A LEGO-like Lightweight Software Component Architecture for Organic Computing

Thomas Schöbel-Theuer, University of Stuttgart

Abstract: The goals of organic computing are difficult to achieve due to the heterogenity of current software components. We believe that in the long term more advanced software architectures are required, in particular "LEGO[1]-like" systems with a small number of interface types and maximum combinability of component instances. High combinability is the key for automated reasoning on and controlling of organizational structures of organic systems (e.g. self-describing systems), and automated reasoning is in turn the key for achieving organic properties such as self-configuration, self-optimizing, self-healing, and so on.

1 Introduction

Self-configuration is one of the main goals of organic computing, among others. First we will make clear that self-configuration is a special case of plain automatic configuration. When a system A is to be configured automatically, some identifiable system B must be responsible for that. Otherwise we have no means to deal with the configuration task in a formalized way. When B is a subsystem of A, we speak of self-configuration. Otherwise, we have a more general configuration system B for system A. In this view, self-configuration is nothing more than making sure that B is some subset or subpart of A.

Thus we split the problem into two parts: (1) build a system B which can (re)configure system A, and (2) ensure that B is a (not necessarily true) subsystem of A. Then A can (re)configure "itself", on behalf of its subsystem B.

In this position paper, we will first discuss property (1) with respect to the needs of organic computing. We will develop recursive means for handling configuration. Property (2) then becomes a special configuration of a solution for property (1).

Automation of configuration tasks means dealing with compositional structures and properties of components. A major issue of current component architectures is *composability*. Current OO design methods often lead to system designs with dozens or even hundreds of class interfaces, and in turn of component interface types. In our opinion, a large number of interface types is counter-productive for composability.

We believe that the success of the LEGO toys [LEG] can teach us another lesson: *uniform interfaces* (or at least a small number of at least partly compatible interfaces) are not just a nice feature of a component architecture, but *fundamental* for composability. In other

[1]Brand names like LEGO and Fischertechnik are the property of their respective owners.

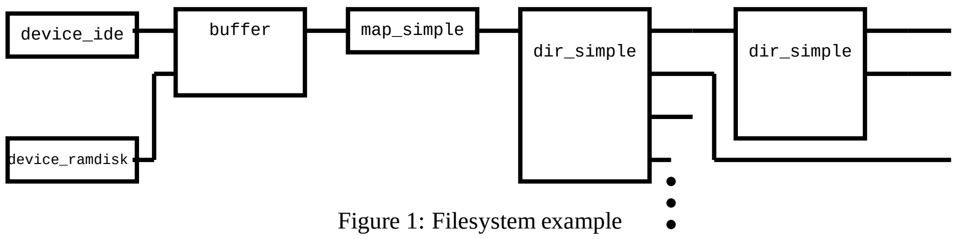


Figure 1: Filesystem example

words, uniformity of interfaces is a *must* for achieving high composability. In the area of mechanics, this is intuitively clear, and the LEGO system is demonstrating the truth of it. Although there exist hundreds of types of LEGO bricks and very complex items like dolls, there is basically only one interface type having the shape of a tenon. Although the corresponding tapholes may have different shapes, their *functionality* of connecting to the same type of tenon is always the same. At a few very special places, other types of interfaces are also used, but a close look at them reveals that they are at least *partly* compatible with the standard interface.

One might argue against the LEGO system that functionality could be reduced by uniformity of interfaces. Other toy brick systems like Fischertechnik [ft] can create higher functionality, but they also use a very low number of basic interface types. So this means that the interfaces have to be carefully designed to be *universal,* at least "universal enough" for achieving the goals of the system. Fischertechnik addresses a different application area from LEGO.

We believe that the problem of functionality can be overcome in many places by the concept of *universal genericity of interfaces* [ST03b]. The LEGO system is a good example for *compositorical genericity* [ST03b] based on a universally generic interface. Whenever the overall functionality is achieved by composition of standard bricks, the interface can be lean and simple, as long as it is universally generic.

In many presentations on component software, the *puzzle* has been used as a metaphor. However, the puzzle is nearly the opposite of a LEGO system: any two pieces which don't belong together will not fit, i.e. have non-compatible shapes (at least in many cases). The pieces of a puzzle can be put together only in *one single* way (although in many different orders), forming one single end result - and thus the system is not compositorically generic. Splitting a software system into components according to the puzzle metaphor is not suitable for organic computing - it is just too inflexible to allow for (self-)configuration, and does not support context awareness and anticipatory behaviour on the structural level.

