Programming Languages for the Future of Design Computation

Robert Aish
University College London
United Kingdom

Al Fisher
Buro Happold
United Kingdom

Dominic Orchard
University of Kent and University of Cambridge
United Kingdom

Jay Torry
University of Cambridge
United Kingdom

Abstract

Design Computation is the use of programming in the design of physical systems such as buildings and infrastructure. This involves embedding both general-purpose textual languages and domain-specific visual languages within geometry modelling and engineering applications in the construction industry. A unique form of entry-level end-user programming has emerged in Design Computation. However, there are significant usability and representational issues; general-purpose languages present barriers to adoption, whilst visual languages do not scale to complex design problems.

In this essay, we explore how advances in programming language research could be harnessed in future Design Computation languages to address these pedagogic, representational and scaling issues so as to improve human-readable program structure and semantics and to enable machine-readable program verification.

CCS Concepts: • **Software and its engineering → Software notations and tools**; *Software organization and properties*; *Integrated and visual development environments.*

Keywords: Design Computation, Programming Languages, Visual Languages, End-User Programming, Cognitive Dimensions, Usability, Collaborative Coding, Type Systems, Units of Measure, Collection Types, Program Verification.

ACM Reference Format:
Robert Aish, Al Fisher, Dominic Orchard, and Jay Torry. 2024. Programming Languages for the Future of Design Computation. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/3689492.3689812

Onward! '24, October 23–25, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1215-9/24/10
https://doi.org/10.1145/3689492.3689812

1 Introduction

Design Computation (DC) combines advanced numerical simulation and interactive geometric modelling to help architects and engineers imagine and realise the *form* and *performance* of buildings.[1]

An example of Design Computation in action is the dome of the Louvre Abu Dhabi museum (Figure 1) [32]. The size, complexity and digital fabrication of this project could only be achieved through highly computational and automated methodologies which were required to facilitate both the design and engineering processes and to generate the design communication output including drawings and Building Information Models (BIMs) [15].

Broadly, DC embeds programming within the architecture, engineering and construction (AEC) industry. However, a distinction can be made between Design Computation and conventional AEC modelling techniques that create static geometric models. Static models represent the outcome of particular design decisions, but not the underlying design logic and temporal decision sequence. In contrast, DC directly represents the design rationale as a program whose execution creates not just a single output (or BIM model) but encourages the exploration of numerous design options. This enables more efficient and performative buildings to be designed beyond the reach of conventional manual, labour-intensive design modelling.

DC can also drive solution space generation and optimisation [3], digital fabrication, off-site construction and robotic assembly [23]. Yet DC is more than a productivity tool: It changes the way designers think, echoing Perlis on the constructive influence of programming languages:

> *"A programming language that doesn't change the way you think is not worth learning"*
> —Alan Perlis, Epigrams in Programming [58].

The users of DC, termed *computational designers*, are a new hybrid of highly-skilled professional architects or engineers who are also end-user programmers, i.e., who write software not as the primary goal of their job but to support its main

[1] It could be argued that Design Computation combines the three types of programming suggested by the authors of the Poplog language: *numbery, bumpy* and *thinky* [66].

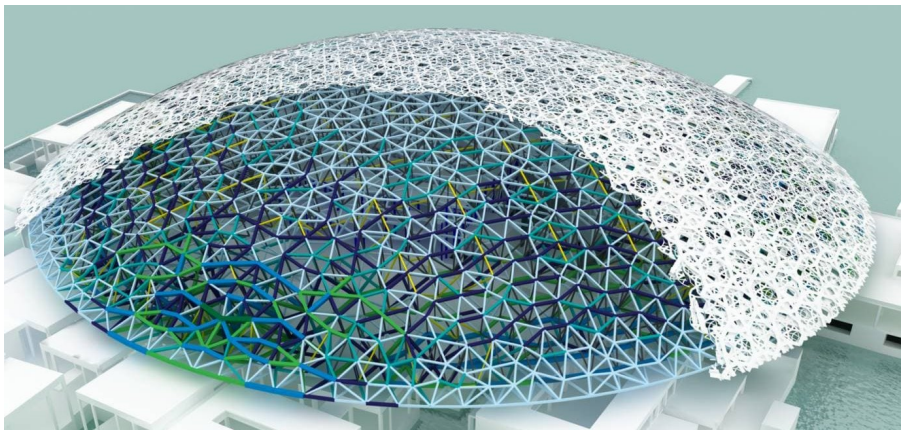

Figure 1. Louvre Abu Dhabi: The 180m diameter intricately latticed roof is designed to balance performance across aesthetic, lighting, environmental, structural, museography and construction requirements. Mediating the local microclimate in the spaces below, through 1.8 percent "rain of light" transmission, the roof is composed of 7,850 individually sized stars supported on a steel space-frame constructing of 10,000 optimised elements [32, 65].

objectives of designing the built environment [39, 50].[2] These users are expected to be immediately productive whilst still learning to program. This contrast, between their domain expertise and their lack of formal training in programming, suggests the need for research into programming languages, tools, and systems to support the field.

Originally, DC used conventional languages, such as C++, as standalone applications to generate inputs files or invoke the APIs of BIM and CAD applications. For example, a C++ implementation of first principle analytical and numerical modelling techniques was used to compute the geometrical configuration of the British Museum Great Court roof [72].

However, the high-level skills required by these languages impeded the mainstream adoption of this approach in DC. Visual Data-Flow Programming (VDFP) removed this bar to DC [7] and is now universally adopted by architectural practitioners and university schools of architecture. Specialised DC languages typically provide a hybrid programming system combining visual and textual programming languages: For example, Dynamo (visual) with DesignScript (textual) [6]; and Grasshopper (visual) with C# (textual) [68].

Although DC is used in the design of some of the most carefully articulated, precision-engineered and prestigious architecture, VDFP does not encourage equivalent rigour in software design: (1) It does not employ any static checking; (2) it facilitates a culture of "copy-and-paste" programming; (3) it has usability issues [2] and (4) it interacts poorly with modern software engineering methods. The widespread adoption of VDFP presents a challenge to both researchers and construction industry managers. It is so ingrained in architectural practice that it is unlikely to be displaced; however, as we shall argue, visual programming can play an important role as part of a wider language system.

[2]We could also use the terminology 'vernacular software developers' [64].

The DC user community may be of particular interest to the programming language research community: It combines various different ways of thinking as potential candidates for programming, including constructs and types of reasoning which may be difficult to represent with existing programming languages. All these challenges combine to create interesting opportunities for programming language innovation. Thus, in this essay, we seek to reach across from Design Computation to the Programming Languages community. We offer an insight into this interesting computational domain, outline its development methodologies and concerns, highlight some challenges faced, and propose avenues for how ideas from the PL literature could be leveraged to advance the start-of-the-art in programming for DC.

The ideas presented here are the result of an ongoing collaboration between members of the Design Computation community and the Programming Language research community. The essay is in three parts.

This first part provides background and context: Section 2 considers the wider organisational and social aspects of programming for DC before Section 3 walks the reader through a case study of a particular example (the computation generation a faceted panel) which gives a taste of visual DC programming. Section 4 sets the scene for computation as a critical aspect of architectural and engineering innovation.

The second part provides our vision of a way forward for DC. Section 5 describes our proposal of a *programming system* comprising a progression of languages to aid the learning of DC programmers. Section 6 focuses on how ideas from PL could be leveraged to better capture domain aspects in the proposed language system. Section 7 returns to the case study of Section 3, imagining how our ideas may be embodied. Section 8 considers how we might be able to transition the existing state-of-the-art towards our new vision.

The third part reflects and discusses. Section 9 presents key usability issues in the deployment of DC languages. Section 10 reflects on the way computation has changed architectural and design thinking. Section 11 then concludes.

2 Context: Organisational and Social Aspects of Design Computation

Design Computation is characterised by three overlapping and complementary dimensions: (1) An educational progression from entry-level to proficient programmer; (2) a design progression from initial sketch to the complete definition of a building to be constructed; (3) a hierarchy of social structures from individual, to team, to organisation. These three dimensions help us understand how existing DC languages are used and how programming language research might help to address many of the current challenges, suggesting ways to improve future DC languages.

Individual Development. For the individual, Design Computation programming is often exploratory and used as part of a corresponding exploratory design process. This involves quickly creating a number of minimum geometric sketch models with which to visualise and evaluate alternative design options: an example of computation supporting a creative and subjective way of thinking.

Individual DC programmers typically start by using domain-specific visual programming languages. There are two principal reasons for this choice. Firstly, languages of this type are built into many of the geometric modelling applications in use within the construction industry. Secondly, visual languages appear to lower the barrier to entry-level programming with the additional potential to directly and intuitively represent high-level descriptions of design and fabrication processes and workflows.

However, these visual programming languages (VPLs) only provide access to a subset of programming concepts and thus do not scale to more complex programming tasks. In addition, numerous usability challenges have been reported with large scale visual programs. Thus, there is an important potential role for future Design Computation languages to help users progress beyond existing limitations and become proficient in more powerful general-purpose languages.

Team Development. As the AEC design and engineering process progresses, there is a gradual change in emphasis from individual to collaborative working and from exploratory to more rigorous design with computation supporting a more analytical and objective way of thinking.

At this collaborative team level, Design Computation languages play a key role as the common platform giving access to a number of critical evaluation and simulation application plugins, for example for structural analysis, energy performance, and environmental impact. The role of the end-user

Levels of Computational Competency

1. ***Entry level flow-based programming*:**
   Ability to use and run scripts authored by others;
   ***progressing to** ability to create a simple visual script which captures a design or engineering workflow.*
2. **Advanced flow-based programming:**
   Ability to create more complex visual programs where the logic and data are properly structured;
   ***progressing to** co-authoring workflows collaboratively across disciplines.*
3. **Entry level text-based programming:**
   Appreciate the need to move beyond visual programming and be able to write macros and short scripts;
   ***progressing to** reuse of functions as part of larger workflows and ability to extend existing applications through extensive scripted processes.*
4. **Advanced level text-based programming:**
   Appreciate the need to move beyond individual programming and embrace collaborative software engineering through use of IDEs, managed code dependencies, debugging, documentation
   ***progressing to** using version control and contributing to centralised code repositories.*

Figure 2. Excerpt from Buro Happold's Computational Competency framework illustrating the progressive levels of capability, engagement and participation required in design computation for scalable and sustainable adoption.

programs is to represent buildings in terms of different analytical models for these processes. The DC program effectively captures the decision logic for a project and in many ways becomes the design and engineering IP.

Organisational Development and Beyond. At an organisational level, a key aspect of the overall computational strategy involves collecting the project-specific programs from different design teams within the organisation. These can then refactored into more general, interoperable and reusable components which can be subsequently deployed across the organisation.

This co-creation and re-use of Design Computation programs is a logical extension of the intensive collaboration already found within AEC organisations. These organisations are characterised by large, distributed and diverse teams, ranging across a wide range of expert skills and thus embracing a wide range of computational proficiencies (Figure 2).

Design Computation programs encode critical design thinking; therefore, these programs, and the languages in which they are written, have an important *explanatory* role to fulfill (particularly where engineering experience/responsibility and computational skills are not sufficiently aligned). This explanatory role can be extended to public engagement, raising awareness about alternative design options, different performance trade-offs and consequential impact.

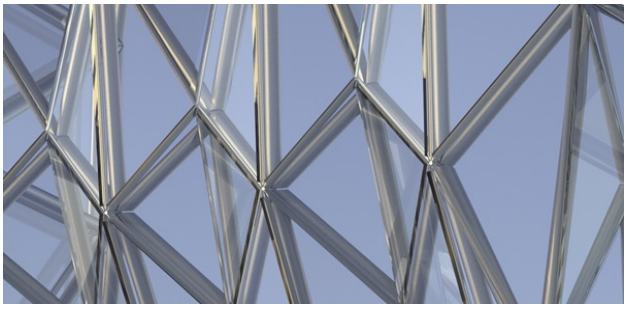

Figure 3. Example output rendering of a faceted façade [23].

More generally, at a policy level, construction represents a substantial proportion of global GDP (on average, ~12% of GDP across most of the world's economies; 2005 data [11]).

When combining emissions from the construction industry with operational emissions: Buildings represent around 37% of total global operational energy and process-related CO_2 emissions as of 2021 [70]. Whilst Integrated Assessment Models can estimate and predict the environmental effects of construction; improving the environmental performance of construction only occurs 'project by project', and therefore 'program by program', alongside broader sector and policy changes. Improved languages could make it easier to develop DC programs, arguably making it easier to develop more performative buildings with reduced environmental impact. There may be additional global challenges: How to support the developing world's justified aspiration for better infrastructure while addressing the wider environmental and climate impact of construction.

3 Case Study: Design Computation in Practice Today

To contextualise development practice in Design Computation, we illustrate the use of the Dynamo visual programming language [10] within Design Computation[3] for a real-world façade engineering project (Figure 3 [23]) with its minimal re-interpretation as a Dynamo program in Figure 5.

In this example in Figure 4, the polygon provides a lightweight geometric *placeholder* for the façade panel. The intent of this example visual program is to create the solid geometry for the physical façade panel. The defining dimensions for the material form are: (a) an inset from the edge of the polygon, then (b) an offset orthogonal to the plane of the polygon and finally (c) a defined thickness to create the geometric solid to represent the façade panel (Figure 4). The façade might be visualised as a collection of polygons, but polygons are abstract geometry with zero thickness. To physically construct the façade, such abstract geometry has to be *materialised* into a physically constructable form.

[3]Visual languages used in Design Computation are predominantly based on data-flow programming, à la Kahn [36], visualised as a DAG, rather than a visual UI for imperative programming, for example, as in Scratch [41].

There are many ways an experienced architect/engineer/programmer might approach this problem. To illustrate the programming method and usability issues, the programming and geometric operations used in this example have been deliberately limited to those which an entry-level end user would have had access to in a typical visual programming application. In reality the frame supporting the panel would also be modelled, but this is not considered for simplicity.

The visual program (Figure 5) comprises four input nodes and five process and output nodes. The user did not have to learn complex syntax; the program was composed using simple drag-and-drop interactions. The input *point data* is elided as it is externally defined and imported into the program.

Visual programming applications are based on the idea of data flow and are designed with *liveness* in mind [67]. Any change to the program data or logic automatically triggers recompilation and re-execution. Thus the user can change the value of the inputs with sliders and these changes are propagated through the program and the geometric output is updated. Alternatively, the user can change the nodes or change the connections between the nodes. With end-user programming the distinction between using and programming is blurred and becomes a single exploratory process. A typical user reaction is: "*This is easy. I can create with this.*"

Visual programming is attractive to users from a design and engineering background who are already familiar with other forms of visual expression and communication. It has enabled a form of programming to be learnt by entry-level programmers where regular text-based programming might have been a substantial barrier to learning to program.

