Evolutionary Developmental Design for Non-Programmers

Patrick Janssen[1], Cihat Basol[2], Kian Wee Chen[3]
[1,2,3]National University of Singapore
[1]patrick@janssen.name, [2]cihatbasol@gmail.com, [3]chenkianwee@gmail.com

Abstract. *Evolutionary developmental design (Evo-Devo-Design) is a design method that combines complex developmental techniques with an evolutionary optimisation techniques. In order to use such methods, the problem specific developmental and evaluation procedures typically need to be define using some kind of textual programming language. This paper reports on an alternative approach, in which designers can use Visual Dataflow Modelling (VDM) instead of textual programming. This research described how Evo-Devo-Design problems can defined using the VDM approach, and how they can subsequently be run using a Distributed Execution Environment (called Dexen) on multiple computers in parallel. A case study is presented, where the Evo-Devo-Design method is used to evolve designs for a house, optimised for daylight, energy consumption, and privacy.*
Keywords. *Evolutionary; developmental; design; performance; optimisation.*

INTRODUCTION

Evolutionary design is 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 evolutionary steps that create, transform, and delete individuals in the population. Each individual represents a design variant, and has a genotype representation and a phenotype expression: the genotype representation encodes information that can be used to create a model of the design, while the phenotype expression is the actual design model. The individuals in the population are evaluated relative to one another, and on the basis of these evaluations, new individuals are created using 'genetic operators' such as crossover and mutation. The process is continued through numerous generations so as to ensure that the population as a whole evolves and adapts.

Evolutionary design differs from other types of evolutionary approaches (such a genetic algorithms) in that it includes a complex developmental step that generates a phenotype by applying the genes in the genotype (Frazer 1995, Bentley and Kumar 1999, Stanley and Miikkulainen 2003, Janssen 2004, Hornby 2005, Kowaliw and Banzhaf 2011). We therefore refer to this as evolutionary developmental design, or Evo-Devo-Design. For designers, the developmental step is crucially important, since it delineates the search space of possible designs. The Evo-Devo-Design method is able to augment the traditional process of design exploration, in which typically only a small number of options will be considered. The advantage of Evo-Devo-Design is that it is able to automatically develop and evaluate large populations of design variants. This method has proved to be well suited to design processes that are typically divergent and exploratory (Janssen 2004).

One of the key drawbacks of such advanced digital design methods has been the need for designers to write and develop their own customised software tools. This has severely limited the general applicability of such methods. This paper describes an alternative approach, whereby designers can apply Evo-Devo-Design methods without having to write any code. The authors have developed a Distributed EXecution ENvironment (Dexen) for population based multi-objective optimisation algorithms. Such algorithms include hill climbing, simulated annealing and evolutionary algorithms [1,7,8]. In this paper, we will focus on using Dexen for Evo-Devo-Design.

Following this introduction, section two gives an overview of the Dexen architecture. Section three focuses on how non-programmers can use Dexen for Evo-Devo-Design. Section four reports on a case-study experiment using Dexen to evolve a house design.

DEXEN SYSTEM ARCHITECTURE

The two main goals of Dexen are speed and flexibility. Speed is an issue since design optimisation problems typically require complex simulations that can be prohibitively slow. Flexibility is an issue since design optimisation problems typically require highly customised evolutionary steps, often requiring the integration of existing simulation programs. In order to achieve these goals, Dexen has been designed with two key features. First, for speed, Dexen is designed to run on multiple computers in parallel. Second, for flexibility, Dexen provides an end-user programming model that allowed users to encapsulate the problem specific aspects within a few key scripts.

Dexen is based on a previous multi-objective evolutionary developmental design environment called EDDE (Janssen 2004, Janssen et al 2005, Janssen 2009). Dexen has been developed with a fundamentally different type of architecture to achieve improvements in both speed and flexibility.

The process of running a population based optimisation problem within Dexen is described as a job. The blueprint for the job is referred to as a job definition or (in the case of design jobs) the design schema (Janssen 2004). The schema defines a set of computational procedures, which are referred to as tasks. When a job is run, the tasks will be executed by Dexen. Each task will act on entities in the population called individuals. An individual represents a complete solution to the problem being optimised.

