codings, Gray codings and the evolution of neural network structures (Harp and Samad, 1991).

- Generative developmental functions typically transform genotypes into phenotypes using recursive processes such as L-systems, production rule systems, and other types of symbolic rewrite processes. A grammar-based process is often used that works as a decompression function. In many cases, phenotypes are created that are exponentially larger than the genotypes. Examples include the evolution of classifier systems (Wilson, 1989), the evolution of neural network structures (Kitano, 1990; Gruau, 1992; Garis, 1994), the evolution of artificial life (Sims, 1994) and the techniques use in the field of genetic programming (Koza, 1992, 1994).

- Adaptive developmental functions use a scheme that dynamically creates the development function during evolution. The development function is generally a generative process that is subject to modification during evolution. Such modifications are usually dependent on either the number of generations that have elapsed, or on other factors such as the diversity of the population. Examples include extensions of genetic algorithms (Shaefer, 1987; Schaudolph and Belew, 1992; Whitley et al., 1991) and extensions of genetic programming (Angeline and Pollack, 1994; Rosca and Ballard, 1994).

Bentley and Kumar (1999); Kumar and Bentley (2003) have proposed an alternative classification of developmental processes. They refer to the developmental process as a *computational embryogeny*. They first classify embryogenies into those where the developmental process is evolvable (the adaptive type in the classification by Angeline (1995)) and those where it is not evolvable. Evolvable types are then further subdivided into those that explicitly describe the developmental process, and those where the developmental process emerges as a result of recursively applying growth rules. Three different classes are defined: *external embryogenies* are non-evolvable; *explicit embryogenies* are evolvable and explicitly describe the developmental process; *implicit embryogenies* are also evolvable but only describe the developmental process implicitly. In order to explore the behaviour of the evolvable classes of embryogenies, Bentley and Kumar developed some simple experiments that involved evolving letter shapes.

Developmental processes for design

Classifications specific to particular fields have also been developed. In design, a number of such classifications have been proposed. Bentley (1999d) identifies four types of evolutionary design: *evolutionary design optimization*, *creative evolutionary design*, *evolutionary art*, and *evolutionary artificial life forms*.

- Evolutionary design optimization is the straightforward optimization and is referred to here as parametric evolutionary design.

- Creative evolutionary design involves the evolution of designs guided by functional performance criteria.
- Evolutionary art focuses on evolutionary systems with interactive interfaces that allows human users to evolve a variety of designs and to judge these designs based on aesthetic appeal.
- Evolutionary artificial life-forms involve the evolution of designs that have both a form and a behaviour.

Hu et al. (2002) have proposed a classification of evolutionary design problems that is dependent on the types of problems. They propose three types of problem: *fixed structure with a fixed number of parameters*, *variable structure with no parameters*, and *variable structure with a variable number of parameters*.

- Fixed structure with a fixed number of parameters are standard optimization problems where the topology is fixed, and certain parameters are allowed to vary.
- Variable structure with no parameters are problems where only structure is important, such as algorithm design, program induction and logic design.
- Variable structure with a variable number of parameters are problems where both the structure and the parameters need to vary. Typically, a structure is sought within a topologically open-ended space, and parameters need to be assigned to key variables associated with the structure. The number of parameters and their semantics may change frequently. Examples, include circuit design, mechanical design and neural network design.

Hu et al. (2002) find that many of the most interesting types of problems fall into the last classification.

4.4.3 Reproduction, evaluation and selection rules

Reproduction rules

Alternative reproduction rules have been developed for both binary string representations and real-valued vector representations. For binary strings, two common alternative recombination operators are *n-point crossover* and *uniform crossover*:

- N-point crossover (Jong, 1975) is a generalization of one-point crossover by increasing the number of crossover points. For example, with $n = 2$, two random crossover point are chosen. Each parent string is then broken into three sections, and the middle sections are then swapped, thereby creating two new child strings. Two-point crossover is the most common type of crossover (Booker et al., 2000).

- Uniform crossover (Syswerda, 1989) does not predefine the number of crossover points in advance. Instead, each string position is considered in turn and a breakpoint is inserted with a predefined probability (usually 50%). The two parent string are then broken into sections and two new child strings are created by swapping these sections.

In addition to these recombination operators, a variety of other operators have been developed for different purposes. For example, recombination operators have also been developed that use more than two parents. Such operators combine genetic material from multiple parents to create multiple offspring. Booker et al. (2000) give an overview of these developments.

For real-valued vectors, new types of recombination and mutation operators have had to be developed. The mutation operators are similar to their binary counterparts. Two common types of mutation are *uniform mutation* and *non-uniform mutation* (Michalewicz, 1996):

- Uniform mutation applies mutation to each real value within the vector with a fixed probability. If mutation is applied, then the old value is replaced by a new random value uniformly selected from a range of possible values.

- Non-uniform mutation also applies mutation to each real value within the vector with a fixed probability. If mutation is applied, then the old value is replaced by a new value that is calculated by taking into account the age of the population. Early on during the evolutionary process, the new value is selected randomly from the range of possible values. However, as evolution progresses, the chance of the new value being different from the old value gradually decreases. The size of the mutations will get smaller as the population gets older.

For the recombination operators, the two main variants are *simple crossover* and *arithmetical crossover* (Michalewicz, 1996):

- Simple crossover is similar to binary crossover and breaks off the tail of each parent string at some random location between two real values, and then swaps these tails to create two new child strings.

- Arithmetical crossover creates a new real-valued vector by calculating the averages of the corresponding values in the parent vectors. The first value in the child vector will be equal to the average of the first values in the parent vectors, the second value will be equal to the average of the second values in the parent vectors, and so forth.

Both Holland (1975, p. 111) and Goldberg (1989, p. 14) emphasise that recombination using crossover should be the main type of operator,

with the mutation operator acting as a 'background operator' supporting recombination. This is why the mutation probability is usually set low. Recently, empirical and theoretical investigations have demonstrated that increased emphasis on mutation can result in performance improvements (Bäck et al., 2000a). Booker et al. (2000) give a formal analysis of the behaviour of various recombination operators.

Evaluation rules

When developing evaluation rules, a problem described as the *scaling problem* may arise. The selective pressure that is applied to a particular population of individuals is dependent on the variation of fitnesses of the individuals in the population. If there is little variation in the fitness of the individuals, then the selective pressure will be small. For example, if a function with a range between 1000 and 1010 is being optimised, then the selective pressure will always be low because the variation in fitnesses is always small.

A number of methods have been proposed to overcome this problem. Two common approaches are *windowing* and *scaling*:

- Windowing uses the worst fitness score in the population as the baseline, and subtracts this value from all other fitness values.
- Scaling attempts to map the fitness values to some new set of values that are less susceptible to stagnation and premature convergence. Two types of scaling are *sigma scaling* and *linear scaling*.

Scaling problems only occur with proportional selection; tournament selection or rank based selection — which will be discussed next — avoid these problems. For these selection schemes, the magnitude of the fitnesses makes no difference, as long as they are different. Furthermore, Whitley (1989) also argues that in many real applications, evaluation functions may only be able to provide approximate measures of fitness. He writes: "In most cases, it may not be realistic to use the value generated by the evaluation function to judge relative differences in fitness. Ranking (or even approximate ranking) may be the best one can expect from an evaluation function."

Anther potential complication that may arise when developing evaluation rules is when more than one objective needs to be evaluated. The majority of evolutionary algorithms have been applied to problems where only a single objective needs to be optimised. However, real-world problems often involve multiple objectives that need to be simultaneously considered. Furthermore, such objectives are often *conflicting* in that a solution that performs well in one objective may perform badly in another. In such a case, the objectives cannot all be simultaneously optimised, and a compromise solution must be sought whose performance is acceptable for all objectives. When an evolutionary algorithm is used in to search for such solutions, it is referred to as *evolutionary multi-objective optimization*.

If only a single objective is being considered, the evaluation function will produce a single scalar value, and the function is called a *scalar valued function*. If multiple objectives are being considered, the evaluation function is a composite function consisting of a list of separate objective functions, one for each objective. The evaluation of this composite function results in a list of scalar values called the *objective vector*. The evaluation function is referred to as a *vector valued function*.

Evolutionary algorithms require the objective vector to be transformed into a single number, referred to as *scalarization*. A distinction has been made between the evaluation function and the fitness calculation (see section 4.3.1 on page 94). In the context of multi-objective optimization, the evaluation function is a vector valued function, and fitness calculation is considered to consist of the process of scalarization of this vector.

The scalarization of the objective vector can be performed either within the evaluation step or the selection step. One option is to numerically combine the objective functions into a single evaluation function. However, different objectives are often *non-commensurate*: non-commensurate objectives measure fundamentally different qualities that cannot be meaningfully compared or combined. This has led to the development of alternative scalarization techniques that rank populations of solutions by comparing how solutions perform for each objective.

One commonly used approach is the Pareto-ranking method. With this method, one solution is said to *dominate* another solution if it performs better for one or more objectives, and performs worse for none. If a particular solution cannot be dominated, then it is described as being *Pareto-optimal*. A solution is Pareto-optimal is it is impossible to change it in such a way that all objectives are either improved or stay the same. Alternatively, Pareto-optimal solutions are solutions where improvement in one objective will always lead to degradation in another objective. In nearly all cases, multiple solutions will exist that are Pareto-optimal. This set of solutions is called the *Pareto-optimal set*. The plot of the objective functions whose vectors are in the Pareto-optimal set is called the *Pareto-front*. Multi-objective optimization methods often present the user with a range of alternative Pareto-optimal solutions to choose from.

Selection rules

For the selection step, the canonical genetic algorithms use *proportional selection*, where the probability of selection is proportional to the fitness of the individual[9]. A common method for implementing this form of selection is *roulette wheel sampling*, whereby each individual is assigned a slice of a roulette wheel, the size of the slice being proportional to the individual's fitness. On each spin of the roulette wheel, the individual under the wheel's marker is placed in the intermediate population.

[9]Holland's original genetic algorithm actually suggested only picking one parent according to fitness. If another parent was required, then it was picked at random.

Researchers have experimented with a variety of other selection methods. In particular, researchers have found that the selection methods provides a way to affect the balance between exploration and exploitation. Three other common approaches are *rank based selection*, *tournament selection* and *Boltzmann selection* (Goldberg and Deb, 1991; Bäck, 1994; Hancock, 1995).

- Rank based selection was first proposed for genetic algorithms by Baker (1985). Rank based selection orders all the individuals in the population according to their fitnesses and then uses their rank to calculate their selection probabilities. Two ways of making this calculation are *linear ranking* and *exponential ranking* (Hancock, 1995). A new main population can then be created in a similar way to proportional selection, by repeatedly selecting individuals using roulette wheel sampling.

- Tournament selection extracts a random sample of individuals from the population and then selects an individual from this sample. A new main population is created by repeatedly extracting random samples and selecting individuals. In some case, the fittest individual in the sample is always selected. In other cases, the fittest individual is selected with some predefined probability (not related to its fitness), thereby allowing less fit individuals a chance of selection. This method is easy to implement and is computationally efficient. In addition, the selective pressure can be easily controlled by increasing or decreasing the size of the sample, or by adjusting the probability parameter. An analysis of this approach was presented by Goldberg and Deb (1991).

- Boltzmann selection creates the new population using roulette wheel sampling in the normal way. However, Boltzmann selection attempts to vary the selective pressure as evolution progresses. A 'temperature' variable controls the rate of selection according to a pre-set schedule, with lower temperatures resulting in higher selection pressure. The temperature starts out high, thereby ensuring that even individuals with low fitness are selected. As the temperature is lowered, the selection pressure is increased and as a result only the fitter individuals are selected. For example, see Goldberg (1990); de la Maza and Tidor (1993).

Tournament and rank based selection approaches discard information about the magnitude of the differences between fitness values. As a result, these approaches may avoid two key problems associated with proportional selection. First, they may avoid stagnation due to lack of selective pressure when individuals all have similar fitnesses. Second, they may avoid premature convergence due to excessive selective pressure when a 'super' individual with unusually high fitness is created (Whitley, 1989).

4.5 Summary

This chapter has given an overview of the field of evolutionary computation. The main points are as follows:

- Two evolutionary architectures are the *general synchronous evolutionary architecture* and the *general asynchronous evolutionary architecture*. In both cases, a centralised control structure is used, whereby a cyclical process invokes and applies the evolution steps to individuals in the population. With the synchronous architecture, two populations are maintained and the evolution steps are used to repeatedly replace these populations. With the asynchronous architecture, a single population is maintained and the evolution steps are used to repeatedly replace individuals in the population with new individuals. The majority of evolutionary algorithms are based on the synchronous architecture.

- Genetic algorithms are the most common type of evolutionary algorithm. Other common evolutionary algorithms include evolution strategies, evolutionary programming, and genetic programming. Each type of algorithm can be described using the general synchronous architecture, thereby highlighting the differences and similarities between these algorithms.

- The evolution steps require rules and representations that define how individuals in the population should be processed. A wide variety of rules and representations exist. For real world problems, there is little theory to guide the researcher, and as a result researchers have experimented with many different types and combinations of rules and representations.

Chapter 5

Evolutionary design

Contents

5.1 Introduction

This chapter consists of five sections, each of which describes a different evolutionary design system or approach. In each case, certain key features are highlighted which are relevant to the proposed architecture described in chapter 7.

In chapter 1, two evolutionary design approaches were described as *parametric evolutionary design* and *generative evolutionary design*. The

first two systems discussed in this chapter are parametric evolutionary systems, while the last three are generative evolutionary systems and approaches.

- In section 5.2, a parametric evolutionary design system developed by Khaled Rasheed is described, called the *Genetic Algorithm for Design Optimization* (GADO). This is an optimization system for use in engineering design. The key feature of this system is its novel architecture specifically designed for situations where the evaluation step is complex.

- In section 5.3, a parametric evolutionary design system developed by Louisa Caldas is described, called the *Generative System*(GS). GS has been used to optimise aspects of building designs, and in particular has been proposed as a method for evolving designs for low-energy buildings. The key feature of this architecture is the integration of an existing simulation application called DOE-2.

- In section 5.4, a generative evolutionary design system developed by Peter Bentley is described, called *Genetic Algorithm Designer* (GADES). This system aims to be highly generic, and Bentley argues that it can be used in a wide variety of design domains. For this system, the key feature is the overall architecture, which provides a clear example of a generative evolutionary design system. It also highlights certain problems and weaknesses associated with creating a highly generic system.

- In section 5.5, a generative evolutionary design approach developed by John Frazer is described, called *concept-seeding*. In this case, the key feature of the concept-seeding approach is that it suggests a knowledge-rich approach to generating designs. This offers an alternative to the generic approach explored by Bentley in GADES. A prototype system using concept-seeding is also described.

- In section 5.6, another generative design approach developed by John Frazer is described, referred to here as *epigenetic design*. The key feature of this approach is that it offers an environmentally sensitive way of generating designs, whereby the designs that are generated differ depending on the environment within which they are generated. This generative method was developed independently of any evolutionary systems, but may be used in such systems in order to implement the development step.

5.2 GADO

5.2.1 Overview

Introduction

Rasheed (1998) has developed a parametric evolutionary design system, called the *Genetic Algorithm for Design Optimization* (GADO), for use in engineering design. GADO was briefly described in section 4.2.2 on page 88. The systems has an architecture with a number of novel features designed to increase its efficiency for design engineering problems, particularly where the evaluation step may involve computationally expensive simulation or analysis routines. It uses a number of new types of rules and search control strategies and has demonstrated a great deal of robustness and efficiency when compared with other systems.

Evolutionary algorithm

GADO uses a steady-state evolution mode, with individuals being added to the population one at a time. Rasheed and Davison (1999) developed a parallel version of GADO. Global parallelism was used where one master processor executes the main evolutionary process — including the reproduction and selection steps — and multiple slave processors perform the evaluation step.

Since evaluations may be complex and may involve computationally expensive simulations of design models, this approach was seen to potentially reduce the execution time by a significant factor. However, the steady-state evolution mode used by GADO is less susceptible to parallelization. When a generational evolution mode is used, the whole population must be evaluated each generation. As a result each slave processor can be assigned a sub-set of the population to evaluate. With a steady-state evolution mode, only one individual is created each generation, and as a result only one individual needs to be evaluated.

In order to allow GADO to be parallelised, the steady-state evolution mode had to be modified to allow more than one individual to be evaluated at a time. Rasheed and Davison (1999) therefore propose an algorithm whereby a new individual can be created even before the evaluation of the previous individual has been completed.

Slave processors are controlled by the master processor and wait to be sent a design model to be evaluate. The master processor continuously monitors all its slave processors. Each time a slave has completed evaluating an individual, the master will take two actions irrespective of the state of the other slaves. First, it will add the individual with its evaluation score to the population. Then, it will select two parent individuals from the population, create a new individual and send it to the idle slave for evaluation. This means that the slave processors will evaluate multiple new individuals in parallel.

This results in a more complex type of steady-state evolution mode

where, at any one point in time, a significant number of new individuals will have been created but not yet added to the population. The number of such new individuals is dependent on the number of slaves being used. (The number of slaves may be larger than the population size, in which case the number of new individuals under evaluation would be larger than the population.) For each individual, there is a delay between being created through reproduction and being available for reproduction as a parent. The size of this delay will depend on the slave processor performing the evaluation. Furthermore, in a heterogeneous computing environment — where slaves may have different processor speeds and memories — the size of the delay will vary from one slave to the next. This results in evaluated individuals being added to the population in a different order form the order in which they were created.

The algorithm used by GADO can be described according to the five steps of the general asynchronous evolutionary architecture[1] (See figure 4.5 on page 90).

- The selection step changes its behaviour as evolution progresses. The maximum number of generations is specified at the start of the evolutionary process. For the first 75% of generations, the selection step uses rank based selection, which is seen as promoting exploration and as a way of avoiding premature convergence. For the last 25% of generations, a more greedy form of selection is used in order to promote exploitation in the final stages of search.

- The reproduction step creates a single new individual each generation using a number of crossover and mutation operators. Some operators have been specifically developed for GADO and change their behaviour as evolution progresses.

- The evaluation step performs a simple simulation in order to calculate the fitness of each individual. This calculation may include a penalty function that reduces the fitness of individuals that violate certain constraints.

- The development step is not used.

- The survival step selects a single individual in the main population to be deleted using a crowding technique that takes into account both the fitnesses of the individuals in the population, and the similarity of the individuals in the population.

Reproduction step

Each individual in the population represents a parametric description of an artefact, where all parameters have continuous intervals. The geno-

[1]The general architecture can be used to conceptually describe how GADO functions, but it may not reflect how GADO was actually implemented.

type representation uses a floating point representation, which is seen as being superior to a binary representation.

The operators used in the reproduction step include five crossover operators and three mutation operators. For the crossover operators, two are standard operators: point crossover and random crossover. The other three operators are novel and have been specifically designed for the types of optimization problems considered by GADO. All three operators consider a genotype of an individual as representing a point in the multidimensional genotype search space. These operators then create new child points by performing geometric operations on the parent points. Of the three mutation operators, two are novel and one is the standard non-uniform mutation operator. The reproduction step will randomly select one of the crossover operators and one mutation operator, with a fixed probability of selection being assigned to each operator. These operators will then be used to create a new genotype.

Evaluation step

GADO also incorporates a technique that minimises the number of evaluations that are necessary by preventing individuals being evaluated that are similar to previously evaluated individuals with low fitness scores. This is performed by a *screening module*.

When a new individual has been created through reproduction, prior to evaluation the screening module will decide whether it is worthwhile evaluating it. The screening module considers the group of individuals in the population that are most similar to the new individual. If at least one of these neighbours has a fitness score higher than some threshold, then it is evaluated; otherwise it is discarded. The size of the group and the fitness threshold are parameters set at the start of the evolutionary process. The similarity of two individuals is measured using a distance metric that considers the differences at a genotypic level.

In order to ensure that the evolutionary process has had a chance to fully explore the search space, this module is only activated after the first 25% of generations have been processed.

Diversity maintenance

For problems that require complex and computationally expensive evaluation processes, the steady-state evolution mode is argued to perform better than the generational mode. This is because the evolution mode retains all the best individuals found during the evolutionary process and imposes a higher selection pressure. However, due to the increased selective pressure, premature convergence is identified as being particularly problematic. An important consideration is the maintenance of diversity in the population, particularly in the early stages of the evolutionary process. This is referred to as *diversity maintenance*.

The diversity maintenance module maintains diversity in two ways: first, it discards individuals created by the reproduction step that are

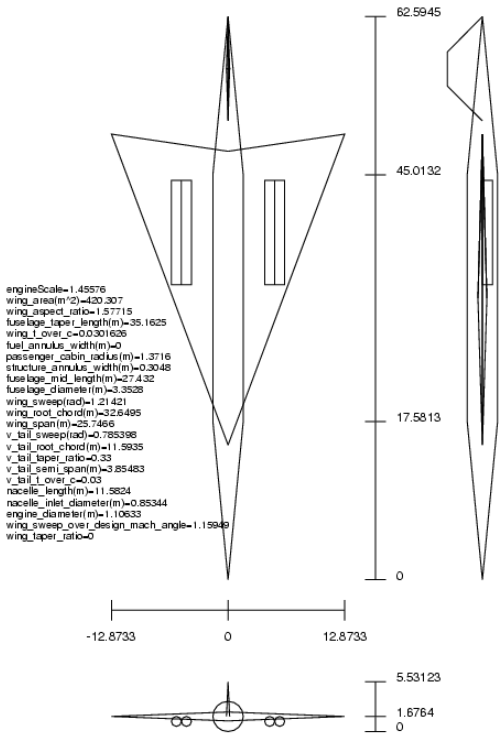

Figure 5.1: Optimization of supersonic aircraft

similar to existing individuals in the population. Second, if a severe loss of diversity is detected, this module will attempt to restore diversity by rebuilding the population using previously evaluated individuals.

In order to decide whether to discard a new individual, the diversity maintenance module finds its closest neighbour in the population using the distance metric. If the distance to the closest neighbour is under some threshold — referred to as the *rejection radius* — then the new individual is rejected. The rejection threshold is gradually decreased as evolution progresses, thereby allowing the process to converge towards the global optimum near the end of the search.

In order to decide whether to rebuild the population, the diversity maintenance module calculates the average distance between individuals in the population. If this average distance is below a certain threshold,the population is rebuilt. The rebuilding process discards all the individuals in the population except the best one and then adds previously evolved individuals to the population, with preference going to those individuals that have both a high fitness score and that are most dissimilar to the retained fittest individual.

5.2.2 Demonstrations

Supersonic Transport Aircraft Design

Application areas discussed in the context of GADO include the optimization of aircrafts and missiles. Rasheed and Davison (1999) describe the results from a series of experiments using GADO to optimise a simplified design for a supersonic transport aircraft. Figure 5.1 on the facing page shows certain parameters for the design of a supersonic aircraft being optimised by GADO.

Random populations were created, with each population containing 120 individuals. Each population was then evolved for 12,0000 iterations. In order to explore the affect of using different numbers of slaves, these populations were evolved using various numbers of slaves. The maximum number of slaves used was 100.

In addition, two slightly different evolution modes were tested. The first evolution mode is the approach described above: individuals may be evaluated at different speeds and may be added to the population in a different order to the order in which they were created. This mode is advantageous when using heterogeneous computing resources, since processor speeds and memory specifications of slaves are likely to vary. They refer to this type of evolution mode as *random return*. However, the affect that this would have on the evolutionary process was uncertain. With the second evolution mode, individuals were forced to be added to the population in the same order as they were created. This evolution mode they refer to as *ordered return*.

Conclusions from experiments

The type of parallelism used by GADO has the potential for close to linear speedup in cases where the evaluation step is computationally expensive. However, Rasheed and Davison (1999) suspected that the speedup that could be achieved would be degrade when large number of slaves were used and when randomised return evolution mode was used. They write: "We were concerned that as the number of slave processors becomes significantly large the deviation from the steady state model — in which a new individual is generated after the previous one has already been evaluated and possibly inserted into the GA population — may degrade the performance and make it more like random search."

The experiments that they performed showed that degradation of the linear speedup was quite limited and was only noticeable when the number of slaves was larger than the population size. In addition, experiments using the random return evolution mode performed as well as those using the ordered return mode.

5.3 GS

5.3.1 Overview

Introduction

Caldas has developed a parametric evolutionary design system, called the *Generative System* (GS), for use in architectural design, focusing on aspects related to the environmental performance of buildings. GS generates populations of alternative solutions from a parametric model and a set of rules and constraints defined by the designer. The system combines a genetic algorithm with a complex building simulation application, DOE-2[2] (Manual, 1993) used for performance evaluation.

The objective of developing GS is as a system to be used in the early conceptual design phase in order to explore alternative design solutions. In particular, it aims to assist designers in researching low-energy architecture solutions. Caldas and Norford (2001) write: "Solutions must not be interpreted as definite or optimal answers, but as diagnoses of potential problems and as suggestions for further architectural explorations, thereby building an innovative and promising interaction between architecture and computation."

Evolutionary algorithm

Genetic algorithms are applied as a search procedure to look for optimised design solutions in terms of thermal and lighting performance in a building. The genotype representation consists of a standard fixed length binary string, which is decoded into a set of values that are then applied to the variables in the parametric model. The evaluation step then evaluates the designs in terms of lighting and thermal behaviour using the DOE-2 application. The results from the simulations is then used as the fitness for the design, with the GA aiming to minimise this value. The GA searches for low-energy solutions to the problem under study.

A special type of genetic algorithm, called a *micro-GA*, is used. The main difference between a canonical GA and a micro-GA is the size of the population: while the former typically use population sizes of between 30 and 200 individuals, the latter use much smaller populations size, and in this case the population of only five individuals is used.

Due to the small population size, micro-GA's quickly converge to a solution. Each time the algorithm converges, the evolutionary process is restarted with a new random population except for the best individual from the previous run, which is allowed to survive. During the evolutionary process, many new random populations will be created, and as a result it is argued that the size of the initial population can be greatly reduced.

[2]DOE-2.1E ©, http://www.doe2.com/

Evaluation step

DOE-2 is a widely used and accepted building energy analysis program that can predict the energy use and cost for most types of buildings. It also performs a simplified form of lighting analysis, based on the daylight factor method. DOE-2 uses a description of the building layout, constructions, usage, and conditioning systems (lighting, HVAC, etc.), along with weather data, to perform an hourly simulation of the building and to estimate energy consumption.

DOE-2 was chosen to be the simulation engine because it performs both thermal and lighting calculations, and because it offers good accuracy for reasonable computation times. In addition, it includes a vast library of construction materials, glazing types, shading devices, HVAC systems, operation modes, control strategies, and so forth. However, Caldas identifies two drawbacks with DOE-2. First, it is a single-node system and cannot predict variations in air temperature at different points in a space. As a result, it cannot compute air movement and air flow patterns. Second, it cannot perform thermal comfort predictions.

Before evolution can start, one of the key tasks for the designer is to create an input file that describes the building and its mechanical and electrical systems. The design needs to be described using the Building Description Language (BDL) used by DOE-2. BDL is flexible enough to describe most type of building geometries, except curved surfaces, which have to be approximated by a number of planes. Surfaces such as walls, roofs and floors can have any orientation, and be tilted in any direction. BDL also allows light sensors to be placed in a space where light levels will be calculated.

The BDL file must include a set of codes that represent the variables that are to be evolved by the GA. Once the evolutionary process has started, the evaluation step in the GA makes a call to DOE-2 each time an individual needs to be evaluated. This call extracts the values encoded in the genotype and inserts them into the BDL file. Any dependent variables not directly encoded in the genotype must be calculated by the evaluation step. DOE-2 then runs the thermal and lighting simulation and returns the annual energy consumption, which then represents the fitness of the design.

Caldas (2001, p. 52) suggests that in future, other applications capable of performing Computational Fluid Dynamics (CFD), thermal comfort predictions (such as EnergyPlus[3] ©), and accurate lighting simulations (such as Radiance[4] ©) could also used. However, as Caldas states, "at the present time, and considering the several hundred evaluations a Generative System performs, including these types of programs would be infeasible, for in a standard personal computer one might have to wait unreasonable amounts of time to obtain results for a moderately complex building."

[3]http://www.eere.energy.gov/buildings/energyplus/
[4]http://radsite.lbl.gov/radiance/

Maver (2000) gives a brief review of important developments in building analysis and simulation. Issues relating to integration and interoperability of systems have been discussed by Bazjanac and Crawley (1997); Hensen (2002); van Treeck et al. (2003); Shea (2004).

5.3.2 Demonstrations

Optimization of facade design

One of the main demonstrations of GS has been the optimization of the sizes and positioning of window openings in a facade, and may in addition also optimise the sizes of overhangs that provide shading for windows. In this case, building geometry, spatial organization, and construction materials are left unchanged. A single objective was considered, consisting of reducing the annual energy consumption, which incorporates both space conditioning and lighting.

One of the building blocks — Tower H — of Alvaro Siza's School of Architecture in Oporto was used as a test case for demonstration. Tower H was chosen due to the rich spatial configurations and variety of architectural light sources. Facades include windows of different proportions and sizes facing distinct orientations, and in some cases have overhangs that provide shade. In addition, the top floor includes roof lights. The tower houses mainly studio teaching rooms, and the control of natural light is an important consideration.

For each space, two light sensors were defined and desired illuminance values were specified according to the type of occupation. The artificial lighting system was assumed to be continuously dimmable, thereby allowing for the quantification of savings in artificial lighting due to natural daylight. The DOE-2 application calculates the annual energy consumption of the building, taking into account the climate for that geographical location.

By studying the original designs by Siza, Caldas derived a set of rules relating to the composition and proportion of the openings. These rules were then used define a set of windows with variable dimensions, bounded by maximum and minimum values. Other constraints related to the composition and proportion of the windows.

Figure 5.2 on the next page shows a set of alternative facades evolved by the system. These facades were evolved for the climate in Oporto. As an experiment, the system was also used to evolve facades for the same building with climate data for Phoenix, Arizona and for Chicago, Illionis. In the Phoenix climate, cooling is dominant, while in the Chicago climate, heating is dominant. These experiments were performed as an academic exercise to compare the results from the system. Caldas concludes that the range of solutions developed by the system for different geographical locations show that the system is able to adapt the design to the local climate.

Figure 5.2: Alternative facade solutions generated by GS for one block of the School of Architecture in Oporto by Álvaro Siza. From (Caldas and Norford, 2001).

Optimization of building geometry

In a number of other demonstrations (Caldas, 2002), the building geometry has also been allowed to change in a highly restricted way. In these cases a parametric model of the building is defined that predefines all adjacencies between spaces. In one example, the model specifies a simple spatial topology consisting of two floors, with four spaces on each floor arranged in a square grid. On the first floor, the four spaces can vary in their length and width, but the height is constrained to be the same. On the second floor, the spaces may also have variable heights. In addition, the roof may also tilt. Walls could have windows of variable height that ran the full length of the wall. This resulted in a variety of building forms being generated and evolved. Figure 5.3 on the following page shows a selection of building forms evolved by GS.

In this case, the building forms that were generated had different volumes. Since buildings with small volumes tend to consume less energy, the volume or area of the building has to be included in the evaluation process. Caldas (2002) experimented with two approaches: first, the buildings with small area could be penalised by reducing their evaluation score. Second, instead of calculating total energy consumption, the energy per unit area could be calculated. Caldas concludes that the second approach is preferable.

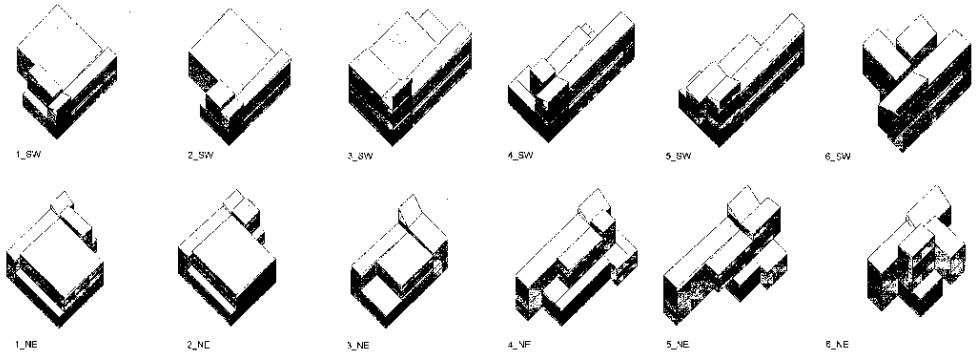

Figure 5.3: Alternative building forms generated by GS. Top row is viewed from the south west, and the bottom row is viewed from the north-east. From (Caldas, 2001, p. 256).

Multi-criteria optimization

Pareto multi-criteria optimization techniques have also been used (Caldas, 2001; Caldas and Norford, 2004). The two conflicting objective functions considered were maximising daylighting use and minimising energy consumption for conditioning the building. These experiments were done using the Chicago climate data, as heating and daylight are elements that conflict. Large openings result in more daylight but also increased heat loss. The system was used to evolve design that reflected the best trade-off between these two conflicting objectives.

GS generated a uniformly sampled, continuous Pareto front, from which a number of designs were selected for visualization. The designs shown in figure 5.3 are a set of such Pareto-optimal designs. The first design represents the best building shape in terms of heating, and the last design is the best building shape in terms of lighting. Intermediate design represents varying trade-offs between heating and lighting.

The experiments performed by Caldas show that the system could simultaneously consider multiple objectives and could take into account interactions between different elements in the building design. The design of a specific element is dependent on its integrated role in the design with respect to the objectives under consideration.

5.4 GADES

5.4.1 Overview

Introduction

Bentley (1996, 1999a) has developed a generative evolutionary design system, called *GADES* (Genetic Algorithm Designer), that he claims is a generic system that can be used for many different types of design. The system evolves designs represented as solid models using a modified

genetic algorithm that incorporates specialised genotype representation, a specialised design model representation, and a complex generative process that creates models from genotypes.

Bentley (1999a) claims that GADES can easily be configured to evolve designs for a wide variety of design problems and that — once properly configured — it can consistently evolve good designs without human intervention. (These claims are seen to be highly optimistic, and will be discussed in more detail in chapter 7.)

Bentley emphasises the generic nature of the system, claiming that the system is able to evolve designs 'from scratch'. The term *from scratch* is used to indicate that little design knowledge needs to be provided. Bentley claims that the default configuration for the system is able to deal automatically with various complex issues such as multiple objectives and design constraints. In particular, the main input required from the user is an initialization file that specifies which evaluation modules should be used in order to evaluate designs.

Evolutionary algorithm

The evolutionary algorithm used by GADES can be described using the general synchronous architecture shown in figure 4.1 on page 81.

- The survival step allows a portion of the individuals in the main population to survive, which are added to the intermediate population. An elitist replacement strategy is used[5], whereby the fittest individuals are allowed to survive.

- The reproduction step creates new individuals by randomly selecting parents form the fittest 80% of individuals in the main population, and performing crossover and mutation using specialised operators. The new individuals are added to the intermediate population. New individuals are created until its size is equal to the main population.

- The development step creates design models for the new individuals in the intermediate population using a complex generative process[6]. This process uses information contained in the genotypes to create a solid model of a design.

- The evaluation step evaluates the new individuals for multiple objectives. A set of custom-written evaluation modules are used that independently evaluate different objectives. Modules were developed that could evaluate design objectives related to size, mass, surface area, stability and aerodynamics.

[5]Bentley (1996, p. 120) describes the replacement strategy as a steady-state replacement strategy. Since a large number of individuals are replaced every generation, this strategy is more accurately described as an elitist strategy.

[6]Bentley refers to the generative process as an *embryogeny*.

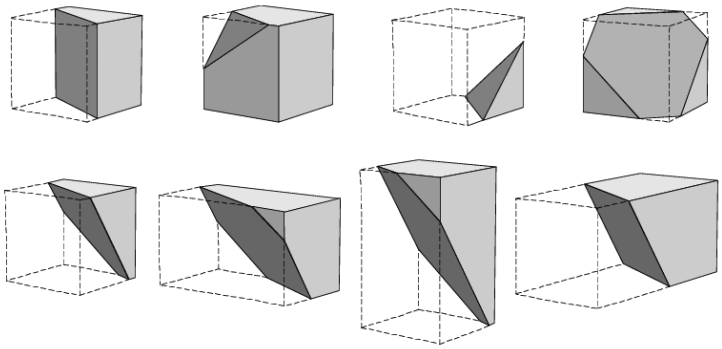

Figure 5.4: Examples of clipped stretched cubes used in the spatial partitioning representation in GADES. From (Bentley, 1996, p. 56)

- The selection step then aggregates these evaluations into a single fitness value, using a specially developed technique called *Sum of Weighted Global Ratios* (SWGR). This technique converts the evaluation score for each objective into a ratio, using the best and worst evaluations for all generations. These ratios for the different objectives are then weighted and summed to provide a single overall fitness value for each individual. The weights reflect the relative importance of each objective, and must be provided by the user. Once fitness values have been calculated for all individuals in the population, the selection step copies all individuals into the main population.

Developmental step

The generative process used in the developmental step must be able to generate solid models that vary significantly from one another. A variety of representational schemes were explored for the solid models, focusing on the number of parameters required in order to define the models. Bentley argues that, for evolutionary design, representations that require few parameters are preferable since this reduces the size of search space. For the final system, a low-parameter representation using a specialised spatial-partitioning representation was developed that defines forms by assembling primitive solids.