It can be observed from this example that visual programs do not require the nodes to be given unique names. This is not required because in a visual data flow language connections between variables are implemented as arcs between nodes, rather than the referencing of named variables (as in textual languages). This may initially simplify the tasks of the entry-level programmer. However, this allows visual programs to be constructed with multiple unnamed nodes of the same type (such as the Number Sliders in Figure 5) and consequently with no clear semantics.

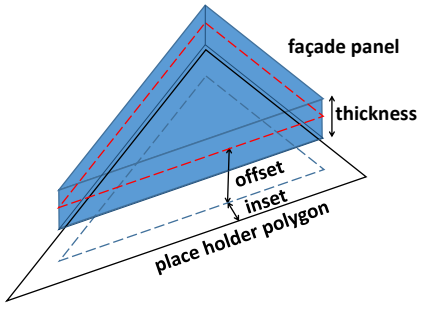

Figure 4. Inset, Offset and Thickness dimensions used in the example façade panel visual program

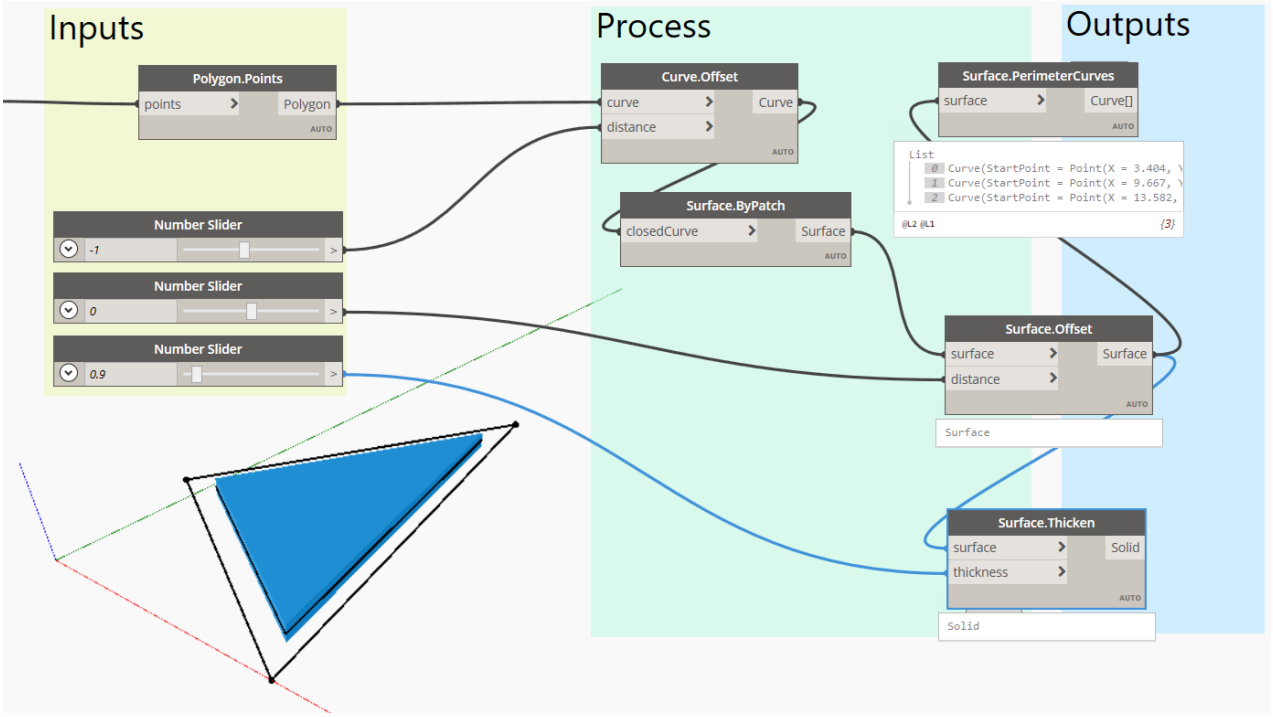

Figure 5. A typical Visual Program, with nodes representing inputs, processing operations and outputs. Notice that there is no obligation to name specific nodes: The user has not bothered to record which specific dimensions each slider controls.

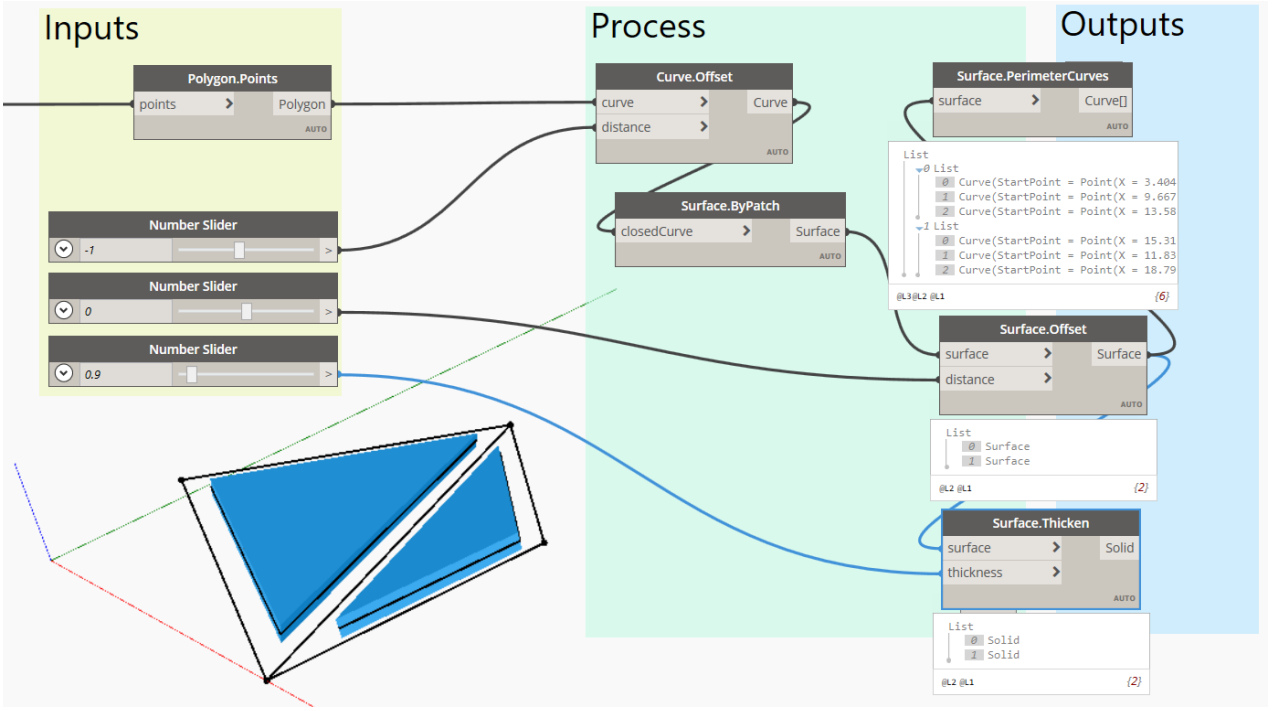

Figure 6. Here the single input value (a polygon) [in Figure 5] has been changed into a collection (of polygons, which is propagated to all dependent nodes, which now become collections).

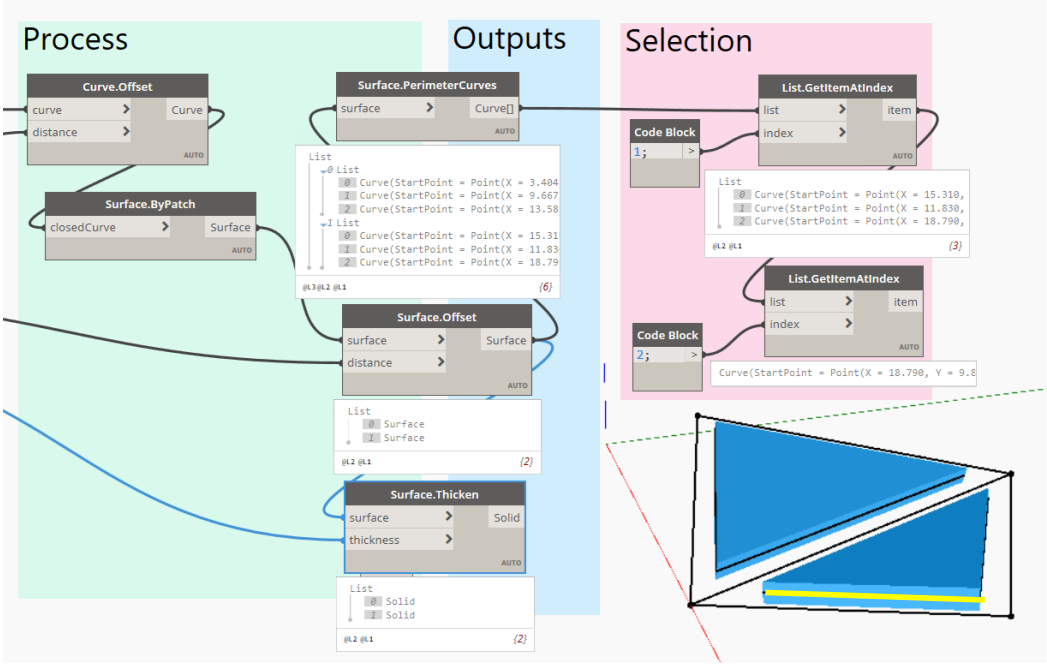

Figure 7. Selecting a member of a 2D collection with visual programming requires multiple nodes. (in this case a single edge of one of the façade panels, highlighted in yellow).

Having proved the design method works with a single polygon (Figure 5), the programmer may now continue to explore how this method might be applied to the complete façade, here represented by a collection of polygons (of which only two are shown) (Figure 6).

Notice that the nodes in the visual languages are overloaded for collections, thus a change in the dimensionality of an input (from a single polygon to a collection) is propagated through the visual program, with the result that the output of each of the different nodes becomes a collection.

As the name suggests, Visual Data Flow languages encourage a very data centric way of programming. In this case, the output nodes in Figure 6 are: a collection of perimeter curves, a separate collection of offset surfaces, and another collection of 'thickened' solids.

In this example, the two-dimensional collection of perimeter curves is implemented as a 'list of lists'. Consequently, to re-assemble the parts of a specific façade panel, the user must use numeric indices to index into the separate collections.

If the user is restricted to only using the nodes provided by the programming environment then some operations, such as the selection from a multi-dimensional collection, often requires quite complex visual programming involving multiple nodes (Figure 7). We can observe that while the geometric result may resemble a collection of façade panels, the way the information is structured, labelled and accessed (as homogeneous collections, as lists of lists) is not how the user might want to think about the façade panel.

Ideally, the façade panel should be defined as a user-defined schema or class which could directly represent a domain specific *chunk* of information, structured as a semantically indexed heterogeneous collection of different types of named subparts and properties (such as *perimeterCurve, offsetSurface* and *Solid*). Although this is approach is possible, it is difficult for the entry-level programmer to discover: they are left using a large number of simple operations (nodes) and effectively unable to express the underlying logical structure.

The reaction of the users is likely to change from:

This is easy. I can create with this.

to:

This is getting awkward, even with just a small model.
Multiple nodes are needed for what should be simple tasks.
This seems unlikely to scale to really complex projects.

We can imagine the user asking: "*So, what to do?*"

3.1 Limitations of Visual Programming

This example can help us understand some of the factors which limit visual programming.

Many visual programming systems implement the ideas suggested by Myers [49] for simplifying programming languages to make initial learning easier. Two aspects of this simplification are: To minimise the need for control structures; and to avoid requiring nodes (i.e. variables) to be identifiable (or named). However, Burnett et al. [16] and other researchers have noted that, generally, visual programming systems do not scale to more complex programming tasks.

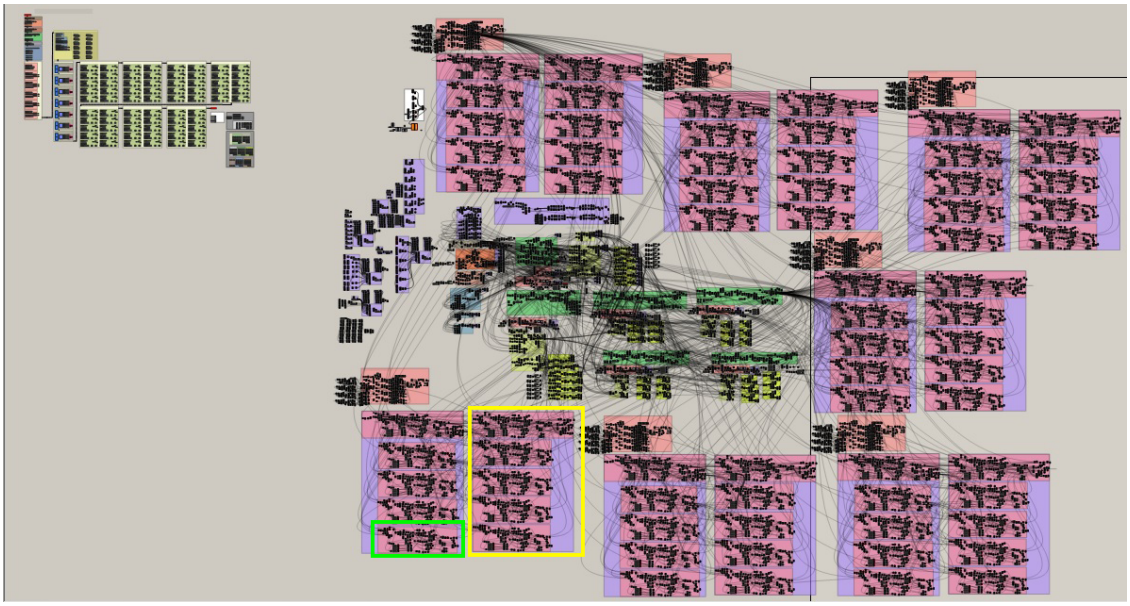

Figure 8. A complex visual program from Al-Jokhadar [8, p.766], composed in the Grasshopper visual language [68]. This program is a direct analogue of the organisation of the repeated nested spatial and construction components of a high-rise building. The green and yellow boxes are added to the original illustration to aid explanation. To reflect the organisation of the building, a defining sub-graph (outlined in green) is copied four times with an additional variant to form the intermediate sub-graph (outlined in yellow) which, in turn, is copied fourteen times, giving seventy copies of the original (green) sub-graph. While the idea of a visual program as an analogue of the hierarchical structure is appropriate, achieving this via copy-and-paste of the original sub-graph (rather than via some form of modularisation) is problematic from a software engineering perspective.