For design optimisation jobs, the schema will typically include three tasks: development, evaluation, and feedback. Development will generate a model of the design. Evaluation will evaluate some performance criteria of the model. Finally, feedback will use the results from evaluation to generate or modify individuals. If the algorithm being used is an evolutionary algorithm, then feedback will kill some low performance individuals, and generate some new individuals using crossover and mutation.

Dexen has been designed for two levels of user, which we refer to as general users and specialist users. General users are assumed to have the required programming skills to developed their own schemas from scratch. Specialist users may not have the required programming skills, but will instead be able to create schemas by using automated schema generators. Specialist users may typically have advanced knowledge and skills in their domain of interest.

Different schema generators can be created for various areas of specialisation. Each schema generator will target specific software tools. Currently, a schema generator has been developed focusing on architectural design using the Sidefx Houdini software, to be discussed in more detail in section 4.

Dexen components

Dexen consists of four main types of components: one server, and multiple clients, masters and slaves. Each of these components can run on separate machines, thereby allowing the computation to be distributed between multiple machines.

- The server is the core of the system, and all other components connect to the server.
- Each client provides a user interface for an end user to start, stop, and monitor the progress of jobs. When a user starts a job, they need to use the client to upload the schema for that job. This schema will include a set of tasks that need to be executed.
- Each master manages one job, including the population of individuals associated with that job. A user my start multiple jobs, in which case Dexen will create multiple masters.
- Slaves execute the user defined tasks associated with a particular job. Typically, many slaves will be running in parallel. Dexen will automatically assign slaves to masters to execute tasks without requiring any action from the user.

A Dexen population consists of a set of individuals, each of which can become a complete solution to the problem being optimised. Initially, when individuals are first created, they contain only the basic parameters (or genes) for a particular solution. As individuals are processed, they may accumulate additional information, and they thereby change their state.

For example, for an evolutionary schema, an individual's state includes it's genotype, phenotype, and performance scores. The individual starts life with only a time of birth and a genotype. The development task creates a phenotype. One or more evaluation tasks calculate the performance scores. Finally, the feedback task kills some existing individuals, and generate some new individuals (who will only have a genotype).

The Dexen population is therefore a heterogeneous population that contains individuals in different states. For example, some may only have genotypes, some may also have phenotypes, and some may also have performance scores.

A schema must define two types of tasks: one master task and one or more slave tasks. The master task will usually be used to configure various settings and to initialize the population. Initialization typically consists of creating a set of new individuals to start the optimization process. Each slave task will then process individuals from the population.

Each slave task performs a specific user defined procedure, and as a result it requires individuals in a particular state. For example, an evaluation task may need an individual that already has a phenotype, but that does not yet have an evaluation score. Individuals that do not meet these criteria need to be rejected. A filtering process therefore has to take place in order to discover which individuals in the population match which slave task. In order to do this, each slave task is assigned a user define boolean function, referred to as the filter function. This is used to decide if a particular individual is valid for processing by that task.

EVO-DEVO-DESIGN FOR NON-PROGRAMMERS

For the general user, writing a schema involves defining the tasks that will be executed by Dexen. The user needs to define one master task, and one or more slave tasks. The programming model that has been defined for these tasks is both simple and powerful. The schema has to be written in Python, and a basic understanding of object-orientated programming is required.

However, for users that are non-programmers, writing a schema may be difficult. Such users may be architects and engineers who are experts in their own field, but who may not have the required programming skills needed to write the schema code. For such users, schema generators can be used in order to automate the process of creating schemas. Schema generators are implemented as part of the client and run on the user's local computer.

Schema generators target specific software applications. The user will be required to define the problem specific aspects of their schemas in some format that will not require them to write code. For example, in a design scenario, the user would be required to define the design development and one or more design evaluation procedures. The schema generator can then be used to generate all the necessary Python code to wrap these core procedures.