The primitives from which solid models are created are referred to as a *clipped stretched cubes* or *blocks*. Figure 5.4 shows example of these blocks. A block consist of a solid cube that can be stretched to form an orthogonal solid of any size, and that can optionally be clipped along one plane. The clipping plane will in effect slice of part of the solid. This allows each block to have a clipped face that is not orthogonal, thereby allowing surfaces at non-orthogonal orientations to be approximated. An important restriction is that the blocks in a solid model must all be non-overlapping.

The generative process creates forms by placing blocks in space relative to the global origin. The position and size of each block is controlled by nine parameters that are encoded in the genotype. The genotype uses a hierarchical representation consisting of *blocks* and *genes* that encode the parameters. Each genotype consists of a variable number of blocks, and each block consists of nine genes. A gene is represented as a 16 bit string that encodes one of the parameters used to define the position and size of a block. The generative process generates a solid model by simply decoding the parameters in the genotype and placing blocks independently from one another.

Since the blocks are placed independently from one another, they may overlap. The generative process uses a repair procedure in order to correct such overlaps. For each pair of overlapping blocks, the process resolved the illegal overlap by reducing the size of the offending clipped stretched cubes until the overlap disappears.

Reproduction step

The reproduction step uses specialised mutation and crossover operators in order to create new genotypes.

- The mutation operator adds or deletes blocks in the genotype. In order to add a block, the operator splits a randomly chosen block into two. In order to delete a block, it simply randomly selects a block and deletes it.

- The crossover operator is similar to the standard one point crossover operator. However, the operator must ensure that the genotype produced by crossover has a valid hierarchical structure. The operator therefore searches the parent genotypes for crossover points that will result in valid child genotypes.

5.4.2 Demonstrations

Generic approach

Bentley argues that GADES can be used in many different domains, to solve many different types of problem. Bentley (1999a) writes, "Most evolutionary design systems do not follow the example provided by natural evolution... Natural evolution... uses the same genetic machinery to generate everything from bacteria to blue whales (Dawkins, 1986). In order to explore the advantages and disadvantages of this highly generalised natural evolutionary approach, GADES makes no distinction between stages of design and is not limited to a single type of design. For every type of problem presented, the system simply evolves the form of new designs from random blobs to optimised shapes." He then demonstrates how GADES has been used to evolve a variety of designs, 'from coffee tables to hospital layouts'.

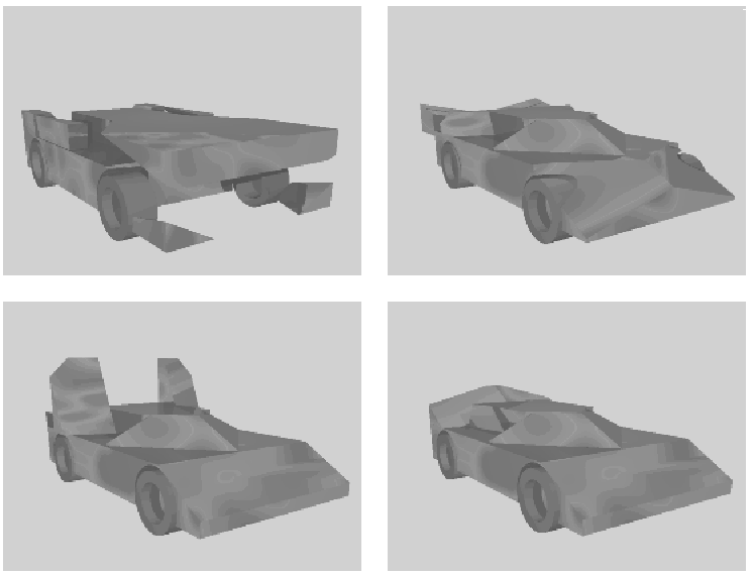

Figure 5.5: Examples of sports car designs at different stages of evolution. From (Bentley, 1996, p. 205)

Example of designs generated by GADES

Bentley (1996, 1999a) has tested GADES for sixteen different design problems from different fields. These include the evolution of tables, sets of steps, heat-sinks, optical prisms, streamlined designs (train fronts, boat bowls, boat hulls, saloon cars, sports cars) and two-dimensional floor plans for hospital layouts.

Figure 5.5 shows four sports car designs at different stages of evolution: top left represents a random design, top-right is the best design after 20 generations, bottom-left is the best design after 200 generations, and bottom-right is the best final design after further evolving the rear of the car.

Figure 5.6 on the next page shows four table designs evolved by the system.

5.5 Concept-seeding

5.5.1 Overview

Introduction

Frazer (1995b) has proposed a generative evolutionary design approach that differs fundamentally from previous approaches in terms of the role of the designer. Many researchers in the generative evolutionary design field first focused on evolutionary systems and later enhanced these systems with generative capabilities. Frazer followed an opposite path; he first focused on generative systems (Frazer, 1974), and then enhanced

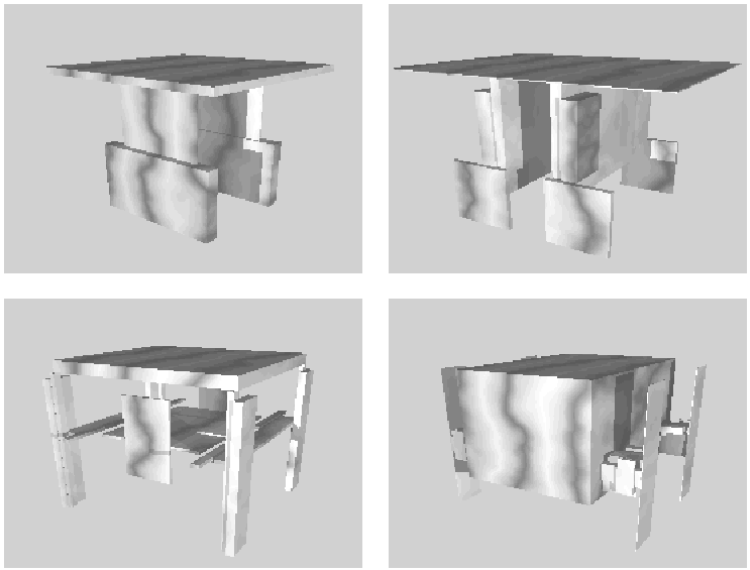

Figure 5.6: Examples of table designs. From (Bentley, 1996, p. 173)

these systems with evolutionary capabilities (Frazer, 1995b). The generative approach developed by Frazer first requires the designer to capture and codify a set of design ideas. These encoded design ideas can then be used by a computer program to generate alternative designs that all embody the design ideas.

This approach enables "the designer to crystallise a generalised design concept which embraces formal, structural, constructional, constructional, aesthetic and other considerations. The program then allows this concept to be manipulated into specific building forms in response to a particular problem" (Frazer and Connor, 1979). This approach was described as *concept-seeding*, with the design ideas being captured as a *seed*.

Generative concept-seeding

A key idea embedded in the concept-seeding approach is that designers must necessarily be an active participant in the software creation process. Whether designers are actually programmers, or whether they collaborates with programmers, these designers must define the design ideas that will be encoded as generative rules. Furthermore, it is suggested that the process of creating rules and generating forms will lead to ideas for new types of rules, resulting in positive creative feedback between the system and the designer. The role of the designer is not limited to simply specifying design rules, which a programmer then implements.

The importance of design ideas in the design process has been discussed in chapter 2. Frazer refers to a similar idea discussed by Sullivan (1967), where he described the development of a 'germ'. Frazer starts his book, *An Evolutionary Architecture* with a quote from Louis Sullivan:

> "... a typical seed with two cotyledons. The cotyledons are specialised rudimentary leaves containing a supply of nourishment sufficient for the initial stage in the development of the germ. The germ is the real thing; the seat of identity. In its delicate mechanism lies the will to power: the function of which is to seek and eventually to find its full expression in form. The seat of power and the will to live constitute the simple working idea upon which all that follows is based..." (Sullivan, 1967).

In the concept-seeding approach, the seed does not contain the generative rules. Instead, this approach requires two types of information to be encoded: first, generative rules need to be codified that can develop a concept-seed into a design; second, a concept-seed needs to be codified that captures a set of design ideas. These are then used by the generative program to generate a design in response to the design environment.

The concept-seeding approach is, in itself, not cyclical. The generative program creates a single design from a single seed in response to the design environment. However, the assumption is that the designer will explore a wide range of design possibilities by making small generative modifications to either the concept-seed or the generative rules. The results is a cyclical process guided by the designer.

Evolutionary concept-seeding

The concept-seeding approach was originally developed as a generative rather than an evolutionary approach. The system generates a single design proposal from a single seed in response to the design environment. However, early on, Frazer hinted at possible development of some kind of automated procedures that were able to improve the forms being generated based on previous experience. Frazer (1974) writes: "Alternative strategies for cultivating the seed are automatically evaluated and the program adjusts itself on a simple heuristic basis to adopt those tactics which have proved most successful in previous attempts". This eventually led the idea of embedding the generative approach in a computational evolutionary system that was able to evolve alternative seeds.

The parametric and the generative evolutionary design methods were discussed in chapter 1, and shown in figures 1.1 and 1.2 on page 12.

The generative concept-seeding approach may be combined with the generative evolutionary approach, resulting in a new type of design method that integrates the advantages of both previous methods. Such an approach was first proposed in (Frazer, 1990), and was further developed in (Frazer, 1995b). This is referred to here as the *evolutionary concept-seeding method*. Figure 5.7 on the next page summarises the main stages of this method.

- The generative concept-seeding method allows designs to be generated that embody particular design ideas. However, the method

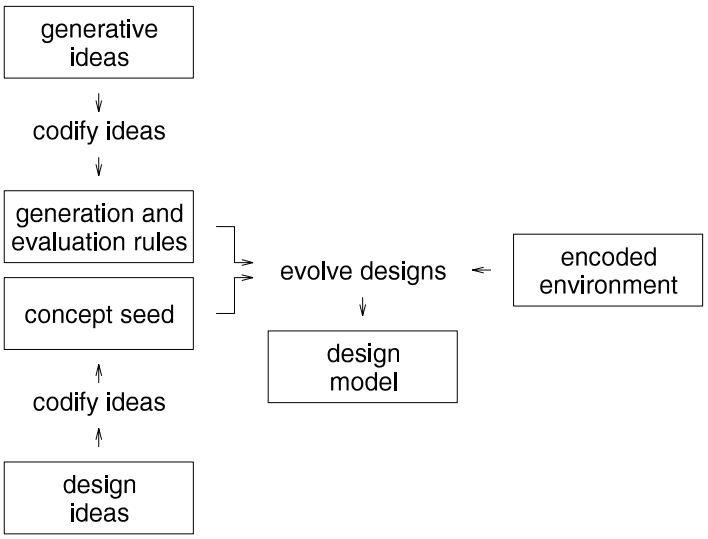

Figure 5.7: The evolutionary concept-seeding design method.

does not use an automated exploration or feedback system, and as a result it is up to the designer to discover the generative modifications that produce the most suitable design.

- The generative evolutionary method allows designs to be evolved that are adapted to their environment in complex ways. However, the designs do not embody any design ideas and the designer has limited control over the types of designs that are created.

The evolutionary concept-seeding method synthesises these two previous methods, thereby allowing designs to be evolved that — as well as being adapted to their environment — also embody particular design ideas.

Frazer (1995b, p. 65) summarises this method as follows: "The evolutionary model requires an architectural concept to be described in a form of 'genetic code'. This code is mutated and developed by a computer program into a series of models in response to a simulated environment. The models are then evaluated in that environment and the code of successful model used to reiterate the cycle until a particular stage of development is selected for prototyping in the real world."

Figure 5.7 shows the evolutionary concept-seeding method. The method includes three stages: codifying generative concepts, codifying design ideas and evolving designs. First, generation and evaluation rules are defined. In this case though, the generation rules must generate designs from the seed rather than from the genotypes. Second, the concept-seed is defined that codifies a set of design ideas. Third, the design alternatives are evolved in response to the design environment. This last stage requires a generative evolutionary system that includes concept-seeding.

The genotypes encode the generative modifications. These modifications will make small changes either to the concept-seed itself or to the

Figure 5.8: The two basic structural units of the Reptile System.

generative rules that transform the seed. These generative modifications result in different designs being produced and evaluated. The generative modifications that result in designs with the highest fitness scores are selected. The genetic operators are then used to create a new population of generative modifications, which will be used to generate a new population of designs, and so forth.

5.5.2 Demonstrations

Reptile system

The first attempt to realise a generative concept-seeding approach (without evolution) was the development of a generative program to create space frame enclosures. From 1966 onwards, Frazer (1974); Frazer and Connor (1979) designed an alternative type of space frame system that he referred to as the "Reptile System". This space frame system was capable of creating a wide variety of enclosures from just two basic structural units, shown in figure 5.8. These units could be connected together in over three hundred different ways to form the skin of the enclosure[7].

The generative Reptile program embodied a set of context-sensitive rules that were based on the geometry of the two units and the various ways that these units could be combined. This allowed the program to take a seed enclosure and to increase its size by using the inbuilt rules and by extrapolating from the existing structure. This could be done with minimal human intervention.

Once a larger enclosure had been developed, it could be further manipulated in an interactive way by applying high-level commands described by terms such as 'stretch' and 'bend'. These commands would not stretch and bend the individual units but would stretch and bend the overall enclosure, inserting and deleting units as and where required. In this way, enclosures with complex geometries could be created that were guaranteed to be valid in both a formal and a constructional sense.

[7]Drawing a Reptile enclosure by hand (which Frazer refers to as 'traditional T-square design') was tedious, and as a result Frazer developed a computer program. A program was developed in 1967 to aid in the process of drawing the enclosures and creating perspective views. (In 1967 the computer hardware was extremely limited in terms of speed, memory, and graphical output capabilities. Much of the programming effort focused on optimizing how these enclosures were stored and manipulated.) However, the program was soon enhanced with additional features, and by 1971 a generative program was developed that was able to automatically generate complete Reptile enclosures.

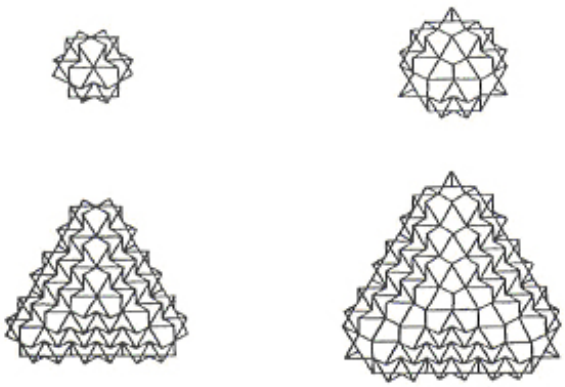

Figure 5.9: Enclosures growing from two different seeds

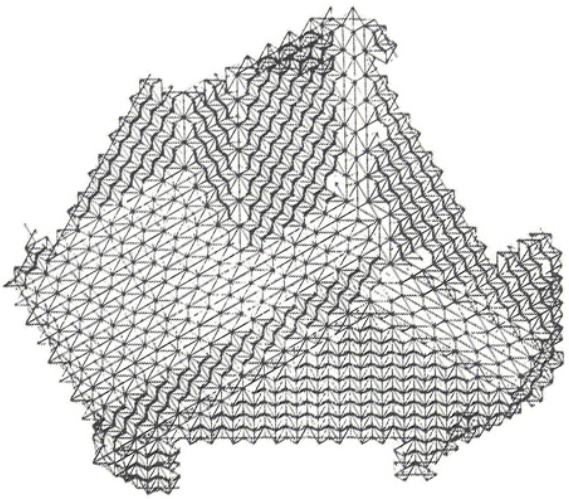

Figure 5.10: Plan of building generated from star seed

Generalization of the Reptile approach

Frazer generalised the approach developed with the Reptile program to encompass more general-purpose systems of building construction. The seed was conceptualised as a compact three-dimensional object that embodied all the key configurations such as a corners or a changes in direction. The elements in the seed were not necessarily a building component as was the case with the Reptile program, but were more usually assemblies of components. The seed may then be used in order to develop — either interactively or automatically — designs for buildings that embody the key configurations in the seed.

Once a designer has developed a particular seed, Frazer imagined that the enclosure would be developed in two steps. In the first step, the enclosure is — as far as possible — automatically generated in response to quantifiable defined in the brief. In the second step, the user evaluates

Figure 5.11: Output from evolutionary system using the evolutionary concept-seeding method.

a series of solutions produced by the first step in terms of non-quantifiable criteria such as aesthetic judgements.

First prototype system

Although Frazer (1995b) has demonstrated various techniques that could be applicable in the evolutionary concept-seeding approach, no complete implementation of an evolutionary system incorporating concept-seeding was developed.

The first generative evolutionary design system[8] that explicitly incorporated the idea of concept-seed was developed by Sun (2001), one of Frazer's Ph.D. students (Frazer et al., 1999). Sun developed an evolutionary system to support product design, and tested the system for designing mobile phones, remote controllers and other hand-held products. Figure 5.11 shows a family of mobile phones that were evolved using her evolutionary system.

Rudiments and formatives

Sun (2001) decomposed the idea of a concept-seed into two new representations, referred to as *rudiments* and *formatives*.

[8]Caldas (2001) discusses an approach to generating and evolving designs that is similar to the concept-seeding approach. One of the key elements in her design method (Caldas, 2001, p. 9) is a set of initial design rules, formulated by the architect, that define elements, variables, constraints, compositional rules, and so forth. This approach seems to have similarities to the concept-seeding approach developed by Frazer. Caldas (2001, p. 41) cites the work of Frazer (1995b) as providing some of the background for the ideas in her thesis. However, the systems implemented by Caldas only incorporate design ideas in a highly simplistic form, primarily embedded in a parametric model and constraints related to dimensional parameters. The systems discussed by Caldas are not seen to incorporate the concept-seeding approach.

- A rudiment is a type of sub-component in a design, built up from geometric primitives consisting of simple solids and surfaces. A rudiment will incorporate a set of parametrised features.

- The formative consists of a set of rudiments and a set of configurational rules that define how the rudiments may be combined.

Both rudiments and formatives are seen to embody design knowledge. In particular, a key area of design knowledge considered in this research was knowledge about the manufacturability of the products.

Sun (2001, p. 52) writes: "A formative is an encapsulated potential design solution, which defines a set of entities and relations, as well as the generative rules involved during the generative process. A rudiment is a composition element of the formative, which defines a set of entities and related design knowledge. In the domain of product design, a potential product design solution corresponds to a formative and it contains the constitutional parts of a product structure, the relationship of these parts, and the configuration rules to build the product embodiment."

Frazer makes a distinction between the seed and the rules that develop the seed into a design. With the Reptile program, Frazer demonstrated the feasibility of using different seeds with the same set of rules. This allows different design models to be generated by mutating the seed. Sun (2001) has taken a slightly different approach. In her case, configurational rules are encapsulated in the formative.

In the system developed by Sun, the designer must define two types of information. First, a set of rudiments must be defined that consist of primitive components with a set of parameterised features. For hand held devices these may include a button, a screen, a piece of casing, and so forth. Second, a set of configurational rules must be define that constrain how the rudiments may be combined. For example, the rules may specify a particular relationship between the button rudiment and the casing component. Together, the rudiments and the rules constitute the formative.

The evolutionary system then evolves both the configuration of rudiments and the parameter values for the features associated with those rudiments. The genotype must encode both the configurational information and the parameter values. In order to achieve this, the genotype uses a hierarchical variable length representation that mirrors the configuration of the rudiments in the design. The first level of the hierarchy encodes a list of rudiments; the second level encodes the features associated with these rudiments; and, at the third and final level encodes the parameter values for each feature. The developmental step then uses a generative process in order to generate design models by instantiating the rudiments encoded in the genotype.

Integration with existing CAD application

Sun (2001) has developed a prototype evolutionary system. This system

has a database of rudiments, an interface for building up the formatives, and an evolutionary system that enables designers to explore and evaluate a large number of design solutions in an interactive manner.

This system is integrated with a CAD application in order to allow different designs to be developed, evaluated and visualised. Initially, AutoCAD[9] ©was used and programs were written in AutoLisp ©(Sun et al., 1999). The final system was integrated with MicroStation/J[10] ©(Sun, 2001, p. 118–120) due to its sophisticated modelling capabilities and programmability using MicroStation Basic, C and Java. By integrating the evolutionary system with such a CAD modelling application, the generative process in the developmental step can make use of complex geometric functions. For example, MicroStation uses the Parasolid modelling kernel, that includes extensive surface and solid modelling functions.

5.6 Epigenetic design

5.6.1 Overview

The environment and the developmental step

The role of the environment in evolutionary design is usually restricted to the evaluation step. In such a case, the developmental step will generate a design based purely on the genetic information contained in the genotype. The evaluation step then analyses and simulates the design within a virtual environment. This environment will model significant aspects of the physical environment in which the design will finally be used. This role of the environment in the evaluation step is valid. However, research by Frazer has repeatedly emphasised that environment may also play a critical role in the developmental step. If the developmental step uses a generative process, rather than a simple mapping process, then the environment may affect the way a genotype is developed into a design model.

Frazer (1995b, p. 65) writes: "In order to create a genetic description it is first necessary to develop an architectural concept in a generic and universal form capable of being expressed in a variety of structures and spatial configurations *in response to different environments*"(emphasis added). This statement includes two key ideas. First, the idea of capturing an architectural concept — described as concept-seeding — has been discussed above. The second key idea relates to the role of the environment.

Frazer (1995b) describes generative processes that are affected by the environment as *epigenetic processes*. With an epigenetic growth process, the final form of the design model is partly influenced by genetic factors, and partly by environmental factors. Epigenetic processes highlight the

[9]http://www.autodesk.com
[10]http://www.bentley.com

fact that the genotype cannot be seen as a blueprint for a design, but must instead be seen as a set of triggers that initiate a growth process that is sensitive to its environment.

Epigenetic generative processes

A variety of generative processes created by Frazer and his researchers developed forms in response to the environment. Solar geometry was one area of particular focus. They also developed a variety of evolution programs that manipulate abstract forms. Form is typically represented and manipulated as cellular structures that inhabit an infinite three-dimensional spatial grid. As a result, the grid can be used to represent both the new form and existing forms that are part of the environmental context. The grid also allows the substitution approach to be effectively used to grow new structures, with genetic rules being used to add and remove cells. However, the genetic rules are sensitive to the environmental context. This means that the growth process is affected by both internal genetic factors and external environmental factors.

For example, in one program by Rastogi (Frazer, 1995b, p. 89), abstract surfaces are generated that represent the interaction between an evolving structure and the environmental context. Both the evolving structure and the environmental context are represented in exactly the same way, as structures constructed of spherical cells arranged in a cellular grid. Frazer and Rastogi (1998) write: "These evolutionary models exist entirely in the computer and comprise 'seeds' of coded descriptions that can divide and multiply. A seed inhabits dataspace and its growth depends on the environment and its own genetic code. Each growth of a seed produces a change in the environment. Certain cells and cell clusters, depending on their place in the data structure and relationship with other cells, may prove to be more adapted to their new environment and will survive while others will perish."

Iso-space

Frazer (1992) has proposed a non-orthogonal cellular grid that is based on the close packing of spheres in space as a way of representing both structure an environment. This grid is described as *isospatial*, in that each cell has neighbours that are all equidistant. This is not the case with the orthogonal grid, where each cell has three type of neighbours: six face neighbours, twelve edge neighbours, and eight vertex neighbours. Frazer writes: "An alternative (and preferable) geometry uses the centrers of close packing spheres (cubic faced-centred packing). This gives each [cell] twelve neighbours of identical geometric conformation and equal centre-to-centre distance. The reduction in number, equidistance between centres and ease of identification in the database all provide significant advantages for certain applications."

Representing the evolving structure and the environment in the same isospatial grid allows the evolving structures to easily interact with the

environment. In addition, cellular-automata based growth processes can then be used to generate alternative structures. A number of generative programs that use isospatial cellular automata have been developed by Frazer and his students. In these cases the transition rules consisted of an antecedent pattern that described the state of the centre cell and its twelve neighbours, and the consequent pattern described the new state of the centre cell. In total, this allows for $2^{12} = 4096$ rules.

Levels of adaptation

The environment may include both design constraints and the design context. The generative process may then create designs that fulfil certain constraints, and may also adapt to certain aspects of the design context. The evolutionary concept-seeding method may be seen to incorporate a number of different levels of design adaptation. These may be described as *concept-seed adaptation*, *generative adaptation* and *evolutionary adaptation*.

- Concept-seed adaptation involves the designer adapting the concept-seed to particular design tasks. The design ideas embedded in the concept-seed will be adapted to certain types of design environments.
- Generative adaptation involves the concept-seed being developed into a design model in a way that is sensitive to information about the environment. This requires the environmental information to be encoded in an appropriate format.
- Evolutionary adaptation involves the evolutionary system adapting the population of designs to the encoded environment. In this case, the evolutionary system does not attempt to adapt individual designs, but adapts the population as a whole by favouring genotypes that result in well adapted designs.

5.6.2 Demonstrations

The Interactivator

Frazer (1995b,c,a); Frazer et al. (1995b,a); Frazer and Rastogi (1998) and his researchers developed a generative evolutionary system, known as the *Interactivator*[11], that emphasised the role of the environment.

A precursor to this experiment was an experiment by Frazer and Graham (1994) where an evolutionary program was used to evolve cellular automata transition rules that tended to generate complex structures. In this case, an isospatial cellular automata was used to grow forms, with

[11]The author was a student at the Architectural Association and involved in developing the Interactivator together with Manit Rastogi and Peter Graham.

more complex forms being allowed to survive. The forms being generated consisted of a cellular structure of close packed spheres.

The Interactivator was part of a generative evolutionary experiment launched to involve global participation in the evolution of a family of abstract structures. The experiment was at the centre of the exhibition entitled An Evolutionary Architecture. It was the work of Frazer, his wife and their students at the Architectural Association, and the School of Design and Communication at the University of Ulster.

The Interactivator evolves the transition rules for an isospatial cellular automata. In this case, the genetic information is stored in *chromosomes* rather than genotypes. A chromosome encodes a single transition rule. Each generation, the developmental step uses a population of chromosomes — described as the *dominant chromosome pool* — to generate one structure. The developmental step uses a cellular automata to simulate growth by cellular division. Cellular structures are generated, where each cell contains a copy of all the chromosomes. Growth starts from a single cell, and this cell divides and multiplies, forming a complex cellular structure. Two types of form — cellular configurations of spherical cells and organic 'skinned' forms — are generated representing different aspects of the generative process.

Frazer writes: "As cellular division takes place, unstable cells are generated. In the next generation this leftover material creates a space of exclusion in the cellular space, which in turn interacts with the physical environment to create a materialization of the model. Boundary layers are identified in the unstable cells as part of their state information and an optimised surface is generated to skin the structure. This material continues to exist throughout the evolution of the model and will initially affect the cellular growth of future generations."

Local and global environment

The cellular growth process used by the Interactivator is an epigenetic process in that it is influenced by the environment. Cell division is based on two types of information: the chromosomes contained in the cell and the environment outside the cell. This environment includes two types of environment, referred to as the *local* environment and the *global* environment (Botsford, 1995): the local environment consisted of neighbouring cells in the virtual space in which the cells were dividing; the global environment consisted of the physical space of the exhibition, where changes in temperature, light levels, noise levels and movement were recorded using a set of sensors.

A chromosome transition rule consists of a typical substitution rule, with an antecedent and a consequent. A chromosome consists of the following 5 parts:

```
[person@address.com][1011010*11**][000011011010][1][192.5]
```

The five parts are decoded as follows:

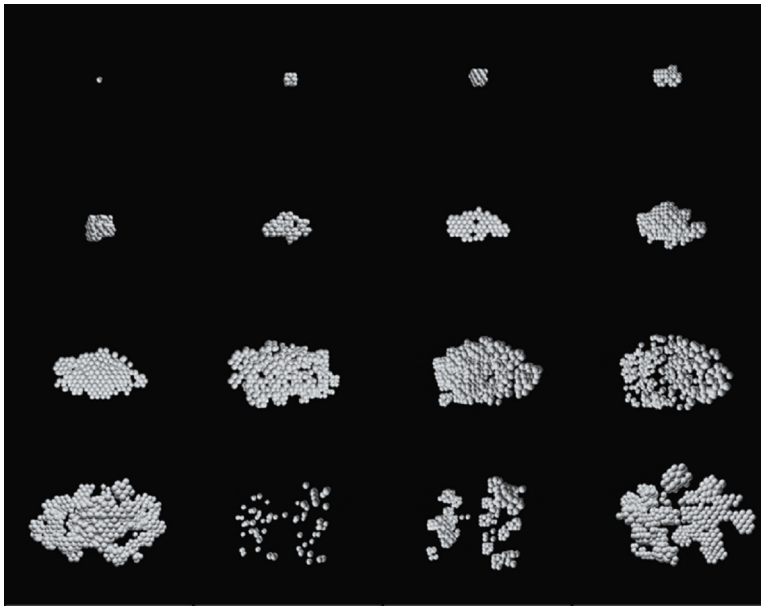

Figure 5.12: The Interactivator: the process of cellular division and multiplication.

- Origin of chromosome: The unique address from where the chromosome is received.
- Antecedent condition: The local environment of a cell (* being a don't care situation)
- Consequent action: The state of the cell in the next generation.
- Flag: Whether a chromosome is dominant or recessive.
- Strength: Fitness of the chromosome with respect to the environment.

Developmental step

The developmental step uses a generative process that consists of three parts, referred to as *cellular growth*, *materialization* and the *genetic search landscape*:

- Cellular growth: Chromosomes are generated by being sent in by a remote user, an active site or are created by the reproduction step. The global environment determines which chromosomes become dominant, thereby being included in the dominant chromosome pool. The local environment of each cell then determines which chromosomes become active. The cell multiplies and divides in accordance with the consequent action specified by the active chromosomes. Figure 5.12 shows a cellular structure dividing and multiplying from one cell.

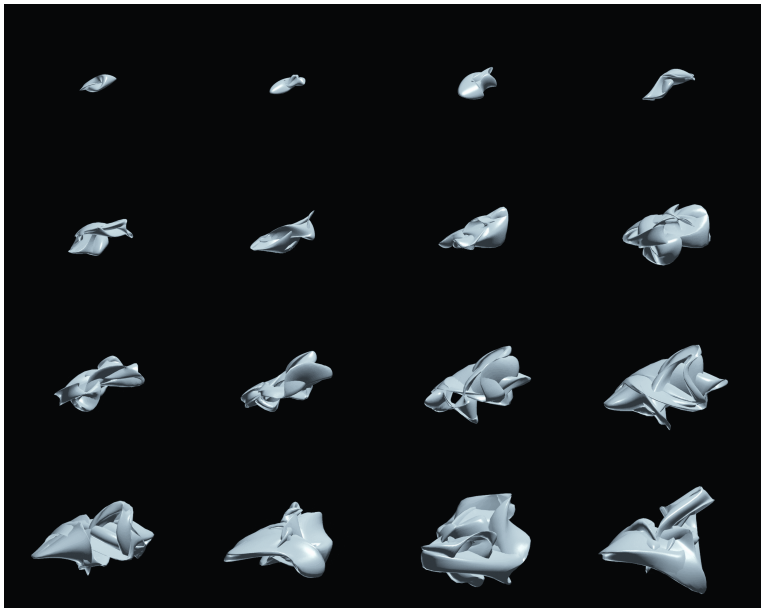

Figure 5.13: The Interactivator: the materialization of left-over cellular material.

- Materialization: As cellular division takes place, unstable cells are generated. In the next generation this leftover material creates a space of exclusion in the cellular space. This space of exclusion interacts with the physical environment to create a materialization. Boundary layers are identified in the unstable cells as part of their state information and a surface is generated to skin the structure. This material continues to exist throughout the evolution of the model and will initially affect the cellular growth of future generations. Figure 5.13 shows the gradual development of the space of exclusion over many generations.

- Genetic search landscape: The selection criteria are not defined but are an emergent property of the evolutionary process, and are based on the relationship between the genotype, cellular structure and the virtual and physical environment over time. A genetic search landscape is generated for each member graphically representing the evolving selection criteria. Form, or the logic of form, emerges as a result of travelling through this search space.

Once the developmental process stabilises, the cellular activity is terminated. The final cellular structure, the materialization and the genetic search space are posted out. A new generation of cellular activity is then initiated. The fittest chromosomes from the parent generation are combined with the new list of dominant chromosomes from the population to form a new population. These new chromosomes are then inserted into single cells, and these cells are then developed new cellular structures.

The evolutionary system was integrated with AutoCAD V. 12[12] ©. This allowed the complex surface modelling operations during the materialization process to make use of the modelling functions provided by AutoCAD. The evolutionary system was developed using AutoLisp — the programming language built into AutoCAD — thereby allowing the developmental step to directly call any modelling functions. In addition, this also allowed the process of cellular growth, materialization and the genetic search landscape to be visualised in the AutoCAD application window.

Co-evolution of structure and environment

In another experiment, Rastogi (Frazer, 1995b, p. 89) developed a generative program that explored the interaction between a growing environment and a growing structure by creating a hierarchical system of cellular automata. The generative program created boundary surfaces between a cellular environment and a cellular structure both developing in an isospatial cellular automata. The development of the environment and the structure were controlled by separate sets of transition rules, and these rules were themselves generated by a pair of two-dimensional cellular automata. As the environment and structure developed, the boundary surface deformed and transformed.

5.7 Summary

This chapter has discussed five evolutionary design systems and approaches. The main points are as follows:

- *Genetic Algorithm for Design Optimization* (GADO) is a parametric evolutionary design system for engineering design optimization that uses parallel master-slave model and a steady-state asynchronous evolution mode. A single population is maintained and new designs are created and added to the population one at a time. In addition, specialised crossover and mutation operators have been developed. A module has also been developed to ensure that diversity in the population is maintained.

- *Generative System*(GS) is a parametric evolutionary design system for building design that is integrated with a sophisticated simulation application called DOE-2. In this case, the systems follows the general synchronous evolutionary architecture, and uses standard rules and representations. The DOE-2 application was used for both lighting and thermal calculations, and in some cases Pareto multi-criteria optimization techniques were used. Experiments were performed using climate data for different global locations.

[12]http://www.autodesk.com

- *Genetic Algorithm Designer* (GADES) is a good example of a generative evolutionary design system. The system follows the general synchronous evolutionary architecture, but uses specifically developed rules and representations. A set of customised evaluation routines are also required. The system is supposed to be highly generic and applicable to any design domain. As such, it highlights certain problems and weaknesses associated with creating a highly generic system.

- The *concept-seeding* approach allows designers to create a set of rules and representations that encapsulate certain design ideas. When this approach is used in an evolutionary system, then the designs evolved by the system will all embody the ideas encoded in the rules and representations for the developmental and evaluation steps. Recently, a prototype system using the evolutionary concept-seeding approach has been developed. This system allows rules and representations to be defined for generating designs for hand-held devices such as mobile phones or remote controls.

- The *epigenetic design* approach allows designs to be generated in response to the design environment. Many evolutionary design systems allow the evaluation step to make use of encoded environmental information such as information about the design constraints or deign context. However, this approach also allows the developmental step to make use of such information. This results in an additional level of design adaptation to the environment.

Part III

Research proposition

Part three consists of three chapters that describe the proposed generative evolutionary design framework.

- Chapter 6 describes the design method introduced in chapter 1. The two phases of the method are described in more detail, and the requirements for the method — conservativeness and synergy — are discussed.

- Chapter 7 describes the computational architecture introduced in chapter 1. The requirements for the architecture – scalability and customizability — are discussed with reference to existing methods.

- Chapter 8 demonstrates the process of encoding a design schema. An example schema is introduced and a set of routines are implemented and used to generate design models.

Chapter 6

Design method

Contents

6.1 Introduction

This chapter describes the design method of the proposed generative evolutionary design framework. It consists of two main sections:

- In section 6.2, an overview of the design method is given. The schema development phase and the design development phase are described, with each phase being broken down into two smaller stages. The schema development phase is then described in more detail.
- In section 6.3, the requirements for the design method identified in chapter 1 are discussed. First, the requirement that the method should be conservative is discussed; and second, the requirement that the method should be synergetic is discussed.

6.2 Overview of method

6.2.1 Structure of method

Four stages

The proposed generative evolutionary design method was introduced in chapter 1 (see section 1.2.2 on page 18) and is shown in figure 1.3 on page 19. One of the key components of this method was a design entity that captures the essential and identifiable character of a family of designs, referred to as the *design schema*.

The method consists of the schema development phase and the design development phase. Each phase requires the design team to carry out a number of tasks that are grouped into stages. Each stage focuses on a different set of skills, and as a result different team members are likely to lead the design process at each stage. The composition of the design team may also change during the different stages of the design method.

The two stages of the schema development phase are as follows:

- The schema conception stage requires the design team to conceptually develop a design schema for a particular niche environment. The main guiding force at this stage are the preconceptions of the design team, which includes their philosophical beliefs, cultural values, and design ideas. The design team must then define the design schema is some relatively explicit way. The design team may, for example, create a number of prototypical designs that capture the variability that they desire. At this stage, the design team works at a purely conceptual level and as a result, no specialised programming or computational skills are required.

- The schema encoding stages involves encoding the design schema in a form that can be used by the evolutionary system. This involves defining a set of evolutionary rules and representations for the evolutionary system. The requirements from the design team are now different, and specialised programming and computational skills are essential. The evolutionary process must be well understood and, in particular, the design team must be aware of the relationships and interactions between the various rules and representations.