2 Some Domain Engineering Requirements of LEGO-like Systems

Like any software system, there are universal requirements for organic computing like correctness, efficiency (in both time and space), usablility, and so on. Since a LEGO-like system addresses a large application domain, the requirements will be rather "generic" (domain engineering instead of requirements engineering). For general organic computing, there are well-known requirements like self- and context-awareness which we don't want to repeat here.

One of the largest problems the goals of organic computing is the heterogenity of current hardware and software components already in use. Many components must be either *altered* to become suitable for a LEGO-like system, or *adapted*, or *re-implemented*. In some cases, the cost for re-engineering of the problem domain and re-implementation could be lower than that for modification or adaptation.

The interface type of the LEGO-like system must be universally generic enough to allow for representation of both active and passive resources of the system, as well as for a suitable abstract representation of the *structure* of the system.

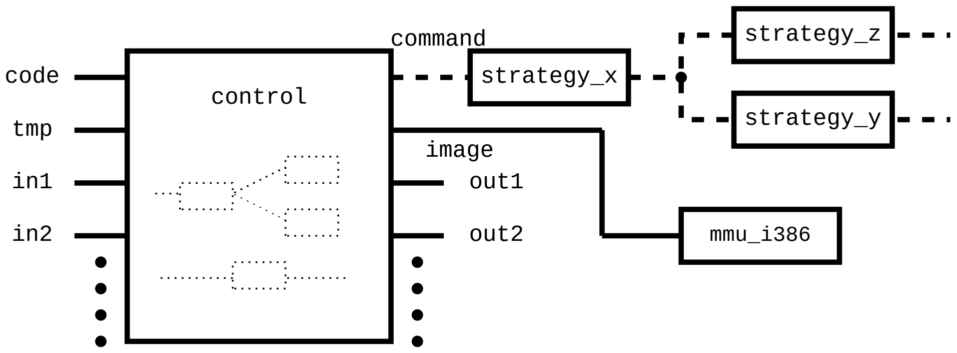
It should be possible to express the subsystem components for self-management, self-organization, self- and context-awareness also in the LEGO-like system (recursively). Otherwise the orthogonality of concepts could be violated, and self-management of the management components would be either impossible or require different concepts.

3 An Example LEGO-like Architecture

In [ST02, ST03a, ST03b], we have proposed a LEGO-like lightweight component architecture for (distributed) operating systems. It could be characterized as a "pipes and filters style" in the sense of software architecture [SG96]. Due to its universal genericity, it should be also suitable for database systems [ST03c], and perhaps other application areas. We are currently working on an operating system prototype called ATHOMUX. The basic infrastructure and the filesystem part are already working. Preliminary performance measurements on its basic operations show results in the same order as roughly comparable basic Linux services, or even better.

The wires in the example (figure 1) may be viewed as "transportation channels" which *logically* transport instances of a memory abstraction called "nest". The boxes are called "bricks" and may be characterized as "transformers" between nest instances. Brick instances are depicted with inputs on their left and outputs on their right, similar to functional units in electrical engineering or automation control [IEC]. Wires are directionally drawn from left to right. By dynamic wiring, we implement anonymous connection-oriented directional communication (as opposed to OO, which employs connectionless undirected communication, often on known partners). Communication normally goes from right to left, in the opposite direction of logical transportation.

Our "pipe and filter style" differs from known implementations of that style in the operating system area, such as UIO [Ch87] or stackable filesystems [HP94], in a number of respects. First, it has no "consuming" semantics like pipes, where processing of data would "destroy" the old data and produce a new version instead. Rather, a brick adds to the ways we may look at the system, by providing a non-desctructive new *view* on the data: both the old view and the new view provided by the transformation will exist in parallel; the "old" view may for example be used by parallel wiring to other "consumers" or "clients". Modifi cations carried out in one view will usually be immediately reflected in the other view; a transformation may thus be characterized as logically acting bidirectionally. Second, we use nests as a *universal* (but nevertheless rather simple) memory abstraction on all layers of the whole operating system, not limited to fi lesystems (in contrast, stackable fi lesystems are based on conventional fi lesystem abstractions such as `inodes`, `vfs`, directory (sub)trees, etc). We take advantage of the fact that bricks may be instantiated dynamically at runtime, resembling the dynamic nature of fi lesystem subtrees directly by recursive instantiation of `dir_*`-bricks. Third, generation and maintainance of instances is done by a recursive and *self-describing* methodology as sketched:



Brick instances of any type are created and maintained by a special brick type called `control`. In the above picture, a `control` instance does not create other brick instances directly, but rather creates an `image` which may be considered a kind of "process image" "containing" the instances and their wiring. In order to really "execute" the instances network, some `mmu` instance like `mmu_i386` has to follow it (possibly indirectly). In our ATHOMUX prototype, we implemented a `control_dummy_linux` without `image` output, running in an ordinary Linux process, but separating both levels logically. Our brick sourcecode is designed to run in other `control_*` environments without alteration.