Having learnt visual programming, but as yet no other languages, many users have few options but to persist with visual languages making even more complex programs as exemplified in Figure 8.

The visual language has given the user the *freedom* to distribute the information in the program between multiple copies of a single subgraph and the unique connections between those copies. We can imagine the dilemma facing the user who subsequently wants to change the original subgraph and apply this change across the whole program: Either change the subgraph and recopy it seventy times (which may destroy the unique connections); or repetitively edit each of the seventy copies. Perhaps we are at the limits of such language *freedoms* if they allow the end-user progamer to adopt an unsustainable form of programming.

We are reminded of George Orwell's rather uncomfortable criticism that "the slovenliness of our language makes it easier for us to have foolish thoughts" [57].

Figure 8 does not illustrate a performance issue. On the contrary, most visual languages are more than capable of computing and displaying thousands of nodes and arcs. Rather, the type of visual program shown above represents a usability issue. It illustrates the influence of visual languages on the computational behaviour of entry-level programmers [35].

In the extreme case reported by Lim [40], increasing the complexity of a visual program can continue until the point is reached where it becomes so complex that the user no longer understands their own program (and is afraid to change it for fear of breaking the program and being unable to recover). The example in Figure 8, and the reports by Burnett and Lim, should make us *pause for thought* about the usability limits of visual programming when the normal, as intended, use of a language becomes self-inhibiting.

The users' behaviour in continuing to add to the visual program (even though these additions are of diminishing value) may be explained in terms of computational behaviour [35]. From this perspective, the relatively low incremental cost of such additions can be compared to the comparatively higher cost and uncertainty of learning a more capable textual language. Additionally, every increase in size and complexity of the visual program raises the cost of conversion. This is compounded by non-standard functionality and terminology in many visual languages [4] further inhibiting this change.

While visual programming may lower the entry barrier to programming, when it is combined with the copy-and-paste method it becomes incredibly easy for the user to create a visual program on a scale where it is practically impossible to perform many of the expected programming tasks such as comprehension, editing, and re-use. In lowering the barrier to adoption, visual languages raise the barrier to *migration*.

In summary: 'The path of least resistance is a dead-end' [1]. *So, what to do?*

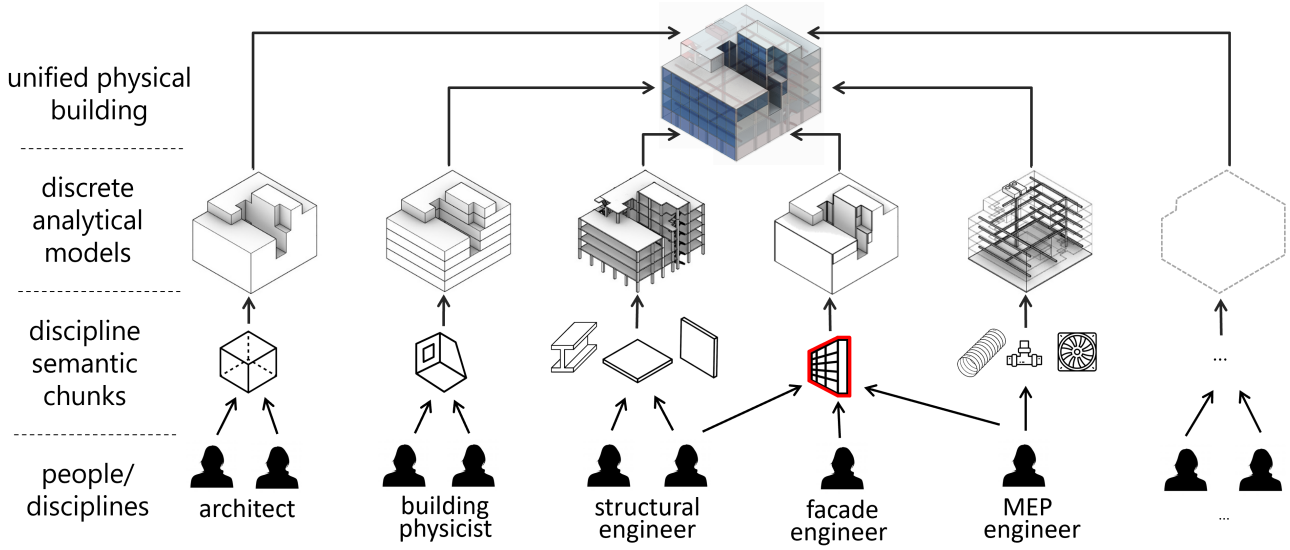

Figure 9. A building is a single coordinated collection of physical components. However, to facilitate its design and engineering it is conceptualised as a number of discrete analytical models, each under the responsibility of different engineering disciplines. Therefore to create a unified and overall performative building requires considerable inter-disciplinary collaboration and negotiation. There are occasions when this collaboration is focused on a specific object which plays a pivotal role in the overall building system. An example (highlighted in red above) is a façade wall panel. This is a single physical entity which must respond to a number of different performative criteria and thus may have a number of different disciple-specific representations. The challenge discussed here is how to represent this type of multi-functional, multi-performative component in future Design Computation languages (and in doing so, to support this type of cross-disciplinary collaboration). The wall example will be featured in subsequent sections as this theme is further developed.

4 Programming Support for Building Systems Integration

A motivation for this language research comes from observing specific changes in the objectives of the construction organisations (to build more performative buildings) and the response within design and engineering teams (to innovate by creating new forms of Building Systems Integration) [62].

Building Systems Integration requires that a building is conceptualised and explicitly represented as a single integrated system and in a way which dissolves the boundaries between the previously separate engineering disciplines. In reality, a building is a single collection of physical components, however it is conceptualised as a collection of separate engineering subsystems (structure, external envelope, energy, lighting, circulation, egress, fire protection, etc.). This is shown diagrammatically in Figure 9 and illustrated in Figure 1. Each subsystem can be formally described by different analytical models, used with corresponding analysis and simulation applications and plug-ins (Figure 10). However, these separate computer applications and plugins effectively reinforce this conventional 'siloed' way of thinking.

In fact many individual physical components within the building, such as a simple wall (Figure 11) are characterised by a number of physical properties and, therefore, play a number of different roles in the overall performance of the building. This performance is the responsibility of different design and engineering disciplines. Thus, architectural design and building engineering already uses computation and requires a high level of cross-disciplinary collaboration. The question is: 'How can innovation in programming languages support new ways of thinking and collaboration required for innovation in Building Systems Integration?'

Each engineering subsystem is a complex network of components, therefore Building Systems Integration requires these networks to be unified to represent the complex physical connections and the performance inter-dependencies between the different subsystems. The challenge for the language designers is to create language features which can represent and manage these complex systems of connections while reducing the complexity of the task facing the end-user programmer.

Both programming and design are social activities. What is interesting is that at a collaborative team level, the process of Building Systems Integration is effectively a form of *constructive negotiation* between different engineering specialists where each is attempting to maximise their contribution to overall performance of the building without constraining the ability of their colleagues to do likewise.

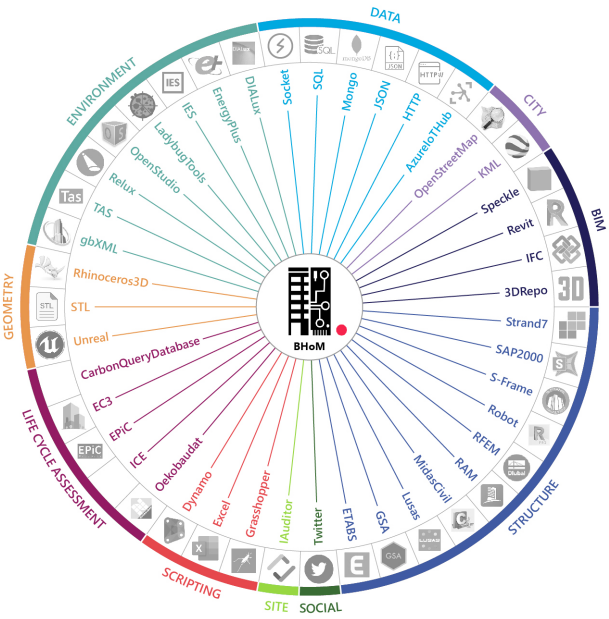

Figure 10. No single software or application can be used to create and analyse the many different analytical models needed for multidisciplinary Design Computation. Here are just 45 of the wide variety of design, analysis and simulation software used in practice illustrating the complexity of data interoperability and design collaboration [13]. The individual nodes in the diagram represent specific engineering, design and construction discipline software, including domain specific schemas and semantics.

To *not constrain* might include not prematurely defining dimensions or performance measures with unnecessary precision if there is no objective reason and certainty, particularly if this might limit the agency of other engineering specialisms. It is only through careful team negotiation that overall precision of the design can be increased and uncertainty reduced to the point where a single unified, resolved and precise description emerges of *what will be constructed* and *how it will perform.*[4]

[4]It is interesting to note the influence of game design on architectural design and engineering. While there is a continuing trend towards the '*gamification*' of design, there are fundamental differences between game design and architectural design in the intent, process, and outcome. Both game design and DC use computational geometry, programming, and simulation. However, in DC the computational model and image are not the *result*; rather, they are used to test the design *virtually* prior to its construction. It is the constructed physical building which is the *result*. Consider that a real building will have to obey the laws of physics (whether or not anticipated); therefore, there should be a high level of fidelity between the response of a virtual building used at the design stage and the equivalent response of the physical building after construction. However, a virtual building used in game design only needs to emulate those laws of physics which are required to '*suspend disbelief*'. Perhaps even this distinction is being blurred by 'design for co-living' in a combined physical and virtual space?

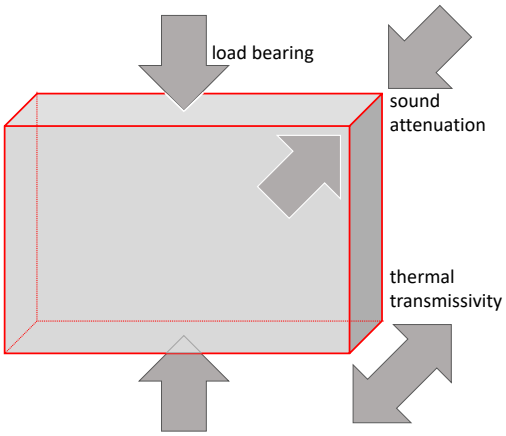

Figure 11. A selection of the many different physical properties of such a fundamental component as a wall and the different role this single physical object plays in a building. Criteria can extend further to light transitivity, porosity for ventilation or quantity of embodied carbon etc. (equally to qualitative and subjective properties such as aesthetics).

Therefore, Design Computation is effectively the integration of two complementary processes: progressive formalisation, in which the structure and semantics of the program is progressively resolved; and Building Systems Integration, in which the design and engineering of the proposed building is progressively resolved. This suggests that future Design Computation languages should support *semantic precision*, the modelling of *complex connectivity between components* and the *explicit representation of uncertainty*.

5 Proposal: A Progressive Language System

In Section 3, we concluded that visual programming does not scale to the complexity of even moderate design challenges; and in Section 4, we concluded that building systems integration is further increasing the complexity of these challenges. Therefore, we might reiterate: "*What to do?*"

Existing DC languages present a number of unresolved educational, usability, and representational issues. Visual languages are easily learnt, but do not scale to complex programming tasks, while existing high-level languages are challenging to learn for entry-level programmers and may not easily facilitate domain-specific thinking.

To address these concerns, we propose a comprehensive programming system based on clear language design principles [12] derived from established programming language features, tools, and methods.

The system comprises four related languages:

1. Initial
2. Transitional
3. General-purpose (i.e., industry standard)
4. Extension

This system of languages is intended to support the dimensions of Design Computation described in Section 2: First, the educational progression for entry-level users; second, the design progression from exploration to a precise engineering model and construction data; and third, the collaborative progression from individuals, to teams, to organisations (by encouraging co-creation, encapsulation, reuse, and explanation). These three dimensions are embodied and supported in our proposed system by a methodology of *progressive formalisation*. In this method, the entry-level programmer gradually learns the advantages of adding structure and human-readable semantics to their program and in so doing become a more proficient programmer.

This approach is similar, in principle, to the Hedy programming system—an educational tool which gradually increases the syntactic complexity and expressiveness of a Python-based language [31]. However, our proposal is tailored specifically to the requirements of Design Computation and incorporates both visual and text-based paradigms. The remainder of this section expands on the Initial, Transitional, and General-purpose languages; whilst Extension languages are explored in Section 6. We propose various approaches and implementations throughout.

5.1 Initial Language

Typically, the Initial language should lower the barrier for the entry-level user to learn programming. Based on the example in Section 3, and its wider critique of visual programming, the *raison d'etre* of a visual language is its usability. Existing visual programming builds on the conventions of dataflow-based diagramming [36], which is recognised as being inherently useful and intuitive as a high-level process description [21].

It is important to note that the nodes (or process boxes) in such high-level diagrams are usually clearly labelled and the overall complexity of the diagram is limited to the number of nodes whose names can be read when drawn on a single whiteboard/A4 sheet/laptop computer screen. These legibility limits apply equally to visual programming.

When these limits are reached (and when the user wants to progress to more complex programming challenges) then it is important that the visual program can gracefully hand over to other more scalable languages. In short, an Initial visual program should lower the barrier to adoption *and also lower the barrier to migration*.

This suggests that visual languages could benefit from being re-conceptualised and re-theorised.

5.1.1 Tenets. Based on these critiques, we propose a number of tenets for the design of visual programming languages which should be included in the agenda for future Design Computation:

- Careful agreement is required on what should be the key programming concepts to be included in an Initial programming language and whether these are suitable to be presented in a visual form;
- Warning the user when the visual program is approaching the limits of legibility;
- Avoiding any non-standard functionality or terminology in the visual language, which will not be available in the Transitional or General-purpose languages;
- Providing easily transformable code and appropriate conversion tools;
- Ensuring that during the migration from visual to text-based languages nothing is required to be unlearnt.

5.1.2 Reflection. There is perhaps a philosophical divide between the designers of the existing visual languages used in Design Computation and the approach proposed here.

Existing visual languages are a distinct form of programming which (in Design Computation) has evolved separately from the development of regular text-based languages.[5]

The entry-level programmer may not be aware (nor concerned) that some of these visual languages may be limited and non-standard and this might subsequently raise the barrier to migration. The immediate focus for the entry-level user is that the language lowers the barrier to adoption. If more advanced functionality is required, then perhaps these concerns are dismissed by the entry-level user imagining that a suitable plugin can be easily downloaded into the visual programming environment.

An alternative and more open approach is proposed here which suggests that visual languages have an important role not just in lowering the initial barrier to programming, but also lowering the barrier for timely migrations to other programming paradigms. This divide could be viewed as a specific form of a more general distinction: Between those for whom the advantage of technology is as an entry point to understanding the underlying principles; and those who consider technology to be an advantage if it can be effectively used without requiring any deeper understanding.