For the design development phase, the two stages are as follows:

- The design evolution stage allows a large variety of alternative designs to be evolved and adapted to a specific design environment. At this stage, the design team will need to encode the design environment for the specific project, and will then need to configure and run the evolutionary system. The process of encoding the design environment is generally much more straightforward than encoding the design schema. The evolutionary process will result in a set of alternative designs. The design team must choose one of these for further detailed design.

- The detailed design stage will need to further develop the design model selected in the previous stage to the level of detail required for construction. This stage involves the design team carrying out a process of detailed design similar to most existing design methods. As with the first stage, no specialised programming or computational skills are required.

Interaction between stages

The four stages do not have to remain distinct. At each stage, the design team will probably find it necessary to 'look ahead' to the next stage and 'look back' to the previous stage. This is particularly true for the two middle stages: the schema encoding stage and the design evolution stage.

In the schema encoding stage, the design team will need to repeatedly test an encoded schema by running the evolutionary system as defined in the next stage. At the same time, the variability of designs defined in the previous stage may be discovered to be to difficult to achieve, and as a result the design schema may require modification. Alternatively, certain unexpected design features may emerge as the result of a particular way of encoding the design schema, which the design team may decide to explore further.

In the design evolution stage, the design team may want to apply an encoded schema to a project for which it was not developed. In such a situation, the design team will have to go back and modify the encoded schema. The design team may also need to go forward and consider issues relating to how the design is detailed.

Schema development phase in more detail

For the design team, the schema development phase is a critical phase. It is during this phase that design ideas are developed and encoded. The two stages in this phase will be discussed in more detail in the next section.

For the schema conception stage, the process of creating the design schema is described. The concept of a design schema is compared and contrasted with the types of abstract body plans used by natural evolution.

For the schema encoding stage, the process of creating the evolutionary rules and representation is discussed. The genericness of these rules and representations is discussed in relation to the performance and re-usability of the evolutionary system.

6.2.2 Schema conception stage

Conception of design schema

During the schema conception stage, the design team must develop a design schema that encompasses a family of designs. This schema is an abstract model from which a variety of designs can be extrapolated. Natural evolution also evolves alternative designs based on an abstract models, often referred to as *archetypal body plans*. In many ways, design schemas appear analogous to archetypal body plans. However, important differences exist.

The most obvious difference is that design schema are defined outside the evolutionary process whereas archetypal body plans have been created by the evolutionary process. This difference is considered less significant when the *historical* and *contingent* aspects of natural evolution are considered. First, the body plans were evolved for historical environmental conditions relevant hundreds of millions of years ago but are no longer relevant today. Second, many biologists have argued that the evolution of body plans is highly contingent, in that the survival of body plans are primarily due to luck[1] rather than to any fitness based selection mechanisms (Gould, 2000).

More significant differences between design schemas and archetypal body plans can be identified at a more detailed level. These differences are related to a stark contrast in variability of natural and artificial design: while natural designs are all based on a small number of body plans, artificial designs are highly variable and resist the discovery of any common characteristics.

The contrasts between archetypal body plans and design schemas are explored in more detail below. First, body plans in nature are briefly introduced. Second, design schemas are compared and contrasted with these body plans, with key differences being emphasised. Third, the variability of natural and artificial designs is investigated. Finally, it is argued that design schemas should focus on a small family of designs that share certain characteristics. From the perspective of the design team, the designs that are most relevant are designs that they themselves created.

Archetypal body plans

The organisms evolved by nature often resemble one another, which has led biologists to identify homologies: a homology is defined by Owen (1843, p. 379) as "the same organ in different animals under every variety

[1] In *Wonderful Life*, Gould (2000) describes the Burgess Shale animals that existed in the pre-Cambrian and Cambrian periods. He emphasises that only a small number of the great variety of body plans that existed in this fauna have survived. He argues that the survival of these body plans was highly contingent, and that if evolution was 're-run', a different set of body plans would survive. Although many aspects of his argument remain contentious (for example, see (Morris, 1999)), the general idea that the history of life is profoundly contingent has become accepted.

of form and function". During the nineteenth and twentieth centuries, a variety of body plans for plants and animals were proposed based on the discovery and analysis of such homologies.

Johann Wolfgang von Goethe explored, from a holistic perspective, the relationship between the body plan of an organism and the growth process for that organism. Goethe was fascinated by plant and animal morphology and proposed an archetypal plant — which he called the *Urpflanze* — from whose form all other plants might be derived. The Urpflanze was not seen as a Platonic absolute category, nor as an historical ancestor in the Darwinian sense. Rather, it was seen as an outward expression of the developmental growth process by which plants come into existence. Goethe imagined that, by extrapolating from the Urpflanze, new plant types could be proposed. Thus Goethe writes "With such a model, and with the key to it in one's hands, one will be able to contrive an infinite variety of plants. They will be strictly logical plants - in other words, even though they may not actually exist, they could exist" (Goethe, 1817).

Such archetypal body plans became one of the major biological controversies of the first half of the nineteenth century. Russell (1982) summarises this conflict as follows: "Is function the mechanical result of form, or is form merely the manifestation of function or activity? What is the essence of life – organization or activity?". Two schools emerged: Georges Cuvier founded the *functionalist* school that maintained that function was primary; Étienne Geoffroy St. Hilaire continued the *formalist* school that maintained that form was primary. Cuvier argued that similarities between organisms could only result from similar functions. Hilaire, on the other hand, argued in his Philosophie anatomique (2 vol., 181822) that all vertebrates were modifications of a single plan of structure.

The theory of evolution by natural selection proposed by Darwin (1968) synthesise the positions of Cuvier and Hilaire. For Darwin, the existence of such body plans were seen as the result of structures that existed in common ancestors modified by natural selection. Darwin (1968, p. 233) writes:

> "It is generally acknowledged that all organic beings have been formed on two great laws - Unity of Type, and the Conditions of Existence. By the unity of type is meant the fundamental agreement in structure, which we see in organic beings of the same class, and which is quite independent of their habitats of life. On my theory, unity of type is explained by unity of descent. The expression of conditions of existence, so often insisted on by the illustrious Cuvier, if fully embraced by the theory of natural selection. For natural selection acts by either now adapting the varying parts of each being to its organic and inorganic conditions of life; or by having adapted them during long-past periods of time: the adaptations being

> aided in some cases by use and disuse, being slightly affected by the direct action of the external conditions of life, and in all cases being subjected to the several laws of growth. Hence, in fact, the law of the Conditions of Existence is the higher law; as it includes, through the inheritance of former adaptations, that of Unity of Type."

When Darwin refers to adaptations being 'subject to the several laws of growth', he acknowledges the importance of the developmental growth process. Darwin (1968, p. 427) highlights that archetypal body plans are often most apparent during the developmental growth process, stating that "community in embryonic structure reveals community in descent". This idea was later pursued by Ernst Haeckel. Haeckel (1874) had claimed that the developmental growth process of an individual from zygote to adult reflects the evolutionary history, in terms of patterns of lines of descent, of the taxonomic group to which the individual belongs. This is summarised in the law of recapitulation: that *ontogeny recapitulates phylogeny*.

Recently, this has led to speculation about how archetypal body plans might be genetically encoded. Although a strong form of recapitulation has been discredited (Richardson et al., 1997), it is generally accepted that phylogeny and ontogeny are closely intertwined, and many biologists are beginning to explore and understand the basis for this connection. For example, the discovery of certain genes that organised pattern formation in the early embryonic stage of development led to speculation that these genes might be the key to the evolution of the archetypal body plan. Slack et al. (1993) proposed a set of genes that encoded a relative positioning system used in the developing organisms. These genes seem to be present in all animals and are therefore seen to be a defining character, or *synapomorphy*, of the animal kingdom. Slack refers to this set of genes as the *zootype*, which may be viewed as the genetic counterpart to the body plan. Slack proposes that these genes must have been present in the last common ancestor of all multicellular organisms.

Design schemas versus archetypal body plans

Design schemas and archetypal body plans are both abstract models from which a variety of specific designs can be extrapolated. In both cases, an evolutionary process evolves alternative designs based on these abstract models. However, the concept of archetypal body plans differs fundamentally from the concept of the design schema. Archetypal body plans are described by Darwin (1968, p. 415) as follows:

> "We have seen that the members of the same class, independently of their habits of life, resemble each other in the general plan of their organization. This resemblance is often expressed by the term 'unity of type'; or by saying that the several parts and organs in the different species of the class

> are homologous... What can be more curious than that the hand of a man, formed for grasping, that of a mole for digging, the leg of the horse, the paddle of the porpoise, and the wing of the bat, should all be constructed on the same pattern, and should include the same bones, in the same relative positions? Geoffroy St. Hilaire has insisted strongly on the high importance of relative connexion in homologous organs: the parts may change to almost any extent in form and size, and yet they always remain connected together in the same order."

This highlights the fundamental difference between archetypal body plans and design schemas. In the case of design schemas, three key features may be identified: first, all designs share a certain essential and identifiable character; second, designs may vary in overall organization and configuration; and third, all designs are adapted to the same niche environment. In the case of archetypal body plans, all three key features are inverted: first, designs may have completely different characters; second, all designs share the same overall organization and configuration; and third, designs may be adapted to totally unrelated environments.

Variability of designs

The difference between design schema and archetypal body plans is due to the difference in design variability of artificial designs and natural designs. It is often emphasised how natural evolution has evolved a vast diversity of natural designs. However, this diversity may actually be seen too be highly limited when the overall organization and configuration is considered. Despite the massive diversity that exists in nature, most designs are based on a small number of body plans. This has been discussed by Gould (2000, p. 49):

> "Biologists use the vernacular term *diversity* in several different technical senses. They may talk about 'diversity' as a number of distinct species in a group... But biologists also speak of 'diversity' as difference in body plans. Three blind mice of differing species do not make a diverse fauna, but an elephant, a tree and an ant do — even though each assemblage contains just three species.... Most people do not fully appreciate the stereotyped character of current life... Stereotypy, or the cramming of most species into a few anatomical plans, is a cardinal feature of modern life... Several of my colleagues (Jaanusson, 1981; Runnegar, 1987) have suggested that we eliminate the confusion about diversity by restricting this vernacular term to the first sense — number of species. The second sense — differences in body plan — should then be called *disparity*."

Variability in character is therefore described as *diversity*, while variability in overall organization and configuration is therefore described as *disparity*. In terms of diversity, artificial and natural designs are similar. In both cases the diversity in character is huge. However, in terms of disparity, artificial and natural designs are very different. The disparity of artificial designs is much greater than the disparity of natural design. As a result, the homologies commonly found in natural designs, are — in the case of artificial design — quite rare. The reason for the greater disparity of artificial designs must be attributed to the process by which such designs are created. In nature, the process is highly restricted, by both the inheritance mechanisms of evolution by natural selection and by the 'several laws of growth'. With artificial design, the cognitive design process is much less restricted and designers will typically strive to impose their individuality on a design.

Due to the lack of disparity in nature, archetypal body plans could be proposed that were highly generic and encompassed significant portions of the designs created by nature. For artificial designs, on the other hand, an archetypal body plan would only be applicable to a small number of designs. A different type of approach therefore needs to be taken. Rather than trying to identify a common body plan, a more flexible approach is required that specifies common characteristics without predefining the overall organization and configuration.

Families of designs

Despite the disparity in artificial designs, some have suggested that common characteristics may nevertheless be found that are highly generic and may therefore encompass a significant portion of the designs created by human designers. For example, Gropius (1962) writes: "A basic philosophy of design needs first of all a denominator common to all... Will we succeed in establishing an optical 'key', used and understood by all, as an objective common denominator of design?" Such a 'key' would provide "the impersonal basis as a prerequisite for general understanding and would serve as the controlling agent in the creative act."

Such a generic common denominator would be of great value, since it could be used by any human designer or by any automated design system to guide the design process. However, to-date, no such common denominator has been discovered. Due to the disparity of artificial designs, it is likely that any common denominators must be restricted to some small family of designs.

Common denominators — or shared characteristics — may then be defined in relation to such a family of designs. One approach may be to investigate a variety of formal devices, such as grammars, typologies, proportioning systems, ordering principles, and so forth. The family of design would then consist of members using the same set of formal devices.

An alternative approach would be to look beyond the actual formal

devices, and instead seek the source of these devices in the design process. This approach will lead back to the preconceptions of the design team. In this case, the family of designs would consist of the body of work of the design team.

The latter approach is preferable for two reasons: first, the body of work of a single design team has a strong unity; and second, designers will find designs that embody their own beliefs, values and ideas more relevant.

- In general, a designer's body of work does not consist of a disparate set of unrelated designs. Instead, the individual designs tend to be related to each other, and can be seen to reflect specific ideas relating to aesthetics, space, structure, materials and construction. The individual designs may often be seen as part of a personal process of exploration and development. The designs may form a stylistic family or a chronological sequence, or parts of one design may be found in another design. As a result of the interrelationships between the designs, each design becomes recognisable as being part of the designers body of work. The unity in a designer's body of work therefore tends to be strong.

- In general, designers have little interest in exploring designs that do not reflect their beliefs, values and ideas. For a designer's body of work, the common denominator becomes the personal and idiosyncratic *design ideas* of the designers. These ideas can then be used to guide the generative process. These design ideas will reflect a set of more general philosophical beliefs and cultural values held by the designers. By incorporating the designer's ideas, the relevance of the designs synthesised are greatly increased.

6.2.3 Schema encoding stage

Encoding rules and representations

After the design team has developed the design schema, the next stage of the design method involves encoding the schema as a set of rules and representations that can be used by an evolutionary system.

For any evolutionary system — whether in the design domain or some other domain — the way that these rules and representations are encoded will affect the trade-off between the *performance* and the *re-usability* of the system. In general, as the rules and representations incorporate more domain- and task-specific knowledge, the performance of the system will increase and the re-usability of the system will decrease.

Broadly, three approaches can be identified that differ in the level of knowledge incorporated into the rules and representations: the *highly-generic* approach, the *domain-specific* approach, and the *task-specific* approach. The levels of performance and re-usability that can be achieved by these approaches is dependent on the domain and the task.

For generative evolutionary design systems, these performance and re-usability issues are related to the *variability problem* and *style problem* introduced in chapter 1(see section 1.1.2 on page 12 and section 1.1.2 on page 15). For the variability problem, task-specific knowledge can be used in order to restrict design variability. As discussed in the section above, the task-specific knowledge that is most relevant to the design team is likely to consist of their own design ideas. However, this aggravates the style problem: incorporating such task-specific knowledge will result in designs with a specific character or style, thereby reducing re-usability.

Performance and re-usability are therefore in direct conflict. One way to resolve this conflict is to allow the design team to customise the evolutionary system with rules and representations specifically developed to reflect their own design ideas. This approach is seen to be similar to the concept-seeding approach developed by Frazer and Connor (1979); Frazer (1995b); Sun (2001).

The different approaches to encoding the evolutionary rules and representations are discussed in more detail below. First, the three main approaches are described, and performance and re-usability issues are discussed. Following this, the discussion then focuses specifically on generative evolutionary design systems. Performance issues are then discussed in relation to the variability problem and the style problem. Finally, the concept seeding approach is discussed as a way of resolving the conflict between performance and re-usability.

Types of rules and representations

The three broad approaches to creating evolutionary rules and representations are described as follows:

- With the highly-generic approach, researchers use standard rules and representations that do not rely on any task or domain-specific knowledge. For example, many researchers use binary string genotypes in combination with one or two point crossover and bit mutation. Any task or domain-specific knowledge is confined to the evaluation step.

- With the domain-specific approach, researchers develop non-standard rules and representations that incorporate domain-specific knowledge. Many researchers have found that the standard rules and representations result in evolutionary systems with poor performance. Researchers have therefore incorporated domain knowledge in the rules and representations in order to improve the performance of the system.

- With the task-specific approach, researchers develop non-standard rules and representations that incorporate task-specific knowledge. In this case, the type of tasks in the domain may be complex and as a result researchers may find it necessary to incorporate such

task-specific knowledge if a reasonable level of performance is to be achieved.

As discussed in chapter 4 (see section 4.4.1 on page 99), highly-generic approaches are described as *weak* while task-specific approaches are described as *strong*.

Performance and re-usability

The idea that knowledge is critical to improving the performance of an evolutionary algorithm is similar to the *No Free Lunch* (NFL) theorem discussed by Wolpert and Macready (1995, 1997) in relation algorithms that search for a maximum or minimum value of a cost function. They argue that the performance of such algorithms can *only* be improved by making them less generic. They show that "all algorithms that search for a maximum or minimum value of a cost function perform exactly the same, when averaged over all possible cost functions" (Wolpert and Macready, 1995). More generally, they state that "...for any algorithm, any elevated performance over one class of problems is exactly paid for in performance over another class" (Wolpert and Macready, 1997).

However, it is also clear that when researchers embed domain- and task specific knowledge in the rules and representations used by an evolutionary system, then the re-usability of the system will decrease. In some cases — for example, genetic algorithms — it may be possible to create rules and representations that are highly generic and also perform reasonably well. As a result, such systems can be re-used for many tasks with little modification. However, for most real-world tasks this is not feasible (see section 4.4.1 on page 99). In such cases, researchers must balance the performance of the system against the re-usability of the system.

The conflict between performance and re-usability is also applicable to evolutionary design: for any evolutionary design system, any elevated performance over one class of designs is exactly paid for in performance over another class. In other words, the performance of the system can be improved by using less generic rules and representations, but the re-usability will also decrease as a consequence. In this case, the *domain* is seen as a particular design domain — such as building design, and the *task* is seen as the process of creating a type design — for example, a design for a particular type of building.

With parametric evolutionary design, design variability is low and as a result the performance of the system using generic rules and representations may be reasonable. For example, GS developed by Caldas (2001) (see section 5.3 on page 118) is based on the canonical genetic algorithm and is therefore highly generic. Some parametric evolutionary design systems may nevertheless include domain-specific rules and representations in order to achieve a better performance: for example, the GADO system (Rasheed, 1998) (see section 5.2 on page 113) uses a real-valued

genotype representation and a number of specially developed crossover and mutation operators that incorporate domain-specific knowledge.

With generative evolutionary design, a much greater variability of design needs to be evolved. This results in a major shift in the balance between performance and re-usability. Since it is not possible to develop a generative process capable of generating any conceivable design in any domain is highly unrealistic, the highly generic approach can be ruled out. This leaves two possibilities: the domain-specific and the task-specific approaches.

Performance of generative evolutionary design system

For generative evolutionary building design, the domain-specific approach is likely to result in poor performance. Although the range of designs to be generated would be limited to the building domain, the developmental step would still have to be capable of generating almost any possible building design. Such a developmental step would have to rely on a generic rules and representations, which would lead to variability being under-restricted. This would then result in poor performance due to the variability problem, and in particular due to the problem of chaotic designs.

A key requirement for the rules and representations is to avoid unrestricted variability, and this can only be achieved by identifying shared design characteristics that can be used to guide the generative process. It has been argued that the most appropriate family of designs is the body of work of one design team. This approach results in a set of rules and representations, that — although not specific to a single design — are specific to one design team. This approach is therefore a task-specific approach.

Re-usability of generative evolutionary design system

The problem of embedding such task-specific rules and representations in an evolutionary design system has been described by Bentley (1999b) as the *style problem*. When discussing evolutionary art systems, Bentley writes:

> "One undesired side-effect of many of these representations is that they generate pieces of art which have distinct styles. Often the style of form generated using a particular representation is more identifiable than the style of the artist used to guide the evolution. This can cause problems if the artist wishes to take credit for the piece. The cause of this 'style problem' is perhaps due to the initial preconceptions and assumptions of the designer of the representation. By limiting the computer to a specific type of structure, or a specific set of primitive shapes and constructive rules, it

> will inevitably always generate forms with many common and identifiable elements" (Bentley, 1999b).

Bentley therefore suggests that the main alternative to parametric evolutionary design[2] is generative evolutionary design systems that do not use task-specific rules or representations; he refers to these types of systems as *evolutionary exploration systems*. Bentley (2000a) writes: "a knowledge-lean representation is used, and ... a set of low-level components is defined. Solutions are then constructed using these components, allowing exploration (sometimes at the expense of size of search space and the ability to locate optima)."

In the development of GADES (see section 5.4 on page 122), Bentley (1999a) has attempted to create an exploration system using only the domain-specific approach. For the developmental step, a generative process was created using only low-level knowledge-lean rules and representations. Bentley claims that GADES is both highly generic and that it can consistently evolve good designs without human intervention. However, both these claims appear overly optimistic.

Regarding the genericness of the system, although the rules and representations used by Bentley are not specific to one designer, they do limit its applicability to a small area of design. The system is only capable of evolving solid objects[3], which already excludes a vast set of possible designs. Furthermore, the solid objects evolved by GADES all have a particular 'blocky' monochrome character; the system cannot evolve other types of geometries such as free-flowing surfaces. In addition, many other aspects — such as symmetry of designs — has to be predefined by the user.

Second, with regard to the quality of designs evolved, the example designs shown by Bentley (1999a) are all seen to be simplistic. For instance, Bentley uses the relatively straightforward problem of designing a table as an example of the capabilities of the GADES system. Bentley (1999a) writes: "Results were good: GADES consistently evolved fit table designs, often with surprising creativity." The examples that Bentley provides do not support this optimism. From a design perspective, the table designs created by the system are very primitive.

The style problem versus the variability problem

The style problem may be contrasted with the variability problem. The style problem relates to re-usability, while the variability problem relates to performance.

- The style problem relates to the *re-usability* of the evolutionary system. If task-specific rules and representations for a particular

[2]Bentley refers to parametric evolutionary design as *evolutionary design optimization*.

[3]Bentley (1999a) has claimed that GADES is capable of evolving other types of design, such as hospital plans. However, this requires a significant amount of modification to the system.

designer are programmed into the system, then the system will be restricted to evolving design that reflect the designer's preconceptions. The style problem suggests that researchers should use the domain-specific approach rather than the task-specific approach to creating rules and representations. This will lead to a system that is capable of evolving designs that are more generic, thereby allowing the system to be re-used by many different designers on many different projects.

- The variability problem relates to the *performance* of the evolutionary system. In the building design domain, both the diversity and disparity of designs are high. If only domain-specific rules and representations are used, then designs will vary in highly unrestricted ways which will result in a poor performance from the evolutionary system. The variability problem suggests that researchers should limit the variability of designs by using the task-specific approach to creating rules and representations. This will lead to a system that, as a result of the restricted variability of the designs, performs better and is capable of evolving designs that a designer may find surprising and challenging.

Attempting to create a generative evolutionary system without task-specific knowledge is seen as a fundamentally flawed approach. In order to evolve surprising and challenging designs, knowledge about the design task must be incorporated in the rules and representations of the evolutionary system.

One way to achieve this is to accept the low level of re-usability and create a new evolutionary system for each designer. This is common for the evolutionary art systems (Bentley, 1999b). In some cases, the designers and programmers may work together to develop a system, while in other cases they be the same people. This is also the approach taken by Soddu (2002), an architect who has created his own generative system capable of producing designs with a distinctive character. The disadvantage of this approach is that designers must all developed their own specialised evolutionary systems from scratch. Many designers simply do not have the resources for such a process. This approach is also wasteful since the underlying evolutionary system should not have to change.

Concept-seeding approach

An alternative approach would be to allow designers to customise an evolutionary system with their own rules and representations. Such an approach would allow for the conflict between the style problem and the variability problem to be resolved. The core of the evolutionary system prior to customization could incorporate only domain-specific knowledge, and could be re-used by many different designers. Rules and representations could be defined that incorporate task-specific knowledge, including

knowledge related to the preconceptions of the designers using the system.

This is similar to the concept-seeding approach developed by Frazer and Connor (1979); Frazer (1995b); Sun (2001). This approach was discussed in chapter 5 (see section 5.5 on page 126). To recap, the concept-seeding approach — when combined with a generative evolutionary design system — allows the designer to capture and encode a set of design ideas in the form of a seed. Such a seed allows designs to be generated that embody the design ideas encoded in it. A generic and re-usable generative evolutionary design system can then be used to evolve modifications to either the seed or the generative process.

However, the concept-seeding approach also has a number of limitations. Firstly, it focuses exclusively on the generative rules and representations used in the developmental step. As discussed in chapters 4 (section 4.4 on page 98) and 5, the rules and representations used in the other evolution steps also have a significant effect on the types of designs that can be evolved. Furthermore, the rules and representations in the developmental step are closely related to both the reproduction step and the evaluation step. It is usually impossible to change the rules and representations in one step without modifying other steps.

In order to support such experimentation, the proposed architecture generalises the concept-seeding approach by including all the evolutionary rules and representations in the 'seed', in this case referred to as the *encoded schema*. Such an encoded schema may still include entities such as the design seeds developed by Frazer and Connor (1979) for the Reptile Program, or the rudiments and configurational rules in the system developed by Sun (2001). The encoded schema will also include all the other rules and representations that are required to manipulate and process these individuals. This approach gives researchers maximum flexibility in encoding the design schema.

Each evolutionary step is characterised as a process that transforms input data into output data. The rules that define this transformation are encapsulated as a self-contained program, referred to as *evolution routines*. These routines can be linked to the evolutionary system and executed by the system whenever a transformation needs to be performed. The representations define the format of the input and output data of these routines.

In total there are seven different routines, and three representations. Figure 6.1 on the following page show the various routines and representations that need to be codified. The routines must be specified explicitly, while the representations are defined implicitly as data formats used by the routines. These routines and representations will be described in more detail in the next chapter.

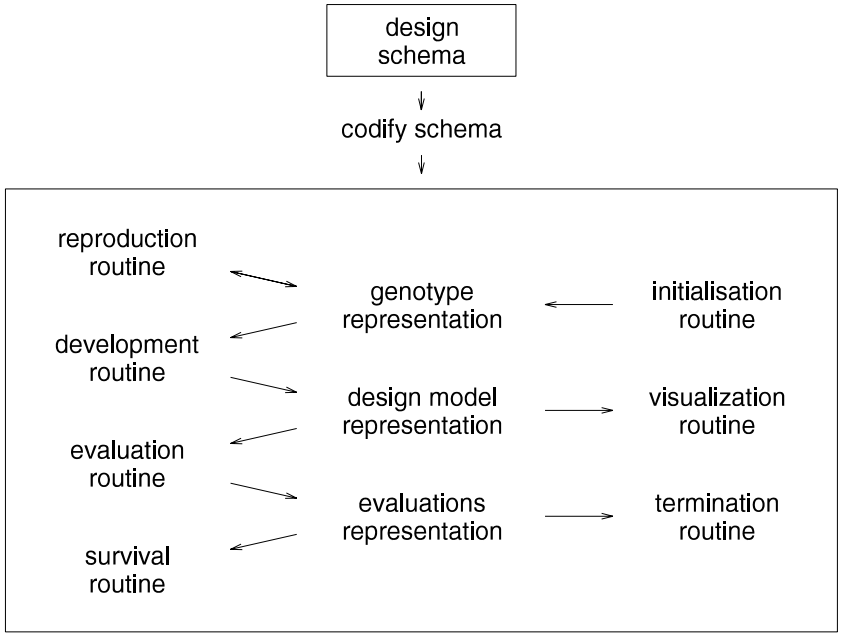

Figure 6.1: The rules that encode the design schema.

6.3 Key requirements

In chapter 1 (see section 1.2.2 on page 18), two key requirements for the proposed method were identified: it should be *conservative* and it should *synergetic*.

6.3.1 A conservative method

As stated in chapter 1 (see section 1.2.1 on page 16), the design method is argued to be *conservative* based on a general similarity between the structure of the design method and a conventional design process commonly used by designers in practice.

Design schemas as design ideas

Chapter 2 reviewed research into the design process. A number of researchers found that, in practice, the design process does not follow the design sequence of analysis, synthesis and evaluation, as commonly proposed in the 1960's. Instead, they argue that many designers — including architects — bring a set of preconceptions into a project, and that these preconceptions play a significant role in the design process. Two types of preconception have been identified: a *design stance* and a set of *design ideas* (see section 2.4.3 on page 46).

- The design stance has been referred to as the designer's *paradigmatic stance* (Broadbent, 1988), *guiding principles* (Lawson, 1997), and *theoretical position* (Rowe, 1987). This stance consists of a designer's broad philosophical beliefs and cultural values, which

emerge over an extended period of time through multiple projects. The design stance often evolves and changes throughout a designers career. However, a designer will generally adhere to a single design stance that is applicable to any projects that they work on.

- Design ideas have been described as *primary generators* (Drake, 1979), *enabling prejudices* (Rowe, 1987), and *working methods* (Frazer, 2002). The main characteristic of such design ideas is that they are not derived from a process of reasoning or analysis, but are self-imposed subjective judgements that define and direct the design process.

The relationship between a set of design ideas and a design project may take a number of different forms. Lawson's analysis of the design process suggest that there are three possibilities (Lawson, 1997). First, the design ideas may be directly related to some specific aspect of a project. Second, the design ideas may be related to the type of project, but not specifically to the one project. Third, the design ideas may not arise from the project but may originate from the designer's own personal design stance.

Design ideas that are not project-specific may be re-used within a variety of projects. This results in a design process that is similar to the schema-based design process. Through previous projects or competitions, designers implicitly develop design schemas that consist of sets of design ideas that may be applied to a range of new projects. The design schemas may include any combination of the different types of design ideas described by Rowe (1987): anthropomorphic analogies, literal analogies, environmental relations, typologies and formal languages.

Existing design schema process

An existing design process can be defined that incorporates both design preconceptions and design schemas. This process is seen as one that some designers loosely follow. This design process is shown in figure 6.2 on the following page. Although the direction of the flow should be understood to be predominantly top to bottom, most designers do not work in a linear manner, but move backwards and forwards between stages.

The design process considers two types of environment that were defined in chapter 1. The *design environment* includes the project specific constraints and context, while the *niche environment* is not one specific environment, but instead encompasses a range of design environments that are similar to one another. The design process consists of two phases: in the *schema development phase* a design schema is created, while in the *design development phase*, the design schema is applied to a specific project and a detailed design proposal is produced.

The schema development phase consists of just one stage:

- During the schema conception stage, a design schema is developed that consists of a set of design ideas for a niche environment. De-

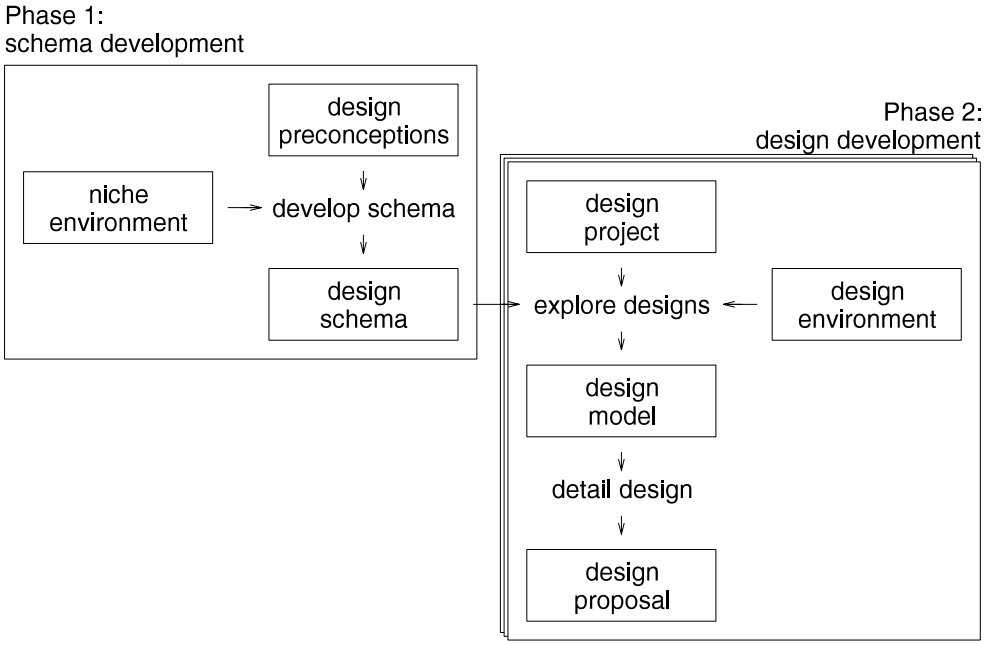

Figure 6.2: A design process used by some designers.

signers will usually develop such ideas by working on a series of similar projects or competitions, and this stage may emerge over a long period of time. Design preconceptions may be seen as initiating the process of idea development.

The design development phase consists of two stages:

- During the design exploration stage, an initial design model is developed by adapting the design schema to the design environment for a specific project. This design environment will include the project brief and a particular site. Designers typically explore and evaluate a range of alternative designs at this stage. This stage involves analyzing the brief and the site, and applying the design ideas in an appropriate manner.

- During the detailed design stage, a design proposal is developed from the design model. At this stage, the role of the design environment is less significant. The design model will define the overall configuration and organization of the design but will not have specified any details.

Proposed design method

The proposed generative evolutionary design method is a modification of the existing design process described above. This method was introduced in section 1.2.2 on page 18 and was further described in section 6.2.1 on page 148. Figure 1.3 on page 19 shows the four main stages of the design method, which may be described as follows:

- The schema conception stage is similar to the equivalent stage in the existing design process. In the existing process, design schemas are usually implicitly defined. The proposed method requires the design schema to explicitly defined.

- The schema encoding stages is an additional stage. This involves defining a set of evolutionary rules and representations for the evolutionary system.

- The design evolution stage replaces the design exploration stage in the existing design process. Rather than explore a small number of alternatives manually, this stage allows a vast variety of alternative designs to be evolved.

- The detailed design stage is the same as the equivalent stage in the existing design process. The design model that is generated in the evolutionary system needs to be further developed to the required level of detail.

The proposed generative evolutionary design method can be seen to share many features with the existing design processes described above. Although designers do not explicitly use design schemas, developing a variety of designs from an abstract set of architectural concepts and ideas is often implicitly part of existing design process.

6.3.2 A synergetic method

The second requirement for the proposed design method is that it should be synergetic. As discussed in chapter 1 (see section 1.2.1 on page 16), this is achieved by making use of the contrasting abilities of the design team and the computational system in a way that is mutually reinforcing. In particular, the design method specifies a process where the design team can focus on those tasks that are predominantly creative and subjective, and where the computational system can be applied to those tasks that are predominantly repetitive and objective.

Designing as a social activity

The creation of a synergetic design method is based on the idea that computers may play an important role in the creative design process. This synergy focuses on the designers encoding the design schema as a set of rules and representations that can then be used by computational systems to evolve designs.

The role of computers was discussed in chapter 2 (see section 2.3.2 on page 40). To recap, computers may either be used as *mere tools* for tasks such as drafting or as a *design support medium* that is an essential counterpart in the creative design process of human designers. In the latter case, the computer is not required to be an intelligent system, but

should have certain knowledge and capabilities in the area of interest. Mitchell (1994) identifies three distinct paradigms that describe how the role of the computer has been changing: the *designing as problem-solving* paradigm emerged in the 1960's; the *designing as a knowledge-based activity* emerged in the 1980's; and most recently, the *designing as social activity* has emerged.

In each of the paradigms identified by Mitchell, researchers have attempted to create computer systems that have gone beyond mere tools, striving to create systems with knowledge and capabilities that support the design process. In each case the approach has been different. The proposed design method and computational architecture follow Mitchell's third paradigm and view design as a social activity.

Encoding design schemas

The proposed method allows a variety of designs to be created that embody the same design ideas. The design ideas have been described as preconceptions that are personal and idiosyncratic to the designers involved.

This approach differs fundamentally from the problem-solving paradigm from the 1960's. In this paradigm, preconceptions were regarded as subjective elements that had to be eradicated. The reasoning was that such preconceptions would impede rational thinking, and were likely to exclude certain design possibilities that might actually be of higher quality.

This approach was often described in terms of search spaces and fitness landscapes: preconceptions would limit the search process to certain areas of the search space and could thereby exclude the optimal solution for the problem. One of the main goals of early design methods was therefore to develop an objective search methodology that would eliminate all preconceptions.

A fundamental flaw in this line of reasoning is that the search process itself requires various preparatory steps to be taken where preconceived ideas are still required. The search process requires four preparatory steps to be taken prior to the actual searching of the space. First the problem must be clearly specified. Second, the search space must be meaningfully structured so that solutions are related to each other in some meaningful way. Third, a fitness function must be defined so that solutions can be compared. Finally, a search procedure must be defined to search the space. The first three steps all require the designer to make various decisions that involve preconceptions.

Since the 1960's, it has become clear that objective design processes can only be used for solving highly constrained and simplified problems. Any design task of even moderate complexity cannot be 'solved' in this way. Instead, a much more complex process mediates between the design task and the eventual design. Design preconceptions are essential and necessary ingredient in this process.

Choice of preconceptions

The problem-solving paradigm is rejected. However, this paradigm embodied an aversion to preconceptions that may nevertheless have some validity. Preconceptions are often thought to lead to designs that are predictable and conventional, and generally of low quality. In contrast, high quality designs are generally considered to be the result of an innovative mind that rejects the conventional in favour of the new and radical.

To a certain extent, this argument must be accepted. In some cases, the designer has to drop certain preconceptions in order to create a radically new design. This allows a whole new set of possibilities to be explored. However, preconceptions are a necessary part of the design process. Two types of preconceptions may be considered. Some preconceptions are *limiting* in that they seem to restrict the freedom of the designer, while others are *enabling* in that they give the designer greater freedom.

