Patent US 6851 104 B1: Awarded Feb 1st 2005, Submitted 2000

System and Method for Programming Using Independent and Reusable Software Units.

Patent Description

This patent describes a way to connect up independent software building blocks (components) to create composite components, establishing a compositional hierarchy (claims 1, 2). At the lowest level, the primitive blocks each consist of a single method (a function accepting input and providing output). Claim 9 then describes a way to inherit structure when defining new components by adding/deleting/replacing inherited elements for reuse.

Claims 3-8 describe how values flow along connectors between the building blocks from input to output gates. This is known as a dataflow paradigm.

I use a variant of (claims 1, 2, 9), but not (claims 3-8). The basic defence I am using against this patent is to demonstrate that all of its structures and techniques were comprehensively described in academic and industrial literature in the 1990s and have been in industrial use for nearly 20 years. My work does nothing different in this regard and tellingly, the patent references none of these works.

Dataflow versus Service Paradigm

The (claims 1, 2, 9) are general enough to describe the basis for part of my work. However, (claims 3-8) implement what is known as a dataflow paradigm where values propagate along connectors in a prescribed order, from input to output gates. My work does not use this, instead using a service paradigm where a client invokes a service which then returns a result, which does not have to trigger an output propagation.

The Darwin system (below) is general enough to implement the dataflow paradigm also. In other words, prior art is a generalised superset of the claims 3-8 also.

As I do not specifically use this paradigm, I do not focus deeply on claims 3-8.

Background of the Patent and Prior Work

The academic field of computer science that covers this area is architecture description languages (ADLs). An ADL describes how to connect up components in a structural fashion, and keeps the structure of a system separate from the logic of its computational units. This is an “electronics circuit board and chips” metaphor applied to software. Applying this at multiple levels creates a compositional hierarchy.

A key advance of ADLs over earlier work is the presence of explicit connectors, which wire up component instances together, like tracks in a circuit board.

Some background of my expertise in this area: I have a PhD from Imperial College, London in the area of ADLs and software engineering, and 20 years of industrial experience before that. My PhD ran from 2005 to late 2009, and I started it after reading one of the Darwin papers from Professors Jeff Magee and Jeff Kramer, who would later become my supervisors. They can lay claim to starting the field of ADLs, and they have published (and implemented) widely from the late 1980's to this present day on this subject.

The ADL they created was called **Darwin**, and it is a superset of the patent description. The techniques were later used by Philips for the software for some of their television products. It also influenced Microsoft's COM system which is present on every Windows computer. This was all done in the 1990s.

My Work

The system I created is in 2 parts. The first part is an ADL called **Backbone**. For Backbone I am primarily concerned about claims 1, 2 and 9. The second part is a graphical modelling tool called **Evolve** which uses Backbone. The techniques used by Evolve are touched on in the patent description, but not in the claims themselves.

Overview of Prior Art

The papers and books I will use are as follows, and full details of these references are at the end of the document. There are many others but I have tried to keep the list small and focussed.

[REX] 1992
This describes Darwin, an ADL for configuring distributed systems, and its runtime executive called Rex. I will show that it describes a superset of patent claims 1, 2 and is general enough to describe the more specific claims of 3-8.

Distributed systems are a superset of the non-distributed systems that the patent addresses. The patent also alludes to its applicability for distributed systems in "XVII. Message Breaking". Further, [KOALA] is a variant of Darwin used at Philips showing the technique also works well for non-distributed systems.

[FORMAL] 1995
This is a formal description of Darwin, which describes the theoretical underpinnings of the hierarchical ADL approach. I have included this because the patent uses a formal description and some of the diagrams in the patent are virtually identical to diagrams in this paper.

[ROOM] 1994
This describes ROOM, an ADL and graphical modelling toolset which was sold to industry by a company called Objectime for developing real-time systems. It uses a Darwin-like model, but was the first to introduce structural inheritance (claim 9). ROOM provided a comprehensive graphical modelling tool. The ROOM program suite was sold to Rational Software in the 1990s and subsequently purchased by IBM.