In order to define the core procedures, a designer could use the Visual Dataflow Modelling approach (VDM). VDM allows users to program by visually linking together graphical nodes with wires. The nodes and wires are arranged by users to create complex networks through which data can flow. Each node represents a function, and the wires represent the data inputs and outputs for the function (Woodbury 2010, Janssen and Chen 2011).

The Houdini schema generator

In order to demonstrate this approach, a schema generator has been developed for a 3D CAD and animation software, called SideFX Houdini.

For development, a Houdini network that generates a phenotype from a set of genes is required. The phenotype will be some kind of model of the design variant. For evaluation, the Houdini network that generates an evaluation score from a phenotype is required. A simulation program may be used in order to perform the evaluation. If more than one criteria needs to be evaluated, then multiple networks can be created.

The Houdini schema generator provides a set of Houdini nodes that the development and evaluation network must use. These nodes are used to define the start and end points of each network, and the user can then create any type of network between these two points. The Python code generated by the schema generator will assume that these special nodes are present and will read and write data from these nodes. For development, a genotype and a phenotype node is provided. For evaluation, a phenotype and an evaluation score node is provided.

The user also needs to set some basic parameters in a settings file for the schema generator. The parameters that can be set include the following:

- The optimisation algorithm to be used. Options include hill climbing, simulated annealing, or evolutionary algorithm.
- The population size, the maximum number of births, and the input sizes for all tasks, including feedback.
- The settings for the feedback task, including the ranking and selection algorithms to use for the birth and death of individuals.
- The names of the Houdini files in which the development and evaluation networks are defined.
- The structure of the genotype, including the types of genes. (For example, genes can be integers, floats, or strings.) The length of the genotype is assumed to remain constant.

In order to generate the schema, the user places the settings file and the Houdini files in a single folder, and then uses the client to execute the schema generator script. This will result in the Python files being automatically generated for the schema, and being placed in the same folder.

The generator will create the Python code for the master task, and each of the slave tasks. For the development and evaluation tasks, Python wrappers will be generated for the Houdini files. For the feedback task, a simple feedback procedure will be generated. In this procedure, the individuals received by the feedback task will be ranked, the low performance individuals will be killed, and the high performance individuals will be used as parents for breeding new individuals.

The user may then upload this schema to the server to start running the job.

A CASE STUDY

In the case study, a Houdini schema was developed for a free-standing house in a residential setting. Three performance criteria were defined: minimization of energy consumption, maximization of daylight, and maximization of privacy. A number of Houdini files were created, and the Houdini schema generator was then used to automatically generate the Python code for the schema.

Slave tasks
In total, four Houdini files were created, one for each slave task: the development task, the energy evaluation task, the daylight evaluation task, and the privacy evaluation task. Each Houdini file contains a network of nodes that define a problem specific procedure to be executed by Dexen.

The Houdini development network maps the genotype to the phenotype. The network starts with a Dexen genotype node and ends with a Dexen phenotype node.

The genotype in this case consists of 55 real valued genes, each in the range 0.0 to 1.0. The phenotype is a three dimensional model of the house, saved in the Houdini format. The model is shown in Fig. 1.

The house is spread over three floors, and has a living room, dining room, a kitchen, and four bedrooms. A stair-core gives access to all three floors. The living room, dining room, and kitchen are always located on the ground floor. In addition, one of these spaces on the ground floor will be a double height space. The bedrooms are all located on the upper floors. Service spaces such as bathrooms and store rooms are not included. A typical (randomly generated) house is shown in Fig 2, and the genotype for this example is show in Fig 1.