An example of an enabling preconception may be the Modulor, Le Corbusier's dimensioning system based on the proportions of the human body (Corbusier, 1982). Such a system limits the dimensions that can be used but Le Corbusier argues that this actually gives the designer greater freedom. Limits provide a framework within which the designer can experiment. Le Corbusier compares his division of space to the way that the continuous phenomenon of sound had been divided "in accordance with a rule accepted to all, but above all efficient, that is flexible, adaptable, allowing for a wealth of nuances and yet simple, manageable and easy to understand".

Designers should aim to discard limiting preconceptions but keep enabling preconceptions. However, identifying which are limiting and which are enabling is not straightforward. Preconceptions not only affect the way a design is created, but also affect the perceived quality of a design. Preconceptions will affect the choice of objectives, the importance assigned to each objective, and the way that each objective is evaluated.

In addition, a preconception that was previously enabling may gradually become limiting. Evans (1995), in discussing the Modulor, similarly describes a conflict between design freedom and design rules: "Any rule carries with it the eventual prospect of reduced liberty, but new rules can be surprisingly unruly, cleaning away customs and habits that have stood in the way for ages. Thus, for a time, perhaps quite a long time, new rules can offer a way round the obvious."

Choosing preconceptions is therefore seen as being highly subjective and should be left to the designers. Humans perform better than computational systems in these kinds of tasks. By creating the evolutionary rules and representations, the design team can have full control over the preconceptions.

Once the rules and representations are defined, large numbers of design alternatives must be explored, and for each design a detailed evaluation step must be performed, involving the assessment of multiple and

conflicting objectives. In these type of tasks, computational systems excel, while humans perform poorly.

6.4 Summary

This chapter has described and discussed the proposed design method. The main points are as follows:

- The design method is argued to be conservative based on its similarity with a conventional design process. This conventional process, although not universal, is commonly used by many designers. As a result, such designers will easily be able to integrate and appropriate the proposed design method in their own working practices.
- The design method is argued to be synergetic based on the way that the inherent talents of the design team and the computational system are exploited. The main tasks for the design team are translating their design preconceptions into a set of design ideas, and then encoding these design ideas in a computer readable form. For the computational system, the main tasks are generating large numbers of design alternatives and then analyzing and simulating each alternative. The abilities of the design team and of the computational system are used in a way that is mutually reinforcing.
- For the design team, the schema development phase is the critical phase during which a design schema is conceived and encoded. In the conception stage, the design schema must be developed based on a family of designs. The family of designs that is most relevant to the design team consists of designs that they have produced or will produce in the future. This family must have three key features: the designs must all share an essential and identifiable character, the designs must vary significantly in overall organization and configuration, and the designs must all be adapted to the same niche environment.
- In the encoding stage, the design team must encode the design schema as a set of rules and representations. The level of knowledge embedded in these rules and representations will affect the performance and the re-usability of the generative evolutionary design system. A trade-off exists between performance and re-usability: for reasonable level of performance, knowledge-rich rules and representation based on a set of design preconceptions must be used; for a reasonable level of re-usability, knowledge-lean rules and representations must be used that are highly generic. A way of resolving this conflict is proposed, whereby the evolutionary system is split into a generic core and a set of specialised components.

Chapter 7

Computational architecture

Contents

7.1 Introduction

This chapter describes the computational architecture of the proposed generative evolutionary design framework. It consists of three main sections:

- In section 7.2, the requirements for the computational architecture identified in chapter 1 are discussed. The existing evolutionary design systems introduced in chapter 5 are analysed with respect to

these requirements. Based on this analysis, an alternative architecture is proposed. First, the requirement that the architecture should be scalable is discussed; and second, the requirement that the architecture should be customisable is discussed.

- In section 7.3, an overview of the main components and their interactions is described. First, the representation of individuals in the population is described; second, the specialised components are described; and third, the generic core is described. Lastly, the various interactions are described.
- In section 7.4, languages and technologies for implementing the different parts, components and interactions described in the previous section are suggested.

7.2 Key requirements

In chapter 1, two key requirements for the proposed architecture were identified: scalability and customizability.

7.2.1 A scalable system

As stated in chapter 1 (see section 1.2.1 on page 16), scalability is fulfilled in two ways: first, the execution time for the overall evolutionary process is minimised by using an asynchronous parallel evolutionary process. Second, the flexibility and robustness of the architecture is maximised by using a decentralised control structure in combination with a client-server model.

Reduction in execution time

A parallel architecture is proposed that is similar to the asynchronous global parallel architecture developed by Rasheed and Davison (1999) for GADO. GADO was described in chapter 5 (see section 5.2 on page 113), and the general asynchronous architecture implemented by GADO is shown in figure 4.5 on page 90. To recap, the GADO architecture uses an asynchronous steady-state evolution mode and uses a master-slave parallel model, with the evaluation step being performed by multiple slaves.

One drawback of the GADO architecture is the lack of a developmental step. This drawback is considered minor since such a step can easily be included. Furthermore, this step may be parallelised in the same way that the evaluation step is parallelised: each time an individual needs to be developed, the master processor can send the individual to a slave processor.

As discussed in chapter 4 (see section 4.4 on page 86), the speed up that can be achieved by parallelising any of the evolution steps is dependent on the *computation cost* and *communication cost* of the steps. To

achieve a maximum speed up, the communication cost should not outweigh the computation cost. The developmental and evaluation steps are good candidates for parallelization: they both have low communication costs because they require only a single individual from the population; the computation costs tend to be high due to the complexity of typical developmental and evaluation processes.

The GADO architecture is seen to be particularly appropriate in domains such as design where the developmental and evaluation processes may be complex and computationally expensive. The parallel approach has been tested using up to 100 slave computers and the performance has been shown to be good. In addition, the steady-state evolution mode has also been shown to significantly reduce the execution time of the evolutionary process.

The centralised master-slave model used by GADO has a number of weaknesses with regard to the flexibility and the robustness of the evolutionary process. The architecture uses a master-slave model where the master controls the evolutionary process. This results in two weaknesses: first, slaves cannot be easily added and removed without stopping the evolutionary process and reconfiguring the system; and second, the failure of one or more of the slaves may leave the master processor in an unstable state, and may cause the system to fail. In the case of generative evolutionary design, these are both important weaknesses.

- Since the execution time for the evolutionary process may stretch for a number of days (even when using a parallel implementation), it is essential that the architecture allows computers to be added and removed from the evolutionary process with the minimum of reconfiguration. Systems in research labs and offices are typically idle for long periods of time but may be intermittently required for other purposes. A typical scenario is a lab or office where computing resources are used during the day but not at night. Computing resources may include globally networked systems connected via the Internet and allow evolutionary systems to make use of computing resources in different time-zones. The systems connected to such networks will typically vary in terms of hardware and software. Such computing resources are described as *heterogeneous computing resources*.

- It is also important that the architecture is able to cope with the failure of systems participating in the evolutionary process in a graceful manner. With generative evolutionary design, the process of developing and evaluating designs can be complex and the failure of one of these processes becomes more likely. The chance of such failure is further increased when the long execution time of the overall evolutionary process is taken into account.

For the above reasons, the centralised master-slave model used by GADO is replaced by a decentralised control structure using a client-server model. With GADO's master-slave model, the master processor

maintains the population and controls all the evolution steps. With the proposed architecture, the server manages the population but has no direct control over the evolution steps. The developmental and evaluation steps are parallelised, with multiple clients performing developmental and evaluation steps in parallel. This allows the architecture to be both more flexible and more robust.

Flexible and robust architecture

With the proposed architecture, the core evolutionary module — referred to as the *population module* — maintains the population. This module is executed on the central server, but does not control the evolution steps. Instead, it passively waits to be contacted by the evolution steps, thereby allowing the evolution steps to act independently from the population module and from one another.

Each of the evolution steps are conceptualised as modular software components that perform transformations: each step requires a number of individuals, processes these individuals, and produces some result. The result is usually a new or updated set of individuals, or — in the case of the survival step — one or more individuals to be deleted. The evolutionary process consists of evolution steps contacting the population module to request a set of individuals, processing these individuals, contacting the population module once more to send the results back again.

This decentralised control structure results in a population where individuals are in various different *states*: some only have a genotype, others have been processed by the developmental step and have a phenotype, while others also have evaluation scores.

Each of the evolution steps requires individuals in a particular state. It is the responsibility of the population module to ensure that each step is sent individuals in an appropriate state. The individuals in an appropriate state are referred to as *candidates*. When the population module receives a request from one of the evolution steps, it will first identify all possible candidates in the population and will randomly select the required number of individuals from these candidates.

Four evolution steps are used: reproduction, development, evaluation and survival. (The selection step is no longer defined as a separate step but is seen as part of the reproduction step.) Figure 7.1 on the facing page shows the main components of the general asynchronous decentralised evolutionary architecture. This architecture is proposed as an alternative to the two existing general architectures discussed in chapter 4: the general synchronous architecture and the general asynchronous architecture.

The four evolution steps are described as follows:

- The reproduction step requests a small pool of parent individuals that have been fully evaluated. These parent individuals are used to create new offspring which are added to the population.

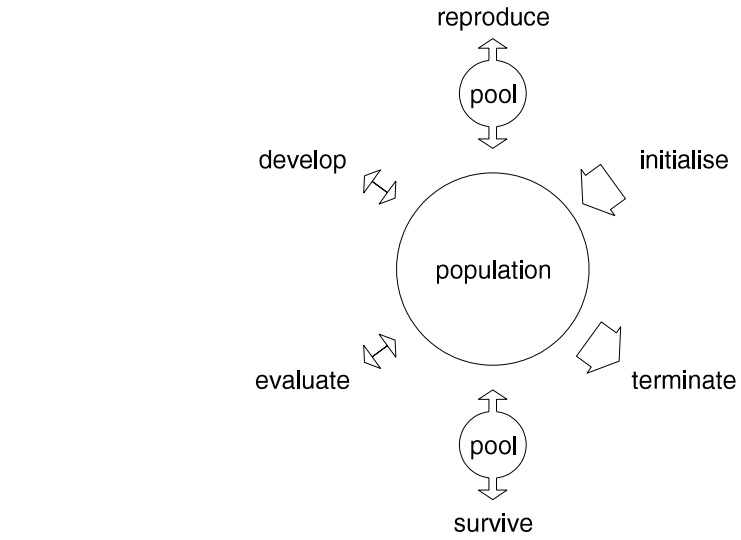

Figure 7.1: General decentralised evolutionary architecture.

- The developmental step requests a single individual that has not yet been developed. This step creates a phenotype for the individual then returns it to the population.
- The evaluation step requests a single individual that has been developed but not yet evaluated. This step evaluates the individual with respect to a predefined objective then returns it to the population. If more than one objective needs to be evaluated, multiple evaluation steps will be used, with each step focusing on one objective.
- The survival step requests a small pool of individuals that have been fully evaluated. The evaluation scores are used to select individuals from the pool, and these selected individuals are deleted from the population.

Figure 7.1 also shows an initialization step and a termination step. The initialization step allows a new initial population of individuals to be created, while the termination step allows termination conditions for the evolutionary process to be specified.

Within evolutionary design systems, a *visualization step* may also be included that allows users to manually select individuals from the population and to visualise their design models. Such a step would not be significant outside the design domain and it is not incorporated in the general architecture shown in figure 7.1. In evolutionary design, the visualization processes may be complex, especially if the phenotype is represented using a non-standard format that requires translation. For evolutionary design systems, it is included as a step in its own right (see figure 1.4 on page 21). This step is similar to the evolution steps in that it also requests an individual from the population for processing. However,

it has no effect on the individuals that it processes and therefore no results are sent back to the population module.

Due to the high *computation costs* and low *communication costs*, the developmental and evaluation steps are seen to be good candidates for parallelization. In contrast, with the reproduction and survival steps, the communication costs are much higher due to the fact that multiple individuals are required, and the computation costs are much lower. These steps are therefore not good candidates for parallelization. The visualization step may also benefit from parallelization, but for different reasons. In this case, parallelization would not affect execution time, but would allow more than one designer to interrogate the design models in the population from any networked location.

The architecture parallelises the development, visualization and evaluation step, which can be performed by client computers, while a server computer can host the population module and perform the reproduction and survival steps. Development, visualization and evaluation clients will contact the population module via the network, while the reproduction and survival steps residing on the server can contact the population module directly.

The proposed architecture is both flexible and robust:

- The decentralised control structure means that developmental and evaluation clients can be added to or removed from the evolutionary process without any reconfiguration the central population module running on the server. The population module will not even be aware of the number of clients. Furthermore, the client-server model only requires computer systems to use compatible communication protocols, but does not require systems to be running the same operating system or software. This approach results in a highly flexible architecture that can make maximum use of heterogeneous computing resources.

- With the proposed architecture, the two most complex steps — the developmental and evaluation steps — are performed by client computers only loosely linked to the evolutionary system. Assuming that multiple clients have been assigned to both these steps, the failure of one of the clients will result in the evolutionary process slowing down, but will not cause the whole system to fail. Furthermore, once the failure has been detected, the client computer can simply be restarted.

7.2.2 A customisable system

The majority of evolutionary systems are provided to users either as source-code libraries or as programming toolkits (Alba and Troya, 1999). As discussed in chapter 4 (see section 4.3.1 on page 91), they are rarely implemented as ready-made menu-driven systems. The proposed architecture follows this trend and specifies a system that falls into the cate-

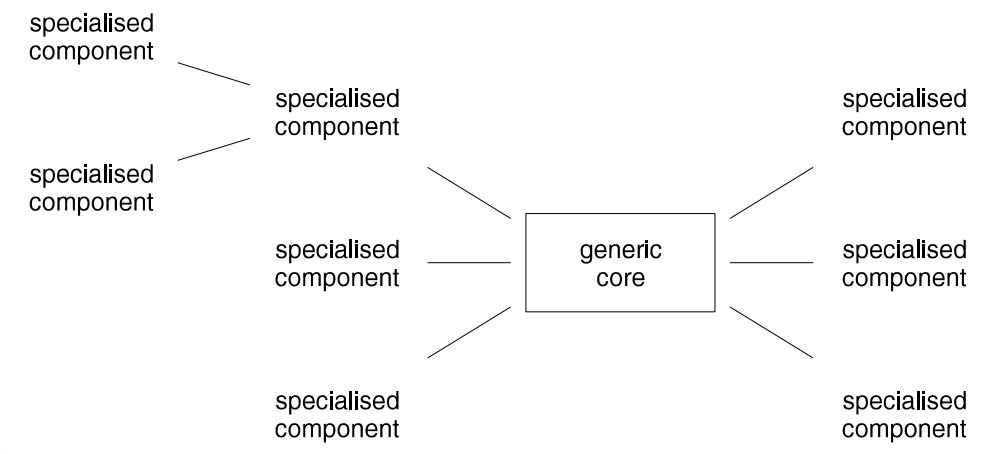

Figure 7.2: Conceptual diagram showing the division between the generic core and the specialised components.

gory of programming toolkit. The users of the architecture are envisaged to be researchers experimenting with the generative evolutionary design approach. The users of the evolutionary system implemented based on this architecture are envisaged to be design teams.

The proposed architecture uses an approach that is a generalised version of the concept-seeding approach (see section 6.2.3 on page 160). In particular, the architecture divides the evolutionary system into two parts: a *generic core* and a set of *specialised components*. The generic core provides the main infrastructure for creating an evolutionary design system. The specialised components, on the other hand, must be defined by the design team and will include a set of rules and representations that define the transformations performed by the evolution steps. Figure 7.2 shows the relationship between the generic core and the specialised components. During the evolutionary process, the generic core will invoke the specialised components, which may in turn invoke other specialised components.

As stated in chapter 1 (see section 1.2.1 on page 16), three different types of specialised components can be created: first, a set of *routines* routines can be created that capture a set of design ideas; second, environmental *data-files* can be created that capture information about the design constraints and the design context; and third, existing software *applications* can be used to support the developmental, visualization and evaluation routines.

Routines

The first and most important level of customization is the creation of a set of routines that encode the design schema. Routines, unlike data-files and applications, are not optional and must be defined for the evolutionary process to be able to run.

In chapter 4, a variety of rules and representations for the evolution

steps were described. Researchers have experimented with a wide variety of rules and representations. This experimental approach is necessary because there is little theory to guide the researcher regarding what the most appropriate rules and representations are in a specific situation. As a result, researchers must rely on intuition and trial-and-error experimentation.

A number of representations for genotypes were discussed, including string based representations such as binary strings and real-valued vectors, and more complex types of representations such as tree structures. For selection rules, rank based selection, tournament selection and Boltzman selection were discussed. Reproduction operators were also discussed, including versions for both binary strings and real-valued vectors. For complex representations such as those used in genetic programming, specialised operators are required. For calculating the fitness of an individual, various scaling and multi-objective scalarization techniques were introduced. In chapter 4, various strategies for creating rules and representations for the developmental step were discussed. Some of these strategies were used in the systems described in chapter 5.

By keeping the routines separate from the generic core, the architecture allows the design team to use any of these rules and representations. The design team can embed their rules and representations within the routines, which will then be executed by the generic core. The four evolution steps each have their own routines. In addition to this, there is also an initialization routine, a termination routine and a visualization routine. (These routines will be described in more detail below.) Together, these routines constitute the encoded schema.

The proposed design method described in chapter 6 discusses the process of capturing and codifying the design schema as a set of task-specific routines. The routines can be used by the evolutionary system to evolve design. This approach allows the conflict between the style problem and the variability problem (discussed in chapter 6, section 6.2.3 on page 159) to be resolved. To recap: on the one hand, if task-specific rules and representations are employed, the system's re-usability will be limited to a small group of designers. On the other hand, if task-specific rules and representations are *not* employed, the performance of the system will be poor. Dividing the evolutionary system into a generic core and a set of specialised components allows task-specific rules and representations to be employed while at the same time ensuring a reasonable level of re-usability.

Data-files

The second level of customization involves the creation of environmental data-files that describe significant aspects of the design constraints and design context. The developmental, visualization and evaluation steps may all make use of environmental information when processing individuals.

The constraints may include the budget, the number of spaces, spatial requirements, and performance targets. The context will include information about the building site and may also include large-scale conditions such as weather data. The constraints and context are encoded in a *data-file* that is available to the routines. This environmental information may be used as follows:

- For the developmental routine, the design team may specify an epigenetic generative processes that creates a design in response to both genetic information and environmental information. The architecture incorporates the approach explored by Frazer (1974); Frazer and Connor (1979); Frazer (1995b), whereby designs are generated in response to the environment (see section 5.6 on page 134).
- For the visualization routine, the design team may create visualizations that take into account environmental information. For example, neighbouring buildings, site dimensions, and other physical conditions may be visualised. The latitude may also allow realistic rendering of the design at different times of the year and day.
- For the evaluation routine(s), the design team may require information about the environment to perform the necessary evaluations. In addition to the environment data-file, some information may also specified in the simulation applications, to be discussed in the next section. For example, energy-related simulation applications use weather files that describe yearly weather patterns for a particular location.

The architecture allows the environment data-files to be modified and replaced independently from the routines.

Applications

The third level of customization allows for the use of existing software applications by the developmental, visualization and evaluation routines. Integrating such applications is likely to save time and lead to more accurate results.

- The developmental routine uses a generative process that may require various modelling functions. Basic geometric functions and representations, such as points, lines, planes and the interactions between them are likely to be required. In addition much more complex functions such as surface and solid modelling functions may also be required depending on the generative process.
- The visualization routine must provide an interactive interface that allows a three-dimensional model of a building to be interrogated by the user. In order to help the user understand the design, such an interface should allow models to be rotated, rendered, sectioned,

and printed. Many such interfaces exist already and may be used: some have been developed as stand-alone visualization programs for standard file formats, some are integrated in CAD applications, while others exist as plug-ins for Internet browsers.

- The evaluation routine evaluates the design models with respect to a set of objectives. For many objectives, existing applications already exist. For example, the website for the U.S. Department of Energy[1] lists nearly 300 different software tools for the analysis and simulation of energy and light in buildings. Any existing application that can be automatically executed by another program, and whose inputs and outputs are text based, may be used.

Rather than integrating existing applications, some researchers have have attempted to implement the required functionality from scratch. This is the approach used by Bentley (1996) in GADES. Bentley proposes developing a library of custom-written evaluation routines that are capable of evaluating any forms, even highly chaotic ones. The weakness in this approach has been highlighted by Bentley himself. He admits that such simplified custom-written routines are incapable of evaluating a design to the desired level of accuracy (Bentley, 1999b, p. 42). On the one hand, many of the processes required by these steps are difficult to implement from scratch, and on the other hand advanced implementations are readily available in existing applications.

The integration of an evolutionary system with existing analysis and simulation applications has been explored by Caldas (2001) and implemented in GS, the parametric evolutionary design system developed by her. To recap, GS evolves three-dimensional design models (represented using the BDL language) that are evaluated with regard to thermal performance and lighting performance. The evaluation step invokes a DOE-2 simulation application to perform an hourly simulation of the designs. The results of the simulation are returned to the evolutionary system and used to calculate a fitness for each design. However, GS is only capable of using the DOE-2 program. As Caldas herself suggests, it should be possible to also integrate other applications. Most building designs need to be evaluated for multiple objectives, and as a result these evaluations may need to invoke different analysis and simulation programs. The architecture allows for the possibility of integrating more than one application.

The integration of CAD modelling applications for both generating forms and visualising forms has been explored by Frazer (1995b) and his students in a variety of different systems. The main applications used have been AutoCAD[2] © and MicroStation[3] ©. Evolutionary systems were developed that were fully integrated in the CAD application. For

[1] http://www.eere.energy.gov /buildings/tools_directory/
[2] http://www.autodesk.com
[3] http://www.bentley.com

example, the Interactivator (see section 5.6.2 on page 136) was developed using a programming language embedded in AutoCAD[4], thereby allowing the generative process to make direct use of the AutoCAD modelling functions. In addition, AutoCAD was also used for visualization, with all feedback from the evolutionary process being displayed in the AutoCAD window. The evolutionary system developed by Sun (2001) (see section 5.5.2 on page 132) was similarly integrated with MicroStation. As with the Interactivator, both the developmental step and the visualization step made extensive use of modelling and visualization functionality available in MicroStation.

Two key obstacles to creating such an architecture may be identified: first, analysis and simulation applications are computationally expensive and if multiple applications are used, the evolutionary process may come to a virtual standstill. Second, the various applications often have conflicting requirements in terms of operating systems and hardware requirements, and cannot be run on the same computer. Both these obstacles are resolved by using the parallel architecture discussed above.

7.3 Overview of architecture

The proposed architecture was broadly described in chapter 1, and figure 1.4 on page 21 shows the overall structure. This structure can be further broken down into a more detailed set of interacting and communicating program components. Figure 7.3 on the next page shows a more detailed view of the proposed architecture. This section describes the main features of the architecture, some of which have already been discussed in section 7.2 above.

The architecture divides the evolutionary system into two parts: a *generic core* and *specialised components*. The generic core does not need to be modified or recompiled by the design team and has a high level of re-usability: it is applicable to a wide range of design tasks. The specialised components must be defined by the design team and are specific to a particular design schema. The generic core and the specialised components must manipulate and exchange the same set of individuals.

7.3.1 Individuals

The parts of an individual

Each individual in the population is broken down into a generic part and a specialised part. The generic part includes two representations: a set of possible flags and a unique ID.

- Flags are boolean values used to store information about the individual. The *phenotype* flag is true if the individual has been

[4]AutoLisp, a dialect of the LISP language, was used. Programs written in AutoLisp can directly call modelling and visualization functions in the AutoCAD system.

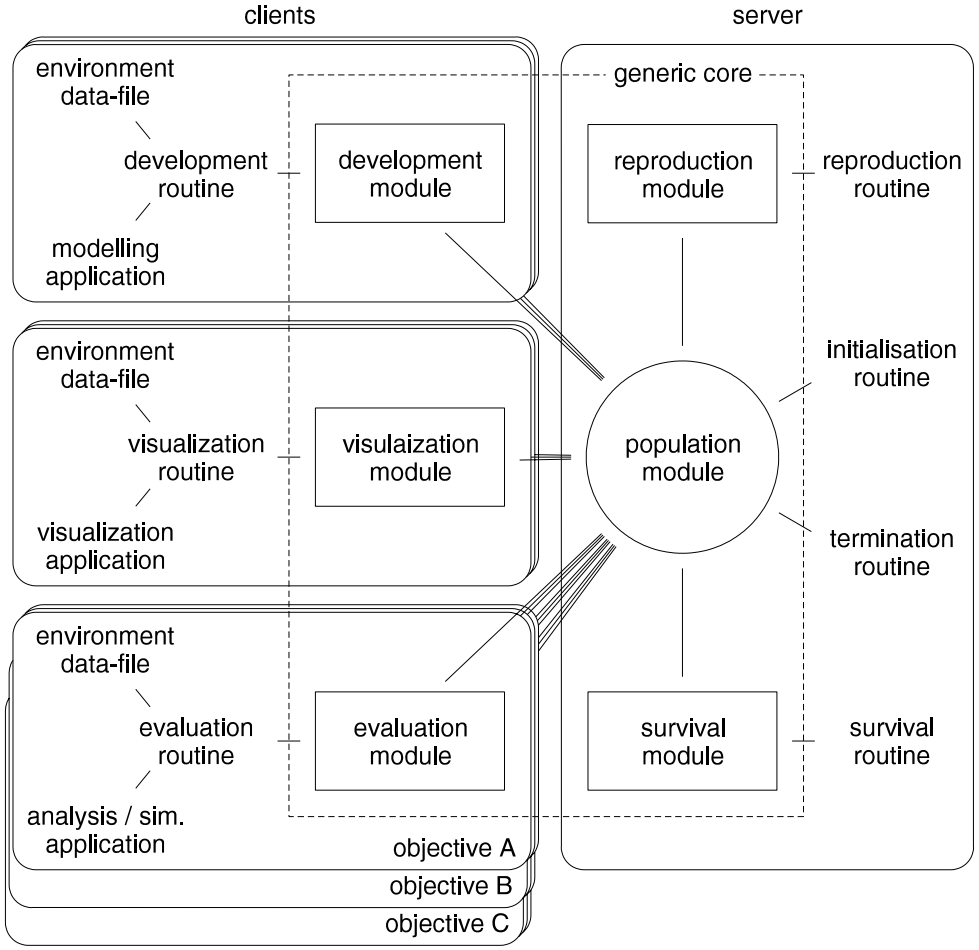

Figure 7.3: The main components of the proposed architecture.

developed, and one or more *evaluation* flags are set depending on which evaluations have been performed. By default, these flags are all set to false. In addition, the *checked out* flag is used to indicate that the individual is being processed by one of the evolution steps.

- The ID is an integer value that is unique in the history of the evolutionary process. The first individual to be added to the population is assigned an ID of 0, and thereafter the ID is incremented for each new individual added. The ID indicates the relative age of the individual. (This age may be of use to some of the evolution routines. For example, the survival routine may choose to delete the oldest individuals.)

The specialised part of an individual consists of three representations: a genotype, a phenotype and a set of evaluation scores.

- The genotype may have any type of representation. For example, the genotype may be a fixed length binary or real-valued string, a variable length string, or some other more complex type of data-structure.

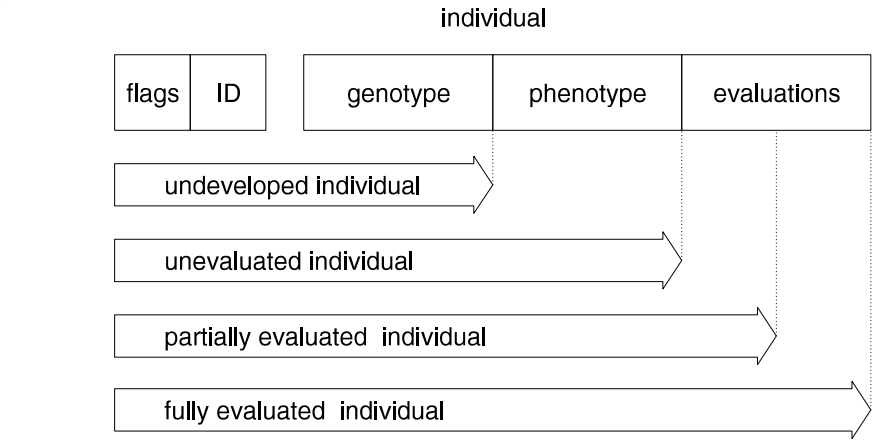

Figure 7.4: The structure of an individual in the population, and the four states an individual may be in.

- The phenotype may also have any type of representation. For example, the phenotype may use some standard format for describing three-dimensional form, or it may use a customised schema-specific data format.
- The evaluations consist of a list of one or more representations. Each individual representation may also be of any type, but will often simply consist of a real number. When an evaluation is performed, the resulting representation is stored in the list.

The state of an individual

An individual in the population may be in a number of *states*, depending on whether the individual has been or is in the process of being developed and/or evaluated. The state of an individual can be deduced from its flags.

Individuals may be described using a number of terms, which relate to the state of the individual. Figure 7.4 shows four commonly used terms, and how these terms relate to the state of an individual.

- An *undeveloped individual* has a genotype, but no phenotype and no evaluation scores.
- An *unevaluated individual* has a genotype and a phenotype, but no evaluation scores.
- A *partially evaluated individual* has a genotype, a phenotype and some, but not all, evaluations have been performed. (In this case, it must be a multiple-objective evolutionary process.)
- A *fully evaluated individual* has a genotype, a phenotype and all evaluation scores.

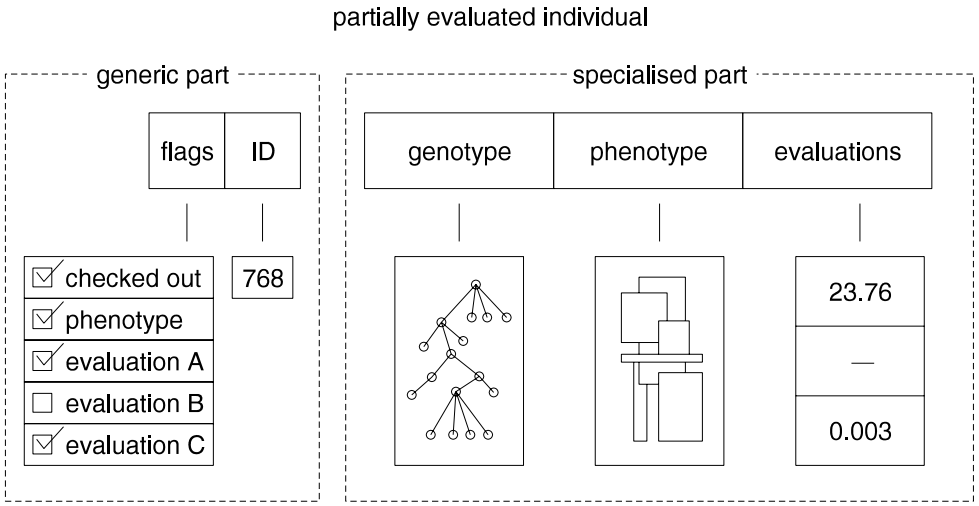

Figure 7.5: Sub-representations of a partially evaluated individual.

An example individual

Figure 7.5 shows an example of a possible individual being evolved for three objectives, labelled A, B, and C.

- The flags show that the individual a phenotype and that two of the objectives — objective A and C — have been evaluated. The flags also show that the individual is currently 'checked-out', which means that one of the evolution steps is processing the individual; this must be the evaluation step for objective 'B'.
- The ID shows that this was the 768th individual to be added to the population. This gives some indication of the age of the individual.
- The diagram of the genotype suggests some kind of hierarchical representations.
- The diagram of the phenotype suggests some kind of orthogonal design that may be represented in a number of ways, including standardised file formats.
- The evaluation scores show that out of three scores, the first and the last have values while the second is blank. This corresponds to the flags.

7.3.2 Specialised components

Types of specialised components

The architecture defines three type of specialised components: *routines*, *data-files*, and *applications*.

- Routines are small user-defined programs that encapsulate the rules and representations used by the evolutionary system. Routines are executed by the generic core.
- Data-files consist of encoded information that describe the design environment, including both the design constraints and the design context. Data-files are accessed by routines.
- Applications are third party software applications that may be integrated into the evolutionary system to perform certain tasks. Applications are invoked by routines.

Types of routines

Routines are created by the design team and encapsulate the rules and representations that define the design schema. These routines are modified and replaced depending on the design schema that is being used. In total there are seven routines: two *population routines* executed by the population module, four *evolution routines* executed by the four evolutionary modules, and one visualization routine executed by the visualization module.

- The two population routines are the initialization routine and the termination routine. The initialization routine is used to create a population of new individuals and specifies the size of the population. In most cases, genotypes will be created randomly. The termination routine specifies a set of termination conditions such as the maximum number of reproductions or a set of evaluation targets.
- Evolution routines all specify a single transformation that transforms a set of one or more input individuals into a set of one or more output individuals. The routine has no direct connection with the population module.
- The visualization routine converts the design model into a data format that can be visualised by an existing application. For example, the visualization routine may convert the design model into a VRML model that can be viewed in a web browser with a VRML plug-in.

These routines, together with the associated representations, constitute the encoded schema as shown in figure 6.1 on page 162.

Manipulation of individuals

The routines directly manipulate the representations in the specialised part of an individual: the genotype, phenotype and evaluation scores. The routines should not change the generic parts: the flags and the ID.

- The reproduction routine will use the evaluation scores to select parents, and will the use parent genotype to construct new genotypes for the offspring.
- The developmental step will use the genotype representation to create a new phenotype representation.
- The visualization step will use the phenotype representation to visualise the design model, but will not actually change any of the representations.
- The evaluation step will use the phenotype representation to create a new evaluation score.
- The survival step will use the evaluation scores to select individuals for deletion.

Table 7.6 on the next page shows the type and number of input individuals and the type and number of output individuals. The first and second columns shows the type and number of individuals that the evolution module must ask for from the population module. The third and fourth columns show the type and number of individuals that an evolution module must send to the population module.

The developmental, visualization and evaluation routines may make use of either design environment data-files or of other existing software applications. The applications most likely to be employed are CAD modelling applications for the developmental step, three-dimensional visualization applications for the visualization step, and analysis and simulation applications for the evaluation step. The applications are invoked and controlled by the corresponding routines, which in turn are controlled and invoked by the corresponding module components.

7.3.3 Generic core

Generic components

The generic core is broken down into a set of *modules*, which are the main program components that define the overall structure of the architecture. In total, there are six modules: a population module, four evolution modules that perform the evolution steps and a visualization module.

These modules interact using an asynchronous evolution mode, a decentralised control structure, and a global parallel client-server model. The server hosts the population and executes the reproduction and survival modules, while multiple clients execute the developmental and evaluation modules. Clients may be duplicated to the point where communication costs start to outweigh computation costs.

The visualization module allows designers to view design models in the population. This module does not affect the evolutionary process in any way. A user interface for selecting individuals in the population may be provided.

	input type	input quantity	output type	output quantity
reproduction routine	fully evaluated individual	one or more	new individual (no ID)	one or more
development routine	undeveloped individual	one	developed individual	one
visualization routine	developed individual	one	———	———
evaluation routine	unevaluated individual	one	evaluated individual	one
survival routine	fully evaluated individual	one or more	individuals (to be deleted)	one or more

Figure 7.6: Input and output of individuals for evolution routines.

Evolution modules

The *evolution modules* perform the evolution steps that process individuals in the population. In total, there are four evolution modules: reproduction, development, evaluation and survival.

These modules will repeatedly perform three actions: they make requests to the population module for one or more individuals, they process these individuals, make requests to the population module to send back the results of the processing. An evolution module sends two types of requests: *get-requests* and *post-requests*. A get-request asks the population module to *get* a certain number of individuals from the population. A post-request has a set of individuals appended to it, and the request asks the population module to perform some action with the appended individuals. The type of action performed depends on the type of module sending the request.

Figure 7.7 on the following page shows the main actions performed by the evolution modules.

- The evolution module will send a get-request to the population module asking for new individuals. If the population module cannot provide the required individuals, for whatever reason,a wait instruction will be sent to the evolution module. In such a case, the evolution module will simply wait for a short time, and then repeat the request.

- Assuming that the population module was able to provide the required individuals, the evolution module will now execute its corresponding evolution routine. The individuals from the population

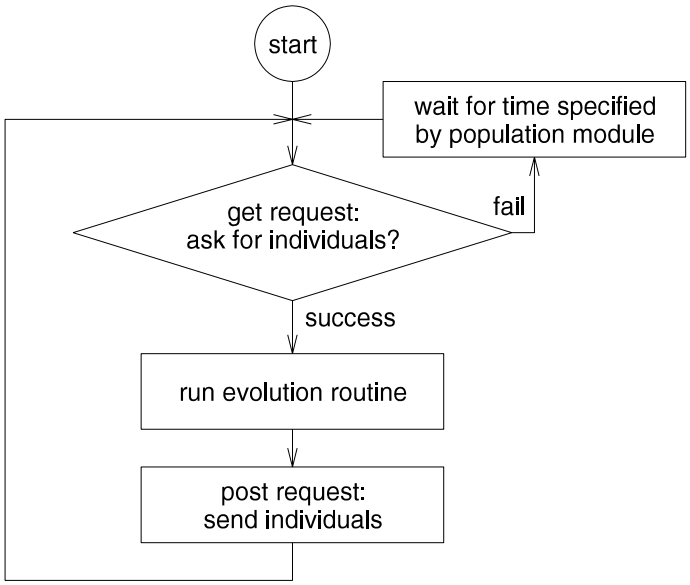

Figure 7.7: Flow diagram of the main actions performed by the evolution modules.

will be provided to the routine as an input. The output will always consist of another set of individuals.