Ancillary documents (covering the patent body rather than the claims per se):

[KOALA] 1996
This describes a Darwin variant called **Koala**, create at Philips to produce software for their

televisions. This shows that the approach was being used in an industrial setting to create non-distributed programs well before the patent was lodged.

[EVOLVING] 1990
This is a description of an evolution approach in the precursor of Darwin, covering the general claim in the patent body regarding runtime software evolution. The patent uses almost identical operations to describe runtime structural evolution.

[SAA] 1998
This describes a Darwin-based graphical modelling tool which covers the modelling tool that the patent body claims would be possible using its techniques. Evolve uses the same techniques.

[BACKBONE] 2006
My paper, describing my ADL called **Backbone**. This paper led the patent submitter to contact me to inform me of the similarities to his patent. He specifically drew my attention to the fact that resemblance (my concept) and structural inheritance (from his patent) are almost identical.

Terminology

The patent uses a number of unusual terms which are not standard. I will use the standard terms.

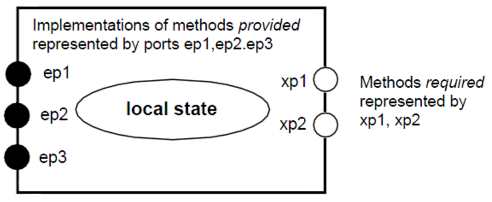
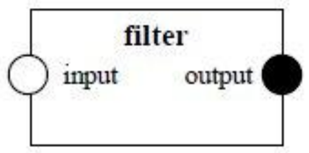
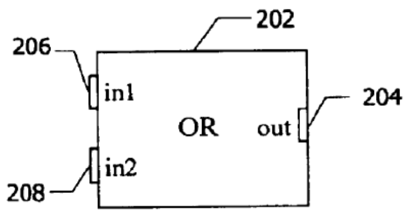
Standard Terms for ADLs	Patent Terms
Component	Connecton, sometimes Block
Component instance	Connecton
Connector	Link or Channel
Provided or Required Service, also Port	Gate

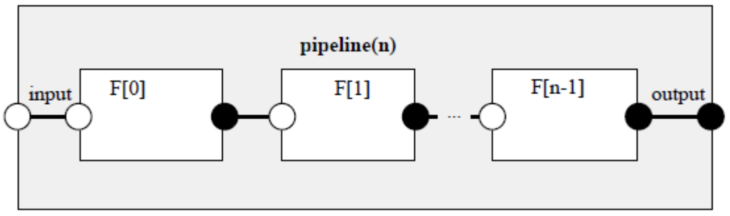
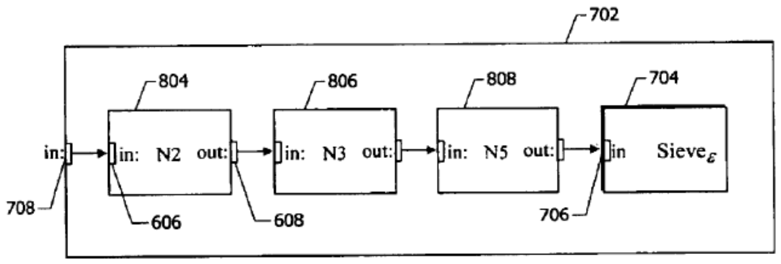
The lack of distinction in the patent between component and component instance is limiting, and the use of the word “connecton” which looks very similar to “connector” is confusing. I will use the standard terminology as far as possible.