[5]Visual programming systems are predominantly based on DAGs (directed acyclic graphs). These evolved as a generalisation of the CSG trees (constructive solid geometry trees) and 'feature' trees which are still used in many solid modelling and mechanical engineering design applications.

In this directed-rooted tree formalism, the leaf nodes are treated as the inputs (typically geometric primitives), the fork nodes are geometric modelling operations and the root node is the output.

Trees are (1) a formalism which is easily understood by the intended user audience and (2) optimised for partial updates: when a node is changed only the dependent nodes need to be re-evaluated. The directed-rooted tree restricts the output of a node to be the input to only one dependent node. This restriction is relaxed with more general DAGs used in Visual Programming, increasing expressiveness, but potentially creating unintended downstream comprehension problems (as Figure 8 illustrates).

5.2 Transitional Language

The objective of the Transitional language is to provide a continuous path for the individual end-user programmer from entry-level visual programming to proficient programming with general-purpose languages–achieved by gradually introducing standard and scalable functionality.

The features and programming tools of Transitional languages are intended to nudge (subtlety encourage a change in behaviour) the entry-level programmer by altering the attractiveness and the cost of an early move from visual to text based programming. In this context, the design of the Transitional program could be considered as a form of 'choice architecture'.

The features we propose for a future Transitional language are developed from ideas originally explored as part of the DesignScript language for Design Computation [6]. DesignScript implemented the following progressions:

- from a visual data-flow notation,
- to a text-based data-flow notation,
- to a text-based imperative notation.

The user is supported in this progression via a Node-to-Code conversion tool, which for the entry-level user's original visual program automatically generates a program in the standard textual notation without requiring the user to first master this notation (Figure 12).

DesignScript (as a prototype Transitional language) anticipated this progression by implementing the visual language as a graphical wrapper around the text-based Data-Flow notation. This ensures that there is a direct correspondence between each visual node and the equivalent text-based statement. The aim of the transitional language is to provide a simple text-based notation which can be directly understood by the entry-level user. For this reason some more complex conceptual and syntactical programming challenges are initially deferred. The pedagogic advantage is that the entry-level programmer can begin to understand the programming notation by observing the correspondence between the visual and text based code of the transitional languages.

The Transitional language therefore encompasses the whole range of proto-programming and programming paradigms, from 'feature' trees, to more general Visual Programming DAGs, to imperative code: and supports the transition of the user's code between these forms.[6]

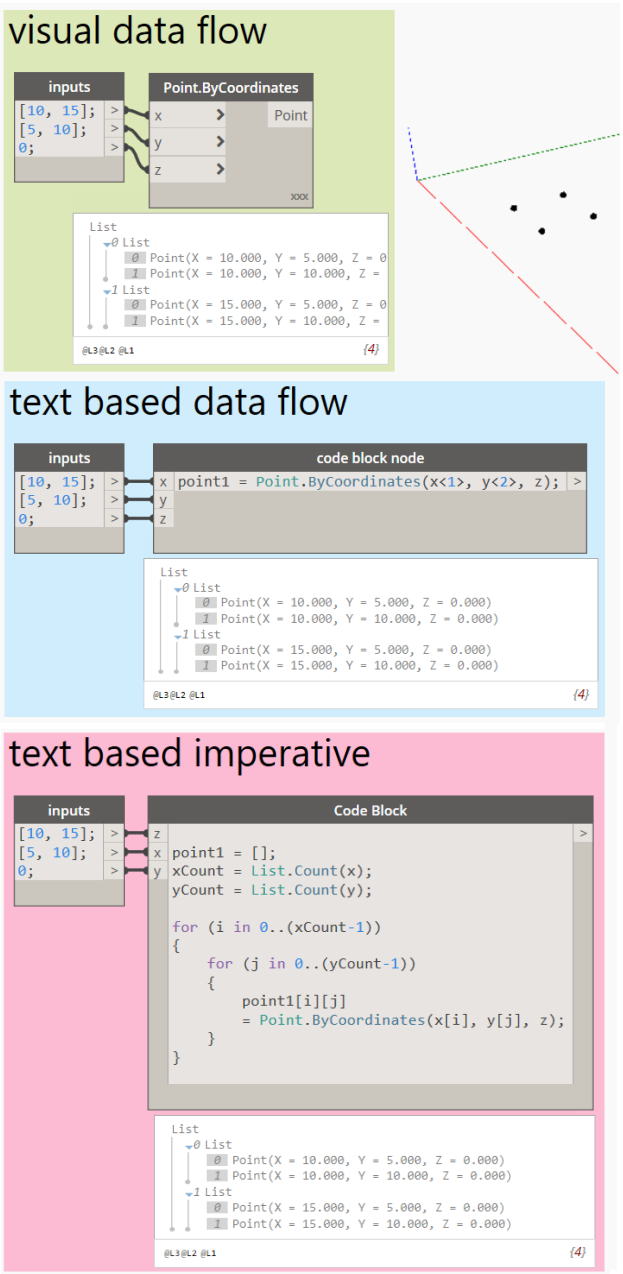

Figure 12. An example of a Transitional language (DesignScript) in use. Here the same logic (to create a 2x2 array of points) is shown in the Visual Data-Flow notation (top), in the text-based Data Flow notation (middle) and in the Text-based Imperative notation (bottom). Note that all nodes and statements in the data flow languages (both visual and text-based) are overloaded for collections (e.g., being lifted pointwise). However, this is not supported in the regular imperative notation used in industry standard General-purpose languages and thus requires code expansion (see, e.g., [6]). We return to a deeper discussion of this in Section 9.

[6]It is interesting to note that the Blueprints visual language [59] does not have a transitional language. The nodes in Blueprints are predefined C++ class. A user can assemble a subgraph of Blueprint nodes which can be composed into a new C++ class. However, in the DC community, C++ is considered as a professional level language. Therefore Blueprints may inadvertently divide the users into entry-level programmers (using the VPL with existing or composed nodes/C++ classes) and professional programmers (who can operate within the nodes or classes). By comparison the intention with the Transitional language is to dissolve the boundaries between users with different computational competencies so that a continuous educational trajectory can be created.

5.2.1 Progressive Formalisation. This is a process in which the programmer adds user-defined semantics (name and type) and structure to a program, thereby enhancing both its human and machine readability.

Timing is important. Typically formalisation starts when program has become sufficiently capable and can demonstrate its potential value, but before it has become too large and complex that formalisation is difficult to apply.

In terms of cognitive dimensions research [14], the user should not be forced to make a *premature commitment* to the early formalisation of an unproven exploratory program, but should also avoid *viscosity* if formalisation is delayed and an exploratory program is allowed to become over complex.

Our case study (Section 3) illustrated the functionality of existing visual languages for Design Computation, which allowed unnamed (or generically named) variables, while the only collection mechanisms provided are for numerically indexed homogeneous collections (such as lists of lists). Figure 13 illustrates how progressive formalisation would be implemented in a future Transitional language. Here, the user is encouraged to start by giving each variable a meaningful name and type. The variables can then be gathered into a defined *schema* and used to generate a new class definition. The use of *clone detection* [60], recognising repeated patterns, and accompanying transformations will then be critical in weaning users off the copy-and-paste technique.

5.2.2 Hybrid Programming. The Transitional language plays a pivotal role in the educational journey of the end-user programmer. Starting with a conventional visual program, the Transitional language allows the user to progress to a form of hybrid visual-textual programming. This combines the advantages of succinctness of text-based coding (within different text nodes) with the advantages of high-level visual diagramming (between nodes). (Figure 14). The user may progress to combine the different text nodes into a single node and leave the Transitional programming phase with a conventional program source file.

Hybrid programming combines both text based programming conventions within code block nodes and visual programming conventions between nodes. This introduces some interesting language design challenges, particularly as there is an overall objective that all aspects of the transitional language should be transformable into regular industry standard formal languages. One aspect is that visual code block nodes should become code blocks in the formal language.

The current Visual languages allow connections between unnamed or differently named variables in different code block nodes, while text-based languages use named references to define such dependencies and also define scoping rules for variables and code blocks. To anticipate the transition to the formal language, the visual language's conventions might need to be modified so that these two conventions are aligned. This would require (a) that arcs can only represent connections between the same named variables in different code block nodes and (b) that the scoping rules in the visual language have to follow those found in formal languages.

Again, this raises the question: Should visual languages develop entirely separate usability features (limiting the transition to text-based languages); or follow the conventions of established text-based languages (and support transition)?

5.2.3 Tenets. We propose the following tenets for Transitional programming languages which should be included in the agenda for future Design Computation:

- By default, the Transitional Language should use the notation of Industry Standard Formal languages.
- Where non-standard notation is introduced in the Transitional language, this should be transformable into the notation of Industry Standard Formal languages.
- Appropriate programming tools should assist the user in all progressions between the different notations (from Visual Data Flow to Text based Data Flow to Text Based Imperative notation) so that these can be achieved without the user first having to learn the new notation and so that the user can understand the new notation by comparing the corresponding statement in the different notations.

5.3 General-Purpose Languages

It is intended that the Initial and Transitional languages will help the entry-level programmer to understand the imperative, object-oriented, and functional programming paradigms offered by General-Purpose Languages (GPLs) such as C# or Python. However, these languages may not offer the kind of domain-specific features required in Design Computation. In a given domain, the success of a GPL is usually dependent on well-developed libraries which, in this context, are provided by the DC programming environments. A Transitional language could perhaps be viewed as a subset, or an easily translated 'sibling language', of a General-purpose language (in the fashion of gradual languages like Racket [24, 69]).

In the following sections, we propose language extensions to enable domain-specific features —based on the specific requirements of Design Computation— whilst also retaining the substantial power and flexibility of GPLs.

6 Proposal: Language Extensions for DC

The fourth part of our proposed system extends the general-purpose approach with *domain-specific* constructs. We argue that more precise representations of both design objects and processes can be enabled by drawing on facets of Programming Languages research with a particular focus on *types*.

One of the most significant interactions between Design Computation and PL research is in the development of type systems. It is recognised that object-oriented languages are a generalisation of what was originally a highly domain-specific concept based on the classification, properties and

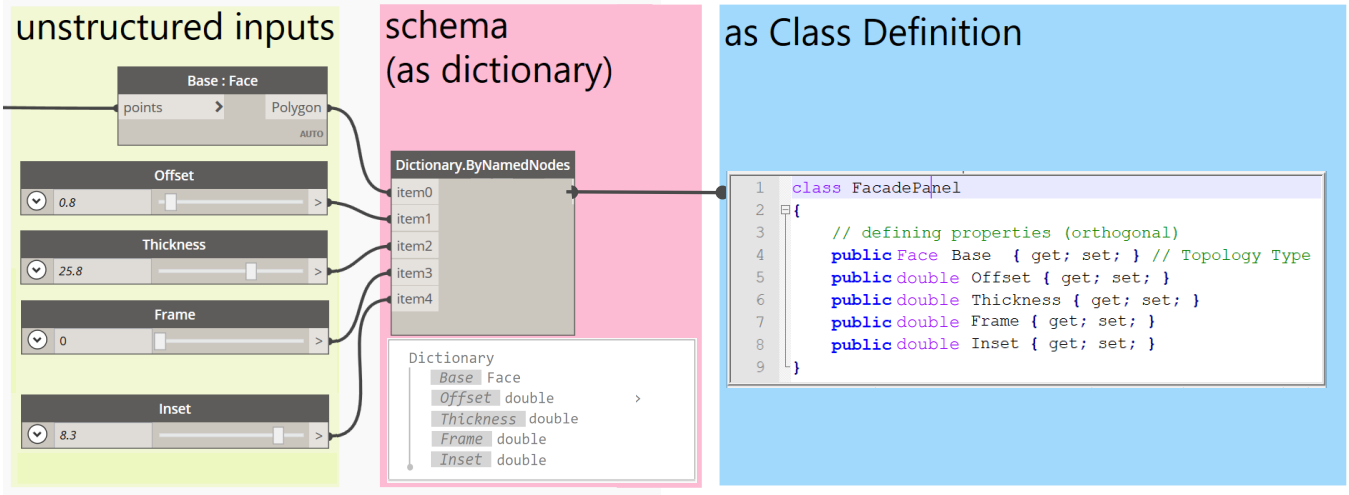

Figure 13. In the progressive formalisation methodology, the user is encouraged to start by first gathering the unstructured inputs as a schema (middle panel), from which a distinctive new class can be defined (right panel). Here each input has been given a meaningful name and type.

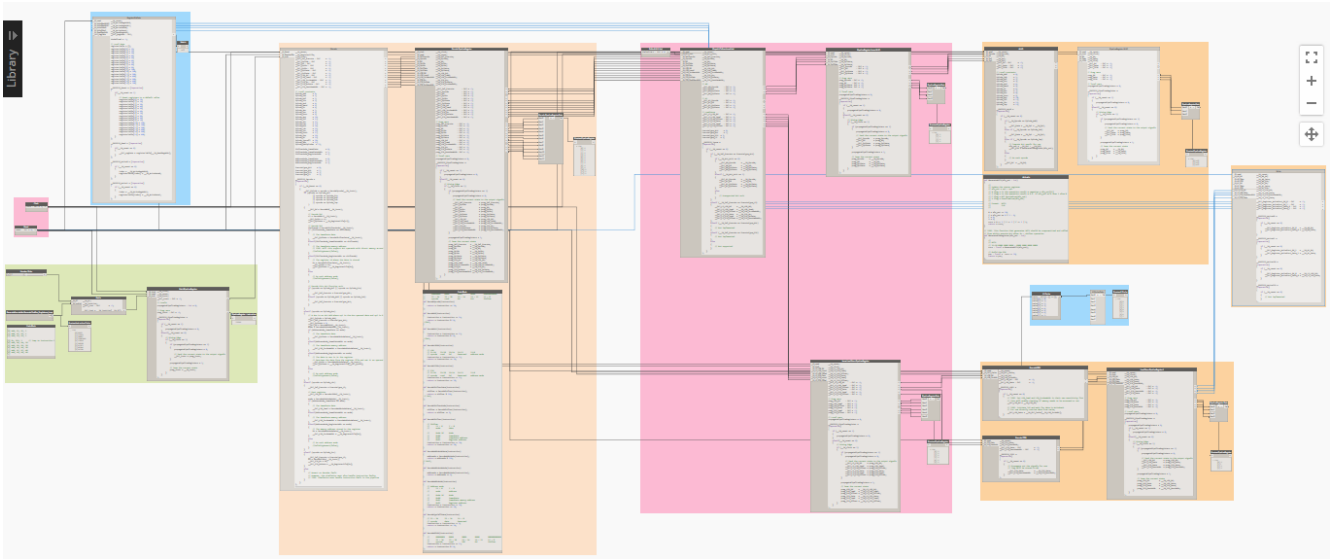

Figure 14. Example hybrid visual and text-based programming from [6]. The visual layout is the high-level process diagram of the MIPS microprocessor pipeline. The code within each node emulates the function of the different regions of the MIPS hardware.

behaviour of physical systems, for example the Simula language. We share the sentiment expressed by the authors of Simula, Kristen Nygaard and Ole-Johan Dahl: that languages "should be problem-oriented and not computer-oriented" [52].