Conceptually, the developmental process can be thought of as a process that transforms an initial simple model into a final complex model. The initial model consists of 12 equal spaces. On each floor, four spaces are packed together around a centre point so that they meet in the middle. The model of the house (i.e. the phenotype) is generated as follows:

- The programmes are assigned to the spaces using 9 genes. The programmes are subject to various constraints. For example, for the ground floor level, the stair-core and living room must be adjacent to one another, and the living room and dining room must be adjacent to one another. Each programme also has a required area.
- The size of each space is defined using 9 genes. Since the area is already known, the genes only need to assign the proportion of the spaces. The three stair-core spaces are also constrained to all have the same size, so that the stack on top of each other.
- The windows are inserted using 12 genes. Each space can have a window in either of its two outward facing walls. The two possible window types are strip window or fully glazed. Each room must have at least one window, but cannot have two fully glazed windows.
- The sun shades are defined using 24 genes. Sun shades are only added to walls that have windows. The genes control the depth of each of the sun shades.
- The orientation of the building is defined using one gene. The building is first placed in the centre of the site, orientated so that the stair-core is facing the road. The gene is then used to rotate the building by a certain amount.

In many cases, the genes are mapped to some other values. For example, the sun shade genes are mapped to a dimension from 0 to 2 meters, and the orientation gene is mapped to an angle from -45 to 45 degrees. In some cases, the genes can also be mapped to a set of discrete variables. For example, for the window genes, each gene is mapped to one of the seven possible choices of window pair choices.

On level 2, a situation can arise where one of the bedrooms is diagonally opposite the stair-core. In such a case, the bedroom would become inaccessible. As a result, if this situation arises, then the

Fig 1. An example of a house generated using the Houdini developmental network.

0.91, 0.61, 0.65, 0.47, 0.24, 0.87, 0.36, 0.30, 0.93, 0.09, 0.29, 0.10, 0.31, 0.45,
0.45, 0.47, 0.44, 0.89, 0.97, 0.25, 0.94, 0.10, 0.15, 0.30, 0.58, 0.54, 0.08, 0.68,
0.20, 0.20, 0.75, 0.91, 0.84, 0.92, 0.41, 0.83, 0.02, 0.20, 0.50, 0.18, 0.34, 0.26,
0.51, 0.84, 0.70, 0.01, 0.22, 0.45, 0.04, 0.11, 0.50, 0.36, 0.27, 0.58, 0.45

spaces are offset in order to create an adjacency between the stair-core and the diagonal bedroom. This situation can be seen in the example shown in Fig. 1.

After the main geometry of the house has been generated, all dimensions are then snapped to a constructional grid. In this case, this grid was set to 0.3 meters. This final step ensures that there are no awkward dimensions. This also means that the area of the rooms will not exactly match the required areas for each programme. However, since the deviation is small, this is seen as being acceptable.

Each Houdini evaluation network uses the phenotype (generated by the development task) to calculate an evaluation score. Each network starts with a Dexen phenotype node, and ends with a Dexen evaluation score node. In addition, custom nodes have been developed to actually perform each type of evaluation. These custom evaluation nodes provide the user with a simple method of running the required simulations. The custom nodes can be inserted into the Houdini network to perform the simulation. The input into the custom node will be the Houdini geometry, and the output will be the simulation results. The custom nodes will also have a set of simulation parameters that can be set by the user.

For energy and daylighting evaluation, the EnergyPlus and Radiance simulation programs are used respectively. The custom nodes will read the Houdini geometry, generate the text-based input file, execute the simulation program, read the text-based results file, and finally import the results back into Houdini. The EnergyPlus node calculates the energy required to keep the house within a certain temperature range using an ideal load air system. The Radiance node calculates the percentage of floor area that has a daylight level of higher than 300 Lux for a standard overcast sky condition. At an early design stage, these are seen as good indicators of the relative performance of the design with respect to energy consumption and daylighting.

For privacy, a custom node is used that calculates the privacy level of each window based on the relative position and orientation of other windows of neighbouring houses. (See Fig. 2.) A value of 100% indicates total privacy, while 0% indicates no privacy. This calculation is performed inside Houdini, so in this case, no external simulation program is required.