Analysis of Patent Claims

Patent claim	Comments and Prior Art
Claim 1	This claim covers the “static” structure of the system. i.e. the design of the components and the connectors between the component instances.
1 . A software ensemble stored on a computer readable medium and executable by a computer, the software ensemble comprising:	This phrase describes a general computer program.
A plurality of software units each software unit of said plurality of software units including	This is quite general, and refers to the collection of components and component instances making up the structure of a hierarchical component model. This is the basis of Darwin described in [REX] and [FORMAL].

a method and data:	
An executive software unit including links between said plurality of software units	This refers to connections between component instances. Described in [FORMAL] in section 2: "Darwin allows distributed programs to be constructed from hierarchically structured specifications of the set of component instances and their **interconnections**". The use of connectors is perhaps the defining feature of ADLs. Described also in detail in [REX].

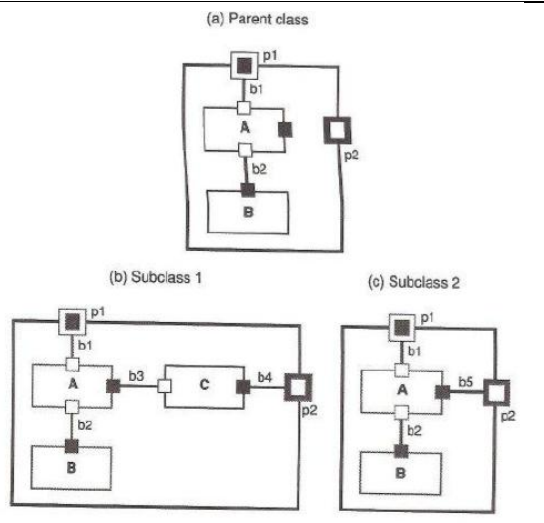
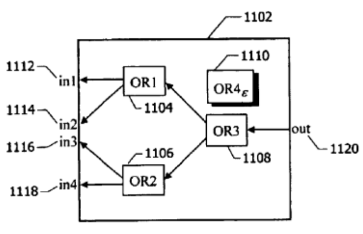
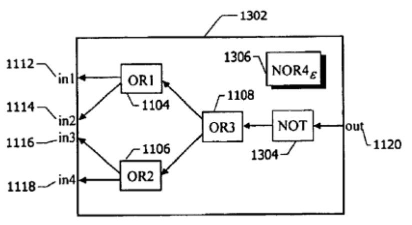
Wherein a software unit of the software ensemble is described by a model M, given by
 	
M = (inGates, {inSign},{a}, Q, q, outgates, {outSign}, {outFunction}
	NOTE: {} indicates a set of elements.
	
	This says that the system comprises a number of method-level components {a}, each of which has an initial state. Each component specifies the parameters it accepts as input {inSign} and what is produces as output {outSign}. The system starts with a state q (the data values), and each function in the set of {a} has a specified set of input parameters as a signature {inSign} and a set of output parameters {outSign}.
	
	This is the same as the Darwin model. [REX] figure 4 shows a primitive component with ports (gates) with provided (input) / required (output) polarity. This has local state as per q in the patent description.
	
	 Implementations of methods *provided* represented by ports ep1,ep2.ep3
	ep1
	ep2
	ep3
	local state
	xp1
	xp2
	Methods *required* represented by xp1, xp2
	Figure 4. - An Object with explicit interfaces
	
	Also, compare and contrast figure 1 of [FORMAL] with figure 2 of the patent. They both describe the same sort of entity.
	
	Figure 1 from [FORMAL] showing a primitive component:
	
	
	component filter{
	provide output<stream char>;
	require input<stream char>;
	}
	
	Figure 2 from the patent showing a primitive "connecton":
	
	
	
	Let us now turn to composite structures:
	