In Design Computation, a type system is much more than a language feature: It is both an ontological representation of what exists and an exploration, or validation, of what *might* exist: how it might be made and assembled; how it might perform and be used; how it might be repaired and recycled; and, crucially, how it might be *dis*assembled.

So for DC, the critical question is: How can the life-cycle of a physical object, whether a product or a building, be represented by a type system as a direct computational analog of existing real worlds (and our conceptualisation of future real worlds)? This extends beyond the tangible objects, such as walls and floors, to the intangible spaces which they enclose.

These requirements match recent advances in programming language research, where a rich vein of work focuses on increasing the expressive power of types to reason about programs in more detail [25, 54, 61].

6.1 Semantic Meaning: Dimensions, Units, Quantity

Previously standard engineering calculation sheets through their tabulated design and units-of-measure annotation helped the user to enforce units-of-measure consistency; but this form of consistency checking has largely been lost with the switch to engineering analysis applications and end-user programming (particularly using conventional languages that do not provide sufficient *domain information*).

Most programming languages provide only a limited set of numeric types (e.g., integers, floating point) to represent completely different physical quantities (e.g., length, force, angle). This lack of dimensional information can lead to undetected numerical errors arising from inconsistent dimensions or units of measurement.[7]

For decades, it has been recognised that dimension and unit information can be seen as an extension of standard type systems [37, 38]. Units-of-measurement typing (for example as developed by Kennedy) instead embeds the rules of dimensional analysis into the programming language. Typically, this would be based on the SI units [51]. Going further, variables could be qualified by a hierarchy of attributes: dimension (e.g., length), units of measure (e.g., metre), and kind-of-quantity (e.g., thickness) [20]. These attributes can then be enforced automatically by the compiler, some attributes can be inferred from use, and conversions can be automatically derived.

The SI system consists of 7 base measures and 23 principle derived measures. A derived units-of-measure type is defined by an expression using base unit types or previously defined derived unit types. We envisage a units-of-measure definition statement which allows derived units-of-measure types to be defined for specific application domains. For example, the U-value, or thermal transmissivity of a building material (Figures 11 and 15) could be defined as a derived unit-of-measure: $W/(m^2 \cdot K)$.

Units-of-measure types may also use refinement types to constrain the range in which a value lies [71], e.g., a number representing angles may be typed as ranging between -2π and 2π. A refinement type system enforces that this invariant holds throughout program execution.

One of the key tests for the compatibility of different units of measure is whether variables can be added or subtracted. For example, consider the dimension variables defined in Figure 4. These are Inset, Offset and Thickness. These are all of the same dimension *length*, but one is a specific kind-of-quantity *thickness*. Also, the Inset and Offset dimensions are defined in orthogonal directions. While the addition of variables of different unit-of-measure type might raise an error, we can envisage a hierarchy of warnings to alert the programmer when variables with the same type, but with different attributes, are added (or subtracted).

Addition and subtraction also apply to integers. Therefore, there is the potential to extend some of the units-of-measure concepts particularly when integers are used as cardinals. If a collection has a specified type, then the count of that collection is a cardinal of that type. Consider a class hierarchy (from general to specific) and collections of these different types. Appending two collections of different sub-types might result in a collection of the common super type. Therefore adding the counts of collections of different sub-types would result in a cardinal of that super type.

Subtraction is slightly more complex: The result (the difference) will be the same type as the minuend, and the subtrahend should be of the same type as the minuend or more general. We can envisage units-of-measure consistency being applied in these cases.

In program development, it is often necessary to distinguish between the units for variable used within the program (for calculation) and the units used in the user interface (to communicate with the users). For example, angular measurements might be calculated internally in *radian* but displayed in *degrees*. How these display units are labeled and understood by the user should be considered as an integral part units consistency. This becomes quite interesting in end-user programming when the user is the programmer.

Implementing units-of-measure types at the language level provides important advantages for Design Computation by improving the human-readable program semantics while a fully units-consistent program (being machine-readable) can be verified as free from any dimensional errors [38, 56]. It is anticipated that some aspects of the units-of-measure types can be retrofitted to industry standard languages, for example using C# attributes, discussed further in Section 8.

6.2 Variability: Tolerance and Uncertainty

Materials, manufacturing, assembly and even measurement itself are characterised by variability, conventionally expressed as tolerances. Yet existing digital modelling tools typically only support nominal absolute sizes, with tolerances only specified as annotations. This severely limits the designer's ability to reason about the critical role of tolerances. Whilst the calculation of numerically precise values is a key aspect of Design Computation, there are occasions when precision might be confused with certainty.

This is particularly relevant at the specification or initial design stages of architectural design, when the performance of the building, or its precise dimensions, have yet to be confirmed. At these stages, it is more useful to define a *performance space* whose dimensions are bounded by acceptable ranges of the defining performance measures (Figure 15).

[7] There are many historical engineering failures associated with *units-of-measure* error. One notable example is the Vasa [48] (a seventeenth century Swedish warship which capsized on its maiden voyage). Archaeological analysis found two types of rulers within the hull, one based on the Amsterdam foot using twelve inches and the other based on the Stockholm foot using eleven inches. Measurements from the hull suggested that the two sides of the ship had been built by different teams of shipwrights, one Swedish and one Dutch, using these different units of measure.

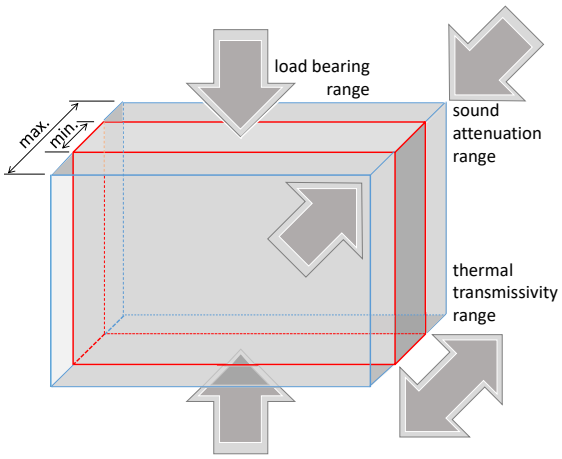

Figure 15. The hypothetical wall from Figure 11. At the specification stage, a performance space can be defined by the acceptable range of the defining performance measures.

In many types of engineering, including buildings, 'compensators' (e.g., seals, bolt slots or glazing spiders) may be designed to pre-empt variability during construction assembly. In this application domain, a tolerance (of a material or manufacturing process) is the expected limit of acceptable *unintended* deviation from a nominal or theoretical dimension, while an allowance (of a compensator) is a *planned* deviation from the nominal or theoretical dimension.

One way to represent uncertainty within computation is via *interval arithmetic* [46], where numerical values are replaced by a pair of values (of the same type) giving a lower and upper bound. Interval computing can be considered as the numeric counterpart to fuzzy logic. We can imagine a generic type defining a general form of intervals over any base type, i.e., `Interval<T> = (T, T)`. Operations are then overloaded according to the standard operations of interval arithmetic. However, in the context of DC, we typically wish to work with an *idealized* measurement for a design within which the *actualization* (i.e, after manufacture) will be known to vary based on manufacturing tolerances.

We therefore propose that a *type-level* notion of variability is much more useful as the static property can represent the additional domain semantics of the measurements. A *tolerance*, giving a 'plus-minus' value by which a value can vary, e.g., $x \pm y$ corresponds to the a value v in range $x - y \leq v \leq x + y$ and could then be captured via a type constructor `Tolerance<Y>` wrapping a floating-point value but where `Y` is a type-level floating point number classifying the variability. *Direct limits*, giving distinct lower- and upper-bounds, could be provided by further constructor `Bounds<X,Y>`.

More fine-grained approaches to representing variability, as codified in the language of Geometrical Product Specification [47] could also be codified at the type level. These numeric properties, akin to refinements codified in the types, can be enforced by the type system, e.g., comparing interfaces of compensators and the tolerance they accommodate versus the variance of components connected to them. Automated theorem provers can then be leveraged by the type-system to discharged complex constraints (see, e.g., quantitative program reasoning at the type-level [55]).

Currently, a numeric variable can only be defined with a less precise type and an over-precise value. Combining units of measurement and intervals radically improves program semantics allowing the direct expression of the specificity of the variable's type and the option to explicitly recognise its uncertain value.

6.3 Connected Collections

It is recognised that buildings are composed of complexly connected collections of physical components and enclose equally complexly connected collections of spaces [19]. While these collections are often hierarchical and can be defined 'top-down', the important queries which allow the user to reason about these relationships need to operate 'bottom-up'.

In standard object-oriented practices, one object can point to another via a field/property denoting some relationship. However, this relationship is not invertible; i.e., knowing which objects 'point back' to an object requires manual accounting.

Many proprietary applications (for structural analysis, energy simulation, digital fabrication, etc.) implement equivalent functionality but often in a proprietary and opaque way. Therefore the end-user programmers may be aware that this functionality exists, but it is not available for more general application and reuse within their programs.

Instead, adding Connected Collections to the language will ensure that inverse mapping from the constituents back to the collection is automatically maintained. This allows designers to directly and conveniently build and query many different types of connectivity such as: complex assemblies and engineering systems, circulation and egress routes, material libraries, simulation results and performance data [5].

Indeed, integrating connected collections with tolerances and dimensional typing will enable diverse but semantically connected analytical models using graph theoretic operations and mereological and topological concepts such as part-whole relations and *vague* boundaries.

The motivation for the Connected Collection language feature comes from both theoretical and practical sources.

Marr and Daloze [42] discuss the relative advantages of providing a few versatile collection types compared to a wide range of specialised collection types. In this context, it is interesting to note that Grasshopper (one of the most widely used VPLs in Design Computation) provides only a single collection type: a 'data tree' (as a list of lists). This effectively imposes a new task on the entry-level programmer: to refactor all potential uses of collections into this single collection

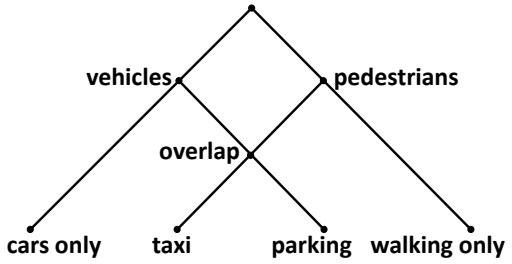

Figure 16. The use of a semilattice to represent two distinctive functions of a city and their overlap (redrawn from Figure 4 of 'The City is Not a Tree' by Alexander [9]).

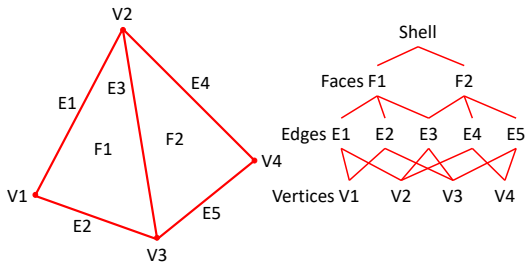

Figure 17. An interesting example of a connected collection is provided by the shell-face-edge-vertex topology (from the faceted Façade model in Figure 3).

type—a form of 'enforced' versatility. The mechanics of using data trees is explained in the tutorial by Rutten [63], but the justification (either pedagogic or performance) for data trees seems quite elusive. Our current proposal for a new connected collection type is intended to be a constructive response to these limitations.

The emphasis on tree collections in Design Computation is additionally surprising since the limitations of trees as representations in architecture was already recognised by Christopher Alexander in his 1965 seminal essay: 'A City is Not a Tree' [9]. From a substantive perspective, Alexander observed that the contemporary approaches to urban planning often resulted in new towns being divided into separate unconnected districts (a tree structure) while traditional cities were more complex and often included overlaps or connections between different districts and functions. Alexander used the semilattice to illustrate the concept of the 'connected' city (Figure 16). In so doing, he used an abstract concept in a highly domain specific way and with a particular normative interpretation (concerning the human-centric design of the urban environment). The more valuable, and general, point is that this abstraction combines a 'top-down' decomposition (of the city into sub-districts or sub-functions) with a representation of the overlap or intersections between the different parts of this decomposition.

In retrospect, neither trees nor semi-lattices may be suitable representations (of cities, buildings, systems, etc.). However, the graphic theoretic concepts which Alexander introduced are highly relevant for Design Computation.[8] Thus, one motivation for our proposed Connected Collections is as a more general response to Alexander's original ideas.

[8]Alexander also proposed the idea of 'Design Patterns' for architecture. While the general idea of Design Patterns is interesting, the specific architectural patterns Alexander proposed were considered within architecture to be too normative and prescriptive and this undermined the acceptance of the general idea with architecture. However, his ideas (now detached from the normative architectural examples) were valued and subsequently adopted within software engineering. This represents another interesting interchange between architectural design concepts and computation.

The Digital Fabrication of the faceted façade example (Figures 3 to 7) illustrates a simple complexly connected collection where a conventional tree data structure would be an insufficient representation. At one level, this type of façade can be defined as a hierarchical system of Topological collections where a Shell contains Faces (representing glass panels) which contains Edges (representing linear frame members) which contains Vertices (representing connectors). Except this is not a conventional tree structure because Faces are connected by sharing common Edges which in turn are connected by sharing common Vertices (Figure 17).

A Connected Collection would facilitate interesting and valuable queries on the façade. The critical query may not be *face.Edge*, but rather *edge.Faces*.

Similarly, a building can be idealised as a hierarchical system consisting of an overall Body (representing the complete building), with Cells (representing rooms) and Faces (representing the construction panels, such walls, floors and roofs). Except this is not a conventional tree structure because rooms (Cells) are connected by sharing common walls (Faces) (see Figure 18). This representation of the complexly connected spaces within a building is an example of non-manifold topology and is the basis for many different analytical models, including those used for thermal, energy, circulation and egress of occupants [5].

In this context, the critical query may not be *cell.Faces*, but rather *face.Cells*. All Faces expect F6 are part of a single Cell and therefore represent external walls, with energy loses or gains to the external environment (depending on the wall orientation). Face F6 is however part of two Cells and therefore represents an internal wall with a possible energy exchange between adjacent rooms at different temperatures [5].