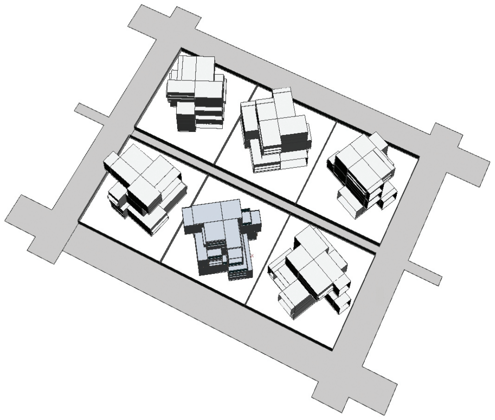
Fig 2. The house on the site, surrounded by five other houses.

RESULTS

The schema generator settings file was used to set the key parameters for the job. The optimisation algorithm was set to use an evolutionary algorithm, and the ranking algorithm was set to use Pareto ranking. The population size was set to 100, and the maximum number of births was set to 10,000. The input sizes for all tasks was set to 1, except for feedback, for which the input size was set to 20.

The job was executed on a cluster of 20 standard desktop PCs and was run overnight. The job took approximately 7 hours to complete.

The Pareto graphs for the results are shown in Fig. 3. Since there are three performance criteria, two Pareto graphs are shown, one plotting energy against privacy and another potting energy against daylight. The Pareto front is plotted on both of these two graphs.

The Pareto graphs show how the number of individuals generated by the evolutionary process gets more dense closer to the Pareto front. The individuals in the initial population were mostly far away from the Pareto front. Through inheritance of favourable genes, the population as a whole gradually evolved, with individuals in the population gradually becoming optimised for the selected performance criteria.

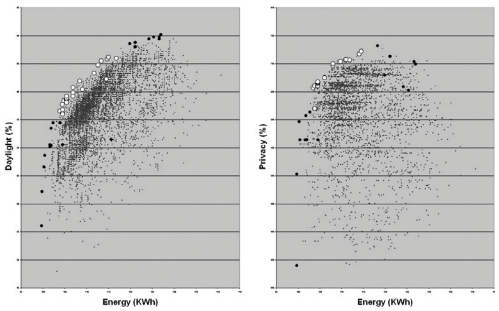

Fig 3. Two Pareto graphs plotting energy against privacy and energy against daylight. Each point represents a design. Larger black and white circles represent designs on the Pareto front. White circles represent designs where energy < 115 KWh, daylight > 75%, and privacy > 65%.

In total, there are 52 individuals on the Pareto front. Of these individuals, most were born at the end of the evolutionary process. (Out of the 52 individuals on the Pareto front, 40 were born during the last 1000 births. However, individual 13 actually turned out to be one of the best, and survived all the way until the end.) These individuals represent different trade-offs between energy, daylight, and privacy. From this Pareto optimal set, individuals that had an energy score of less than 115 KWh, or a daylighting score of less than 75%, or a privacy score of less than 65% are eliminated. This then leaves 25 individuals, from which the designer can select a preferred design. One of the best individuals is shown in Fig 4.

CONCLUSIONS

Initial experiment using Dexen have shown that the use of schema generators lowers the threshold for non-programmers to start using advanced optimisation techniques. In fact, it is now possible to run complex optimisation algorithms using only graphical CAD tools.

Dexen also achieves its two main goals of speed and flexibility. In terms of speed, the distributed master-slave architecture means that Dexen can easily be deployed on compute clusters, and as a result, large and complex optimisation jobs that would otherwise take days to run can now be completed overnight. In terms of flexibility, the Dexen allows a wide variety of optimisation problems to be defined.