The `command` output of `control_*` is a nest instance containing some *abstract representation* of the instances network, satisfying property (1). In our ATHOMUX prototype, we chose ASCII strings describing instances, inputs and outputs, and their wiring in a C-like syntax. This simplifi es manipulations and transformations of virtual brick networks by searching and replacing via regular expressions. By "writing" such representation strings into the command channel depicted with dashed lines, (de)instantiation of bricks, dynamic inputs/outputs, and wires are performed. In general, an abstract representation of a (virtual) network of (virtual) instances can be both queried and manipulated via the dashed control wires, providing a controllable *description* on the system structure and the interrelations between instances. There may exist multiple descriptions *in parallel*. In the picture, different `strategy_*`-instances are non-destructively *transforming* between descriptions or creating virtual descriptions which need not to exist in "reality".

The wiring of `strategy_*`-instances may itself be controlled by another `control` instance connected with `strategy_strategy_*`-instances. This may be continued recursively, forming a hierarchy of control levels. At a certain level the hierarchy can either terminate by a level which controls itself, or at a level where never any modifications are necessary (e.g. a conventional bootstrap mechanism). In order to form an organic system with self-control, we simply select the first alternative to obtain property (2). When a `command` output contains its own controlling and strategy bricks, we get *self-description* in the sense of organic computing. Other properties of organic systems can be achieved by usage of appropriate `strategy_*` bricks in such a self-describing configuration.

As an example for `strategy_*` brick types, a specific `strategy_transparency` may provide network transparency, by automatically inserting `remote` instances into the system wherever it is necessary to span several hosts on a computer network, and by providing a single virtual system image. The automatically inserted `remote`-instances may be hidden for users of `strategy_transparency`, so that they get the virtual impression that no network would exist at all, and as if everything would execute on a single "virtual computer". Further examples are `strategy_*` brick types for providing different operating system *personalities*, code morphing [cod] for creating hardware platform transparency, for automatic adaptation to varying workloads (load balancing) or to unpredictable faults (fault tolerance), for achieving recoverability (transactions), for centralized enforcement of security policies by automatically inserting `check_*` instances at strategic places, emulation of AOP [K+97] as pointed out in [ST03b], and much more.

In particular, the desired behaviour of organic systems such as context-awareness can be implemented by appropriate `strategy_*` bricks. A LEGO-like system allows easy composition of strategies for organic behaviour. In extension to coordination models [CH96], transformations at the strategy level can be carried out non-destructively by recursive and self-describing abstract representations of the system.

In summary, creating multiple descriptions based on separation between instances and their relations enables advanced functionality such as organic behaviour in a modular way.

4 Performance

Our ATHOMUX implementation is a highly experimental and evolving prototype of a LEGO-like component system. Currently it runs in an ordinary Linux process. The following results and considerations are related to a snapshot of the current stage.

The space overhead in bytes for the brick infrastructure is shown in the following table, comparing the 32bit and 64bit Linux version. We discriminate between the space overhead in the instances and the overhead in our specific `control_dummy_linux`. The lower two lines are specific for the latter, and may change for other types of `control_*` and for stateless / pseudo-stateless implementations.

space overhead (bytes)	32bit	64bit
input overhead in instance	8	16
output overhead in instance	12	24
input/output overhead in `control`	28	56
brick overhead in `control`	28	48

The runtime performance of instantiation / deinstantiation is shown in the next table. In addition to the ASCII string representation of instance networks, we also implemented redundant (de-)instantiation operations for bricks, input/outputs and connections due to performance reasons. We measured on an otherwise unloaded AMD Athlon XP 2200+ with 1350MHz running Redhat Linux 8.0 with kernel 2.4.18-14, and on a Dual Opteron with 1800MHz clock running the 64-bit version of SuSe Linux 9.0 and kernel 2.6.0-1-smp. The following results are intended to show up the *order* of performance only. Each pair of operation has been repeated 1.000.000 times.

time in ms	32bit	64bit
brick instantiation / connect / deinstantiation	1548	676
`open()` / `close()` on `/etc/passwd`	3313	2375
pair of `getpid()`	449	178

The first line shows the results for instantiation of a small brick with one input and one output, connection to another brick output, and finally de-instantiation (which automatically disconnects also). For rough comparison with a somewhat related functionality, we show the timings of repeating 1.000.000 `open()` on `/etc/passwd` for reading immediately followed by `close()`. Since the overhead for crossing the kernel boundary is expected to significantly contribute to that, we also measured 1.000.000 pairs of `getpid()`, which is a system call doing nearly nothing. Even when the results from the lower line are subtracted from the middle line, our runtime overhead remains competitive. However note that we do not yet implement locking. Our results are just for getting an impression on the *order* of the performance which is achievable with lightweight LEGO-like components.