- The evolution module will send a post-request to the population module. The results from previous execution of the evolution routine will be appended to this request.

Population module

The *population module* is the central module that all other modules must communicate with. This module performs three actions: it initialises the population; it responds to get- and post-requests; and it verifies if the termination conditions have been met.

The main loop performed by the population module is shown in figure 7.8 on the next page. After initialization, the evolutionary process can be started. The population module repeatedly performs three main actions.

- The population module first checks if the termination conditions have been met. The termination conditions may either be empty, in which case the process will continue until it is terminated by the user, or it may specify a condition such as the maximum number of reproductions or the fulfilment of a particular evaluation objective.

- The population module passively waits for requests from the evolution modules. Request may arrive at any time and in any order. Requests will be dealt with one at a time in the order that they arrive.

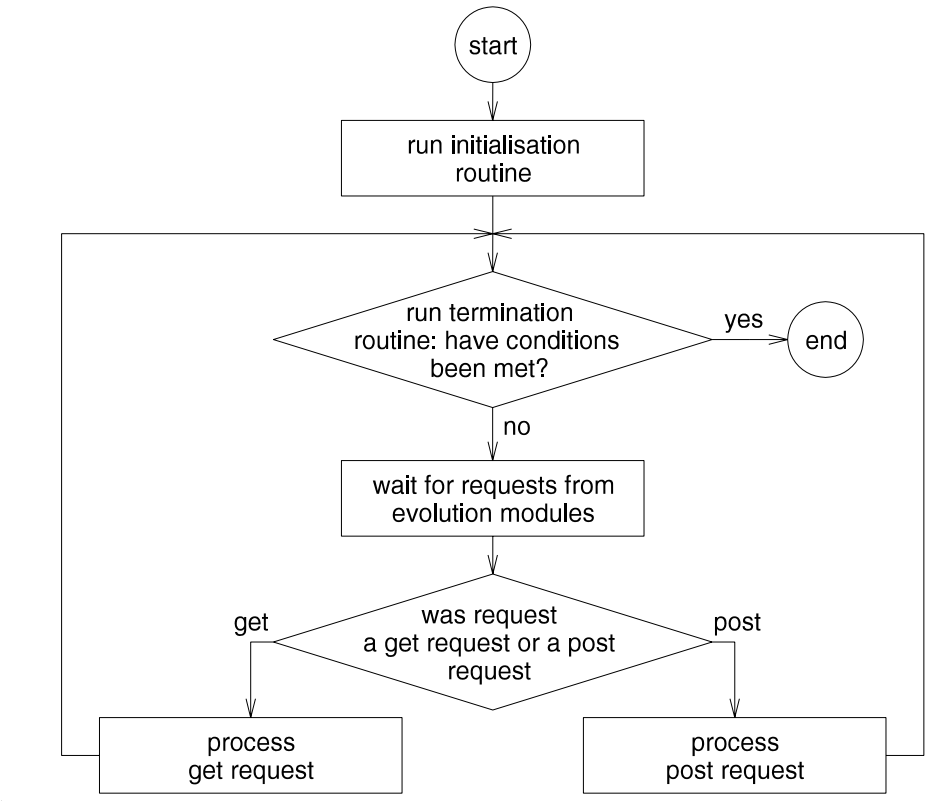

Figure 7.8: Flow diagram showing the main loop for the population module.

- If the population module receives a request, it processes the requests in the appropriate manner depending on whether it is a get- or a post-request.

The way that the population module processes get and post-requests is discussed in more detail below. Figure 7.9 on the following page shows how a get-request is processed, and figure 7.10 on page 190 shows how a post-request is processed.

Response to a get-request

When the population module receives a get-request from one of the evolution modules, it must attempt to send the requesting module the required individuals. Each evolution module requires individuals in a particular state and the population will consist of individuals in different states: some individuals will be undeveloped, some will be unevaluated, some will be partially evaluated and some will be fully evaluated. It is up to the population module to attempt to find individuals in the population in an appropriate state. It will scan the population and identify all appropriate individuals, referred to as the possible *candidates*. If there are enough candidates in the population to fulfil the get-request, the population module will randomly select the required number of individuals from

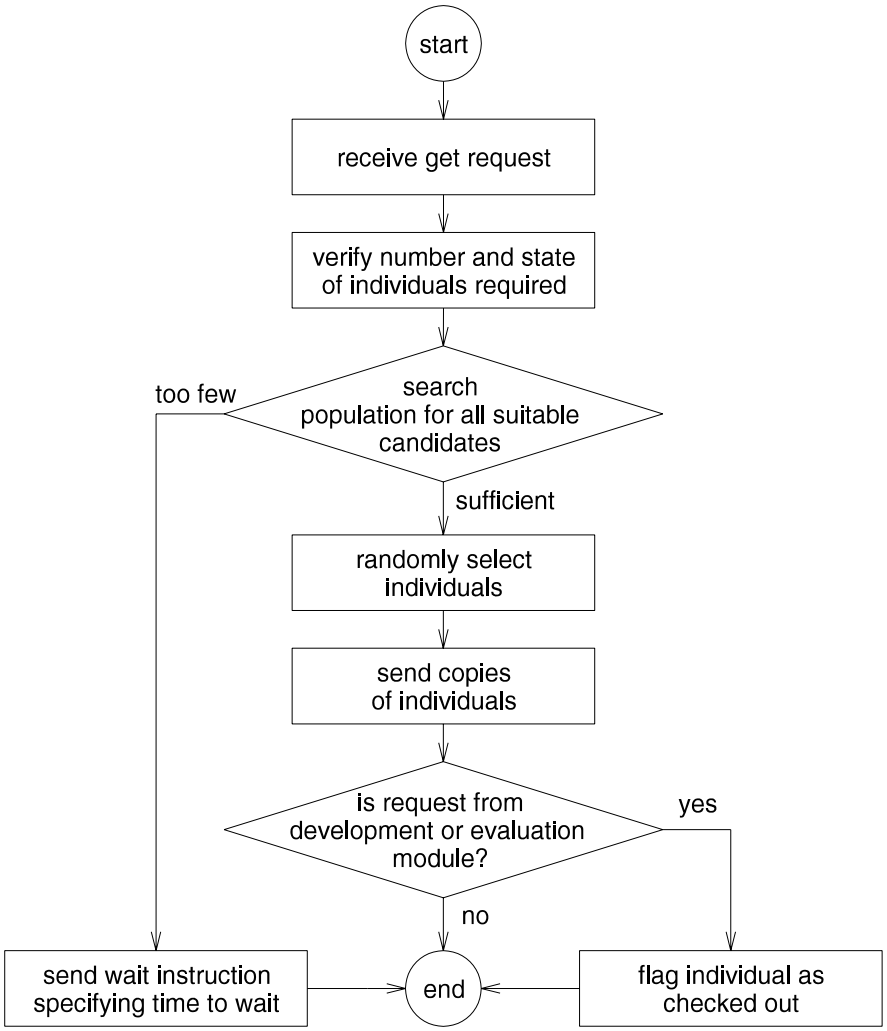

Figure 7.9: Flow diagram showing the response of the population module to a get-request.

the candidates and send copies to the requesting module. The originals in the population are flagged as being 'checked out'.

Figure 7.9 shows the main actions performed by the population module when it processes a get-request.

- The population module receives a get-request, which will specify the number and state of individuals that are required. The required state depends on the type of module making the request, and the required number of individuals is specified as an initialization parameter.

- The population module will search the population for possible candidates. It will disregard any individuals flagged as 'checked out'. If the population contains too few candidates, it will send a wait instruction back to the requesting module, and then terminate.

- Assuming that sufficient candidates were found in the population,

the population module randomly selects the required number of individuals from the candidates. Copies of the selected individuals are made and sent to the requesting module.

- If the requesting module is a developmental or evaluation module, the selected individuals in the population also need to be flagged as being checked out. Flagging the individuals avoids them being developed or evaluated multiple times.

Response to a post-request

When the population module receives a post-request, one or more individuals will be appended to the request. The population module will respond differently depending on the type of evolution module that is sending the request. When the population module receives a put request from the reproduction module, it will add the appended individuals to the population, creating new IDs in the process. When the population module receives a put request from either the developmental or evaluation module, it will use the ID of the appended individual to find the matching individual in the population and replace it. When the population module receives a put request from the survival module, it will delete the existing individuals in the population with IDs that match the appended individuals.

Figure 7.10 on the next page shows the main actions performed by the population module when processing a post-request. (See figure 7.8 on page 187 for a view of the overall behaviour of the population module.) It performs six actions.

- The population module receives a post-request with a set of individuals appended to the request.
- If the requesting module is the reproduction module, new unique IDs can be created for each appended individual, and the individuals can be added to the population.
- If the requesting module is not the reproduction module, the appended individuals will have matching individuals in the population. The population module finds the matching individuals by searching for their IDs in the population.
- If the requesting module is the survival module, the individuals found in the population must be deleted.
- Otherwise, the requesting module must either be a developmental or evaluation module, and only a single individual will be appended to the request. The flags for this individual must be updated to reflect its new state.

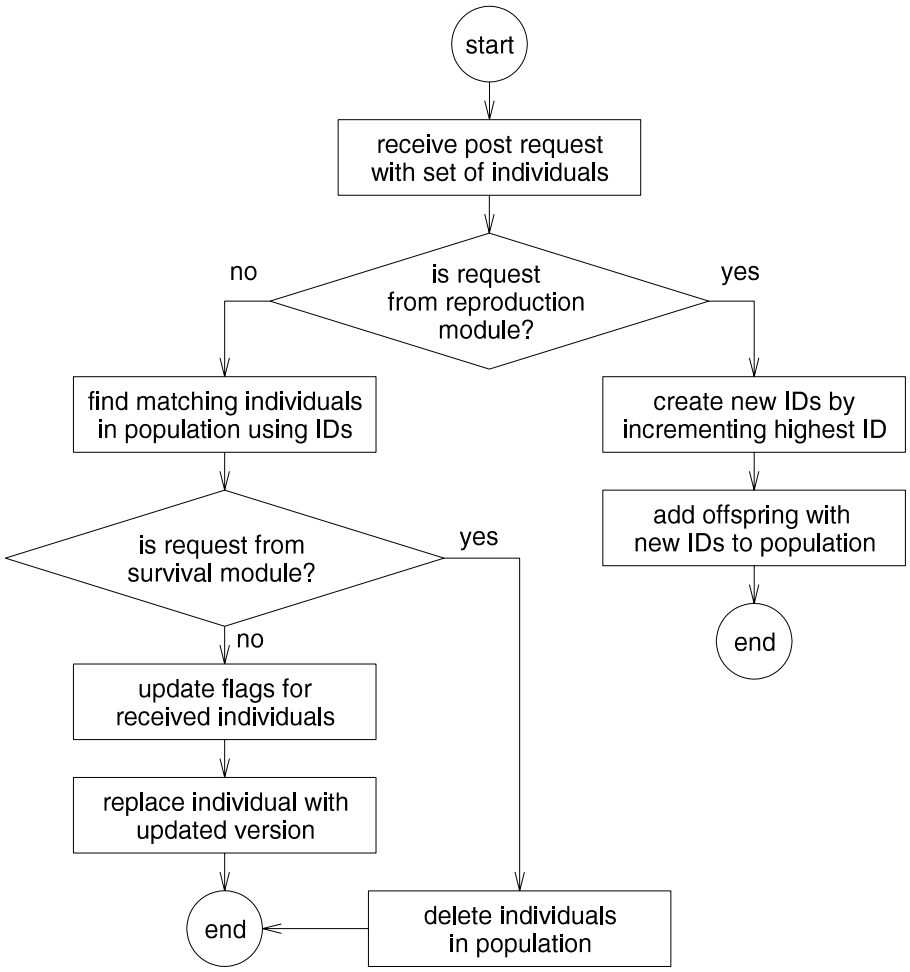

Figure 7.10: Flow diagram showing the response of the population module to a post-request.

- Finally, the appended individual will replace the individual found in the population. This will also automatically remove the flag indicating that the individual was checked out.

7.3.4 Interactions between components

Four evolution steps

Each of the four evolution steps includes a generic module and a specialised routine. The evolution modules repeatedly contact the population module for one or more individuals, process these individuals by executing the routine, and contact the population module again to send back the new or updated individuals. Once an evolution module has completed this process, it will immediately contact the population module again and repeat the process. If all individuals in the population have already been processed, the population module will ask the evolution module to wait for a short time. The time period for this *waiting instruction* can be adjusted by the population module and is the only

control that it has over these modules.

The four evolution steps were briefly discussed in section 7.2.1 on page 172, and will be described here in more detail.

- The reproduction module requests a pool of fully evaluated parents, which the population module will randomly select from all suitable candidates. One or more new individuals are produced by executing the reproduction routine. The size of the pool is specified as an initialization parameter. If two parents reproduce to create one child, the reproduction routine may ask for a pool of ten parents and create five children. Alternatively, the reproduction routine may be designed to impose some selective pressure by choosing the fittest two parents from the pool, and creating just one child.

- Each developmental module requests a single undeveloped individual, which the population module randomly selects from all suitable candidates, and flags as checked-out. The individual is then developed by executing the developmental routine.

- Each evaluation module requests one individual that has not been evaluated for the objective, which the population module randomly selects from all suitable candidates, and flags as being checked-out. The individual is evaluated by executing the evaluation routine. Such an evaluation is an attempt to predict the performance of the design model, should the design be implemented.

- The survival module requests a pool of fully evaluated individuals, which the population module randomly selects from all suitable candidates. One or more individuals are selected for deletion by executing the survival routine. The size of the pool is specified by the survival routine. For example, the survival routine may deterministically choose the individuals in the pool with the worst evaluations. (This is tournament selection, with the pool constituting the tournament.) If multiple objectives are evaluated, the survival routine will need to use a scalarization technique to rank the individuals in the pool.

Size of pool

The asynchronous evolution mode allows individuals in the population to be added one at a time. The evolutionary process does not need to wait for the whole population to be developed and evaluated. As soon as an individual has been evaluated, the evolutionary process can start to either discard or incorporate its genetic information.

For both the reproduction step and the survival step, the size of the pool may be specified in advance as one of the initialization parameters. However, defining an appropriate pool sizes is complicated by the fact that these sizes will affect both the asynchronous evolution mode and the selection pressure:

- The larger the pool, the more likely it is that requests for individuals will be rejected due to there being too few suitable individuals in the population at that point in time. As the size of the pool approaches the population size, such rejections will become more common. The evolution mode will become similar to the synchronous evolution mode.

- The smaller the pool, the lower the selection pressure will be, with the evolutionary process becoming more exploratory. As the pool size approaches one, the evolutionary process will become similar to random search.

A balance must be found between these conflicting requirements. One solution is to allow this size to vary dynamically, and to be equal to the number of candidate individuals in the population at the time. This would ensure that the largest possible pool of individuals would be used without requiring the evolutionary process to stall.

Evaluation of multiple objectives

If multiple objectives are being evaluated, at least one evaluation module per objective must be defined. Each evaluation module will be associated with a different evaluation routine, and each evaluation routine may make use of a different analysis or simulation application.

For example, a design schema may include three main objectives: minimization of the building's construction cost, minimization of building's energy consumption and maximization of interior daylight levels. Three routines would need to be defined: a cost analysis routine, an energy simulation routine, and a daylight simulation routine. Each routine would be controlled by its own module. For example, the cost analysis routine would be controlled by its own module that would specifically request individuals from the population that had not yet had their cost evaluated. Similarly, the energy and daylight simulation routines would each be controlled by their own modules.

Different objectives are evaluated independently from one another. An individual in the population may have multiple evaluations performed in any order. For example, the cost module will request any developed individual that has not had its cost evaluated. Whether or not the individual has had daylight levels or energy consumption simulated makes no difference.

In an ideal situation, each objective will have one or more clients assigned to it. However, the number of clients used will in practice depend on the computing resources available. If such resources are limited, it is possible for a client to be assigned to more than one type of evaluation, or to be assigned to perform both the evaluation step and the developmental steps. It is even possible to run the whole system on a single computer, acting as both client and server.

Controlling the size of the population

One consequence of using a decentralised approach is that the population size will naturally tend to vary over time. The variance in the population size will depend on the rate at which the reproduction module adds new individuals to the population and the rate at which the survival module deletes individuals. If some form of control were not applied, the population size would be unlikely to remain stable. The population module must control the size of the population by issuing appropriate waiting instructions to the reproduction and survival modules.

Before the evolutionary process starts, the population size is predefined. The population module attempts to maintain the population at this size. If the population is greater than the population size, the reproduction module will be instructed to wait. If the population is smaller than the population size, the survival module will be instructed to wait.

7.4 Implementation strategies

In order to implement an evolutionary system based on the proposed architecture, various decisions need to be made concerning the languages and technologies used. Languages and technologies for three key areas would need to be decided upon: the generic core, the individuals, and the specialised components. Each of these areas have different requirements.

7.4.1 Language and technologies for the generic core

Key requirements

The generic core consists of six different modules distributed between one server and a set of clients. The server hosts the population module and two evolution modules. The other three modules — the developmental, visualization and evaluation modules — reside on client computer systems that communicate via the network with the population module. One of the key requirements for the generic core is a language that has good networking capabilities. The other requirement is that the modules residing on the client computers remain platform independent. This allows clients to use the operating systems required by existing applications.

Java is suggested as an ideal language because it has built in support of standard networking technologies, and is platform independent. There are a number of Java-based technologies that may be used to implement the communication between the modules. Two possible approaches that exploit existing technologies are identified: an implementation using *Java Servlets* technology and an implementation using *JavaSpaces* technology.

Implementation using Java Servlets

Java Servlets is a technology that facilitates building server based web applications. Servlets are Java programs that reside on a server and are managed and executed by a server software — called a *servlet container* — that provides network services for communication between the server and its clients. Clients interact with the servlet container using the request-response model and will typically use the Hypertext Transfer Protocol (HTTP).

A program on the client will send a HTTP request to the server. The container will receive the request and, depending on the type of request, will pass the request onto a servlet. After the request, the servlet will perform some kind of processing, possibly accessing resources stored in a database, and will produce a HTTP response. The container will pass the response back to the client.

In many cases, the program running on the client will be a web browser, and the request will be for a web page which the servlet generates dynamically. However, the servlet programming paradigm is flexible and powerful, and is not limited to generating web content. The paradigm has also been used to develop a wide variety of distributed applications.

For the proposed architecture, this approach may be used to implement the interaction between modules. The population module could be implemented as a servlet. The servlet would store the population in a database and would read and write individuals to the database each time a request was received. The other modules would be implemented as client programs that communicate with the servlet using the HTTP protocol. The HTTP protocol allows for various types of requests to be sent, including `GET` requests and `POST` requests. These may be used to either ask for or send individuals to the population module.

The advantage of using this approach is that many aspects relating to communication, persistence, and scalability are taken care of by the servlet container. Containers implement the Java Servlet specification developed by Sun, and Tomcat[5] ©is a freely available servlet container that is used in the official Reference Implementation. Many other servlet containers have been developed, and are usually a component of web and application servers[6]. Such servlet containers also typically provide web based user interfaces that allow web applications and servlets to be managed remotely.

Implementation using JavaSpaces

JavaSpaces is a technology that facilitates building distributed applications in networked environments. Building distributed applications with

[5]http://jakarta.apache.org/tomcat/

[6]For example, commonly used web and application servers include BEA WebLogic Application Server, IBM WebSphere, Sun Java System Web Server, Sun Java System Application Server

conventional network tools usually entails passing messages between processes or invoking methods on remote objects. With the JavaSpaces framework, in contrast, processes do not communicate directly, but instead coordinate their activities by exchanging objects in an area of shared memory, called a *space*. A client can *write* new objects into a space, *take* objects from a space, or *read* objects in a space. When taking or reading objects, processes use simple matching, based on the values of fields, to find the objects that matter to them. If a matching object is not found immediately, a process can wait until one arrives.

For the proposed architecture, the population module could be implemented as a space, with individuals in the population existing as objects in the space. The other modules in the generic core would be implemented as clients that write, take and read objects from the space. The use of JavaSpaces technology for implementing an evolutionary system was first proposed by Setzkorn and Paton (2004).

Objects in spaces are identified using a matching process that is similar to the matching process performed by the population module. With the population module, when one of the other modules makes a request for one or more individuals, it needs to find individuals in the population in an appropriate state. By specifying a template that contains values for a set of fields, the matching process can be performed by the JavaSpaces framework. If the values for the fields correspond to the required state of the individual, any objects returned will be individuals in the correct state.

As with the Java Servlet approach, the advantages of this approach is that communication, persistence, and scalability are all taken care of by the JavaSpaces framework. In addition, the communication costs of this approach are likely to be lower than the Java Servlet approach.

7.4.2 Language and technologies for representing individuals

Key requirements

The generic part of an individual consists of simple types of representations with fixed data-structures. The flags may simply consist of a variable length list (depending on the number of objectives) of boolean values, and the ID is a single integer value. The specialised representations, however, may be much more complex. Furthermore, the data-structures for these representations cannot be specified in advance because different design teams are likely to want to experiment with different types of data-structures.

A flexible type of representation is required that allows design teams to experiment. The most complex representation is likely to be the phenotype representation. This representation must go beyond the types of geometry-based representations used in CAD modelling applications. As well as geometrical information, the design model must also incorporate

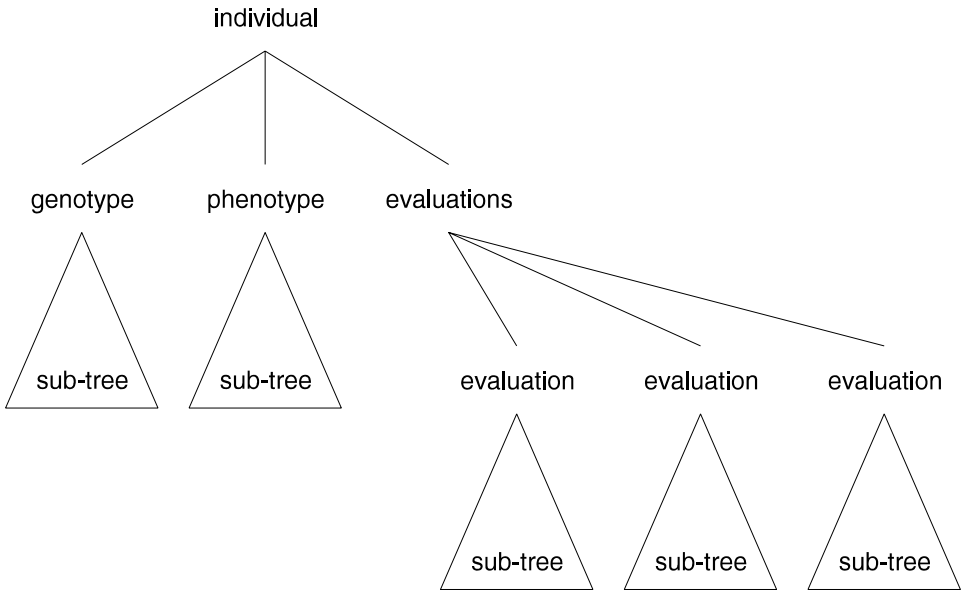

Figure 7.11: Individual represented using a single tree structure with various sub-trees.

information required by analysis and simulation applications. Such applications typically require three type of information: component-based information (e.g. cost estimation), space-based information (e.g. thermal simulation), and networked based information (e.g. structural analysis) (Mahdavi, 1998).

Implementation using XML

An appropriate language for creating such representations is *eXtensible Markup Language* (XML), a flexible text based data format for structured data. XML may be thought of as a meta language which allows data representation languages to be defined. XML is also an open standard, is human-readable, and is widely supported by most common programming languages including Java.

XML data is defined as a set of nodes structured as a hierarchical inverted-tree. Each node may have a set of attributes and one or more sub-nodes. (Non-hierarchical references between nodes allows most types of commonly used data-structures — such as lists and graphs — to be easily represented.) If XML is used to represent an individual, each individual would be represented by a single tree structure. A possible structure for this tree is shown in figure 7.11. The top node is the *individual* node which has an *ID* attribute and a number of *flag* attributes. This node has three sub-nodes called *genotype*, *phenotype* and *evaluations*. The *genotype* and *phenotype* nodes would have sub-trees (shown in the figure as triangles) containing the genotype and phenotype data. The *evaluations* node has one or more *evaluation* nodes that each have sub-trees that contain the results from each evaluation.

For representing design models, XML may be used to define a custom representation incorporating all the information required by the analysis and simulation applications. In the AEC/FM industry, the development of comprehensive building representations that can be used by a wide variety of software applications, referred to as *interoperability*, has become an important research area.

Interoperability between software is based on the idea of a *Building Information Model* (BIM) (sometimes described as a *virtual building model* or *building product model*). A BIM is a term used to describe a type of representation developed specifically for describing objects and relationships specific to buildings. The Industry Foundation Classes (IFC) by the International Alliance on Interoperability (IAI) represents the latest effort, jointly by research organizations and commercial vendors, to develop a BIM (Eastman, 1999). The analysis and simulation applications now capable of importing the IFC BIM include thermal comfort applications, energy simulation applications, airflow simulation, structural analysis applications, and so forth. Furthermore, the IFC BIM is also supported by a number of key modelling applications, including Autodesk, Bentley, Graphisoft, and Nemetschek.

The IAI have also developed a version of IFC that uses XML, called ifcXML. This type of representation may be appropriate for use as a representation for the design model. The key advantage of using a standard representation such as ifcXML is that it would make the integration of analysis and simulation applications easier. Using a standardised representation that is compatible with these applications would avoid the need for conversion routines that translate one representation into another representation. This also applies to the visualization of design models.

7.4.3 Language and technologies for specialised components

Key requirements

Routines encode all the evolutionary rules and representations and must be defined as small programs that can be executed by one of the Java modules. When executed, the routines must process one or more individuals, which are defined using XML. In addition, routines must be able to read data stored in the environmental data-files, and they must be able to execute or interact with existing applications. As with the modules, routines must also be platform independent, thereby ensuring that clients are not limited to using only one operating system.

The links between the Java modules and the routines need to be implemented in a way that does not require the modules to be re-compiled every time the routines are modified or replaced. The links should be specified in a flexible manner. This could be achieved by using a configuration file that specifies the routines to be executed and that is read

by the Java module at the time of execution. However, in addition to executing the routine, the Java module also needs to pass the XML data of the individuals to the routine for processing, and subsequently needs to retrieve the results from the processing, which will also be a set of XML data.

Two key implementation issues need to be decided: first, the languages and techniques used for implementing the link between the modules and the routines; and second, the language used to define routines. For creating the link between the modules and routines, a language is proposed — called the *Extensible Stylesheet Language: Transformations* (XSLT) — that allows rules to be defined for manipulating and transforming XML data. Java is again proposed as the most suitable language for routine. There are a number of other languages that may be used, these are described below.

Links using XSLT

XSLT is a language that can be used to specify a set of rules that transform one XML tree, called the *source tree* into another XML tree, called the *result tree*. In order to perform the transformation, a program called an *XSLT processor* is required. Such a processor will, when provided with an XML source tree and a set of XSLT transformation rules, generate an XML result tree. Many processors have been developed that can easily be embedded in another program. The Apache Xalan-Java processor is a Java implementation of such a rule processor.

The XSLT language allows a set of transformation rules to be defined, each consisting of a *template* and a *pattern*. An iterative matching process is used that attempts to match the templates to structures in the source tree, and if a match is found the pattern of the matching rule is added to the result tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, nodes from the source tree can be filtered and reordered, and arbitrary structure can be added.

Within the proposed architecture, the evolution modules may each incorporate an XSLT processor and use this processor to transform a source tree that represents one or more individuals. Since XSLT cannot process more than one tree, multiple individuals need to be aggregated into a single tree. For each get-request, the population module will create single a source tree that contains one or more individuals. These individuals may, for example, have been requested by the reproduction module as parents to be used to create new offspring. This source tree is sent to the requesting module, and this module applies the rules specified in the XSLT file to the tree structure, thereby generating a result tree. The population module receives the tree structure, disaggregates it into separate trees for each individual, and then performs the appropriate actions.

Although the XSLT language is powerful, it is not suitable for the

performing complex types of processing typically required in the developmental and evaluation steps. For such transformations, functions can be called to perform the required processing. The XSLT language includes a set of functions that can be used in rules to perform various functions. In addition to the standard function provided, users may also define their own functions — called *extension functions* — and call these functions from within their XSLT rules. When a function is called, the function can be passed to any part of the XML source tree for processing, and the result of the processing can be placed in the result tree.

The routines for the evolution steps can be implemented as a set of functions. The link between the module and the routine can be specified in a file that contains a set of XSLT rules that call the required functions.

Each of the modules could be implemented in a similar way: each module would read a file and extract a set of XSLT transformation rules that include calls to functions defined as separate routines. The population module would also be implemented in a similar way with respect to initialization and termination routines. The XSLT transformation rules could be defined as a set of files created by the design team. The generic core would not have to be changed in any way. Another advantage of using XSLT is that it is a standard developed by the World Wide Web Consortium. This means that proprietary languages do not have to be learnt.

Language for routines

Extension functions for XSLT may be written in Java and in a variety of scripting languages. The website for the Apache Xalan-Java processor lists the following scripting languages: Javascript[7], Python[8], Tcl[9], NetRexx[10], VBScript[11], PerlScript[12], Groovy[13], and ObjectScript[14]. The possibility of using powerful scripting languages rather than full-blown programming languages may be a useful feature because it considerably reduces the knowledge threshold for design teams interested in developing and encoding design schemas. However, it is unclear whether these types of languages are capable of invoking and interacting with third-part applications. Java is therefore seen as a good compromise.

[7] http://www.mozilla.org/rhino
[8] http://www.jpython.org/
[9] http://www.scriptics.com/java
[10] http://www2.hursley.ibm.com/netrexx
[11] http://msdn.microsoft.com/scripting
[12] http://www.activestate.com/
[13] http://groovy.codehaus.org/
[14] http://objectscript.sourceforge.net/

7.5 Summary

This chapter has described and discussed the proposed computational architecture. The main points are as follows:

- The architecture is highly scalable. This is achieved by using an asynchronous parallel model in combination with a decentralised control structure. The asynchronous parallel model is similar to that used by the GADO system, and allows the execution time to be significantly reduced. The master-slave control structure used by GADO is replaced with a decentralised control structure using a client-server model. This control structure results in an architecture that is both more flexible and more robust.

- The architecture is highly customisable. This is achieved by dividing the evolutionary system into a non-customisable generic core and a highly customisable set of specialised components. The generic core incorporates no schema-specific knowledge, thereby ensuring that it can be used by any design team. The specialised components incorporate schema-specific knowledge. Three types of specialised components are defined: routines that encapsulate the evolutionary rules and representations; environmental data-files that list information relating to the design constraints and design context; and, existing applications that can be used to perform modelling, visualization and evaluation functions.

- The generic core consists of six generic modules: a population module that manages the population, four evolution modules that autonomously perform the four evolution steps, and a visualization module that allows users to visualise design models. The evolution modules extract individuals from the population and then process these individuals by invoking the corresponding evolution routine. The representation of an individual consists of a generic part and a specific part. The generic module can manipulate and modify the generic part, while the specialised routines can manipulate and modify the specialised part.

- In order to implement a system based on the proposed architecture, appropriate programming languages and associated technologies must be identified. Three different areas of the architecture must be considered: the representation of individuals in the population, the development of the generic core and the development of the specialised components.

Chapter 8

Demonstration

Contents

8.1 Introduction

This chapter demonstrates the process of encoding a design schema. The proposed design method was discussed in chapter 6, and started with the schema development phase. During this phase the design team conceives of a design schema that captures the character of a family of designs, then encodes this schema as a set of evolution routines. This chapter introduces a design schema for a particular family of designs and describes a set of evolution rules and representations for this schema. The initialization, developmental and visualization routines have been implemented and designs have been generated and visualised.

The chapter consists of three main sections:

- In section 8.2, the design schema that is to be encoded is described, and then the process of encoding this schema is discussed.

- In section 8.3, a generative process is described that may be used by the developmental step to generate designs.
- In section 8.4, the evolution routines implemented for the demonstration are described, and generated designs are then discussed.

8.2 Overview

8.2.1 Schema conception stage

A design schema

A basic design schema has been developed to demonstrate how it might be encoded. This schema is for a family of multi-story buildings constructed using standard concrete frame construction. The site is assumed to be flat and open, with no structures adjoining the site. The site is also assumed to be substantially larger than the building.

Character of design schema

The character of the design schema may best be understood by considering a set of examples. Figure 8.1 shows a range of designs created using the generative process that will be described in the next section. The main feature of these designs is their variability in terms of the overall building form, the organization of spaces, and the treatments of facades. Some additional complications such as sloping walls have been included, but not curved walls.

The designs will have the following characteristics:

- The design will have about three or four levels. On each level, there may be about five to ten spaces.
- The geometry of the design consists entirely of flat planar faces. There are no curved walls or roofs. Windows may be of a number of different types. The windows in the wall of a particular space must all be of the same type.
- The design is confined to a non-orthogonal oblong volume defined by six faces: a floor face, four vertical faces and a roof face. The floor face is assumed to always be level with the ground. The vertical faces may be rotated and inclined to the ground plane. The roof plane may slope in any direction.
- The design defines a set of floors and walls that divide the volume into a set of smaller spaces. These floors and walls are arranged in an orthogonal manner.
- Within this volume, the design has an open and airy massing. The design does not completely fill the volume, and includes many exterior spaces 'carved out' of this volume. These exterior spaces may

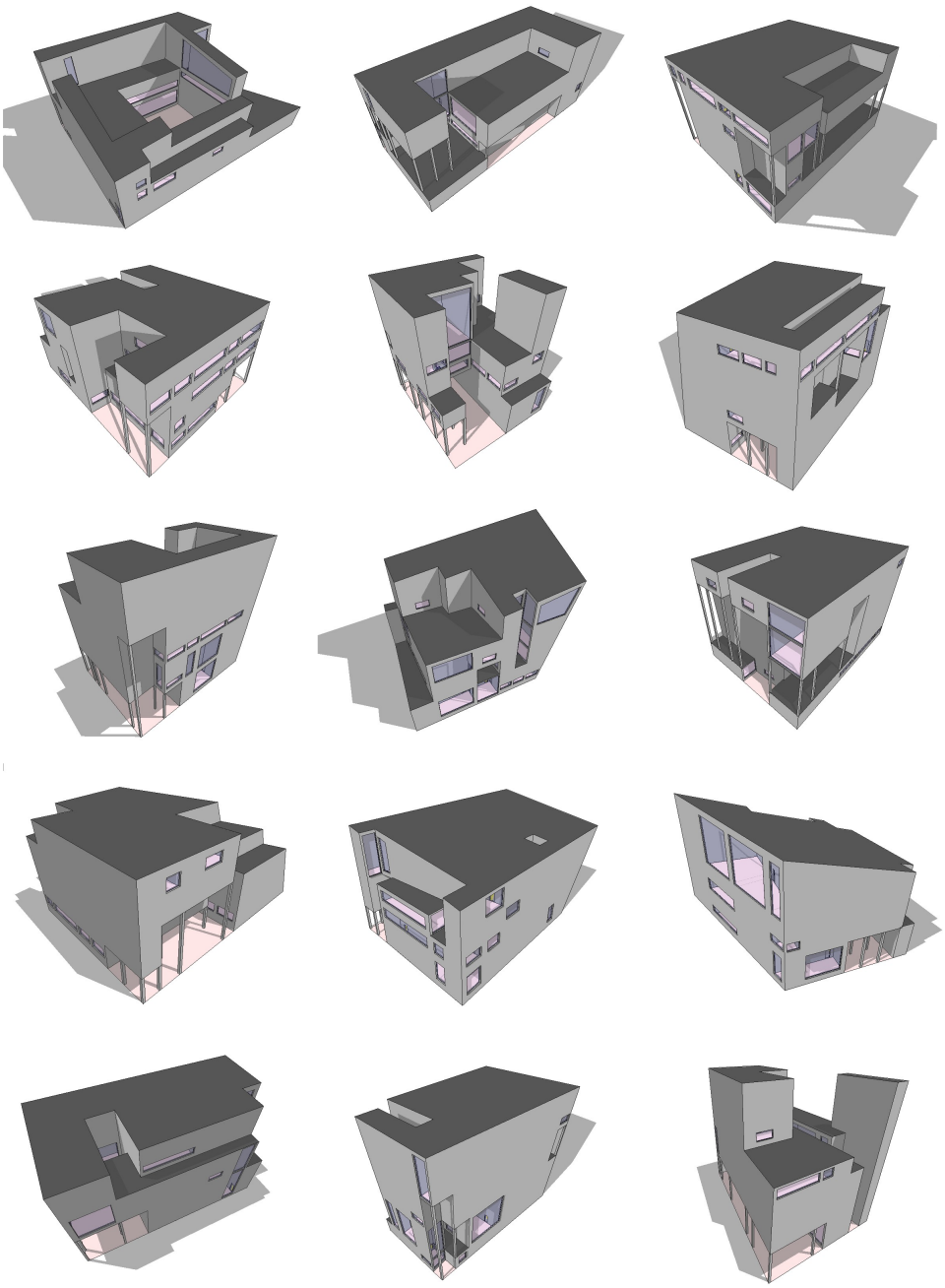

Figure 8.1: A set of generated designs.

be anywhere in the building, provided that they are in contact with the open air. If spaces are carved out at the lower levels, this may result in columns being required to support the upper levels.