	[FORMAL] figure 2 shows how component instances are connected

	together to make a composite component. This is the foundation of the hierarchical approach. Compare and contrast with figure 8 in the patent, they are almost identical. Figure 2 from [FORMAL] showing a composite component:  Figure 8 from the patent showing a composite "connecton":  702 804 806 808 704 in: in: N2 out: in: N3 out: in: N5 out: in Sieve 708 606 608 706 These are slightly different examples, hence the lack of an "output" in the latter figure. However, it makes the point that the patent describes a subset of the pre-existing Darwin language. In other words, the Darwin configuration language can model the patent system completely. Like the patent, Darwin initially worked at the method level also. I am going to withhold my opinion about whether the patent submitter actually knew about Darwin and failed to reference it. The examples are eerily similar.
Where inGates...	Just describes the above terms.
Claim 2	This builds on claim 1, using similar language, and describes the runtime state of the program. i.e. the changeable state while running, which starts as per the initial design (claim 1) which can then deviate.
2 . A software ensemble stored on a computer readable medium and executable by a computer, the software ensemble comprising:	As before, just describes a general computer program in all its forms.
A plurality of software units each software unit of said plurality of software units including a method and data:	As before.
An executive software unit including links between said plurality	As before.

of software units	

Wherein the software ensemble, E, is described by E = (inGates, {inSign}, ∈, M, outgates, {outSign}, {outFunction})	This logic description references M of claim 1. It is basically saying that a running program has an ensemble executive ∈ that keeps track of the (possibly changing) structure of the system
	
	The ensemble executive is a reference to the runtime of the system which manages the configuration of component instances and connections. Such a runtime is described in [EVOLVING] as the configuration manager. The purpose of this was to track the connections during runtime, as well as other aspects.
	
	The operations established by [EVOLVING] for the executive are described in section B of that paper as:
	
	`create(component instance), remove,`
	`link, unlink`
	
	In an almost identical fashion, the patent in section XVIII describes the operations as:
	
	`addConnecton(component instance), removeConnecton, replaceConnecton`
	`link, unLink`
	
	Perhaps the only difference is in the use of replace, which in Darwin was synthesised by a delete/add.
	
	[REX] describes a variant of the Darwin runtime called Rex. [FORMAL] describes a later, improved version, called Regis.

Where ∈ is...	Just describes the above terms.
Claim 9	Using structural inheritance to reuse existing components
The software ensemble according to claim 2, wherein structural inheritance is utilized to build new software ensembles from existing software ensembles.	This claim is directly relevant to my [BACKBONE] work, as I define a similar construct called resemblance, which allows for structural inheritance. The notion of structural inheritance was first developed in [ROOM] and was a key selling point of the ROOM toolset. ROOM was an ADL where structure could be inherited and added to, deleted or overridden (replaced). The patent description of structural inheritance is almost identical to the description in the [ROOM] book figure 9.5. A similar example is used in the patent in figures 11 and 13. [ROOM] figure 9.5 showing structural inheritance of 2 subclasses of a parent class:

	 (a) Parent class p1 b1 A p2 b2 B (b) Subclass 1 b3 C b4 (c) Subclass 2 b5 Patent figures 11 and 13 showing structural inheritance of a subclass (13) from a parent class (11). Figure 13 adds the NOT instance and 2 connectors, in the same way that figure (b) above adds the C instance and b3/b4 connectors.  1102 1110 1112 in1 OR1 OR4ε 1104 1114 in2 in3 1116 OR3 out 1120 1106 1108 1118 in4 OR2 Fig. 11  1302 1306 NOR4ε 1112 in1 OR1 1104 1108 1114 in2 in3 1116 OR3 NOT out 1120 1304 1106 OR2 1118 in4 Fig. 13 The patent only allows for single inheritance, as per the Smalltalk inheritance approach. ROOM also had this exact limitation. This is perhaps not surprising – both ROOM and the patent describe Smalltalk systems By way of contrast, my work allows for multiple inheritance, and also deals with the complex conflicts that can result.

Claims 3-8 are not really applicable to my work, as they describe the dataflow paradigm. However, Darwin is general enough to be able to model this approach also.

Claims 3, 5	Describes how the system handles the propagation of a single output to many output paths via an order function. Neither Darwin nor my work explicitly uses this feature. If pushed, you could model this paradigm in Darwin, however, showing Darwin to be a generalised superset of the claims in the patent. In other words, Darwin is a general approach which can model a number of paradigms. The patent claims 3, 5 are one example of this. I therefore will not analyse this further. Depends on claim 2.