Connected Collections can represent a range of inclusion and connection concepts (from general to domain specific): (1) very general inclusion relationships where members may be part of multiple *intersecting* sets; (2) *network relationships* between the members; (3) more specific network relationships which define a (potentially closed) *containment boundary* (as in the spatial building model, Figure 18); and (4) where containment boundaries may, in turn, define other

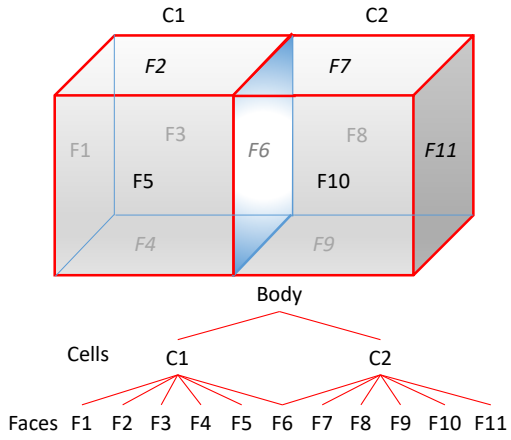

Figure 18. Another example of a connected collection is provided by an idealised spatial model of a simple building using non-manifold topology. Here the adjacent Cells (representing rooms) are enclosed by Faces (representing walls and floors). All but one of the Faces represent *external* construction. However, the two rooms are connected by sharing a common Face (representing an *internal* wall panel) (F6), as described in Figures 11 and 15 and referenced in Figure 9.

more general inclusion relationships (for other sets whose members are so *contained*). In addition, the same member may be referenced by different collections, but these references may imply completely different domain semantics.

Connected Collections are a recurring theme in Design Computation. Previously, each application implemented their own bespoke, hard-coded version which maintained the additional references from the members back to the different parent collections. While end-users could harness these applications and write custom DC programs using the related APIs; the end-user programmer could not construct new, independent, connected collections.

The proposal is that Connected Collections become a standard feature of DC languages, directly accessible by the end-user programmer, potentially re-using and extending conventional collection syntax (with the management of the additional references masked from the programmer and implemented within the language system).

Connected collections allow the user to reference multi-functional components as common members of different engineering subsystems. However, this raises interesting language design issues around the most suitable way to represent such multi-functionalism: as multiple inheritance; as multiple interfaces; or as a lightweight composition using the aspect-oriented programming paradigm (which is the approach adopted in Design Computation practice [13, 22]).

A prototype of Connected Collections has recently been completed and tested with the examples used in this paper, including figure 17) . [18]

6.4 Tenets

As with the other languages in our system, we outline a few core tenets for the Extension language:

- Leverage existing type checking support and syntax;
- Use type inference, but allow compilers to insert inferred types into the code automatically to aid documentation and user understanding;
- Introduce new syntax sparingly (only when required to express new functionality which cannot be emulated by the use of established functionality and familiar syntax);
- Enable user-defined extension of domain concepts to suit the project or engineering sub-discipline;
- Ensure backwards compatibility with other language layers in the system;
- Allow progressive formalisation, in the style of *optional typing*, such that type information can be gradually added;
- Leverage *gradual typing* to enable complex static types to initially be treated dynamically [17].

From an end-user perspective: The functionality of the interval computing feature naturally extends to complex engineering tolerancing [33]; and the functionality of the connected collections feature naturally extends to the geometry and topology of solid modeling [5].

However, there is necessarily a clear boundary between such specialist applications and the functionality of the language itself. As such, the extension language should provide common language features which can reduce the complexity of developing specialist applications and plugins (and to provide end-user programmers with a standard way to integrate these plugins).

Finally, it is envisaged that a proficient programmer could directly use the Extension language features with a general-purpose language and independent of the Initial and Transitional languages.

7 Case Study (Continued): Using the Extension Language Features

This section resumes the case study of the Façade Panel starting from Figure 13, which illustrated the features of a programming language where every type of the numeric variables was defined as `double`. This can be contrasted with Figure 19 where the program semantics has been significantly improved by defining the different linear dimension inputs with precise units-of-measure or tolerance types.

Figure 20 shows how the class definition can be expanded to include additional methods which capture the geometric process defined in the original visual program (Figures 5 and 6). The more richly described `FacadePanel` class in the Extension language can then be instantiated as a component in the Initial/Transitional setting (Figure 21).

This methodology has emerged from our experience in practice and applies at both a team and an individual level.

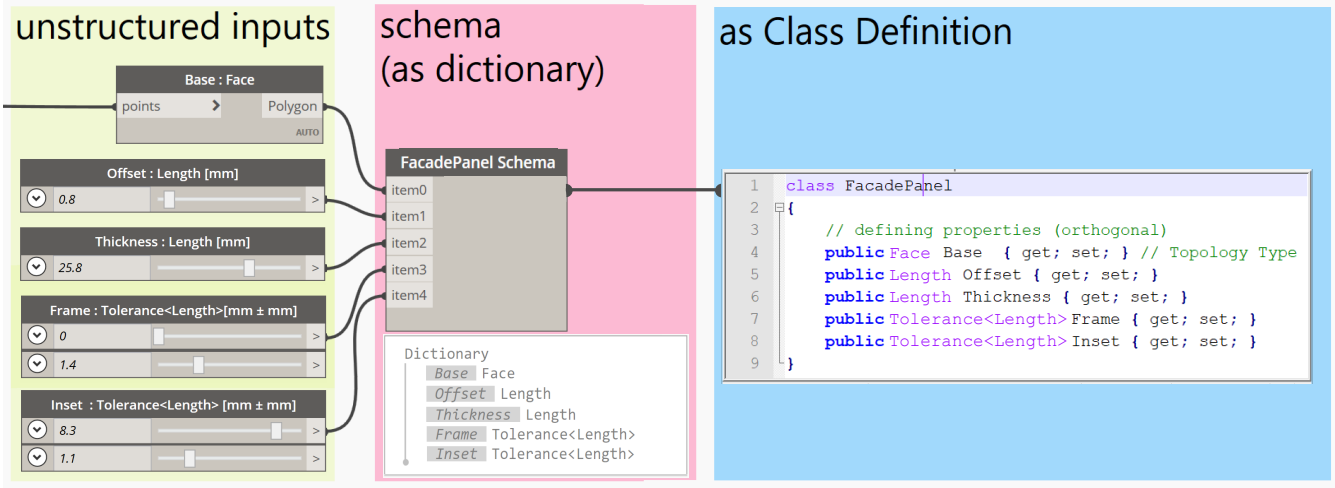

Figure 19. Using the features of the language extension, the program semantics can be significantly improved by defining the different dimensional inputs with precise units of measure or tolerance types. It is envisaged that the Visual language UI will support custom 'widgets' providing simple and intuitive access to these options. The user can understand the notation by comparing the visual program with the generated text-based Class definition.

```
1  class FacadePanel {
2      public readonly Volume PanelVolume;         // properties defined as unit of measure type
3      public readonly Tolerance<Length> SealWidth;// as a tolerance type
4      public readonly Face ContextFaces;          // as a connected collection type
5
6      public static ByOffsetAndInset (Face Base, Length Offset, Length Thickness,
7                    Tolerance<Length> Frame, Tolerance<Length> Inset)  {
8      // application of standard geometric modelling, from the original visual program
9      insetCurve    = Curve.Offset(Base, (Inset.Nominal + EdgeWidth.Nominal);
10     insetSurface  = Surface.ByPatch(insetCurve);
11     CentreSurface = Surface.Offset(insetSurface, Offset);
12     Panel         = Surface.Thicken(CentreSurface, Thickness);
13
14     // Extension features: Units of Measure, Tolerance Typing and Connected Collections
15     Length^3 PanelVolume                = CentreSurface.Area*Thickness // inferred Units of Measure Type
16     Tolerance<Length,0.05> SealWidth    = Inset - EdgeTolerance;       // Units of Measure combined with tolerances
17     ConnectedCollection<Faces> ContextFaces = Face.AdjacentFaces;      // Connected Collection: find the Faces
18   }                                                                    // (connected by common Edges)
19 }
```

Figure 20. The FacadePanel class can be expanded to include additional methods which capture the geometric process defined in the original visual program (Figures 5 and 6). line 16: Calculates a new value from an Area and a Length therefore the units-of-measure system has inferred the resulting variable is of type Volume. line 17: Computes a new interval value via arithmetic expressions over interval terms, with resulting type `Tolerance<Length,0.05>`. line 18: The façade system is constructed out of a connected collection of Shell-Faces-Edges-Vertices in which constituents at one level are connected to constituent at the next lower level. Having built such a connected collection, the user can make useful queries (for free).

At an individual level, we should encourage the entry-level programmers to harness progressive more powerful computational constructs.

At a team level, we can mirror the classical Design Process by encouraging the gradual transition from an exploratory unconstrained and implicit sketching/programming to the use of more precise and sharable representations: Effectively applying established design principles to the way the end user software itself is written.

The motivation for our proposed Design Computation language system comes from our experience that it is difficult to apply this preferred methodology in a coherent and elegant way with existing languages and tools. With the proposed

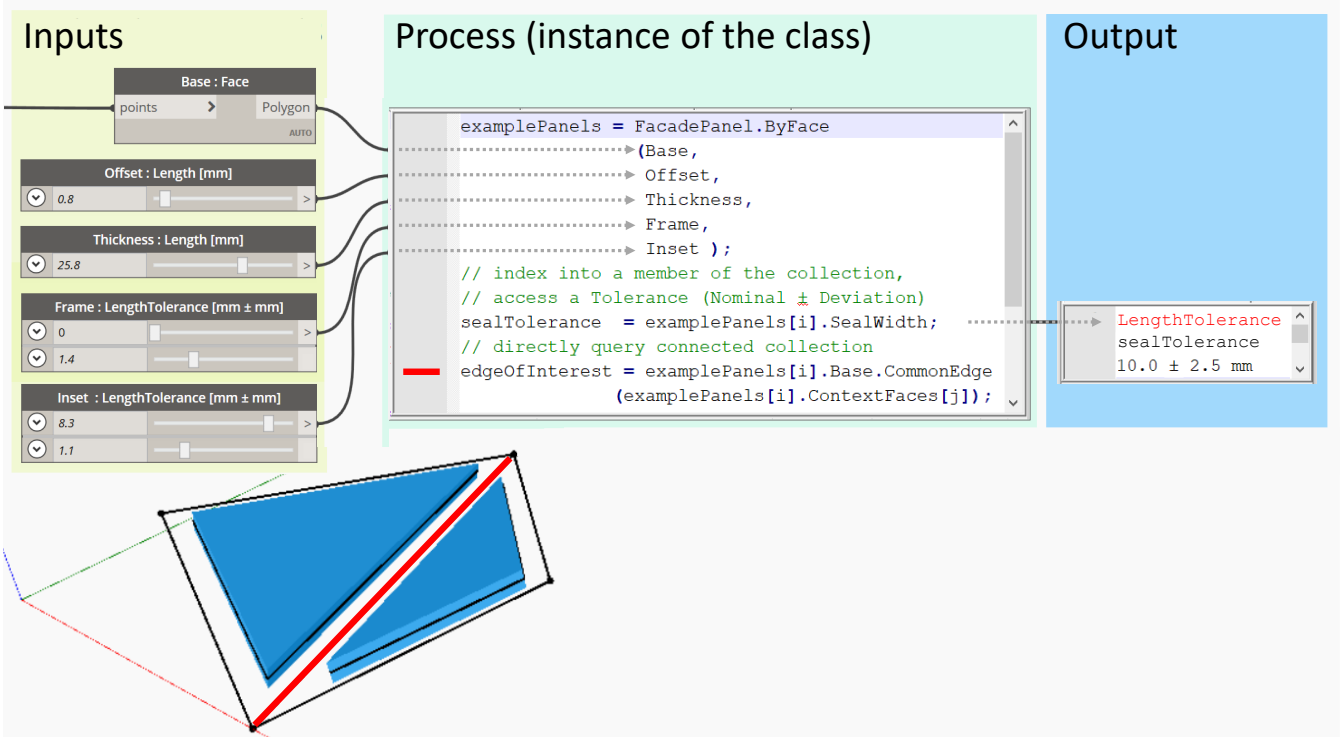

Figure 21. The new Façade Panel class can be instantiated, replacing the process in the original visual program (Figures 5 and 6). Because the inputs are a collection, the result (`examplePanels`) is also a collection. However, the user can index into this collection and access constituent properties using the standard 'dot' notation. Queries from the connected collection such as the `edgeOfInterest` are highlighted in red.

system, progressive formalisation starts with the Initial visual language and with unstructured data and informal use of language, geometry primitives and process.

This progression continues via the Transitional language, where human-readable semantics, such as type and structure is added to create a conventional data schema and an object-oriented class definition.

Progressing to the General-purpose language and then Extension language features, the program can be enhanced with more precise dimensional typing, tolerance typing (which explicitly recognizes imprecise and uncertain values) and the use of connected collections to more precisely represent connectivity relationships.

Adding the additional type information enables the program to become machine readable, allowing verification through inference. This results in semantically precise, domain-specific classes.

The overall process is cyclic, with the possibility that other entry-level users can deploy these new classes as nodes in their visual programming.

8 Paths Forward From the State-Of-The-Art

The proposed workflow outlined in Sections 6-7 assumes an object-oriented approach to the implementation of the General-Purpose and Extension languages.

An advanced type system is implemented by class definitions representing physical dimensions (augmented with additional information such as type-level floating-points to capture tolerance). We consider here existing work which could serve as a transitional path from the current state-of-the-art to our proposed workflow; which, in turn, may increase the adoption potential of the overall progressive formalisation strategy.

8.1 Domain-Specific Descriptions via C# Attributes

One project (which has sought to implement some of the concepts discussed in the preceding sections) is the *Buildings and Habitats object Model* (BHoM) [13], an open collaborative project that aims to create an inclusive computational framework for coauthoring Design Computation workflows.

Specifically, and in direct response to the range of user requirements and skills highlighted in Figure 2, the BHoM is designed to be compatible with both visual flow-based programming (e.g. Grasshopper, Dynamo, Excel) and text-based programming (e.g., conventional C#).

Implemented as a hybrid model for code architecture, it is designed to integrate well in the workflow of any professional in the AEC industry, regardless of their level of computational proficiency.

The underlying framework is implemented in C#, but includes various constraints and conventions to achieve equivalence between the visual nodes and the corresponding text-based functions and objects when the code is reflected in visual programming interfaces, as per DesignScript and the proposed Transitional languages discussed in Section 5.

Particular emphasis has therefore been placed on defining clear object schemas; named, labelled and decorated with semantic contextual information to enable chunking of the community's collective engineering knowledge as reusable modular design representations.

The necessity for this schema-led approach can be seen in our case study (Section 3) where manipulation of unstructured, unlabelled data quickly becomes intractable.