- The spaces of the buildings generally have a simple geometry. Spaces are mostly quadrilateral in plan, and some may also be L-shaped. Spaces may also be double height (or more), and may be L-shaped in section. This allows spaces to interlock in both plan and section. Spaces may not have mezzanine levels since this would complicate vertical circulation.

8.2.2 Schema encoding stage

Specialised components

The design schema must be encoded as a set of evolution routines. In addition, environmental data-files and existing applications may also be used.

Routines

For the design team, the most challenging part of encoding a design schema will probably be the rules and representations related to the developmental routine. This routine must define a generative process capable of transforming a genotype into a phenotype, which in this case is a three-dimensional model of a design. Creating this generative process also requires the genotype and phenotype representations to be defined. Once these representations are defined, it also becomes possible to create the initialization routine and the visualization routine. Both these routines are helpful when creating the developmental routine. The former allows the developmental routine to be tested, while the latter allows the output from the generative routine to be visually inspected.

These three routines — development, initialization and visualization — are highly interrelated and must be developed in parallel. They are demonstrated and implemented in this chapter. Figure 8.2 on the next page identifies these routines. The focus of the demonstration is on the generative process of the developmental routine.

The reproduction, evaluation and survival routines are also discussed. However, they are not implemented as standard rules and representations can be used.

Data-files

The environmental data-files can be used to specify the design constraints and the design context. This information may be used by the developmental, visualization and evaluation routines.

In the case of the developmental routine, the generative process may be sensitive to both the design context (which allows for epigenetic

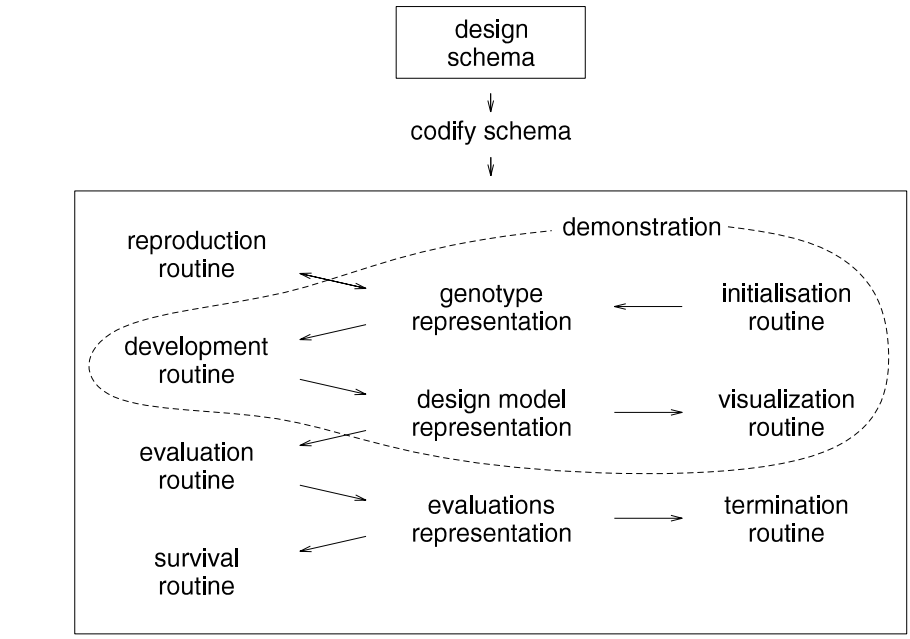

Figure 8.2: The parts of the encoded schema that have been implemented and demonstrated.

growth processes) and the design constraints. For example, if the constraints specify the number of spaces that are required, the generative process may ensure that the designs all have the correct number of spaces.

In this demonstration, the environment data-file is used to specify the site boundary and a number of constraints. The constraints include the number of floors for the building, the number of spaces that are required, minimum dimensions for spaces, and various other parameters that affect the types of spaces that can be created.

The site boundary is used by the generative process to ensure that the design never extends beyond this boundary. The constraints are used to generate designs that fulfil these constraints. For example, all designs will have the correct number of spaces.

Applications

Existing applications may be used in the developmental, visualization, and evaluation routines. In this demonstration, the developmental routine does not use applications. It is implemented as a stand-alone routine that does not require other applications or libraries.

The visualization routine makes use of a application called Ecotect[1] ©, which is an environmental analysis and simulation application. For the visualization routine, the analysis and simulation functionality is not used. Instead, the application is used as a way of quickly visualising three-dimensional models of designs. Ecotect provides a number of rendering and sectioning tools that make the process of interrogating models

[1] http://www.squ1.com

easy. In order for the design to be visualised in Ecotect, it is translated into a compatible format.

The evaluation routine is not implemented in this demonstration. Ecotect is suggested as an application that may be used in this routine. It is capable of performing fast solar, lighting and thermal analysis using various simplified methods. It is also capable of performing highly accurate simulations by integrating with and executing other applications such as Radiance, EnergyPlus and ESP-r.

8.3 Developmental routine

8.3.1 Overview

Controlled variability

In chapter 1, various problems associated with unrestricted variability were identified. With the proposed design method, the design team must create a developmental routine that is capable of producing controlled variability. To recap, controlled variability involves finding a balance between over-restricted variability that results in the generation of predictable designs, and under-restricted variability that results in a system with poor performance. In order to achieve controlled variability, a generative process needs to be defined that consists of a carefully crafted set of rules and representations.

Generative techniques

In chapter 3, a number of techniques were described for generating three-dimensional forms. Three main approaches were identified: the parametric approach, the combinatorial approach and the substitution approach. For this demonstration, a generative process has been created that includes techniques from both the parametric approach and the combinatorial approach. The substitution approach is not used in this case.

The overall generative process is based on the gradual modification of an orthogonal three-dimensional grid. The grid consists of a set of cuboid cells, the dimension of which are predefined as part of the schema and are set at 3.0 meters. The length, width and height of the grid, in numbers of cells, are specified in the environment data-file.

An eight step generative process is defined that transforms the grid into a building design. These steps are shown in figure 8.3 on the facing page. In two of the steps, the model is shown in section to show internal modifications: creating that staircase, and inserting doors.

In figure 8.4 on page 208, the starting and ending conditions of the transformation are shown in plan view: starting with the featureless grid on the left, and ending with an example of a floor of one of the designs on the right. The eight generative steps that transform the grid will be described using this example plan.

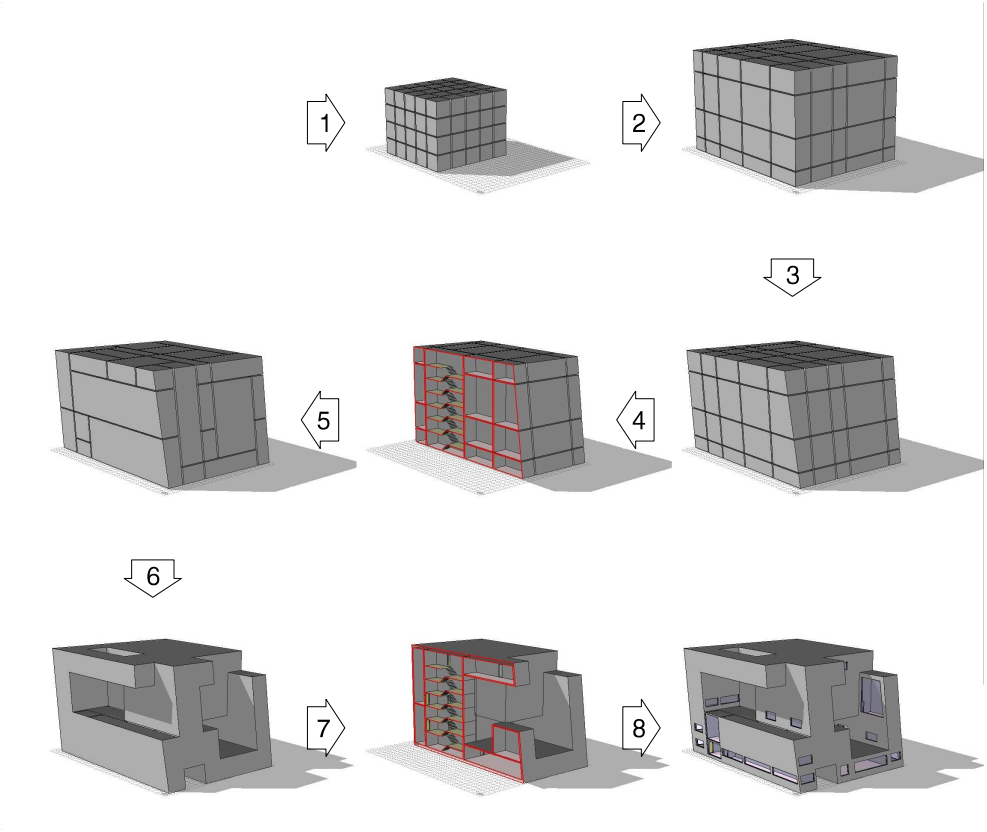

Figure 8.3: The eight generative steps used to generate the design models.

Grid terminology

In order to help explain the generative process, certain terms relating to the grid need to be defined. The grid consists of three main types of entities: *grid-faces*, *cells*, and *spaces*. Figure 8.5 on the following page shows a two-dimensional plan view of these three types of entities within the grid. Cells may be further broken down into *cell-faces*, and spaces into *space-faces*.

- Grid-faces are a set of intersecting quadrilateral surfaces that define the grid. Initially, they are perpendicular to the Euclidean x, y, and z axes.
- Cells are spatial units within the grid. Each cell has six cell-faces. These faces are parallel to and offset from the grid-faces.
- Spaces are larger spatial units that consist of one or more cells. If the space is oblong shaped, then the space will also be defined by six space-faces. However, spaces may also have more complex shapes which will result in more space-faces. The space-faces are coplanar with the cell-faces, and each space-face will consist of one or more cell-faces.

Each grid-face is thought of as having a front that faces the positive direction of the axis to which it is perpendicular, and a back that faces

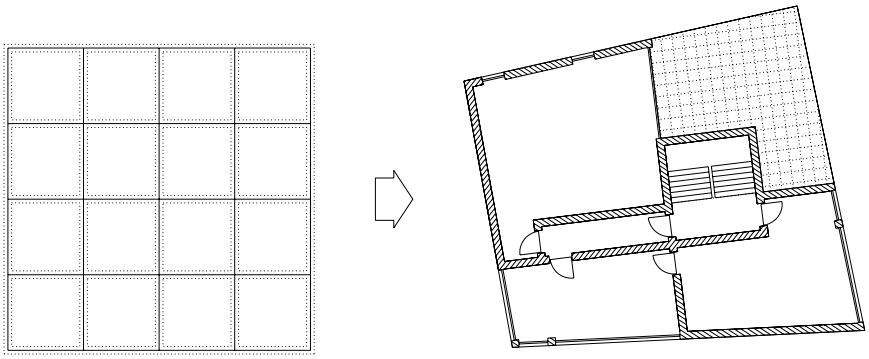

Figure 8.4: The transformation of the grid into a design.

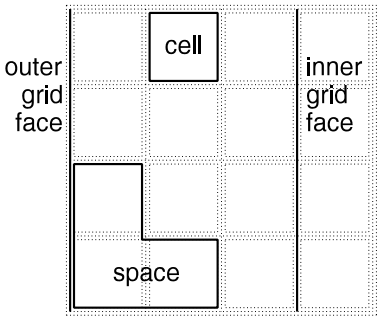

Figure 8.5: Terminology used to describe entities within the grid.

the negative direction. Grid-faces that are parallel to the ground are referred to as *horizontal grid-faces*, with the top grid-face being the *roof grid-face*, and the bottom one being the *floor grid-face*. Grid-faces that are not horizontal are referred to as *vertical grid-faces*, even when they are inclined and not strictly vertical. A distinction is also made between *inner grid-faces* that have adjacent grid-faces on either side and *outer grid-faces* that only have one adjacent grid-face.

Constraints

The generative process must modify the grid without violating five key constraints related to the geometry of the grid:

- First, adjacent grid-faces cannot touch or intersect.
- Second, the order of the grid-faces must remain the same. One grid-face cannot be translated beyond another grid-face.
- Third, the horizontal and vertical dimensions of the cells within the grid must remain greater than the minimum distances specified in the environment data-file.
- Fourth, the outer grid-faces must stay within the boundary of the site, as defined in the environment data-file.

- Fifth, the floor grid-face must remain level with the ground.

In addition to these grid constraints, there are three key space constraints:

- First, all spaces must be accessible from the staircase, either directly or indirectly via another space.
- Second, spaces must have only one floor face, and mezzanines are not allowed.
- Third, simple types of spaces such as oblong shaped spaces or L-shaped spaces are preferred.

8.3.2 Generative steps

Eight steps

The generative process consists of a sequence of eight generative steps: positioning of the grid, translation of the grid-faces, inclination of outer grid-faces, insertion of the staircase, creation of spaces, selection of outside spaces, insertion of doors, and insertion of windows.

Most steps require a set of parameters encoded within the genotype. These parameters are always encoded as real values in the range 0.0 to 1.0. The encoded value may be mapped by the generative step to a value within a different continuous range as required. Some steps may also require certain parameters or data encoded in the environment data-file.

The eight generative steps are shown in figure 8.6 on the next page and each step is described in more detail below.

Step 1: Positioning of the grid

The first step involves positioning the grid within the site. The grid is first placed in the centre of the site. The first three parameters in the genotype then encode a rotation followed by a translation.

The first parameter defines a rotation of the whole grid around the vertical axis. In order to define a rotation, the parameter, which is in the range 0.0 to 1.0, is mapped to the range 0.0 to 360.0 and interpreted as an angle of rotation.

The other two parameters encode a translation of the whole grid from the centre of the site to some new location. The translation is encoded as an angle and a displacement. The angle is encoded in the same way as for the rotation parameter and specifies the direction in which the grid will be displaced. The maximum displacement in this direction that keeps the grid within the site boundary is then calculated. The displacement distance is then calculated by multiplying the maximum displacement by the displacement parameter encoded in the genotype. The grid is then moved to its new location.

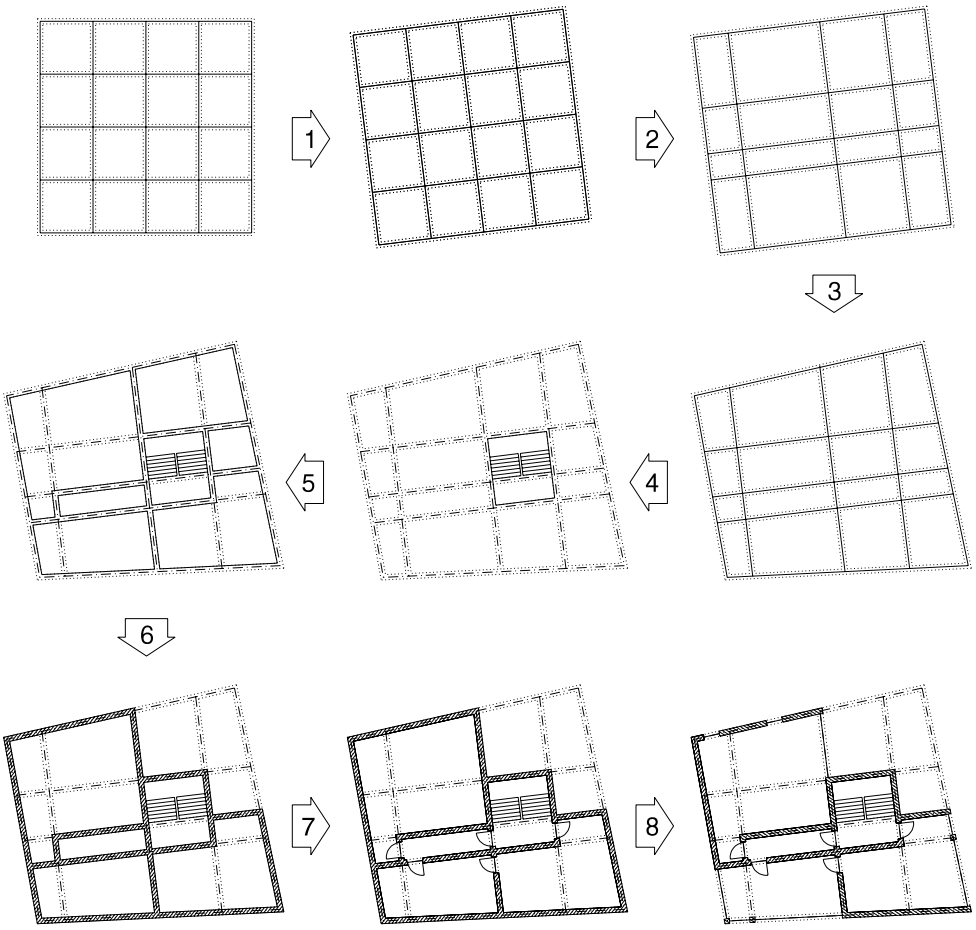

Figure 8.6: Eight generative steps.

Figure 8.7 on the facing page shows the process of positioning of the grid within the site. The numbers indicate the order of the operations and the order of the parameters within the genotype.

Step 2: Translation of grid-faces

The grid-faces that define the grid are translated in directions orthogonal to the grid-faces. This stretches or compresses the cells in the grid. The translation distance for each grid-face is calculated using a parameter encoded in the genotype. The genotype contains a separate translation parameter for each grid-plane.

Each grid-face is selected in turn. The selected grid-face then remains stationary and a set of grid-faces parallel to it — either the grid-faces in-front or those behind — are translated. The parameter in the genotype is mapped to a real value in the range -3.0 and +3.0, which is interpreted as scale factor of the existing distance between grid-faces.

For inner grid-faces, a positive value indicates that all the grid-faces in front are translated, while a negative value indicates that all the grid-faces behind are translated. For example, a value of 0.5 would indicate

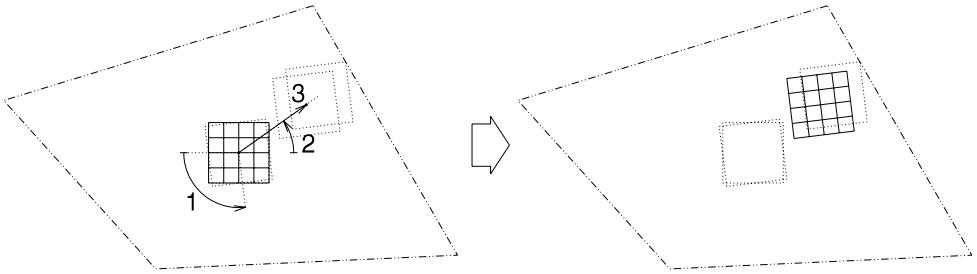

Figure 8.7: Positioning of the grid within the site boundary.

that all the grid-faces in-front should be translated backward. The distance of translation would be equal to half the distance from the grid-face to the next grid-face in front.

For outer grid-faces, the process is similar. Since outer grid-faces only have one adjacent grid-face, an additional grid-face is temporarily added to turn the outer grid-face into an inner grid-face. This additional grid-face is always offset 3.0 meters from the grid.

Step 3: Inclination of outer grid-faces

All outer grid-faces except the floor grid-face may then be inclined. A grid-face is inclined by rotating it around two axes that are not parallel to one another. In order to specify the inclination, the genotype needs to encode two parameters for each grid face. Since five grid faces may be inclined, ten parameters are required.

The grid-faces are first rotated around a one axis and then around a second axis that is not parallel to the first. The two axes of rotation are created by defining lines that bisect opposite edges of the grid-face. As with the rotation step, the maximum angle of rotation that does not violate the grid constraints is first calculated, and the parameter in the genotype then specifies a fraction of this angle.

Step 4: Insertion of the staircase

The staircase is now created and inserted into the building. The key decision is the positioning of the staircase within the grid. The genotype does not contain any parameters for defining this position. Instead, the staircase position is defined based on the geometry of the grid.

A simple type of dog-leg staircase is used, which consists of opposing flights of steps that connect the main landings to the half landings. The staircase must have a minimum of two flights of steps in opposing directions, with each flight having the same number of steps. In such a case, two landings will be required which will be level with the floors, and one half landing will be required. More than two flights may be used if the floor-to-floor height is large. In general, the maximum number of flights is used that does not violate the minimum head-room for people walking up the staircase. (This ensures that the plan dimensions of the stair are

minimised.) In order to ensure that the landing is always on the same side, even numbers of flights are always used.

The dimensions of the staircase, such as the minimum width, the riser height and the tread depth, are predefined as part of the schema. In order for the landings to be level with the floors, all floor-to-floor heights must be adjusted so that they are multiples of double the riser height.

A vertical space — or stairwell — must be created into which the staircase can be inserted. The stairwell will run the full height of the building and will take up two adjacent columns of cells. One column of cells is used for the landings, and the other column is used for the flights of steps and the half landings.

In order to define the position of the stairwell, two adjacent columns of cells must be identified. These two columns are chosen based on three stair constraints. First, the stairwell is constrained to not have any outer grid-faces. This rules out any of the outer columns of cells. The second constraint requires that the stair, whose minimum dimensions are defined in the environment data-file, will actually fit. The third constraint requires that the area of the stairwell (in plan) be as small as possible. All the possible positions are analyzed and for each position, if the staircase will fit, the area is calculated. The position that results in the stairwell with the smallest area is then chosen. In figure 8.8, the fourth position is to narrow for the staircase. Of the three other positions, the second has the smallest area and this one would be chosen.

If the geometry of the grid is such that no suitable position for the staircase can be found, then the generative process may be aborted. In this case, no building model will be generated and the genotype will be assigned a minimum fitness.

Step 5: Creation of spaces

Next, the spaces within the building are created. Spaces consist of one or more cells that are merged together. For each cell in the grid, the genotype encodes a value that is referred to as the *pressure* within that cell. This pressure value is used to decide which cells should be merged to create a space.

The number of spaces that are required is defined in the environment data-file. This number should be substantially smaller than the total number of cells in the grid. Initially, one space is created for each cell in the grid except the stairwell cells. The stairwell is treated as a special space that cannot be merged with any other spaces. Spaces are defined by their interior surfaces, which are offset from the grid-face by half the wall, floor or roof thickness depending on the surface in question. The walls, floors and roof are all centred on the grid-faces. The value for the wall thickness is predefined as part of the schema.

Pairs of spaces are then repeatedly merged together until the number of spaces has been reduced to the required number. The pressure in a space that contains more than one cell is equal to the average of the

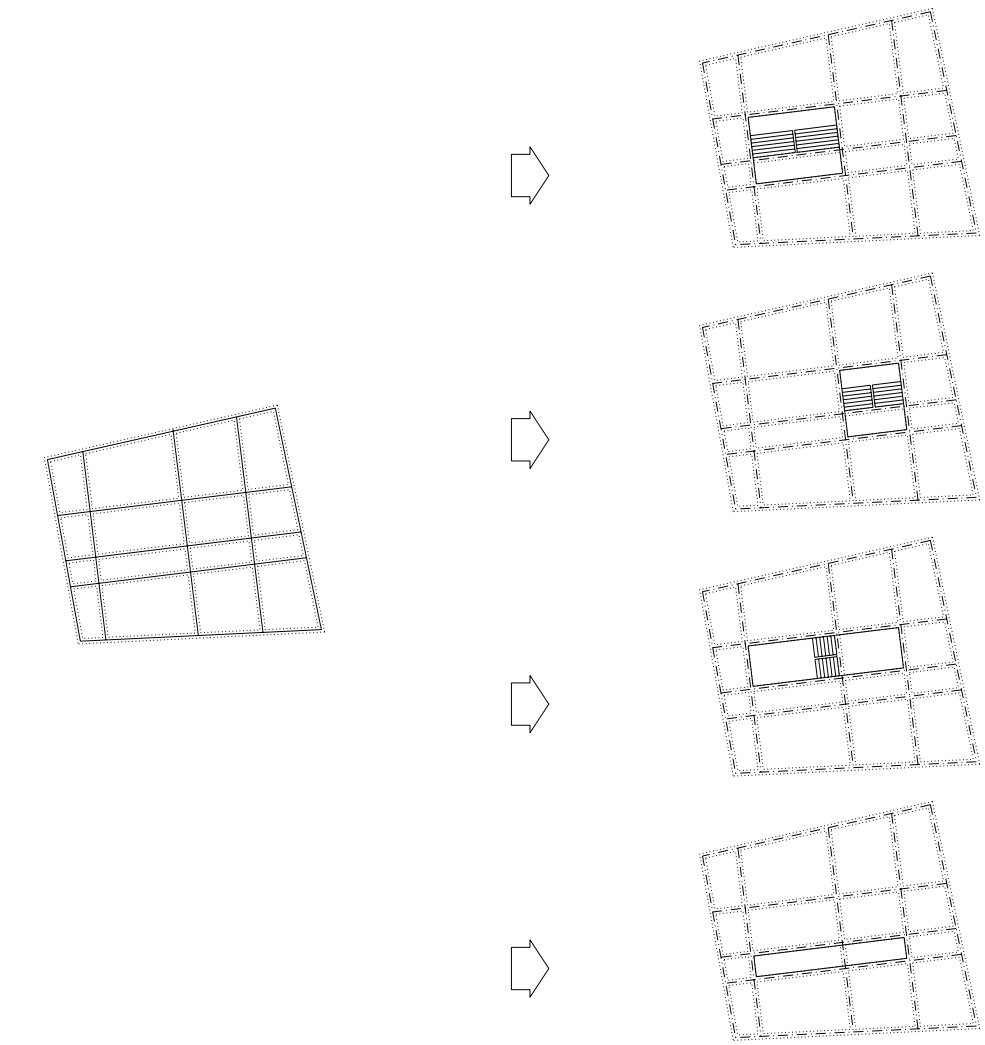

Figure 8.8: Possible positions for the stairwell.

pressures of all the cells in the space. In order to choose a pair of spaces to merge, the difference in pressure between all possible pairs of adjacent spaces is calculated. The pair with the highest pressure differential is then chosen.

The merging of spaces must take into account the space constraints discussed above. In order to fulfil the accessibility constraint and the mezzanine constraint, the merging of any spaces that violate these constraints are disallowed. Each time a merge is disallowed, then the pair of spaces with the next highest pressure differential is considered.

The third constraint — the preference for simple spaces — is more complex. In order to fulfil this constraint, the merging process is split into three different phases. In the first phase, spaces will only be merged if the result is an oblong shaped space (not necessarily orthogonal). As with the other constraints, if the merge is not going to result in an oblong space, then the pair of spaces with the next highest pressure differential is considered. This phase may run out of spaces that can be merged, in

which case the second phase will be started. In this phase, spaces can be merged if the result is an L-shaped space, either in section or in plan. If this phase also does not manage to sufficiently reduce the number of spaces, then the third phase is started in which any kind of merge is allowed.

Step 6: Selection of outside spaces

The next step involves selecting the spaces that are actually exterior spaces, and which need to be removed from the building volume. The total number of exterior spaces is specified in environment data-file. This step uses the same pressure values as the previous space creation step. Once again, the staircase space is treated as a special space that cannot be deleted.

For each space that has an exterior wall, the pressure difference between the space and the outside is considered. As in the previous step, the pressure in each space is calculated by taking the average of the pressure of all the cells in the space. The pressure for the outside of the building is calculated as the average of the pressure of all the spaces in the building. The space with the highest pressure differential is then deleted.

In this case, only one of the space constraints — the accessibility constraint — needs to be considered. If deleting a space would result in another space becoming inaccessible, then the deletion of this space is disallowed. In such a case, the space with the next highest pressure differential will be considered. Each time a space is deleted, the spaces behind the deleted space may become exposed and may then also be considered for the next deletion.

Once the required number of spaces have been deleted, the exterior surface of the building must be generated. As with the interior surfaces, this exterior surface is offset from grid-face by half the wall, floor or roof thickness depending on the surface in question.

Finally, columns are also inserted during this step. Columns are inserted under any part of the building that is not directly supported. The insertion process places columns of fixed dimensions on each grid position below the unsupported part of the design.

Step 7: Insertion of doors

Doors are then inserted to allow access between spaces. For this demonstration doors are simple inserted between all possible spaces. There are no parameters in the genotype that directly affect the insertion of doors. The position of a door in a wall is defined so that the door is not in the middle of the wall, but is close to corners in both spaces.

It would be possible to specify a required set of adjacencies between spaces within the environment data-file. This would allow doors to be inserted only where required.

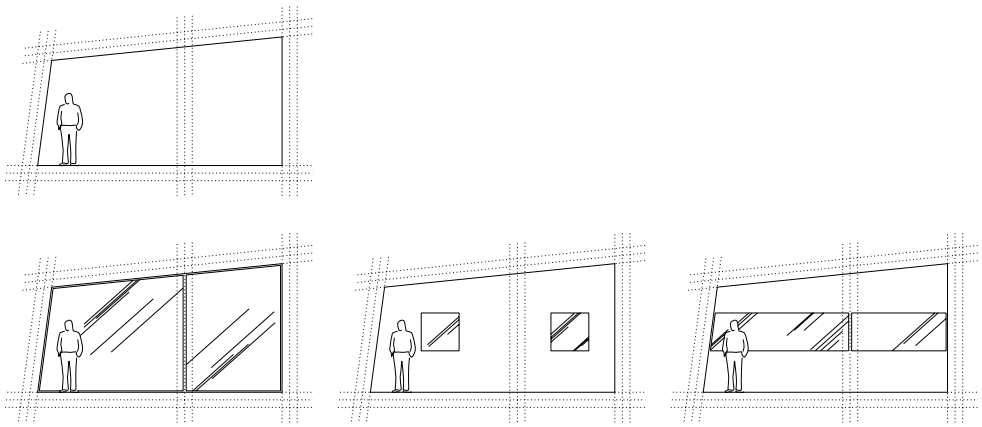

Figure 8.9: Interior elevation of four possible window types.

Step 8: Insertion of windows

Finally, the last step involves inserting windows in the exterior walls. The genotypes encodes values that specify either no window or a window type. Three window types have been specified within this schema: fully glazed window, square window, and horizontal strip window. The possibility of no window is actually treated as another window type, and there are four possible types. These window types are shown in figure 8.9. The geometry and position of the window is calculated relative to the geometry of the wall.

One way to encode the window type parameters in the genotype is to encode a separate parameter for each cell-face. However, most cell-faces will not be exterior walls. Encoding window parameters for all cell-faces would be wasteful and make evolution more difficult. A method needs to be found for reducing the number of window parameters.

One way of achieving this is to associate the parameters with the spaces rather than the cell-faces. This is the approach taken here. Spaces in the building are first listed from largest to smallest. The genotype then encodes one window parameter per space in the list.

Spaces are considered to have four types of vertical faces that are orientated in different directions. (These directions may be thought of as north, south, east and west.) If the space is oblong in shape, then there will be one face for each direction. For L-shaped spaces and other more complex types of spaces, each direction may consist of more than one face. Space-faces that are orientated in the same direction will all be assigned the same window type. The genotype then needs to encode four window types for each space. The window insertion step will then decode these parameters and for each exterior space-face, insert the window type specified by the parameter. If the space-face is not an exterior face, then it is ignored, along with any window parameters assigned to it.

For each space, the four directions may be assigned one of four window types. This results in (4^4) 256 possibilities. The genotype encodes these possibilities as a real value between 0.0 and 1.0, which is mapped to the

range -0.5 to +255.5 and is then rounded off to the closes integer value. The integer value is then converted to base 4, which results in a number with digits in the range 0 to 3. There will be a maximum of four digits, which represent the window types for each orientation.

8.4 Implementation

8.4.1 Overview

In order to verify the character and variability of the designs that would be produced by the generative steps described above, the initialization, developmental and visualization routines have been implemented.

Representation of an individual

An individual is represented using XML, as discussed in the previous chapter (see section 7.4.2 on page 196). To recap, the XML tree has one top level node called *individual*. This node has a number of attributes, including an *ID*, and three sub-nodes called *genotype*, *phenotype* and *evaluations*.

The genotype node contains a long string of comma separated real numbers that are the parameters values for the generative process. If the individual has been developed, then the phenotype contains the representation of the model, encoded using an XML representation. Finally, the evaluations node will contain one or more evaluation score nodes, each of which encodes the result of analyzing one objective. Since the evaluation routines were not implemented in this demonstration, this part was not used.

Genotype representation

Each individual must have a genotype that will be used to create the phenotype. The genotype representation consists of a fixed-length list of real numbers (a real-valued vector), with all values in the rage of 0.0 to 1.0. This uniformity simplifies the creation of crossover and mutation parameters.

- Three values encode the position of the grid within the site.
- A set of values encode translation parameters. A translation parameter is required for each grid-face except the floor grid-face.
- Ten values encode inclination parameters for the roof grid-face and the four outer vertical grid-faces.
- A set of values encode pressure parameters that are used for creating spaces and selecting outside spaces. A pressure parameter is required for each cell in the grid.

- A set of values encode the window type parameters. A window type parameter is required for each space in the grid.

The length of the genotype depends on the side of the grid and the number of spaces. The length of the genotype can be calculated using the following formula:

$$gl = x + y + z(xy + 1) + s + 15$$

where gl is the genotype length, and x, y and z are the number of cells in the direction of the x, y and z axes respectively, and s is the number of internal spaces in the building. For example, for $x = y = z = 4$, and $s = 15$, the genotype length will be 106; for $x = y = z = 5$, and $s = 30$, the genotype length will be 185.

Phenotype representation

The phenotype representation of an individual is the representation used to define and store design models. This representation uses the latest Industry Foundation Classes (IFC) specification[2] published by the International Alliance on Interoperability (IAI).

The advantage of using IFC is that it allows for the representation of the different types of information required by simulation and analysis applications, including physical components such as doors and walls, and abstract concepts such as spaces. The IFC classes describe an abstract structure that may be encoded in a number of way. In this case, the IFC data is encoded using ifcXML[3].

Initialization routine

The initialization routine is used to generate a population of individuals with randomly generated genotypes, but with no phenotypes or evaluation scores. This routine calculates the length of the required genotype using the formula specified above, and creates a random value for each parameter. The input to this routine is the required number of individuals. Figure 8.10 on the next page shows the inputs and outputs for this routine.

Developmental routine

The developmental routine is used to create phenotypes for each individual. The generative process used by this routine has already been described above in section 8.3 on page 206. The inputs and outputs to this routine, shown in figure 8.11 on the next page, are both XML

[2] http://www.iai-international.org /iai_international/Technical_Documents/R2x2_final/index.htm

[3] http://www.iai-international.org /iai_international/Technical_Documents/IfcXML.htm

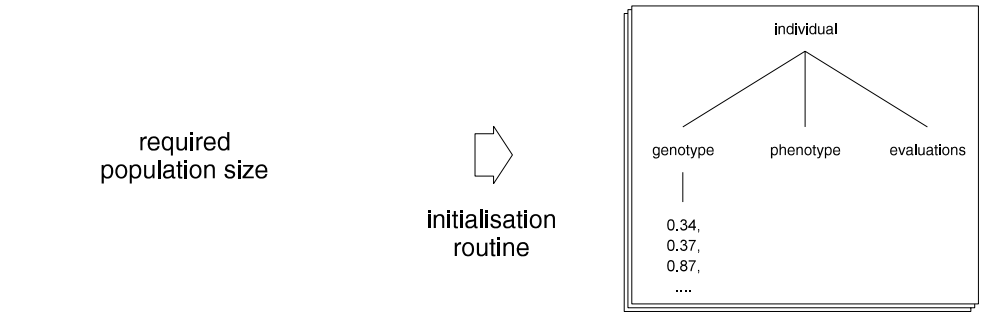

Figure 8.10: Inputs and outputs for the initialization routine.

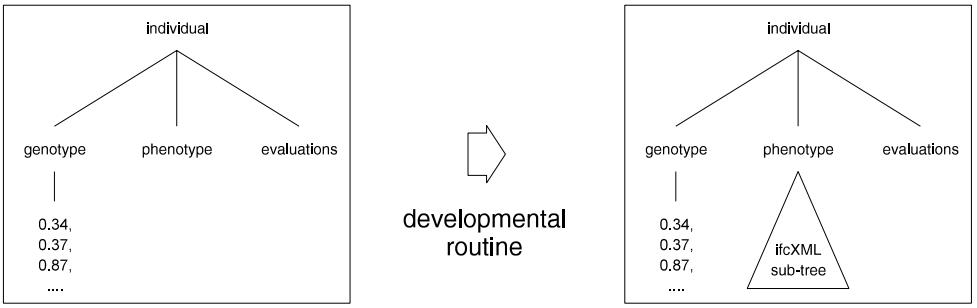

Figure 8.11: Inputs and outputs for the developmental routine.

trees. The input tree is an individual with an empty phenotype node, while the output individual is the same individual but with the ifcXML representation of the design model added to the phenotype node.

The developmental routine also requires various parameters and data stored in the environment data-file. These include the grid size, the number of internal spaces, the number of external spaces, and the site boundary.

Visualization routine

A visualization routine has been created that uses Ecotect to visualise the design models that are generated. This routine extracts the phenotype from each individual, translates the ifcXML representation to the model representation used by Ecotect, and then allows the user to visualise the design using the Ecotect interface. Figure 8.12 on the facing page shows the inputs and outputs for this routine.

8.4.2 Other routines not implemented

Other routines

Issues relating to the implementation of the reproduction, evaluation and survival routines routines is briefly discussed below.