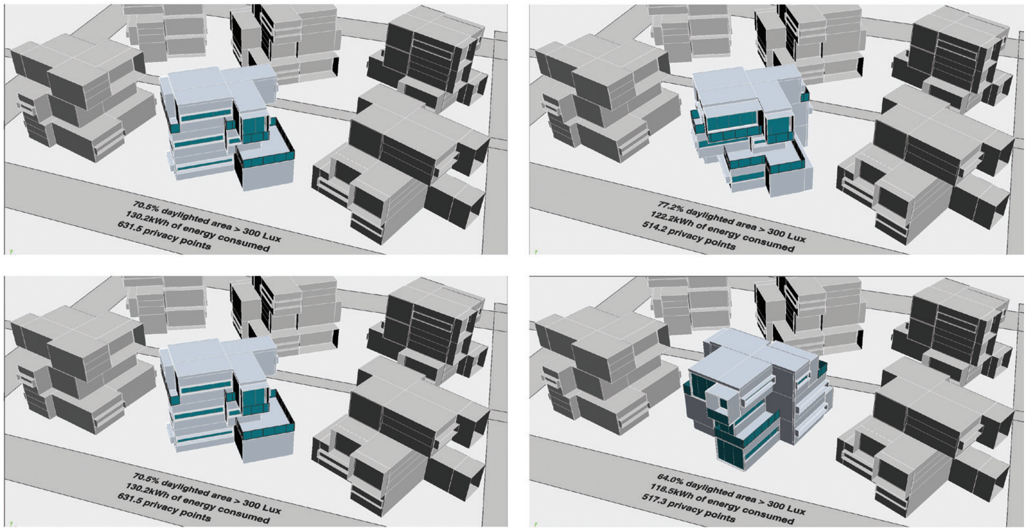
70.5% daylighted area > 300 Lux
130.2kWh of energy consumed
631.5 privacy points
77.2% daylighted area > 300 Lux
122.2kWh of energy consumed
514.2 privacy points
70.5% daylighted area > 300 Lux
130.2kWh of energy consumed
631.5 privacy points
64.0% daylighted area > 300 Lux
118.5kWh of energy consumed
517.3 privacy points

Fig 4. Four examples of evolved design variants for a suburban detached house. The examples shown were randomly selected from a population of 100 individuals, after approximately 8000 births.

REFERENCES

Bentley, P and Kumar, S 1999, Three ways to grow designs: A comparison of embryogenies of an evolutionary design problem, in *Proceedings of the 1999 conference on Genetic and evolutionary computation*, Morgan Kaufmann, pp 35–43.

Caldas, L 2001, *An Evolution-Based Generative Design System: Using Adaptation to Shape Architectural Form, Doctoral dissertation,* Massachusetts Institute of Technology.

Frazer, J. H. (1995) *An Evolutionary Architecture*, AA Publications, London, UK.

Hornby, G 2005, Measuring, enabling and comparing modularity,regularity and hierarchy in evolutionary design, in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, vol 2, pp 1729-1736.

Janssen, PHT 2004, *A design method and a computational architecture for generating and evolving building designs*. Doctoral dissertation, School of Design Hong Kong Polytechnic University (October 2004).

Janssen, PHT, Frazer, JH, and Tang, MX 2005, Generative Evolutionary Design: A Framework for Generating and Evolving Three-Dimensional Building Models, in *Proceedings of the 3rd International Conference on Innovation in Architecture, Engineering and Construction* (AEC 2005), pp 35-45.

Janssen, PHT 2009, An evolutionary system for design exploration, in *Proceedings of the CAAD Futures '09*, pp. 259-272.

Janssen, PHT and Chen, KW 2011, Visual Dataflow Modelling: A Comparison of Three Systems, in *Proceedings of the CAAD Futures '11*, (to be published).

Janssen, PHT, Chen, KW, and Basol, C 2011, Iterative Virtual Prototyping: Linking Houdini with Radiance and EnergyPlus, in *Proceedings of the CAAD Futures '11*, (to be published).

Kowaliw, T and Banzhaf, W 2011, Mechanisms for Complex Systems Engineering through Artificial Development in Morphogenetic Engineering: Toward Programmable Complex Systems, to appear in *Morphogenetic Engineering: Toward Programmable Complex Systems*, Springer-Verlag.

Mitchell, M 1996, A*n introduction in Genetic Algorithms,* MIT Press, Massachusetts Institure of Technology, 1996.

Shea, K 1997, *Essays of Discrete Structures: Purposeful Design of Grammatical Structures* by Directed Stochastic Search, Doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA.

Stanley KO and Miikkulainen, R 2003, A taxonomy for artificial embryogeny, in *Artificial Life*, 9(2):93–130.

Woodbury, R 2010, *Elements of Parametric Design*, Routledge, NY.