As a concrete example of enriching C# classes with additional semantics, primitive Properties of type `double` can be additionally defined as a type of Quantity through assignment of custom Attributes [27]. This value now has additional meaning which can be used to enhance interpretability for the end-user, exposed both in the programming IDE and the Visual Programming Environment.

For example, a `Circle` class may have property `Radius` of type `double`. This can be assigned attribute `Length` [26] implementing the abstract base attribute `QuantityAttribute` [27]:

```
[Length]
[Description("Distance from the Centre to any
     point on Circle.")]
public virtual double Radius { get; set; } =
    0.5;
```

Figure 22 shows this rendered in the Visual Programming Environment.

8.2 Other Approaches

There are various other ways that domain-specific concepts might be 'retrofitted' to existing languages.

The Python Pint library uses * operators to assign units of measure to variables, e.g. `radius = 0.5 * ureg.metre`, yielding a new variable of type Quantity which encapsulates both value and unit information [29]. Through the development of a bespoke Design Computation library, one might imagine something similar being applied to a combination of domain-specific concepts, e.g. `radius = 0.5 * lib.dreg.length (unit=lib.ureg.metre)`. Since many programming languages use libraries, this type of approach could be fairly language-agnostic; however, the potential impact on runtime performance and behaviour could be significant and backwards compatibility with existing projects and workflows may not be realistic.

As a less disruptive option: The CamFort project uses structured comments to verify units of measure in Fortran-based computational science models [53]. In principle, concepts such as dimensions, tolerance or connected components could be implemented in a similar way (e.g. `!=dimension( length)::radius`). While this might provide a high level of cross-language support and backwards compatibility; comments are generally not subject to syntax checking by compilers or interpreters, and they may be overlooked as the main body of code is developed. It is noted that, unlike comments, Python 'docstrings' become a special attribute of the module, function, class, or method within which they are written [28] (and can therefore be accessed and processed in different ways).

Finally, it may be possible to utilise general-purpose languages with existing support for certain constraints. For example, the F# language [45] allows units of measure to be specified alongside variable types, e.g. `let radius : float<m> = 0.5<m>`. Units are checked at compile time and discarded at runtime, meaning no impact on runtime performance is expected [45]. However, in order to incorporate tolerances, connected collections or other domain-specific concepts; some kind of language extension or retrofitting is still likely to be required. This approach could also be highly disruptive if another general-purpose language, such as C#, is already preferred (although it is noted that F# and C# benefit from the interoperability of the Common Language Runtime [44]).

Ultimately, it is the *user-programmer* requirements and experience that should determine the type of approach.

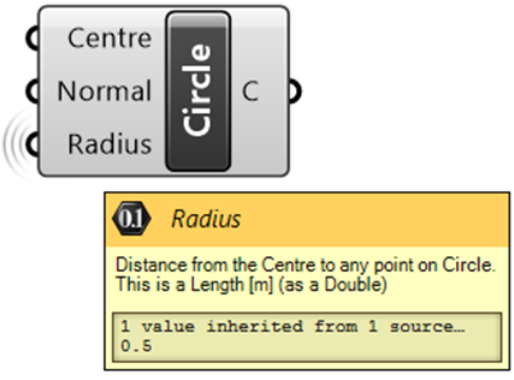

Figure 22. Example of enriched type semantics reflected to the end-user in the visual programming interface, in this case Grasshopper (integrated within Rhinoceros 3D modelling software). The code example given above, of the Circle Radius being defined as a Length type, is shown.

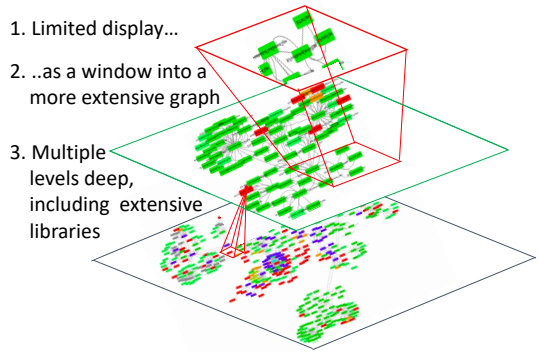

Figure 23. Composed of multiple visual representations from Cook [19]. The lower panel is a visualisation of a class hierarchy, from which the user may select different classes. The example class library here is part of Industry Foundation Classes [19, 34]. The middle panel illustrates a typically extensive visual data flow program (usually with an execution flow from left to right). The top panel illustrates a limited view into the program. This simplified diagram could be extended in practice to comprise multiple sub-views and levels.

9 User Experience: Options and Trade-offs

The current discussion is largely focused on the design of our proposed languages. However, users experience programming languages as part of a wider system which includes their programs, IDE, libraries and the classes they create.[9]

One of the main purposes of the complete system (of the language, program, library, display, as illustrated in Figure 23) is to allow the user to build a mental map of the program. This map necessarily has to extend to include those parts of the program and libraries which are not currently displayed. A mental map may have multiple points of interest, at different levels of depth and with different extents. Mental maps of programs may be useful in *optioneering*: Imagining a coordinated set of potential changes in multiple locations within the program structure and evaluating the necessary implementation strategies and consequences (prior to selecting and applying the most advantageous option).

Whether the user is aware or not, the use of the complete system (language, program, library, display) involves multiple trade-offs. These are summarised in Figure 24.

[9]There are a number of characteristics of libraries which may influence this user experience: (1) the depth of the inheritance hierarchy, (2) whether the inheritance hierarchy is a user-oriented classification or designed for execution efficiency, (3) the size of the semantic chunks of the instantiatable classes and (4) the compositional freedoms supported.

Understandably, a class library used for standardisation (such as the IFC) may require a deep, fine-grained classification to represent the generality of *what exists*. A different library structure may be more suited to represent what is *imagined*: what has *yet to exist*. For example, the BHoM library, offers less complex semantic chunks but substantially enhanced compositional freedoms allowing a lightweight, more expressive, approach better suited to exploratory design and programming.

visual *or* **text** based language/program

succinct *or* **expanded** notation

shallow + wide *or* **narrow + deep** program structure

extent *or* **legibility** of the displayed program

lay explanation *or* **professional navigation** purpose

Figure 24. Complex trade-offs in the choice of language, program structure and display.

The example of hybrid visual-textual programming in Figure 14 provides an interesting illustration of the trade-off between *extent* and *legibility*. Thus, for a given display size and resolution, it may not be possible for the whole program structure to be displayed and for the text to be legible. In the case of Figure 14, the layout of the visual program is a direct analog of a physical system (the MIPS processor), but this example also illustrates the more general point about the programmer's choice between width and depth. For example, an initial choice (wide and shallow) may have to be revised as the program becomes more complex.

Of course, many of the view-extent versus legibility trade-offs (and related navigational issues) can be addressed by established UI and windowing techniques. The more relevant question is: how can the design of the language present the user with a range of programming *usability* options (depth vs. extent vs. succinctness vs. legibility) and what choices and trade-offs does the user make.

Succinctness (Space-Efficiency). The transitional language proposed as part of the system of languages for Design Computation provides three levels of succinctness (or space-efficiency) (illustrated in Figure 12) as follows:

- visual data flow: with nodes overloaded for collections
- an intermediate notation: a succinct text-based notation where each program statement corresponds to a node in the visual program (therefore a data flow language with statements overloaded for collections);
- an expanded imperative text notation, indistinguishable from a formal text-based language, with explicit flow control statements.

With the method of *progressive formalisation*, each increase in the level of expanded notation may suit users with increasing level of computational competencies (see Figure 2). Therefore, a completely *fluid* system of languages would defer the choice of succinctness to the users. Such *fluidity* suggests that each of these programming notations should ideally be fully convertible to the other forms.

However, the succinctness of the intermediate notation in the transitional language has its limitations. In fact, the typography of the expanded language notation (with {} or indentation for nested code blocks) may be more easily understood than the (more succinct) intermediate notation.

In addition, a conditional statement embedded within an iterative construct (in the expanded notation) cannot be represented in the intermediate notation. Therefore, there may be limits as to how a program written in the expanded formal language notation can be transformed back into a succinct intermediate notation (and hence back into a visual language).

Why is this reversibility important? We envisage that future programming languages should support *lay explainability* as the complement of *progressive formalisation.* This is where a complex program can be visually summarised so that it can be understood by senior decision makers and lay participants with limited computational experience.

There is a long tradition of visual diagramming and flowcharting which can be harnessed here. Indeed, anticipating that the complete program is *explainable* in this way might become a key objective in program design. For example, the layout of the hybrid visual-textual programming example (Figure 14) has been designed so that nodes are gathered into clearly labelled groups (and these groups can be used as the basis for a simplified visual presentation). Other automated graph node clustering methods may also be applicable. So, while the direct reversibility of an expanded notation back into a more succinct notation may be not be feasible; the idea of being able to create a simplified presentation of a complex program is very relevant.

Crucially, the summary visual presentation used to *explain* the completed program to the lay participants may differ substantially from the visual *exploratory* program used at the start of the progressive formalisation process. The transformation between visual and text-based notations and the trade-offs (such as depth vs. extent vs. succinctness vs. legibility) are not fixed but may be continuously varying depending on the direction and purpose of the communication and the audience being addressed. The discussion here reminds us that *usability* is a key issue in language design, particularly for end-user programming.

10 Wider Context

Maver suggested that the role of computing is to 'amplify the intellect' [43]: To offer cognitive advantages through the provision of (computable) abstractions with which to think more precisely and to apply those thoughts more widely and more consistently.[10] Abstractions are a key aspect of design and of computation and therefore of Design Computation. Abstractions transform context-specific ideas into robust succinct generalisations. The ideas may be easy to understand in their original context but otherwise may be difficult to apply more generally. Abstractions may initially be more difficult to understand (a cognitive load) but subsequently may be more broadly applied and give more consistent results (providing important cognitive advantages).

In the context of Architecture and Engineering, *designing* can be thought of as an abstraction of *making*, *programming* as an abstraction of *designing*, and *language design* as an abstraction of *programming*, with each level of abstraction amplifying the preceding cognitive advantage.

Historically, a *craft* process combines the exploration of ideas and the physical realisation of those ideas into a single process. When craftsmanship reached the limits of complexity and effort, *designing* emerged as a distinct activity, using conventional media such as drawings and physical models. Designing offered a number of (cognitive) advantages. A drawing, as a simplified, light-weight representation of the salient features of the intended artefact allowed the artefact to be reasoned about without the time and cost of making the physical objects. For example, the Vasa [48] was built just at the point of transition between making and designing, when some ship builders were still directly building without drawings and others had switched to drawings.

In our own time, designing (with drawings, and other physical media) in turn reached the limits of complexity and effort. Computing has now replaced drawing as the principal medium of architectural expression, providing a new computable abstraction with cognitive advantages and further possibilities. Again, it might be interesting to identify a specific building which was being designed at this moment of transition. One such building is the Waterloo International Railway Station [2, 30]. The original design was conceived as a parametric design, with an underlying design rule which was intended to be executed with a series of different *driver* dimensions to create the resulting design model.

What is interesting is that the design architects were starting to think computationally before there were suitable applications and languages available. So this design logic could not be externalised as a program and executed. Instead, the architects were forced to manually interpret the result of this (internal) design logic as a series of static drawings.

Subsequently, one of the first Design Computation applications became available and we were able to help the architects to recover the underlying design rule and guide them as they used these design rules to explore numerous options that would have been prohibitively exhausting to draw with conventional computer graphics editors.

Now more recently, Design Computation using existing visual and text-based languages has arguably reached the limits of complexity and effort. In our experience, architectural ideas (and the related engineering and environmental analysis) have now reached a level of complexity where it is difficult for these to be expressed with the currently available programming language and without consuming valuable cognitive resources of the end-user programmers.

[10] Bronowski had observed that mechanical tools had amplified physical abilities, instruments had amplified the senses, media had amplified communication. Maver extended Bronowski's observations, adding that 'computing has the potential to amplify the intellect'.

We argue that much of this cognitive effort currently being consumed by the use of existing languages could be substantially reduced if there was a suitably well-crafted language which directly supported the key design and engineering abstractions. The cognitive resources saved could then be redeployed to help address the more complex design problems which are currently challenging to solve.

Essentially, we are anticipating a future moment of transition when new Design Computation languages will provide further cognitive advantages enabling further design and engineering possibilities. However, we cannot assume that new abstractions can be added to a programming language and that a *perfectly rational* programmer will use these features exactly as the language designer intended.

As we have observed, programming is a highly individual activity and computational behaviour (including programming) often involves single programmers making subjective trade-offs between short and long term costs and benefits, between continuing to use existing language features and the risks and potential rewards of exploring new features. The cognitive advantages of computing extends beyond the individual programmer to team collaboration and potentially to a wider public engagement. We are looking beyond the immediate advantages to the programmer which such a language might provide, so that the program logic, hence the design decisions, hence the consequential impact of the resulting building is *explainable* to wider stakeholders.

Future Design Computation is likely to be a conversation between human reasoning and machine intelligence, with each posing problems and explaining methods and solutions to the other. But what language will this conversation use? One which should naturally and precisely express domain-specific semantics, including units of measurement, uncertainty and tolerances, and physical and systems connectivity.

11 Conclusions

As we have discussed, a future language for end-user programming in Design Computation might usefully support:

- A continuous *educational trajectory* from simple visual programming to industry standard formal programming, by which an entry-level programmer can gain proficiency and confidence;
- *progressive formalisation*, in which the user's program can express human-readable domain semantics and structure, and support machine-readable verification;
- additional domain-specific functionality for units-of-measure, uncertainty, and connected collections;
- structuring the resulting program so that it is *explainable* to decision makers and lay participants.

While the motivation for these language features was initially a response to the particular needs of the Design Computation user community, many of these features, and indeed the underlying motivation, are quite universal. Therefore, it is hoped that the implementation of these features, initially in a domain-specific language, will lead to their wider adoption in general-purpose languages.

Design Computation combines two subjects with completely different evolutionary timelines–Computing is measured in decades, architecture in millennia. However, both are focused on the expression of underlying concepts, and both stand to benefit from the expressive powers of the other. In this essay, we have explored programming methods and constructs that may be of significant benefit to architecture; but concepts in architecture may provide equally significant opportunities for programming language research and development. By focusing on this 'common ground', we hope that new forms of expression may be introduced to benefit programmers, researchers and architects alike.

Inherent in Design Computation is a further opportunity: Design combines creative exploration of new architectural forms with an objective response to project-specific requirements and conformance to precise norms and statutory regulations. Just as a building has to unify these separate concerns, there may be an equivalent opportunity to unify exploratory and defensive programming in Design Computation.