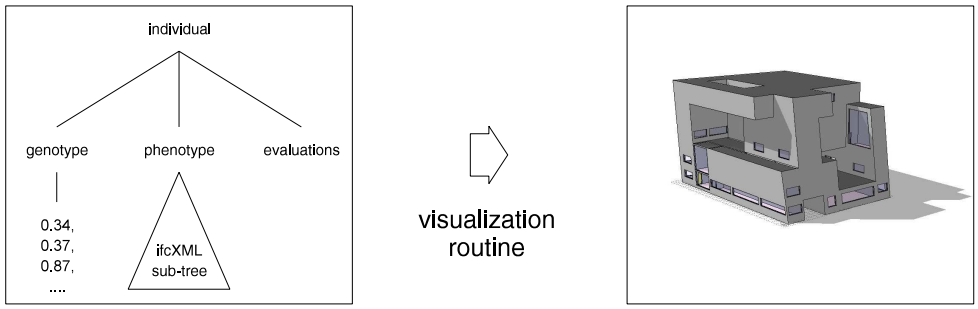

Figure 8.12: Inputs and outputs for the visualization routine.

Reproduction routine

The reproduction routine needs to extract a pool of parent individuals from the population and creates new offspring. Due to the simple structure of the genotype representation, the reproduction routine could use standard crossover and mutation operators for real-value vectors, as described in chapter 4 (see section 4.4.3 on page 105). For example, uniform mutation and simple crossover could be used. The size of the reproduction pool could be just two individuals.

Evaluation routine

The evaluation routine calculates an evaluation score for the phenotype of an individual with respect to a particular design objective. The designs are likely to be evaluated for multiple objectives. Multiple evaluation routines would need to be created, one for each objective.

Ecotect could be used to analyse daylight, artificial lighting and thermal performance. Design models would need to be translated into a format that can be read by Ecotect. In addition, the data required for Ecotect to perform the required analyses and simulations would need to be included in the phenotype and conserved through this translation process.

Ecotect can import various formats such as DXF and 3DS. However, these formats only describe geometry and do not describe critical information about spaces within the building. A more appropriate format is a format specifically developed for Ecotect, called the *model* format. This format is encoded using a human-readable ASCII and the ifcXML representation can easily be translated into this format. Although the evaluation routine has not been implemented, this translation process has been successfully implemented and tested.

Survival routine

The survival routine needs to extract a pool of individuals from the population and select individuals to be deleted. As discussed in the previous chapter, the greater the pool size, the greater the selection pressure. One

approach would be to specify that the pool should contain all fully evaluated individuals currently in the population. The selection routine then needs to rank individuals in the pool using a scalarization technique, as discussed in chapter 4 (see section 4.4.3 on page 107). Since objectives such as light-levels and energy consumption are non-commensurate, a Pareto-based scalarization technique may be used.

8.4.3 Results

Generation and visualization of designs

The three routines were used to generate and visualise a population of individuals. The individuals all have random genotypes and have not yet been evolved. The designs are therefore not expected to be of high quality with respect to any set of objectives. The main aim of generating and visualising these designs is to verify the character of design and the variability that can be achieved.

Examples

Figures on the next page, on page 222 and on page 222 show three examples of design models that were generated. In this case the grid size was set to 5 cells by 6 cells in plan, and 4 cells high. The number of internal spaces was set to 12, and the number of external spaces was set to 8. This produces a genotype length of 162.

Creating a grid four cells high results in all designs having 4 floors. Spaces will have an average of six cells per space, and half the grid volume is likely to consist of exterior spaces.

In the first design example, a building with a courtyard has been generated. Another interesting feature of this example is that there are no spaces on the top floor. Only double height spaces from the floor below rise up to the top floor. The staircase would however provide access to the roof.

In the second example, a building with a number of large spaces has been generated. On the top two floors, a long corridor has been used to provide access to various spaces from the staircase.

In the third example, a complex exterior space has been created that penetrates the roof and reappears on the facade. Some of the spaces have a fairly complex geometry. A number of tall and narrow spaces have also been created. On the top floor, there is only one space.

Controlled variability

One of the important requirements of the designs generated by any developmental routine is controlled variability. The designs that are generated must all share the same character and must also vary significantly in overall organization and configuration, thereby allowing surprising and challenging designs to be generated.

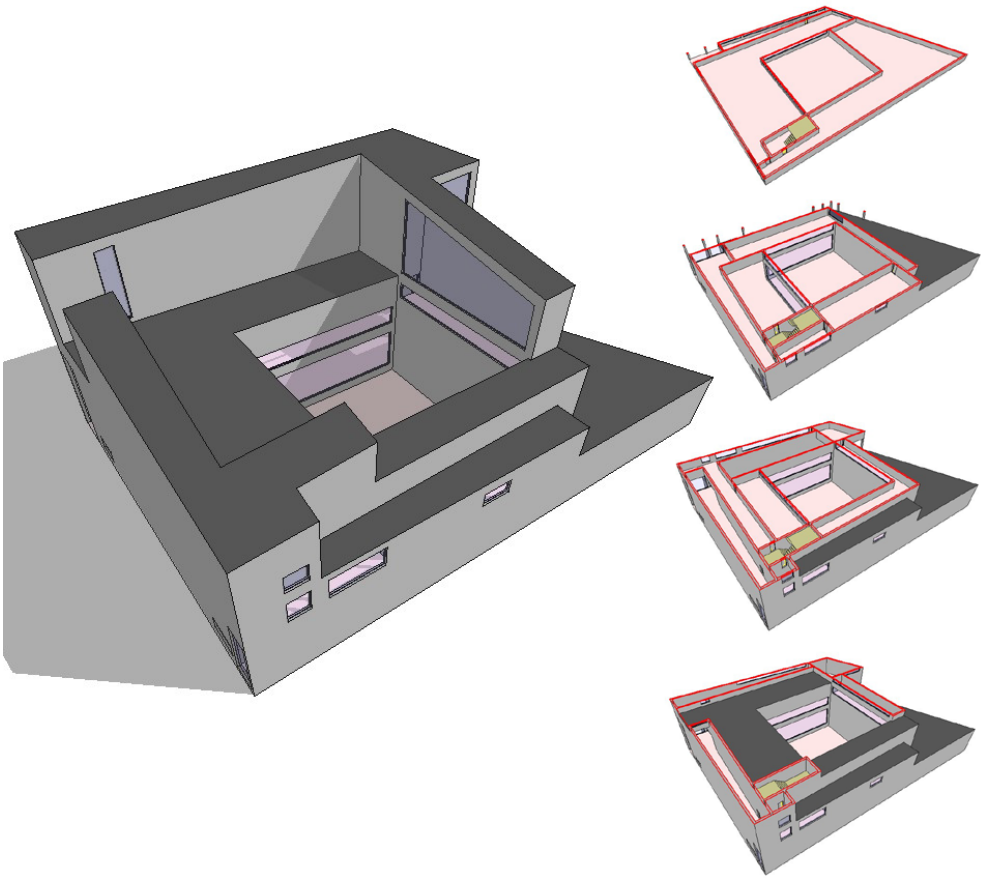

Figure 8.13: First design example.

However, if designs vary in highly unrestricted ways, then the performance of the evolutionary system will deteriorate. In chapter 1 (see section 1.1.2 on page 12), key problems associated with unrestricted variability have been identified — the problem of interpreting chaotic forms, the problem of ranking designs that differ fundamentally from one another, and the problem of the semantic level of the representations of designs.

The generative process must fulfil four key conditions if controlled variability is to be achieved: first, it should be capable of generating designs with the required level of complexity; second, it should generate designs that all share a certain essential and identifiable character; third, it should generate designs that differ significantly in terms of their overall organization and configuration; fourth, it should not generate chaotic forms that are not directly and straightforwardly interpretable as designs.

The developmental routine implemented in this demonstration fulfils all four conditions by carefully controlling the variability of the design models.

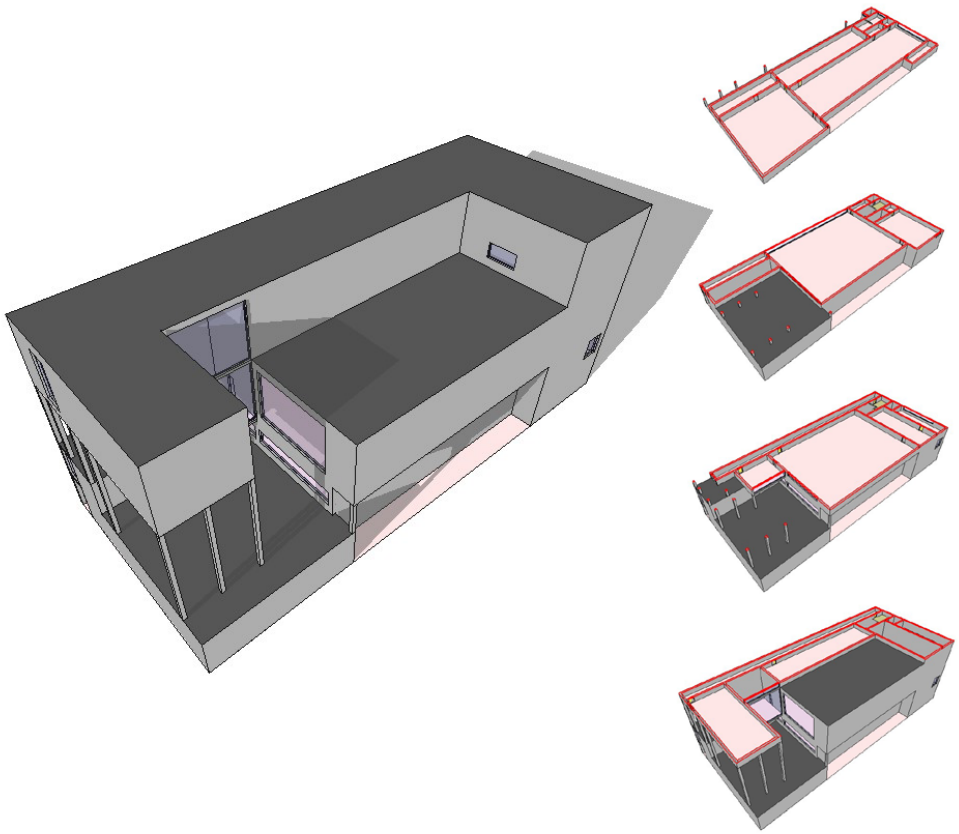

Figure 8.14: Second design example.

- First, the design includes the typical kinds of constructs that characterise buildings, such as spaces, walls, floors, windows and doors.
- Second, the designs all share the same design character that is described by the schema introduced above.
- Third, the designs vary significantly in terms of the overall form, the organization of spaces, and the treatment of the facades.
- Fourth, all designs are represented using high-level semantic constructs that are constrained so as to ensure that only valid building designs can be created.

8.5 Summary

This chapter has demonstrated the process of encoding a design schema. The aim of this demonstration is to support the proposed generative

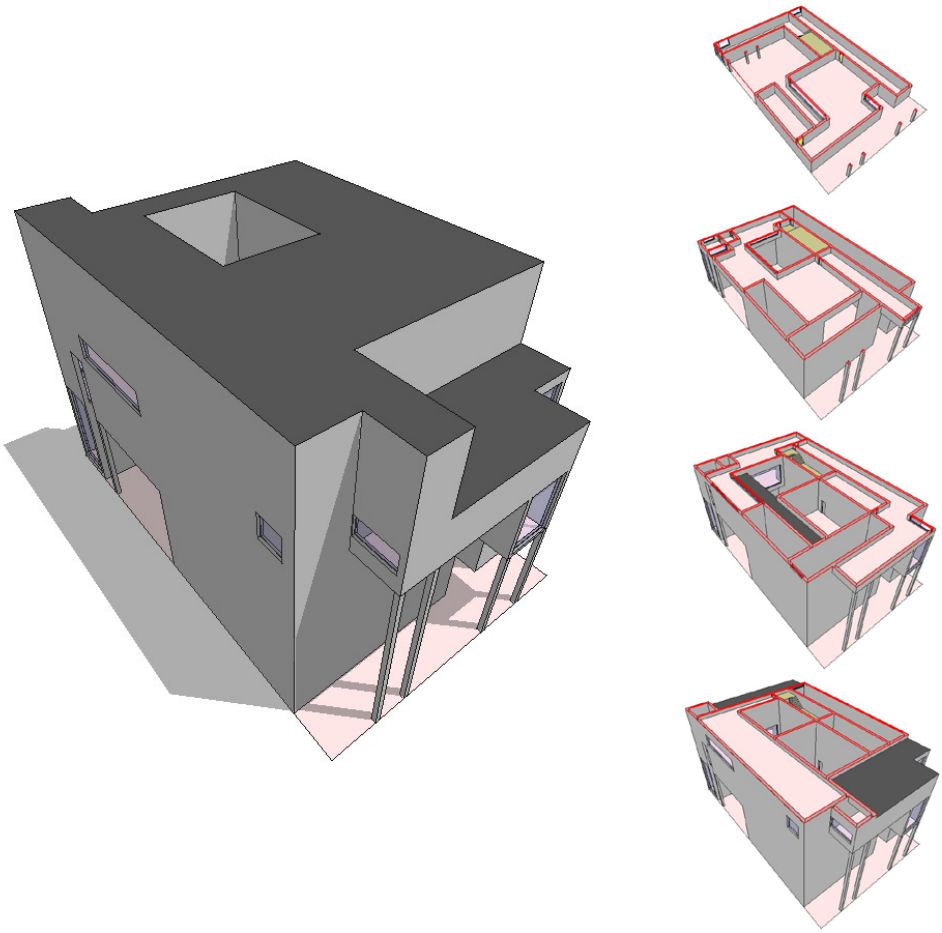

Figure 8.15: Third design example.

evolutionary design framework described in the previous two chapters. The main points are as follows:

- An example design schema has been created for a family of multi-story buildings. The overall building form, the organization of spaces, and the treatments of facades may all vary significantly. Some additional complications such as sloping walls have been included, but not curved walls.

- A generative process has been described for generating design models in the example schema. This process consists of a series of transformations that gradually transform a three-dimensional orthogonal grid structure into a design for a building.

- A developmental routine, an initialization routine and a visualization routine have been implemented for the example schema. These routines have been used to generate and visualise a variety of design models. The designs that are generated have four key

characteristics: first, they are of a complexity typical of buildings; second, they all share the same design character; third, they vary significantly from one another; fourth, they are all directly and straightforwardly interpretable as designs.

The demonstration has shown that it is possible to create a generative process that can produce three-dimensional models of design that are neither over-restricted nor under-restricted. Controlled variability has been achieved.

Part IV

Conclusions

Part three consists of a concluding chapter that identifies the contributions made by this research and briefly discusses possible directions for future work in both the short and the long term.

Chapter 9

Conclusions and future work

Contents

9.1 Contributions

9.1.1 Summary of objectives

Evolutionary systems

This research is concerned with the development of a comprehensive framework that specifies how a team of designers can use computational systems to generate and evolve surprising and challenging designs.

The framework consists of a design method and a computational architecture. The design method describes a set of main tasks to be performed by the design team, one of which requires a generative evolutionary design system. The computational architecture provides an implementation plan for such a system, and defines how the significant hardware and software components can be structured and organised. This

architecture may be used by researchers to build the generative evolutionary design system.

In developing this framework, five main contributions have been made. First, a key problem related to design variability has been identified. Second, a design method for the generative evolutionary approach has been developed. Third, a general computational architecture for a generative evolutionary design system has been proposed. Fourth, based on the general architecture, a detailed architecture has been developed. Finally, a generative process that is capable of producing controlled variability has been demonstrated. Each of these contributions is discussed in more detail below.

9.1.2 Variability problem

Generative evolutionary design

A distinction has been made between parametric evolutionary design and generative evolutionary design. With parametric evolutionary design, a parametric model of a design is predefined and an evolutionary system is used to evolve a set of parameters. With generative evolutionary design, a growth process that generates alternative design models is defined and an evolutionary systems is used to evolve a set of parameters or other modifications associated with this generative growth process.

The key difference between the parametric and the generative approach is related to design variability. With the parametric approach, design variability is highly restricted, whereas with the generative approach, variability may be much more unrestricted.

The variability problem is related to the performance of the evolutionary system. With parametric systems, the restricted variability of the designs allows for an evolutionary system with good performance to be achieved. With generative systems, if variability is highly unrestricted, the process of evaluating the design models becomes complex. As a result, the performance of the evolutionary system will degrade significantly.

If an evolutionary system is to be created that has reasonable performance and is also capable of evolving surprising and challenging designs, design variability must be carefully controlled. In order to control the variability, the evolution steps — in particular, the developmental step — need to be defined to control the types of designs that can be generated and evolved.

Controlling design variability

Controlling design variability involves deciding which designs should be included in the restricted variability, and which should be excluded. A set of characteristics shared by all included designs may be identified, and these characteristics can be used to define the rules and representations for the evolution steps. The rules and representations must be crafted to

ensure that the included designs will be generated and evolved, and — more importantly — that the excluded designs (and chaotic forms) will not be generated and evolved.

Two main approaches regarding which designs should be included have been discussed. A highly generic approach can be taken whereby a significant portion of all possible designs are included (Bentley, 1996). Alternatively a more focused approach can be taken whereby only a relatively small family of designs that are related in some way are included (Frazer, 1974; Frazer and Connor, 1979).

The problem with the first approach is that designs for buildings tend to vary widely and highly generic shared characteristics do not exist. It becomes difficult to create evolution steps that prohibit the generation and evolution of excluded designs and other chaotic forms.

The design schema approach

The second approach should therefore be taken. A much smaller family of designs must be identified that share certain characteristics. For the design team, the most relevant family of designs is their own body of work. Such a family of designs will tend to have a strong unity and will reflect the preconceptions of the design team, including philosophical beliefs, cultural values and design ideas.

The design team has to be involved in creating the evolutionary rules and representations. The essential and identifiable character of the family of designs has to be captured as an abstract conceptualization called a *design schema*. This schema must be encoded as rules and representations, which are used by an evolutionary system to generate and evolve designs that embody the character of the design schema.

9.1.3 Design method

Three design methods

Two existing design methods for generative evolutionary design have been identified, and an alternative design method has been proposed. The existing methods are the *general generative evolutionary method* and the *evolutionary concept-seeding method*. The proposed method is the *schema-based evolutionary method*.

- The *general method* (see figure 1.2 on page 12) is a generic method that is often not explicitly described. The method consists of two stages: first, based on various generative concepts, a set of rules and representations are defined and encoded; second, these rules and representations are incorporated into an evolutionary system that can be used to evolve alternative designs. The first stage usually focuses on creating a generative process for the developmental step, and a set of evaluation routines (or fitness functions) for the evaluation step. Alternative designs can be generated by making

small modifications to the generative process. In the second stage, the evolutionary system generates alternative design by encoding the generative modifications in the genotype. The system evaluates each design using the evaluation routines and information about the design environment.

- The *concept-seeding method* (see figure 5.7 on page 129) (Frazer, 1974; Frazer and Connor, 1979) consists of three stages. The first stage is similar to the previous method, and involves defining a generative process for the developmental step, and a set of evaluation routines for the evaluation step. In this case, the generative process requires a starting condition — referred to as a *concept-seed* — to be defined. This seed encapsulates certain design ideas and the generative process generates designs by progressively manipulating and modifying this seed. The second stage of this method consists of creating this concept-seed based on a set of design ideas. In the third stage, the rules and representations from the first step and the concept-seed from the second step are used to generate and evolve designs. In this case, genotypes may encode modifications to either the generative process or the concept-seed.

- The *schema-based method* (see figure 1.3 on page 19) is similar to the concept-seeding method, in that a set of design ideas are encoded in a format that can be used by the evolutionary system. But this method also differs in various ways. Most importantly, the generative concepts and the design ideas are combined into one entity, which is the design schema. This schema is encoded by creating the rules and representations for all the evolution steps. Two additional stages have also been added to the beginning and the end of the method. Overall, the method consist of four stages: first, design ideas are developed and a design schema is defined; second, this schema is encoded as a set of evolutionary rules and representations; third, designs are generated and evolved; and fourth, one design is selected and further developed into a detailed design. One other important difference between this method and the concept-seeding method is the concept of a niche environment. This concept describes a type of design environment that includes both design constraints and design context. This concept also allows the design method to be split into a schema development phase and a design development phase. In the first phase, design ideas are developed and encoded for an environmental niche, while in the second phase, designs are evolved for a specific design environment.

Advantages of the proposed method

These three design methods may be analysed with regard to the two requirements identified in chapter 1: conservativeness and synergy. A conservative design method minimises the changes in working procedures

that are required by the design team. A synergetic design method maximises the potential of the design team working with the computational systems.

The general method is considered to be neither conservative nor synergetic. With regard to conservativeness, it does not explicitly address how the evolutionary design approach should be incorporated in the overall design process. With regard to the synergy between the design team and the computational system, the role of the design team is not considered. The relationship between the generative concepts and the types of design that are generated and evolved is not clearly defined.

The concept-seeding method addresses this issue more explicitly. For each different type of design, the design team creates a different concept-seed. The evolutionary system is used to generate and evolve designs that embody the design ideas encapsulated in the seed. However, the conservativeness and synergy of this method are not examined. With regard to conservativeness, the method does not consider two key processes: the process of developing the generative concepts and design ideas; and the process of further developing the designs that have been evolved. With regard to synergy, the method does not allow the design team to define their design ideas in a flexible manner. In particular, the design ideas are defined separately from the generative process. This is seen as problematic, because it is often easier to define design ideas in a procedural manner in the rules and representations of the generative process.

The schema-based method has been specifically developed to fulfil the requirements of conservativeness and synergy. With regard to conservativeness, the method is similar to an existing design process commonly used by many designers in practice. The whole design process is considered, starting with the preconceptions of the design team and leading to a detailed design proposal. It only deviates from this existing process when it is essential to the success of the evolutionary design approach. The method deviates in two key stages: the schema encoding stage when a set of evolutionary rules and representations are defined; and the design evolution stage when alternative design models are generated and evolved. With regard to synergy, the method maximises the return on the labour invested by the design team in developing and encoding a design schema. The design schema is personal and idiosyncratic to the design team, and the task of developing and encoding such a schema is highly subjective. On the other hand, evolving and evaluating large numbers of alternative design is a highly repetitive and objective task. The design team is assigned the first task, and the computational system is assigned the second task. A key advantage of this method is that once encoded, the schema can be used to evolve designs for a large number of projects.

9.1.4 General architecture

Three general architectures

Two existing computational architectures have been identified, and an alternative architecture has been proposed. The existing architectures are the *general synchronous evolutionary architecture* and the *general asynchronous evolutionary architecture*. The proposed architecture is the *general decentralised evolutionary architecture*. Each of these three architectures may be parallelised using global parallelism, whereby a single population (possibly split into different parts) is maintained and one or more evolution steps are performed in parallel.

These architectures are described according to their *evolution mode* and *control structure*. The evolution mode refers to the order in which the evolution steps are applied, with the two possible modes being the synchronous mode and the asynchronous mode. The control structure refers to the way in which the evolution steps are controlled, with the two possible control structures being centralised control and decentralised control.

- The *synchronous architecture* (see figure 4.1 on page 81) (Holland, 1975) uses a synchronous evolution mode in combination with a centralised control structure. The populations of individuals is split into two parts — a main population and an intermediate population — which are manipulated by five evolution steps: survival, reproduction, development, evaluation and selection. A cyclical process is created whereby the main population is repeatedly replaced by a the intermediate population. For the synchronous evolution mode, three more specific sub-modes are commonly used that differ in the number of individuals allowed to survive from one population to the next: the generational mode does not allow for any survival; the elitist mode allows a small number of the best individuals to survive; and the steady-state mode allows all individuals except a small number of the worst to survive. This architecture may be parallelised using the global master-slave parallel model (see figure 4.4 on page 86). The master controls the evolutionary process and delegates certain steps — usually the evaluation step — to be performed by multiple slaves. In this case, the parallel implementation reduces the execution time but does not change the fundamental behaviour of the algorithm. As with the non-parallel version, the evolutionary process must wait for all individuals to be processed by one step before proceeding onto the next step.

- The *asynchronous architecture* (see figure 4.5 on page 90) (Cantu-Paz, 1998; Rasheed and Davison, 1999) uses an asynchronous evolution mode in combination with a centralised control structure. A single population is manipulated by five evolution steps: selection, reproduction, development, evaluation, and survival. An iterative

process is created whereby individuals in the population are selected and used to create new individuals, which are inserted back into the population by replacing existing individuals. Individuals are usually added to the population one at a time, with the population remaining dynamically stable. The asynchronous mode used by this architecture may also be described as a steady-state evolution mode. This architecture is commonly parallelised using the global master-slave model (see figure 4.6 on page 91). In this case, as well as reducing the execution time, the parallel implementation also changes the fundamental behaviour of the algorithm. The order in which individuals are created does not necessarily need to be the same as the order in which they are reinserted back into the population. This means that the evolutionary process does not have to wait for all individuals to be processed.

- The *decentralised architecture* (see figure 7.1 on page 173) uses an asynchronous evolution mode in combination with a decentralised control structure. A single population is manipulated by four autonomous evolution steps: reproduction (which includes a selection mechanism), development, evaluation and survival. These four steps are defined as independent iterative processes that extract individuals from the population, process them, and either insert the resulting individuals back into the population, or — in the case of the survival step — delete individuals in the population. The population is managed by a separate process that ensures that the size of the population remains dynamically stable. This architecture may be parallelised using the global client-server model (see figure 1.4 on page 21). The server manages the population and performs the reproduction and survival steps, while clients perform the developmental and evaluation steps.

Advantages of the proposed general architecture

These three computational architectures can be analysed with regard to their scalability. The requirement for scalability was the first of the requirements identified in chapter 1, the second being customizability. Customizability will be considered in the next section with a more detailed description of the architecture. In chapter 7, two aspects of scalability were discussed: the reduction in execution time, and the improvement in flexibility and robustness.

For all three architectures, parallel versions have been discussed that can significantly reduce the execution of the evolutionary process. With the asynchronous and decentralised architectures, the execution time is further reduced by using an asynchronous steady-state evolution mode. This mode has been shown to result in close to linear speed-up in situations where the evaluation step is complex and expensive (Rasheed and Davison, 1999). The advantage of this mode over the synchronous mode is that the evolutionary process does not need to wait for all individuals

to be processed before proceeding to the next evolutionary step. The evolutionary process can start to incorporate or reject the genetic material of the individual, as soon as an individual has been fully evaluated.

The main advantage of the decentralised architecture over the asynchronous architecture is the improvement in flexibility and robustness due to the decentralised control structure. Flexibility is improved because client computers may be added and removed while evolution is in progress without requiring any reconfiguration on the server. This allows computing resources available in research labs and offices to be intermittently used for other purposes and re-assigned to the evolutionary process. Robustness is improved because the failure of any one of the clients will not significantly affect the evolutionary process (providing there are other clients available to perform the same task). Furthermore, the failed client can simply be restarted and re-assigned to the evolutionary process.

The decentralised architecture also allows for a collaborative process between members of the design team in different locations. The client computers may be globally distributed, thereby allowing the different evolution steps to be executed in locations where the required expertise exists. This would be most relevant for the evaluation step. Evaluation may use complex simulation and analysis software applications for which expert knowledge is required. For example, the client computer performing complex lighting simulations may be located in the office of the environmental engineers.

9.1.5 Detailed architecture

Style problem

The style problem (Bentley, 1999b) is related to the re-usability of the evolutionary system. In general, re-usability decreases as the rules and representations embedded in the system become more knowledge-rich. In the extreme case, if the rules and representations are based on a design style or character that is specific to one designer, the re-usability of the evolutionary system becomes limited to one person. The variability problem and the style problem lead to opposite conclusions.

The variability problem illustrates that knowledge-rich rules and representations must be used to achieve reasonable performance. The design schema consists of knowledge about the design character of a family of design. This knowledge is used by the design team to create a set of rules and representations that control design variability.

The style problem illustrates that knowledge-lean rules and representations must be used to achieve reasonable re-usability. Since the design schema is specific to a design team, an evolutionary system created using the schema-based approach will have limited re-usability. (Even for a single designer, the design schema for one project may not be applicable to another project.) As a result, a new evolutionary system would have

to be created, which is an impractical approach.

Summary of detailed architecture

A detailed computational architecture (see figure 7.3 on page 180) has been developed based on the parallel version of the *general decentralised evolutionary architecture* described above. The evolutionary process consists of an asynchronous steady-state evolution mode in combination with a decentralised control structure.

The conflict between the variability problem and the style problem is resolved by creating an evolutionary system that consists of a generic core that is highly re-usable and a set of specialised components that can incorporate the knowledge-rich rules and representations (see figure 7.2 on page 175). The generic core may be re-used by any design team for any design task. Each design team must develop their own design schemas, and based on these schemas, develop a set of knowledge-rich rules and representations.

Customizability is achieved by decoupling the evolutionary process from the rules and representations used by that process. This results in each evolution step being split into a generic *module* and a specialised *routine*. The generic core of the architecture consists of the evolution modules and a module that manages the population. This generic core defines the overall evolutionary process by specifying the interactions between the evolution modules and the population module.

However, the generic modules do not include any of the rules and representations required for processing individuals. These rules and representations are encapsulated in the routines. Each time an evolution module needs to process an individual, it will invoke the specialised routine. The rules and representations used by the evolutionary system can easily be changed by replacing the routines. The routines are independent executable programs without any restrictions as to what types of rules and representations may be incorporated.

For certain evolution steps — including the development and evaluation steps — the routines may invoke two other types of specialised components: environment data-files and existing applications. These specialised components allow for a further level of customization. Environment data-files allow for the routines to make use of information relating to the design constraints and design context. (For example, an epigenetic generative routine may be defined.) Existing applications — such as CAD systems, visualization programs and analysis and simulation systems — may be invoked by the routines for their specialised functionality.

9.1.6 Controlled variability

Encoding the design schema

A key task in the proposed design method involves the design team developing a generative process that produces *controlled variability*. This is part of the schema encoding stage, which entails defining the rules and representations for the evolution steps. The generative process consists of a set of generative rules used by the developmental step to create a variety of three-dimensional design models.

In order achieve controlled variability, the generative process needs to fulfil four key objectives. First, it should be capable of generating the required level of complexity. Second, it should generate designs that all share a certain essential and identifiable character. Third, it should generate designs that differ significantly in terms of their overall organization and configuration. Finally, it should not generate chaotic forms that are not directly and straightforwardly interpretable as designs.

Demonstration of controlled variability

In order to verify the feasibility of creating such a generative process, an example schema has been defined and a generative process has been created. Genotype and phenotype representations have been defined and a developmental routine, an initialization routine and a visualization routines that manipulate these representations have been implemented.

The initialization routine has been used to create a population of individuals with random genotypes, and for each individual the developmental routine was used to generate a design model. The visualization routine was used to inspect these design models. This inspection showed that the four objectives identified above were fulfilled, thereby demonstrating that controlled variability can be achieved.

9.1.7 Summary of main contributions

Main contributions

The main contributions are summarised as follows:

- The variability problem has been identified as the key factor in the performance of the evolutionary system.
- A design method has been developed that is more conservative and more synergetic than other generative evolutionary methods.
- A general computational architecture has been proposed that is more scalable than other general architectures.
- A detailed computational architecture has been developed that is highly customisable.

- A generative process has been demonstrated that is capable of producing controlled variability.

Communities of users

As discussed in chapter 1, the design method and the computational architecture are targeted at different communities of users. The design method is targeted at a community whose primary goal is to create designs for buildings, referred to as *designers*; the architecture is targeted at a community whose primary goal is the development of a computational design system, referred to as *researchers*. The labels 'designer' and 'researcher' should not be understood in a stereotypical sense: the design team may include computer experts and programmers, just as the research team may include architects. (In some cases, the design team and the research team may be the same set of people.)

Although the architecture is targeted at researchers, a system based on this architecture would be used by designers. If a research team implements the system, this system will be transferred to the design community, to be used by design teams for designing buildings. When developing the architecture, it was necessary to take both these communities into account: researchers being immediate target community, and designers the eventual target community.

9.2 Future work

9.2.1 Short term

Five stages

In chapter 1, a research process was described that consisted of five stages (Nunamaker et al., 1991): constructing a conceptual framework; developing a system architecture; analyzing and designing the system; building the (prototype) system; and observing and evaluating the system (see figure 1.5 on page 29).

This research consists of the first three stages. For the first stage, the design method has been developed that defines how the system would be used. For the second stage, the computational architecture has been developed that identifies the significant hardware and software components and their interactions. Finally for the third stage, the components of the architecture have been analysed in detail and possible implementation strategies have been considered.

Build the (prototype) system

The next stage will involve building a prototype system based on the architecture developed in this research. A more comprehensive specification can be created that defines how the system should be built. This process will highlight problems that may exist in the proposed architecture

and result in the architecture being further refined. Once a prototype system has been completed, the next stage — involving observation and evaluation of the system in use — can be commenced.

Observe and evaluate the system

For initial evaluation, the encoded schema developed in the demonstration can be used as a first test case. A design environment and a set of design objectives will need to be defined (possibly based on a real design task). The environment will include both a set of design constraints and a description of the design context. The system may be used to start evolving a population of designs. One way to measure evolutionary process is to track the performance of the individuals in the population with respect to the design objectives. The progress of this population as a whole can be monitored and analysed.

In order to further evaluate the system, a set of design schemas need to be defined with varying levels of complexity. These schemas can be encoded and used to test the system. Such a process may need to consider additional factors when measuring the success of the system. (For example, should the measure of success include the perceptions of the designer using the system?)

9.2.2 Long term

Cyclical research process

The five stage research process described above is cyclical. The last stage of observation and evaluation will result in new theories and models of how the computational system and the design method could be improved. It is envisaged that, in the long term, such a generative evolutionary design framework would itself evolve. One potential area of future research is providing support for the schema encoding stage.

Support for schema encoding

The schema encoding process is complex and time consuming. This process may be supported in three main ways: first, a set of interfaces to commonly used applications could be created; second, a library of commonly used functions and procedures could be developed; and third, a visual programming environment could be created.

- Design teams are likely to want to use many of the same applications for the developmental, visualization and evaluation steps. In many cases, they are also likely to use standard representations for describing design models. This will allow a library of interfaces to be developed for linking to these applications. These interfaces could incorporate file format translations and other configurational issues.

- Design teams are also likely to require many similar functions and procedures. The process of encoding a range of different schemas is likely to result in certain patterns emerging. Some of the functions and procedures used to encode these schemas may start to repeat themselves. Such patterns will allow a library of functions and procedures to be defined. For example, the transformable-grid construct in the demonstration could be used in many other types of schemas.

- Software may be developed that would allow design teams to create encoded schemas in a visual graphical environment, rather than by programming. This approach is described as *visual end-user programming*. End-user programming software uses graphical elements to enable users to create complex programming logic by drawing two-dimensional diagrams (Aish, 2000). Appropriate new metaphors are required that will allow schemas to be developed and encoded with just a minimum understanding of basic programming concepts.

9.3 Conclusions

The research presented in this thesis attempts to create a comprehensive generative evolutionary design framework. This framework considers both broad theoretical issues related to the design process and specific computational issues related to the implementation of a design system. The framework is based on previous design methods and computational architectures, and includes new features.

The goal of developing such a framework is to contribute to the development of a practical generative evolutionary design approach. It is hoped that this framework will provide a basis for other researchers to implement evolutionary design systems.

Glossary

Computational architecture: An implementation plan of how significant software and hardware components of a computer system are structured and organised. This includes the functions and interactions of different components. Communication protocols and data formats may also be defined.

Control structure: The way in which evolution steps are controlled. Two main control structures are *centralised* control and the *decentralised* control. With centralised control, a cyclical process invokes and applies evolution steps to individuals in the population. With decentralised control, the evolution steps are autonomous processes that manipulate individuals in the population.

Design environment: The constraints and context for a particular design. The constraints describe the requirements that the building must fulfil and may include factors such as budget, spatial requirements, and performance targets. The context describes the building site and may include site conditions, neighbouring conditions and weather conditions. The design environment covers all those conditions that influence the success or failure of the design, but that are not part of the design itself.

Design method: A semi-formalised design process that explicitly prescribes a way of designing a type of product. The process is structured as a set of tasks to be carried out by the designer or design team, possibly in some specific order. A design method is a conjecture of a potentially useful design process. It is useful to the extent that its application will lead to products that embody certain design qualities that are seen to be beneficial or desirable.

Design schema: A design conceptualization that captures the essential and identifiable character of a varied family of designs by one designer or design team. It encompasses those characteristics common to all members of the family, possibly including issues of aesthetics, space, structure, materials and construction. Although members of the family of designs share these characteristics, they may differ considerably from one another in overall organization and configuration. Design schemas are seen as formative design generators; their intention is synthetic rather than

analytic.

Evolution mode: The way in which the evolution steps process individuals in the population. The two main evolution modes are the *synchronous* mode and the *asynchronous* mode. With the synchronous mode, the evolutionary process stops and waits for the processing of all individuals by one evolution step to be completed - before proceeding onto the next evolution step. With the asynchronous evolution mode, evolution steps process individuals or small groups in the population as soon as they become available.

Evolutionary algorithm: An algorithm loosely based on the neo-Darwinian model of evolution through natural selection. A population of individuals is maintained and an iterative process applies a number of evolution steps that create, transform, and delete individuals in the population. Individuals are rated for their effectiveness, and on the basis of these evaluations, new individuals are created using 'genetic operators' such as crossover and mutation. The process is continued through a number of generations with the aim of improving the population as a whole.

Evolutionary design: A design approach that relies on evolutionary software systems to aid in the process of designing. Such a system employs evolutionary algorithms to evolve whole populations of design alternatives. The software may be used to evolve complete designs or parts of designs.