Finally, we present some results on the runtime overhead of our current experimental `dir_simple` implementation. It includes the overhead caused by indirect function calls when crossing brick instance borders, but also the overhead for remapping of logical addresses between different nest instances.

time in ms	32bit	64bit
`get` / `put` on 1 `dir_simple`	749	309
`get-transfer` / `put` on 1 `dir_simple`	2330	716
`get` / `put` on a chain of 10 `dir_simple`	1830	903
`get-transfer` / `put` on 10 `dir_simple`	3139	1302
`lseek()` / `read()`	2696	1314

The first line shows the results of repeating 1.000.000 pairs of `get` / `put` operations at logical address 0 on a `dir_simple` instance which is in turn connected with two other pseudo instances for testing (simulating a ramdisk). In this configuration, a data block of size 4096 bytes is passed by reference. In the next line, a `transfer` is additionally executed, leading to a `memcpy()` of 4096 bytes. The next two lines show the same for a chain of 10 `dir_simple` instances. As expected, this adds some further overhead. For rough comparison, the last line shows the timing of 1.000.000 `lseek()` to position 0 followed by a `read()` of 4096 bytes from `/etc/services`, which also leads to a copy. Although the `read()` Linux system call has been highly optimized over the many years of its development, we are competitive at least on the 64bit platform. However, we are not comparing *exactly* the same functionality, since access permissions are not yet checked by our prototype, so these numbers are to be taken as a rough measure of the *order* of the performance.

5 Conclusions

We have argued that it is advantagous to build organic systems on LEGO-like lightweight component systems, which should be able to describe themselves recursively. Recursive self-description is a key to automated reasoning on the system at the structural level, and it is crucial for achieving organic properties such as self-confi guration, self-healing, and many others. Our ATHOMUX prototype demonstrates that LEGO-like systems are feasible and may have a high potential for becoming a high-performant fundamental infrastructure for organic systems.

References

[Ch87] Cheriton, D. R.: Uio: A uniform i/o system interface for distributed systems. *Transactions on Computer Systems*. 5(1):12–46. 1987.

[CH96] Ciancarini, P. und Hankin, C. (Hrsg.): *Coordination Languages and Models*. LNCS 1061. Springer Verlag. 1996.

[cod] Code morphing. `http://www.transmeta.com/technology/architecture/code_morphing.html`.

[ft] `http://www.fischertechnik.com`.

[HP94] Heidemann, J. S. und Popek, G. J.: File-system development with stackable layers. *Transactions on Computer Systems*. 12(1):58–89. 1994.

[IEC] Iec standard 61131-3. `http://www.holobloc.com/stds/iec/sc65bwg7tf3/html/news.htm`.

[K$^+$97] Kiczales, G. u. a.: Aspect-oriented programming. In: *European Conference on Object-Oriented Programming (ECOOP)*. LNCS 1241. Springer-Verlag. 1997. `http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP9%7/for-web.pdf`.

[LEG] `http://www.lego.com`.

[SG96] Shaw, M. und Garlan, D.: *Software Architecture, Perspectives on an Emerging Discipline*. Prentice Hall. 1996.

[ST02] Schöbel-Theuer, T. Skizze einer auf nur zwei abstraktionen beruhenden betriebssystemarchitektur: Nester und bausteine. Arbeitspapier und Vortrag auf dem Herbsttreffen der GI, Fachgruppe Betriebssysteme, Berlin. 7. – 8.11.2002.

[ST03a] Schöbel-Theuer, T. Eine neue architektur für betriebssysteme. Unveröffentlichtes Manuskript einer Monographie. 2003. Erhältlich auf Anfrage bei `schoebel@informatik.uni-stuttgart.de`.

[ST03b] Schöbel-Theuer, T.: On variants of genericity. In: *Proceedings of the Fifteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2003)*. S. 359–365. Knowlegde Systems Institute. 2003.

[ST03c] Schöbel-Theuer, T.: A way for seamless integration of databases and operating systems. In: *Proceedings of the International Conference on Computer Science and its Applications (ICCSA-2003)*. S. 82–89. National University. 2003.