Lastly, we may want to consider the wider impact of Design Computation languages. Recalling a presentation at SPLASH 2016, when Andrew Black quoted James Noble:

> "*Software is the most important infrastructure for... basically everything... Software is totally dependent on programming languages... (therefore) programming languages are the most important infrastructure for writing software... and thus for anything and everything!*"

How might we reply? We see Design Computation as the tangible connection between the computational infrastructure of programming languages and the physical infrastructure of the built environment.

Importantly, Design Computation directly amplifies and propagates the ideas of programming language research through the thousands of more precise DC programs written, the hundreds of thousands of more efficient buildings constructed and the well-being and fulfillment of the millions of people who use those buildings.

Acknowledgements

We thank the anonymous reviewers for insightful comments and suggestions. We also thank Andy King and Chris Leung for many fruitful conversations which fed into the ideas of this position paper and Amer Al-Jokhadar for permission to use the image in Figure 8. Part of this work was supported by the UKRI Centre for Doctoral Training in the Application of Artificial Intelligence to the study of Environmental Risks (reference EP/S022961/1). Orchard received support through Schmidt Sciences, LLC and acknowledges the support of the Institute of Computing for Climate Science at the University of Cambridge.

References

[1] [n. d.]. Personal communication with Chris Leung.

[2] Robert Aish. 1992. Computer-aided design software to augment the creation of form. In *Computers in Architecture: Tools for Design*. Longman, 97–104.

[3] Robert Aish, Al Fisher, Sam Joyce, and Andrew Marsh. 2012. Progress towards multi-criteria design optimisation using DesignScript with SMART form, robot structural analysis and Ecotect building performance analysis. In *Proceedings of ACADIA*. 47–56. https://doi.org/10.52842/conf.acadia.2012.047

[4] Robert Aish and Sean Hanna. 2017. Comparative evaluation of parametric design systems for teaching design computation. *Design Studies* 52 (2017), 144–172. https://doi.org/10.1016/j.destud.2017.05.002

[5] Robert Aish, Wassim Jabi, Simon Lannon, Nicholas Mario Wardhana, and Aikaterini Chatzivasileiadi. 2018. Topologic: Tools to explore architectural topology. In *AAG 2018*. 316–341.

[6] Robert Aish and Emmanuel Mendoza. 2016. DesignScript: a domain specific language for architectural computing. In *Proceedings of the International Workshop on Domain-Specific Modeling, DSM@SPLASH 2016, Amsterdam, Netherlands, October 30, 2016*. ACM New York, NY, USA, 15–21. https://doi.org/10.1145/3023147.3023150

[7] Robert Aish and Robert Woodbury. 2005. Multi-level interaction in parametric design. In *International symposium on smart graphics*. Springer, 151–162. https://doi.org/10.1007/11536482_13

[8] Amer Al-Jokhadar. 2018. *Towards a socio-spatial parametric grammar for sustainable tall residential buildings in hot-arid regions learning from the vernacular model of the Middle East and North Africa*. Ph. D. Dissertation. Cardiff University. https://orca.cardiff.ac.uk/id/eprint/111874/

[9] Christopher Alexander. 1967. A City is Not a Tree. *Ekistics* 23, 139 (1967), 344–348.

[10] Autodesk. [n. d.]. Learn - Dynamo BIM. https://dynamobim.org/ Accessed 25th April 2024.

[11] The World Bank. 2013. *Measuring the Real Size of the World Economy*. Technical Report. The World Bank.

[12] Barbara Rita Barricelli, Fabio Cassano, Daniela Fogli, and Antonio Piccinno. 2019. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software* 149 (March 2019), 101–137. https://doi.org/10.1016/j.jss.2018.11.041

[13] BHoM. 2018. The Buildings and Habitats object Model. https://bhom.xyz. Accessed 24th April 2024.

[14] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind*. Springer, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/3-540-44617-6_31

[15] André Borrmann, Markus König, Christian Koch, and Jakob Beetz. 2018. Building Information Modeling: Why? what? how?. In *Building Information Modeling: Technology Foundations and Industry Practice*. Springer, 1–24. https://doi.org/10.1007/978-3-319-92862-3_1

[16] I. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. 1995. Scaling up visual programming languages. *Computer* 28, 3 (March 1995), 45–54. https://doi.org/10.1109/2.366157

[17] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G Siek. 2019. Gradual typing: a new perspective. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32. https://doi.org/10.1145/3290329

[18] Alexis Cheron. 2024. *Connected Collections: Implementing an Automatically Populated Reverse Map of Related Objects*. MSc Dissertation (unpublished). University of Kent.

[19] Alastair Cook. 2024. Industry Data Model Change, Industry Foundation Classes. Presentation. https://doi.org/10.5281/zenodo.12682064

[20] René Dybkaer. 2010. ISO terminological analysis of the VIM3 concepts 'quantity' and 'kind-of-quantity'. *Metrologia* 47, 3 (2010), 127. https://doi.org/10.1088/0026-1394/47/3/003

[21] Jonathan Edwards, Stephen Kell, Tomas Petricek, and Luke Church. 2019. Evaluating programming systems design. In *PPIG 2019, 28-30 Aug 2019 (unpublished)*. Newcastle upon Tyne, United Kingdom. https://kar.kent.ac.uk/id/eprint/79905

[22] Diellza Elshani, Alessio Lombardi, Al Fisher, Steffen Staab, Daniel Hernández, and Thomas Wortmann. 2022. Knowledge Graphs for Multidisciplinary Co-Design: Introducing RDF to BHoM. In *LDAC@ESWC*. 32–42.

[23] Gustav Fagerstrom, Erik Verboon, and Robert Aish. 2014. Topo-façade: Envelope design and fabrication planning using topological mesh representations. In *Fabricate 2014: Negotiating Design and Making*.

[24] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

[25] Marco Gaboardi, Suresh Jagannathan, Ranjit Jhala, and Stephanie Weirich. 2016. Language based verification tools for functional programs (dagstuhl seminar 16131). *Dagstuhl Reports* 6 (2016). Issue 3. https://doi.org/10.4230/DagRep.6.3.59

[26] BHoM (GitHub). [n. d.]. LengthAttribute.cs. https://github.com/BHoM/BHoM/blob/main/Quantities_oM/Attributes/Length.cs. Accessed 25th April 2024.

[27] BHoM (GitHub). [n. d.]. QuantityAttribute.cs. Available at https://github.com/BHoM/BHoM/blob/main/Quantities_oM/Attributes/Abstract/QuantityAttribute.cs. Accessed 25th April 2024.

[28] David Goodger and Guido van Rossum. [n. d.]. PEP 257 – Docstring Conventions. https://peps.python.org/pep-0257/ Accessed 25th April 2024.

[29] Hernan Grecco and Pint Developers. [n. d.]. Pint: makes units easy - pint documentation. https://pint.readthedocs.io/en/stable/ Accessed 25th April 2024.

[30] Grimshaw. [n. d.]. International Terminal Waterloo. https://grimshaw.global/projects/rail-and-mass-transit/international-terminal-waterloo/ Accessed 18th July 2024.

[31] Felienne Hermans. 2020. Hedy: a gradual language for programming education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 259–270. https://doi.org/10.1145/3372782.3406262

[32] Frédéric Imbert, Kathryn Stutts Frost, Al Fisher, Andrew Witt, Vincent Tourre, and Benjamin Koren. 2013. Concurrent geometric, structural and environmental design: Louvre Abu Dhabi. In *Advances in architectural geometry 2012*. Springer, 77–90. https://doi.org/10.1007/978-3-7091-1251-9_6

[33] ISO. 2020. ISO 23952:2020 Automation systems and integration — Quality information framework (QIF) — An integrated model for manufacturing quality information. https://www.iso.org/standard/77461.html

[34] ISO. 2024. ISO 16739-1:2024 Industry Foundation Classes (IFC) for data sharing in the construction and facility management industries—Part 1: Data schema. https://www.iso.org/standard/84123.html

[35] Christian Johansen, Tore Pedersen, and Audun Jøsang. 2016. *Reflections on behavioural computer science*. Technical Report. University of Oslo. https://www.duo.uio.no/handle/10852/57451

[36] Gilles Kahn and David Macqueen. 1976. *Coroutines and Networks of Parallel Processes*. Research Report. https://inria.hal.science/inria-00306565

[37] Andrew Kennedy. 1994. Dimension types. In *Programming Languages and Systems — ESOP '94*, Donald Sannella (Ed.). Springer, Berlin, Heidelberg, 348–362. https://doi.org/10.1007/3-540-57880-3_23

[38] Andrew Kennedy. 2010. Types for Units-of-Measure: Theory and Practice. In *Central European Functional Programming School: Third*

Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures, Zoltán Horváth, Rinus Plasmeijer, and Viktória Zsók (Eds.). Springer, Berlin, Heidelberg, 268–305. https://doi.org/10.1007/978-3-642-17685-2_8

[39] Amy J Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, et al. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 1–44. https://doi.org/10.1145/1922649.1922658

[40] Jason Lim. 2016. *YOUR: Robot programming tools for architectural education*. Ph. D. Dissertation. ETH Zürich. https://doi.org/10.3929/ethz-a-010748012

[41] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15. https://doi.org/10.1145/1868358.1868363

[42] Stefan Marr and Benoit Daloze. 2018. Few versatile vs. many specialized collections: how to design a collection library for exploratory programming?. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, Stefan Marr and Jennifer B. Sartor (Eds.). ACM, 135–143. https://doi.org/10.1145/3191697.3214334

[43] T. W. Maver. 1984. What is eCAADe? *The Third European Conference on CAD in the Education of Architecture [eCAADe Conference Proceedings] Helsinki, Finland* (Sept. 1984). https://doi.org/10.52842/conf.ecaade.1984.x.d0s

[44] Microsoft. 2023. Common Language Runtime (CLR) overview. https://learn.microsoft.com/en-us/dotnet/standard/clr Accessed 25th April 2024.

[45] Microsoft. 2023. Units of Measure - F#. https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure Accessed 25th April 2024.

[46] Ramon E Moore. 1979. *Methods and applications of interval analysis*. Society for Industrial and Applied Mathematics.

[47] Giovanni Moroni, Stefano Petro, and Wilma Polini. 2017. Geometrical product specification and verification in additive manufacturing. *CIRP Annals* 66, 1 (2017), 157–160. https://doi.org/10.1016/j.cirp.2017.04.043

[48] Lisa Mullins and Rhitu Chatterjee. 2013. New Clues Emerge in Centuries-Old Swedish Shipwreck. *The World* (2013). https://theworld.org/stories/2013/08/15/new-clues-emerge-centuries-old-swedish-shipwreck/ Accessed 18th July 2024.

[49] Brad A. Myers. 2002. Making Programming Easier by Making it More Natural. *Presentation at first EUD-Net workshop in Pisa, Italy* (2002). Slides available at: http://hiis.isti.cnr.it/projects/EUD-NET/pisa.htm.

[50] Brad A Myers, Amy J Ko, and Margaret M Burnett. 2006. Invited research overview: end-user programming. In *CHI'06 extended abstracts on Human factors in computing systems*. ACM New York, NY, USA, 75–80. https://doi.org/10.1145/1125451.1125472

[51] David B Newell and Eite Tiesinga. 2019. *The international system of units (SI): 2019 edition*. Technical Report NIST SP 330-2019. National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.SP.330-2019

[52] Kristen Nygaard and Ole-Johan Dahl. 1978. The development of the SIMULA languages. In *History of Programming Languages*. ACM, 439–480. https://doi.org/10.1145/800025.1198392

[53] University of Cambridge and University of Kent. [n. d.]. CamFort: Automated evolution and verification of computational science models. https://camfort.github.io/ Accessed 25th April 2024.

[54] Dominic Orchard. 2011. The four Rs of programming language design. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, Portland Oregon USA, 157–162. https://doi.org/10.1145/2089131.2089138

[55] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30. https://doi.org/10.1145/3341714

[56] Dominic Orchard, Andrew Rice, and Oleg Oshmyan. 2015. Evolving Fortran types with inferred units-of-measure. *Journal of Computational Science* 9 (2015), 156–162. https://doi.org/10.1016/j.jocs.2015.04.018

[57] George Orwell. 2021. *Politics and the English language and other essays*. epubli.

[58] Alan J Perlis. 1982. Special feature: Epigrams on programming. *ACM Sigplan Notices* 17, 9 (1982), 7–13. https://doi.org/10.1145/947955.1083808

[59] Satheesh Pv. 2021. Introduction to Blueprints. In *Beginning Unreal Engine 4 Blueprints Visual Scripting: Using C++: From Beginner to Pro*, Satheesh Pv (Ed.). Apress, Berkeley, CA, 21–31. https://doi.org/10.1007/978-1-4842-6396-9_2

[60] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199. https://doi.org/10.1016/j.infsof.2013.01.008

[61] John C Reynolds. 2000. The meaning of types from intrinsic to extrinsic semantics. *BRICS Report Series* 7, 32 (2000). https://doi.org/10.7146/brics.v7i32.20167

[62] Richard David Rush et al. 1986. *The building systems integration handbook*. Wiley.

[63] David Rutten. 2013. Grasshopper Masterclass With David Rutten. Video Series. https://vimeopro.com/rhino/grasshopper-masterclass-with-david-rutten

[64] Mary Shaw. 2022. Myths and mythconceptions: What does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages* 4, HOPL (2022), 1–44. https://doi.org/10.1145/3480947

[65] Chris Shrubshall and Al Fisher. 2011. The Practical Application of Structural Optimisation in the Design of the Louvre Abu Dhabi. In *35th Annual Symposium of IABSE/52nd Annual Symposium of IASS/6th International Conference on Space Structures, London, United Kingdom, September 2011*.

[66] Aaron Sloman. 2011. *TEACH PRIMER – AN OVERVIEW OF POP-11 (Fourth Edition 2011 - For Poplog V15.6.4 30 Oct 2011)*. University of Birmingham. https://www.cs.bham.ac.uk/research/projects/poplog/primer.pdf

[67] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[68] Arturo Tedeschi. 2010. *Parametric architecture with Grasshopper : primer*. Le Penseur. https://cir.nii.ac.jp/crid/1130000796360313216

[69] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 132–141. https://doi.org/10.1145/1993498.1993514

[70] United Nations Environment Programme. 2022. *2022 Global Status Report for Buildings and Construction: Towards a Zero-emission, Efficient and Resilient Buildings and Construction Sector*. Technical Report. https://www.unep.org/resources/publication/2022-global-status-report-buildings-and-construction

[71] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013 (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). 209–228. https://doi.org/10.1007/978-3-642-37036-6_13

[72] Christopher JK Williams. 2001. The analytic and numerical definition of the geometry of the British Museum Great Court Roof. In *Mathematics & design 2001*. Deakin University, 434–440. https://researchportal.bath.ac.uk/en/publications/the-analytic-and-numerical-definition-of-the-geometry-of-the-brit