Generative evolutionary design: A design approach that uses an evolutionary system to evolve surprising or challenging design alternatives, for ill-defined design tasks that embody multiple and conflicting objectives. Some kind of growth process is used to generate design alternatives that vary significantly from one another. The system then relies on either human judgement or evaluation algorithms to evolve a population of design alternatives.

Niche environment: A type or category of design environment, encompassing a range of possible constraints and a range of possible contexts. If a specific design environment falls in such a environmental niche set, this design environment is described as *matching* or *falling within* the environmental niche.

Parametric evolutionary design: A design approach that uses an evolutionary system to search for optimal or satisficing design solutions to well defined design problems. The overall design is predefined and those parts thought to require improvement are parameterised. This results in a parametric model into which values for parameters can be inserted to create alternative design solutions. The evolutionary system uses a set of fitness functions

or objective functions to evolve an optimal or satisficing set of parameters.

Bibliography

Abelson, H., Sussman, G. J., and Sussman, J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press, Cambride, MA.

Agabani, F. A. (1980). *Cognitive Aspects in Architectural Design Problem Solving*. Doctoral dissertation, University of Sheffield.

Aish, R. (1977). Prospects for design participation. *Design Methods and Theories*, 11(1).

Aish, R. (2000). Custom objects: a model-oriented end-user programming environment. In *Materials of the Workshop on Visual Languages for End-User and Domain-Specific Programming*.

Alba, E. and Troya, J. M. (1999). A survey of parallel distributed genetic algorithms. *COMPLEXITY*, 4(4):31–51.

Alba, E. and Troya, J. M. (2001). Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Comput. Systems*, 17(4):451–465.

Alexander, C. (1971). The state of the art in design methodology (replies to questions by M. Jacobson). *DMG Newsletter*, pages 3–7.

Alexander, C., Ishikawa, S., Silverstein, B., and Others (1977). *A Pattern Language*. Oxford University Press, New York, NY.

Alexander, C., Ishikawa, S., Silverstein, B., and Others (1979). *The Timeless Way of Building*. Oxford University Press, New York, NY.

Angeline, P. J. (1995). Morphogenic evolutionary computations: Introduction, issues and examples. In McDonnell, J., Reynolds, B., and Fogel, D., editors, *Evolutionary Programming IV: The Fifth Annual Conference on Evolutionary Programming*, pages 387–401. MIT Press.

Angeline, P. J. and Pollack, J. B. (1994). Coevolving high-level representations. In Langton, C. G., editor, *Proceedings of Artificial Life III*, pages 55–71, Reading, MA. Addison-Wesley.

Archer, B. (1979). The three Rs. *Design Studies*, 1(1):18–20.

Arenas, M. G., Collet, P., Eiben, A. E., Jelasity, M., Merelo, J. J., Paechter, B., Preuß, M., and Schoenauer, M. (2002). A framework for distributed evolutionary algorithms. In Merelo Guervós, J. J., Adamidis, P., Beyer, H.-G., Fernández-Villacañas, J.-L., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature - PPSN VII*, volume 2439 of *Lecture Notes in Computer Science*, pages 665–675. Springer-Verlag.

Bäck, T. (1993). Optimal mutation rates in genetic search. In Forrest, S., editor, *Proceedings in the 5th International Conference on Genetic Algorithms*, pages 2–8, San Mateo, CA. Morgan Kaufmann.

Bäck, T. (1994). Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 57–62, Piscataway, NJ. IEEE Press.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, NY.

Bäck, T. (2000). Introduction to evolutionary algorithms. In Bäck et al. (2000b), chapter 7, pages 59–63.

Bäck, T., B., D., Whitley, D., and Angeline, P. J. (2000a). Mutation operators. In Bäck et al. (2000b), chapter 32, pages 237–255.

Bäck, T., Fogel, D. B., and Michalewicz, T., editors (2000b). *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics Publishing, Bristol and Philadelphia, 1st edition.

Back, T., Hammel, U., and Schwefel, H.-P. (1997). Evolutionary computation: comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1):3–17.

Bäck, T. and Schwefel, H.-P. (1996). Evolutionary computation: An overview. In *Proceedings of the Third IEEE Conference on Evolutionary Computation*, pages 20–29, Piscataway, NJ. IEEE Press.

Baker, J. (1985). Adaptive selection methods for genetic algorithms. In Grefenstette, J., editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 101–111. Lawrence Erlbaum Associates.

Baron, P., Fisher, R., Mill, F., Sherlock, A., and Tuson, A. (1997). A voxel-based representation for the evolutionary shape optimisation of a simplified beam: A case-study of a problem-centred approach to genetic operator design. In *2nd On-line World Conference on Soft Computing in Engineering Design and Manufacturing (WSC2)*.

Baron, P., Fisher, R., Tuson, A., and Mill, F. (1999). A voxel-based representation for evolutionary shape optimization. *AI EDAM Special Issue: Evolutionary Design*, 13(3):145–156.

Bazjanac, V. and Crawley, D. B. (1997). The implementation of industry foundation classes in simulation tools for the building industry. In *Proceedings of Building Simulation '97 Conference*, pages 203–210.

Beasley, D. (2000). Possible applications of evolutionary computation. In Bäck et al. (2000b), chapter 2, pages 4–19.

Bell, A. D. (1986). The simulation of branching patterns in modular organisms. *Philosophical Transactions of the Royal Society of London, SeriesB: Biological Sciences*, 313:143–159.

Bemis, A. F. (1936). *The Evolving House*. MIT Press, Cambridge, MA.

Bentley, P. (1999a). From coffee tables to hospitals: Generic evolutionary design. In Bentley (1999d), chapter 18, pages 405–423.

Bentley, P. (1999b). An introduction to evolutionary design by computers. In Bentley (1999d), chapter 1, pages 1–73.

Bentley, P. (1999c). Special issue: Evolutionaey design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AIEDAM)*, 13(3):143.

Bentley, P. (2000a). Exploring component-based representations - the secret of creativity by evolution? In *Proceedings of The Fourth International Conference on Adaptive Computing in Design and Manfacture (ACDM 2000)*, page unpaginated, University of Plymouth, UK.

Bentley, P. (2000b). Special section: Evolutionary design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AIEDAM)*, 14(1):1.

Bentley, P. J. (1996). *Generic Evolutionary Design of Solid Objects using a Genetic Algorithm.* Doctoral dissertation, Division of Computing and Control Systems, Department of Engineering, University of Huddersfield.

Bentley, P. J., editor (1999d). *Evolutionary Design by Computers*. Morgan Kaufmann Publishers, San Francisco, CA.

Bentley, P. J. and Corne, D. W., editors (2002). *Creative Evolutionary Systems*. Academic Press, London, UK.

Bentley, P. J. and Kumar, S. (1999). Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Genetic and Evolutionary Computation Conference (GECCO '99)*, pages 35–43, Orlando, Florida, USA.

Booker, L. B., Fogel, D. B., Whitley, D., Angeline, P. J., and Eiben, A. E. (2000). Recombination. In Bäck et al. (2000b), chapter 33, pages 256–307.

Botsford, G. (1995). Solar logic. Unpublished research proposal developed in Diploma Unit 11 at the Architectural Association, under the direction of John Frazer.

Broadbent, G. (1981). The morality of designing. In Jacques and Powell (1981), pages 309–328.

Broadbent, G. (1988). *Design in Architecture: Architecture and the Human Sciences*. David Fulton Publishers, London, UK.

Bruggen, C. v. (1997). *Frank O. Gehry: Guggenheim Museum Bilbao*. Harry N. Abrams, New York, NY.

Caldas, L. (2001). *An Evolution-Based Generative Design System: Using Adaptation to Shape Architectural Form*. Doctoral dissertation, Massachusetts Institute of Technology.

Caldas, L. (2002). Evolving three-dimensional architectural form: An application to low-energy design. In Gero, J. S., editor, *Proceedings of the Seventh International Conference on Artificial Intelligence in Design (AID'02)*, pages 351–370, Cambridge, UK. Kulwer Academic Publishers.

Caldas, L. and Norford, L. (2001). Architectural constraints in a generative design system: Interpreting energy consumption levels. In *Proceedings of the Seventh International IBPSA Conference*, pages 1397–1404, Rio de Janeiro, Brazil.

Caldas, L., Norford, L., and ao Rocha, J. (2003). An evolutionary model for sustainable design. *Management of Environmental Quality*, 14(2/3):383–397.

Caldas, L. G. and Norford, L. K. (2004). Shape generation using pareto genetic algorithms: Integrating conflicting design objectives in low-energy architecture. *International Journal of Architectural Computing*, 1(4):503–515.

Cantu-Paz, E. (1997). A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171.

Cantu-Paz, E. (1998). Designing efficient master-slave parallel genetic algorithms. In Koza, J., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D., Goldberg, M. G. D. E., Iba, H., and Riolo, R., editors, *Genetic Programming: Proceeding of the Third Annual Conference*, San Fransisco, CA. Morgan Kaufmann.

Cantu-Paz, E. and Goldberg, D. E. (1999). On the scalability of parallel genetic algorithms. *Evolutionary Computation*, 7(4):429–449.

Chien, S.-F., Donia, M., Synder, J. D., and Tsai, W.-J. (1998). SG-CLIPS: A system to support the automatic generation of designs from grammars. In *Proceedings of The Third Conference On Computer Aided Architectural Design Research in Asia (CAADRIA '98)*, pages 445–454.

Chomsky, N. (1956). Three models for the description of language. *IRE Trans. Info. Theory*, 1:113–124.

Chong, F. S. and Langdon, W. B. (1999). Java based distributed genetic programming on the internet. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'99)*, page 1229, Orlando, Florida, USA. Morgan Kaufmann.

Churchman, C. W. (1967). Wicked problems. *Management Science*, 14(4):(B–141) – (B–142).

Coates, P., Broughton, T., and Jackson, H. (1999). Exploring three-dimensional design worlds using lindenmayer systems and genetic programming. In Bentley (1999d), pages 323–341.

Corbusier, L. (1982). Modulor ii. In de Francia, P. and Bostock, A., editors, *Modulor I and II*. Cambridge University Press, Cambridge, MA. Modular II was first published in 1955.

Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M., and Gero, J. S. (1990). *Knowledge-Based Design Systems*. Addison-Wesley, Reading, MA.

Cross, N. (1993). A history of design methodology. In de Vries, M. J., Cross, N., and Grant, D. P., editors, *Design Methodology and Relationships with Science*, pages 15–27. Dordrecht, Netherlands: Kluwer Academic Publishers.

Cross, N. (1999). Natural intelligence in design. *Design Studies*, 20(1):25–39.

Cross, N. (2000). Design as a discipline. In Durling, D. and Friedman, K., editors, *Proceedings of the Conference Doctoral Education in Design: Foundations for the Future*, pages 93–100, La Clusaz, France. Staffordshire, UK: Staffordshire University Press.

Cross, N. (2001). Designerly ways of knowing: Design discipline versus design science. *Design Issues*, 17(3):49–55.

Darwin, C. (1968). *The Origin Of Species*. Penguin Books. (First published by John Murray, 1859).

Dasgupta, D. and Michalewicz, Z., editors (1997). *Evolutionary Algorithms In Engineering Applications*. Springer-Verlag.

Davis, G. B. (2000). Information systems conceptual foundations: Looking backward and forward. In Baskerville, R., Stage, J., and DeGross, J. I., editors, *Organizational and Social Perspectives on Information Technology*, pages 61–82, Massachusetts, MA. IFIP TC8 WG 8.2, Kluwer Academic Publishers.

Davis, S. (1987). *Future Perfect*. Addison-Wesley, Reading.

Dawkins, R. (1983). Universal Darwinism. In Bendall, D. S., editor, *Evolution from Molecules to Men*, chapter 20, pages 403–425. Cambridge University Press, Cambridge, UK.

Dawkins, R. (1986). *The Blind Watchmaker*. Norton, New York, NY.

de la Maza, M. and Tidor, B. (1993). An analysis of selection procedures with particular attention paid to proportional and bolzmann selection. In Forrest, S., editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 124–131, San Mateo, CA. Morgan Kaufmann Publishers.

Drake, J. (1979). The primary generator and the design process. *Design Studies*, 1(1):36–44.

Eastman, C. (1999). *Building Product Models: Computer Environments Supporting Design and Construction*. CRC Press, Boca Raton, FL.

Eastman, C. M. (1970). On the analysis of the intuitive design process. In *Emerging Methods in Environmental Design and Planning*. MIT Press, Cambridge, MA.

Eshelman, L. J. and Schaffer, J. D. (1993). Real-coded genetic algorithms and interval-schemata. In Whitley (1993), pages 187–202.

Evans, R. (1995). *The Projective Cast: Architecture and its Three Geometries*. MIT Press, Massachusetts, MA.

Flake, G. W. (1998). *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. MIT Press, Cambridge, MA.

Fogel, D. B. (1995). Evolutionary computation: Towards a new philosophy of machine intelligence. *IEEE Press*.

Fogel, L. J. (1963). *Biotechnology: Concepts and Applications*. Prentice Hall, Englewood Cliffs, NJ.

Frazer, J. (1991). Can computers be just a tool? *Systemica: Mutual Uses of Cybernetics and Science*, 9:27–36.

Frazer, J. H. (1974). Reptiles. *Architectural Design*, 4:231–239.

Frazer, J. H. (1982). Use of simplified three-dimensional computer input devices to encourage public participation in design'. In *Proceedings of the Computer Aided Design 82 Conference*, pages 143–151. Butterworth Scientific.

Frazer, J. H. (1990). A genetic approach to design - towards an intelligent teacup. In *Proceedings of the Many Faces of Design Conference.*

Frazer, J. H. (1992). Data structures for rule-based and genetic design. In *Proceedings of the 10th International Conference of the Computer Graphics Society on Visual Computing : integrating computer graphics with computer vision*, pages 731–744, New York, NY. Springer-Verlag.

Frazer, J. H. (1995a). Architectural experiments in cyberspace. *Architectural Design - Architects in Cyberspace*, pages 78–79.

Frazer, J. H. (1995b). *An Evolutionary Architecture.* AA Publications, London, UK.

Frazer, J. H. (1995c). The interactivator. *AA Files*, 72–73.

Frazer, J. H. (2002). Creative design and the generative evolutionary paradigm. In Bentley and Corne (2002), pages 253–274.

Frazer, J. H. and Connor, J. (1979). A conceptual seeding technique for architectural design. In *Proceedings of International Conference on the Application of Computers in Architectural Design and Urban Planning (PArC79)*, pages 425–434, Berlin. AMK.

Frazer, J. H., Frazer, J. M., and Frazer, P. A. (1980). Intelligent physical three-dimensional modelling systems. In *Proceedings of the Computer Graphics 80 Conference*, pages 359–370. Online Publications.

Frazer, J. H. and Rastogi, M. (1998). The new canvas. *Architectural Design: Architects in Cyberspace II*, 68(11/12):8–11.

Frazer, J. H., Rastogi, M., and Graham, P. (1995a). Biodiversity in design via the Internet. In *Digital Creativity: A Conference on Computers in Art & Design Education (CADE 95)*, pages 97–106.

Frazer, J. H., Rastogi, M., and Graham, P. (1995b). The interactivator. *Architectural Design - Architects in Cyberspace*, pages 80–81.

Frazer, J. H., Sun, J., and Tang, M. (2000). Research on applications of genetic algorithms to computer aided product design - case studies on three approaches. In *Proceedings of the 3rd International Conference Computer Aided Industrial Design and Conceptual Design (CAID & CD '2000)*, pages 223–228, Beijing, PRC. International Academic Publishers,.

Frazer, J. H., Tang, M., and Sun, J. (1999). Towards a generative system for intelligent design support. In *Proceedings of the Fourth Conference on Computer Aided Architectural Design Research in Asia (CAADRIA'99)*, pages 285–294. Shanghai Scientific and Technological Publishing House.

Funes, P. and Pollack, J. (1999). Computer evolution of buildable objects. In Bentley (1999d), pages 387–403.

Gardner, M. (1970). Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American*, 223(4):120–123.

Gardner, M. (1971). Mathematical Games: On cellular automata, self-reproduction, the Garden of Eden and the game of 'life'. *Scientific American*, 224(2):112–117.

Garis, H. D. (1994). Growing an artificial brain: The genetic programming of million-neural-net-module artificial brains with trillion cell cellular automata machines. In Sebald, A. V. and Fogel, L. J., editors, *Proceedings of the Third Annual Conference on Evolutionary Porgramming*, pages 335–343, London. World Scientific.

Ghosh, A. and Tsutsui, S., editors (2003). *Advances in evolutionary computing: theory and applications*. Springer-Verlag, New York, NY.

Gips, J. (1975). *Shape Grammars and Their Uses: Artificial Perception, Shape Generation and Computer Aesthetics*. Birkhäuser, Basel.

Gips, J. (1999). Computer implementation of shape grammars. Report, NSF/MIT Workshop on Shape Computation, Department of Architecture, School of Architecture and Planning, Massachusetts Institute of Technology.

Glanville, R. (1998). Researching design and designing research. In Strandman, P., editor, *No Guru, No Method? Discussion on Art and Design Research*, B 55. Publication series of the University of Art and Design Helsinki UIAH.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA.

Goldberg, D. E. (1990). A note on boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing. *Complex Systems*, 4:445–460.

Goldberg, D. E. and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. In Rawlins, G., editor, *Foundations of Genetic Algorithms*, pages 69–93, San Mateo, CA. Morgan Kaufmann.

Gould, S. J. (2000). *Wonderful Life: The Burgess Shale and the Nature of History*. Vintage, London.

Graham, P. C., Frazer, J. H., and Hull, M. C. (1993). The application of genetic algorithms to design problems with ill-defined or conflicting criteria. In Glanville, R. and de Zeeuw, G., editors, *Proceedings of Conference on Values and, (In) Variants*, pages 61–75.

Grefenstette, J. (1981). Parallel adaptive algorithms for function optimization. Technical Report CS-81-19, Vanderbilt University, Nashville, TN.

Gropius, W. (1962). *The Scope of Total Architecture*. Collier Books, New York, NY.

Gruau, F. (1992). Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In Schaffer, J. D. and Whitley, D., editors, *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pages 55–74. The IEEE Computer Society Press, Los Alamitos, CA.

Haeckel, E. (1874). *Anthropogenie oder Entwickelungsgeschichte des Menschen*. Engelmann, Leipzig.

Hancock, P. J. B. (1995). Selection methods for evolutionary algorithms. In Chambers, L., editor, *Practical handbook of genetic algorithms: new frontiers*, pages 67–92. CRC Press.

Harp, S. A. and Samad, T. (1991). Genetic synthesis of neural network architecture. In Davis, L., editor, *Handbook of Genetic Algorithms*. Von Nostrand Reinhold, New York, NY.

Hensen, J. L. M. (2002). Simulation for performance based building and systems design: some issues and solution directions. In *Proceedings 6th International Conference on Design and Decision Support Systems in Architecture and Urban Planning*.

Hirschheim, R. (1985). Information systems epistemology: An historical perspective. In Mumford, E., Hirschheim, R., Fitzgerald, G., and Wood-Harper, A., editors, *Research Methods in Information Systems*, Amsterdam, NL. IFIP TC8 WG 8.2, Elsevier Science Publishers. (Colloquium entitled *Information Systems Research - a doubtful science?*).

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.

Hu, J., Seo, K., Li, S., Fan, Z., Rosenberg, R. C., and Goodman, E. D. (2002). Structure fitness sharing (SFS) for evolutionary design by genetic programming. In Langdon, W. B., Cantú-Paz, E., Mathias, K. E., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C.,

Miller, J. F., Burke, E., and Jonoska, N., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 780–787, New York, NY. Morgan Kaufmann Publishers.

Iivari, J. (1987). A methodology for IS development as an orgainsational change: A pragmatic contingency approach. In Klein, H. and Kumar, K., editors, *Information Systems Development for Human Progress in Organisations*. North Holland, Amsterdam, NL.

Iivari, J. (1991). A paradigmatic analysis of contemporary schools of is development. *European Journal of Information Systems*, 1(4):249–272.

Iivari, J., Hirschheim, R., and Klein, H. K. (1998). A paradigmatic analysis contrasting information systems approaches and methodologies. *Information Systems Research*, 9(2):164–193.

Jaanusson, V. (1981). Functional thresholds in evolutionary progress. *Lethaia*, 14:251–260.

Jackson, H. (2002). Toward a symbiotic coevolutionary approach to architecture. In Bentley and Corne (2002).

Jacques, R. (1981). Introduction. In Jacques and Powell (1981), pages ix–xii.

Jacques, R. and Powell, J., editors (1981). *Design : Science : Method. Proceedings of the 1980 Design Research Society Conference*. Westbury House, Guildford, UK.

Janikow, C. and Michalewicz, Z. (1991). An experimental comparison of binary and floating point representations in genetic algorithms. In Belew, R. and Booker, L., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 31–36.

Järvinen, P. (1996). The new classification of research approaches. In Zemanek, H., editor, *36 years of IFIP*. IFIP Secretariat, Laxenburg, Austria. (Internet location: http://www.ifip.or.at/36years/36years.html).

Järvinen, P. (1999). *On Research Methods*. n.p., Opinpaja, Tampere.

Järvinen, P. (2000). On a variety of research output types. In Svensson, L., Snis, U., Srensen, C., Fägerlind, H., Lindroth, T., Magnusson, M., and Östlund, C., editors, *Proceedings of the IRIS 23. Laboratorium for Interaction Technology*, Sweden. University of Trollhättan Uddevalla.

Jones, J. C. (1970). *Design Methods: Seeds of Human Futures*. Wiley, London, UK, 1st edition.

Jong, K. A. D. (1975). *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. Doctoral dissertation, University of Michigan. (Dissertation Abstract International, 36(10), 5140B.).

Jong, K. A. D. (1993). Genetic algorithms are not function optimizers. In Whitley (1993), pages 5–17.

Kanal, L. and Cumar, V., editors (1988). *Search in Artificial Intelligence*. Springer-Verlag.

Kargupta, H. (2003). Gene expression and scalable genetic serach. In Ghosh and Tsutsui (2003), pages 293–319.

Kauffman, S. A. (1993). *The Origins of Order: Self Organisation and Selection In Evolution*. Oxford University Press, New York, NY.

Keen, P. G. W. (1987). MIS research: current status, trends and needs. In Buckingham, R. A., Hirschheim, R. A., Land, F. F., and Tully, C. J., editors, *Information systems education. Recommendations and implementation*, chapter 1, pages 1–13. Cambridge University Press, Cambridge, UK.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476.

Knight, T. (1994). *Transformations in Design: A formal approach to stylistic change and innovation in the visual arts*. Cambridge University Press, Cambridge, UK.

Knight, T. (1999). Applications in architectural design, and education and practice. Report, NSF/MIT Workshop on Shape Computation, Department of Architecture, School of Architecture and Planning, Massachusetts Institute of Technology.

Koza, J. R. (1990). Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Stanford University.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Kumar, S. and Bentley, P. J. (2003). Computational embryology: Past, present and future. In Ghosh and Tsutsui (2003), pages 461–477.

Lawson, B. (1994). *Design in Mind*. Butterworth-Heinemann, Jordan Hill, Oxford, UK.

Lawson, B. (1997). *How Designers Think: The Design Process Demystified*. Architectural Press, Oxford, UK, 3rd edition.

Lawson, B. R. (1972). *Problem Solving in Architectural Design*. Doctoral dissertation, University of Aston, Birmingham.

Lindenmayer, A. (1968). Mathematical models of cellular interactions in development, I & II. *Journal of Theoretical Biology*, 18:280–315.

Lindenmayer, A. (1982). Developmental algorithms: Lineage versus interactive control mechanisms. In Subtelny, S. and Green, P. B., editors, *Developmental order: Its origin and regulation*, pages 219–245. Alan R. Liss, New York, NY.

Lintermann, B. and Deussen, O. (1999). Interactive modeling of plants. *IEEE Computer Graphics and Applications*, 19(1):2–11.

Luke, S. (1998). Genetic programming produced competitive soccer softbot teams for robocup97. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 214–222. University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann.

Mahdavi, A. (1998). A middle way to integration. In *Proceedings of the 4th Design and Decision Support Systems in Architecture and Urban Planning Conference*.

Mandelbrot, B. B. (1975). *The Fractal Geometry of Nature*. W. H. Freeman, New York, NY.

Manual, U. (1993). *DOE-2 Supplement - Version 2.1E*. Simulation Research Group, Lawrence Berkeley National Laboratory. LBL-34946.

March, S. T. and Smith, G. F. (1995). Design and natural science research on information technology. *Design Support Systems*, 15:251–266.

Maver, T. (2000). A number is worth a thousand words. *Automation in Construction*, 9(4):333–336.

Maver, T. and Petric, J. (2003). Sustainability: real and/or virtual? *Automation in Construction*, 12(6):641–648.

McLachlan, F. and Coyne, R. (2001). The accidental move: accident and authority in design discourse. *Design Studies*, 22:87–99.

Michalewicz, Z. (1993). A hierarchy of evolution programs: An experimental study. *Evolutionary Computation*, 1(1):51–76.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin Heidelberg: Springer-Verlag, 3rd edition. (First edition 1992).

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

Mitchell, M. (1999). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

Mitchell, W. (1990). *The Logic of Architecture: Design, Computation and Cognition*. MIT Press, Cambridge, MA.

Mitchell, W. J. (1977). *Computer-Aided Architectural Design*. Van Nostrand Reinhold, New York, NY.

Mitchell, W. J. (1994). Three paradigms for computer-aided design. In Carrara, G. and Kalay, Y., editors, *Knowledge-Based Computer-Aided Architectural Design*, pages 379–388. Elsevier Science B.V.

Monedero, J. (2000). Parametric design: a review and some experiences. *Automation in Construction*, 9:369–377.

Morgan, G. (1980). Paradigms, metaphors and puzzel solving in organizatio theory. *Adminestrative Science Quarterly*.

Morris, S. C. (1999). *The Crucible of Creation: The Burgess Shale and the Rise of Animals*. Oxford University Press, reprint edition edition.

Newell, A., Shaw, J. C., and Simon, H. A. (1967). The process of creative thinking. In Gruber, H., Terrell, G., and Wertheimer, M., editors, *Contemporary Approaches to Creative Thinking*, pages 63–119. Atherton Press, New Yory, NY. (Original publication: 1957).

Nowostawski, M. and Poli, R. (1999). Parallel genetic algorithm taxonomy. In Jain, L. C., editor, *Proceedings of the Third International Conference on Knowledge-Based Intelligent Information Engineering Systems (KES'99)*, pages 88–92, Adelaide. IEEE Press.

Nunamaker, J. F. and Chen, M. (1990). Systems development in information systems research. *IEEE Press*, pages 631–639.

Nunamaker, J. F., Chen, M., and Purdin, T. D. M. (1991). Systems development in information systems research. *Journal of Management Information Systems*, 7(3):89–106.

O'Neill, M. and Ryan, C. (2000). Incorporating gene expression models into evolutionary algorithms. In Wu, A., editor, *Proceedings of GECCO 2000 Workshop on Gene Expression*, pages 167–173, San Francisco, CA. Morgan Kaufman Publishers.

Owen, R. (1843). *Lectures on the Comparative Anatomy and Physiology of the Invertebrate Animals, Delivered at the Royal College of Surgeons*. Longman, Brown, Green, and Longmans, London, UK.

Page, J. K. (1966). Contribution to building for people. In *1965 Conference Report*.

Paul Schwefel, H. (2000). Advantages (and disadvantages) of evolutionary computation over other approaches. In Bäck et al. (2000b).

Polkinghorne, D. (1983). *Methodology for the Human Sciences: Systems of Inquiry*. State University of New York Press, Albany, NY.

Prusinkiewicz, P. (1995). Visual models of morphogenesis. In Langton, C. G., editor, *Artificial life: an overview*. MIT Press.

Prusinkiewicz, P. and Lindenmayer, A. (1990). *The algorithmic beauty of plants*. Springer-Verlag, New York, NY.

Radcliffe, N. J. (1991). Equivalence class analysis of genetic algorithms. *Complex Systems*, 5:183–205.

Rasheed, K. and Davison, B. D. (1999). Effect of global parallelism on the behavior of a steady state genetic algorithm for design optimization. In Angeline, P. J., Michalewicz, Z., Schoenauer, M., Yao, X., and Zalzala, A., editors, *Proceedings of the Congress on Evolutionary Computation (CEC'99)*, volume 1, pages 534–541. IEEE Press.

Rasheed, K. M. (1998). *GADO: A Genetic Algorithm for Continuous Design Optimization*. Doctoral dissertation, Department of Computer Science, Rutgers University, New Brunswick, NJ. Technical Report DCS-TR-352.

Rechenberg, I. (1973). *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog Verlag, Stuttgart, Germany.

Richardson, M. K., Hanken, J., Gooneratne, M. J., Pieau, C., Raynaud, A., Selwood, L., and Wright, G. M. (1997). There is no highly conserved embryonic stage in the vertebrates: implications for current theories of evolution and development. *Anat. Embryol.*, 196:91–106.

Rittel, H. (1973). The state of the art in design methods. *Design Research and Methods (Design Methods and Theories)*, 7(2):143–147.

Rosca, J. P. and Ballard, D. H. (1994). Learning by adapting representations in genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 407–412, Piscataway, NJ. IEEE Press.

Rosenman, M. A. (1996a). An exploration into evolutionary models for non-routine design. In *AID'96 Workshop on Evolutionary Systems in Design*, pages 33–38.

Rosenman, M. A. (1996b). The generation of form using an evolutionary apporach. In Gero, J. S. and Sudweeks, F., editors, *Proceedings of the Artificial Intelligence in Design Conference (AID '96)*, pages 643–662.

Rosenman, M. A. (2000). Case-based evolutionary design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AIEDAM)*, 14:17–29.

Rosenman, M. A. and Gero, J. S. (1999). Evolving designs by generating useful complex gene structures. In Bentley (1999d), pages 345–364.

Rowe, P. G. (1987). *Design Thinking*. MIT Press, Cambridge, MA.

Rudolph, G. (2000). Evolution strategies. In Bäck et al. (2000b), chapter 9, pages 81–88.

Runnegar, B. (1987). Rates and modes of evolution in the mollusca. In Campbell, K. S. W. and Day, M. F., editors, *Rates of evolution*, pages 39–60. Allen and Unwin, London, UK.

Russell, E. S. (1982). *Form and Function: A Contribution to the History of Animal Morphology*. University of Chicago Press. (First edition: 1916, London: Murray.).

Schaffer, J. D. (1987). Some effects of selection procedures on hyperplane sampling by genetic algorithms. In Davis, L., editor, *Genetic Algorithms and Simulated Annealing*, pages 89–103. Morgan Kaufmann Publishers.

Schaudolph, N. N. and Belew, R. K. (1992). Dynamic parameter encoding for genetic algorithms. *Machine Learning*, 9:9–22.

Schmitt, G. (1999). *Information Architecture: basis of CAAD and its future*. Birkhäuser, Basel, Switzerland. (Original edition "Information Architecture. Basi e futuro del CAAD" published in 1998 by Testo & Immagine, Turin, Italy).

Schwefel, H.-P. (1965). *Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik*. Diplomarbeit, Technische Universität, Berlin.

Setzkorn, C. and Paton, R. C. (2004). Javaspaces - an affordable technology for the simple implementation of reusable parallel evolutionary algorithms. Technical Report ULCS-04-011, University of Liverpool, Department of Computer Science.

Shaefer, C. G. (1987). The ARGOT System: Adaptive representation genetic optimizing technique. In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, Hillsdale, NJ. Lawrence Erlbaum.

Shea, K. (1997). *Essays of Discrete Structures: Purposeful Design of Grammatical Structures by Directed Stochastic Search*. Doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA.

Shea, K. (2001). An approach to multiobjective optimisation for parametric synthesis. In *13th International Conference on Engineering Design (ICED 01) - Design Methods for Performance and Sustainability, WDK 28*, pages 203–210.

Shea, K. (2002). Creating synthesis partners. *Architectural Design: Contemporary Techniques in Architecture*, 72(1):42–45.

Shea, K. (2004). Directed randomness. In Leach, N., Turnbull, D., and Williams, C., editors, *Digital Tectonics*, pages 10–23. Academy Press.

Simon, H. (1981). *The Sciences of the Artificial*. MIT Press, Massachusetts, MA, 2nd edition. (First edition: The Massachusetts Institute of Technology, 1969).

Sims, K. (1994). Evolving 3D morphology and behaviour by competition. In Brooks, R. and Maes, P., editors, *Proceedings of Artificial Life IV*, pages 28–39, Cambridge, MA. MIT Press.

Slack, J. M. W., Holland, P. W. H., and Graham, C. F. (1993). The zootype and the phylotypic stage. *Nature*, 361:490–492.

Soddu, C. (2002). Recognizability of the idea: The evolutionary process of argenia. In Bentley and Corne (2002), chapter 2, pages 109–127.

Stiny, G. (1975). *Pictorial and Formal Aspects of Shape and Shape Grammars: On the Computer Generation of Aesthetic Objects*. Birkhäuser, Basel.

Stiny, G. (1980a). Introduction to shape and shape grammars. *Environment and Planning B*, 7:343–351.

Stiny, G. (1980b). Kindergarten Grammars: Designing with Froebel's building gifts. *Environment and Planning B*, 7:409–462.

Stiny, G. (1994). Shape rules: closure, continuity and emergence. *Environment and Planning B: Planning and Design*, 21:49–78.

Stiny, G. and Gips, J. (1972). Shape grammars and the generative specification of painting and sculpture. *Information Processing*, 71:1460–1465.

Suckle, A., editor (1980). *By Their Own Design*. Whitney, New York, NY.

Sullivan, L. H. (1967). *A system of architectural ornament according with a philosophy of man's powers*. Eakins Press, New York, NY. (First published in 1924).

Sun, J. (2001). *Application of Genetic Algorithms to Generative Product Design Support Systems*. Doctoral dissertation, Hong Kong Polytechnic University.

Sun, J., Frazer, J., and Tang, M.-X. (1999). Application of evolutionary techniques in design for manufacturability. In *Proceedings of the Fifth International Conference on Computer-Aided Conceptual Design (CACD'99)*.

Sylvan, R. and Bennett, D. (1994). *The Greening of Ethics*. White Horse Press, Cambridge, UK.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algoithms*, pages 2–9. Morgan-Kaufmann.

Syswerda, G. (1991). A study of reproduction in generational and steady-state genetic algorithms. In Rawlins, G., editor, *Foundations of Genetic Algorithms*, pages 94–101. Morgan-Kaufmann.

Szalapaj, P. (2000). *CAD Principles of Design, An Analytical Approach to the Computational Representation of Architectural Form*. Architectural Press.

Todd, S. and Latham, W. (1992). *Evolutionary Art and Computers*. Academic, London, UK.

Todd, S. and Latham, W. (1999). The mutation and growth of art by computers. In Bentley (1999d), chapter 9, pages 221–250.

van Treeck, C., Romberg, R., and Rank, E. (2003). Simulation based on the product model standard ifc. In *Proceedings 8th Int. IBPSA Conference Building Simulation*.

von Buelow, P. (2002). Using evolutionary algorithms to aid designers of architectural structures. In Bentley and Corne (2002), pages 315–336.

Warfield, J. N. (1994). *The science of generic design: managing complexity through systems design*. Iowa State University Press, 2nd edition.

Watkin, D. (1977). *Morality and Architecture*. Clarendon Press, Oxford, UK.

WCED (1990). Our common future (the brundtland report). Technical report, World Commission on Environment and Development, Melbourne.

Whitey, D. and Kauth, J. (1988). GENITOR : A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference On Artificial Intelligence*, pages 118–130, Denver, CO.

Whitley, D. (1989). The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Schaffer, J. D., editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA. Morgan Kaufman.

Whitley, D., editor (1993). *Foundations of Genetic Algorithms 2*, San Mateo, CA. Morgan Kaufmann Publishers.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85.

Whitley, D., Mathias, K., and Fitzhorn, P. (1991). Delta coding: an iterative search strategy for genetic algorithms. In Belew, R. K. and Booker, L. B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 77–84, San Mateo, CA. Morgan Kaufmann.

Wilson, S. W. (1989). The genetic algorithm and simulated evolution. In Langton, C., editor, *Artificial Life: Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems*, volume 6, pages 157–166, Reading, MA. Santa Fe Institute Studies in the Sciences of Complexity, Addison-Wesley.

Wolfram, S. (1983). Cellular automata. *Los Almos Science*, 9:2–21.

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media, Inc.

Wolpert, D. H. and Macready, W. G. (1995). No Free Lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM.

Wolpert, D. H. and Macready, W. G. (1997). No Free Lunch Theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Wright, A. H. (1991). Genetic algorithms for real parameter optimization. In Rawlins, G. J., editor, *Foundations of genetic algorithms*, pages 205–218, San Mateo, CA. Morgan Kaufmann.

Wright, S. (1932). The roles of mutation, inbreeding, crossbreeding and selection in evolution. In Jones, D. F., editor, *Proceedings of the Sixth International Congress on Genetics*, volume 1, pages 356–366.

Zwicky, F. (1967). The morphological approach to discovery, invention, research and construction. In Zwicky, F. and Wilson, A., editors, *New Methods of Thought and Procedure: Contributions to the Symposium on Methodologies*, pages 273–297. Springer, Berlin.

Zwicky, F. (1969). *Discovery, Invention, Research - Through the Morphological Approach*. The Macmillian Company, Toronto.