Claim 4	Describes a filter and reverse filter applied to each message as it transforms through a connector. These can be modelled as separate instances in the general Darwin model. My work does not use this feature. Depends on claim 3.
Claim 6	Indicates that the structure of the components and their interconnections dictate the messages sent to the output gates. Neither Darwin or my work use this.
The software ensemble of claim 3, wherein the output gate of a software unit and the channels determine the messages to be sent through the output gates of the software ensemble.	This doesn't really apply to Darwin or my work, because the messages to be sent are not so prescriptive in ADLs. The logic inside Darwin primitive components determine the messages to be sent rather than the direct connections. It would be possible to force this prescriptive order in Darwin, by implementing the primitive components according to dataflow principles.
Claim 7	This is saying that if you invoke a service along a connector, it will invoke the appropriate method. Again, it is describing it in terms of data flow rather than generalised method calls. Darwin doesn't work this way, and I feel it would be difficult to argue that this applies to Darwin or my work at current.
Claim 8	This is saying that the result of an action propagates along an output connector. Again, a dataflow paradigm. In contrast to this, Darwin does not have a hardcoded notion of output connectors, and actions do not result in value propagation along any connectors unless the primitive component instance decides to do so.

In summary, the claims can be broadly divided into the following categories:

a. Claims 1 & 2 which define hierarchical component structures with connectors. This is covered by Darwin [REX] [FORMAL], the original ADL which established the field and Regis the Darwin runtime executive. Koala [KOALA] and ROOM [ROOM] are other examples which were used in industry well before the patent filing date of 2000.

b. Claim 9 which describes the use of structural inheritance to effect component reuse by allowing connectors and component instances to be added, deleted or replaced in a subclass. This technique is identical to that described [ROOM] and was in common industrial use in the early 1990s and beyond.

c. Claims 3-8 which describe a dataflow paradigm where values flow prescriptively along connectors. Darwin and other ADLs instead use a service paradigm which does not follow this model, although Darwin is general enough to model this paradigm also if required.

References

[REX] 1992
Configuring Object Based Distributed Programs in REX
Jeff Kramer, Jeff Magee, Morris Sloman, Naranker Dulay
Distributed Computing Systems, pp.187–205, IEEE Computer Society Press

[FORMAL] 1995
Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. 1995.
Specifying Distributed Software Architectures.
In Proceedings of the 5th European Software Engineering Conference.
Lecture Notes In Computer Science, vol. 989. Springer-Verlag, London, 137-153.

[ROOM] 1994
Real-Time Object-Oriented Modeling
Bran Selic, Garth Gullekson, Paul T. Ward.
John Wiley & Sons; 1 edition (April 22, 1994), ISBN-10: 0471599174

[KOALA] 1998
Koala, a Component Model for Consumer
van Ommering, R.
Electronics Product Software,
In Development and Evolution of Software Architectures for Product Families, LNCS 1429

[EVOLVING] 1990
The Evolving Philosophers Problem: Dynamic Change Management.
Kramer, J. and Magee, J.
IEEE Trans. Softw. Eng. 16, 11 (Nov. 1990), 1293-1306.
DOI= http://dx.doi.org/10.1109/32.60317

[SAA] 1995
The Software Architect's Assistant-a visual environment for distributed programming.
Ng, K., Kramer, J., Magee, J., and Dulay, N.
In Proceedings of the 28th Hawaii international Conference on System Sciences (Jan 04 - 07, 1995).
HICSS. IEEE Computer Society, Washington, DC, 254.

[BACKBONE] 2006
Using Resemblance to Support Component Reuse and Evolution
A. McVeigh, J. Kramer, J.Magee
In Proceedings of Structure and Verification of Component Based Systems, Portland Oregon, 2006