TOWARDS A GENERIC DESIGN FOR GENERAL-PURPOSE SENSOR NETWORK NODES

Position Paper

Stefan Gruner

Department of Computer Science, University of Pretoria, Lynnwood Road, 0002 Pretoria, South Africa

Keywords: Standardisation, Sensor network node, Cybernetic interaction model, Micro operating system, Software application interfaces.

Abstract: The key to mature and efficient industrial software engineering is standardisation more than an aggressive struggle for innovation only for the sake of its own. This is assumption is also held for the area of sensor networks development, which is becoming an increasingly important field at the interface between software and hardware engineering. This short position paper proposes and outlines a generic design for the nodes of such sensor networks, which could be used in the future as the basis of almost any conceivable sensor network application. On such a basis of generic standardisation, the development of specific and particular sensor network applications will then be mainly a matter of hardware-independent programming with APIs, as it is already well known in the classical domain of operating systems in ordinary desktop PCs.

1 MOTIVATION AND RELATED WORK

Software engineering for common-place devices (like personal desktop computers) has become comparatively easy during the past decades, mainly because of reasonably high levels of *standardisation* in the design of hardware platforms, *operating systems* (OS) and *application program interfaces* (API) in the context of those common devices. Software engineering for *non-standard* devices, such as *robots* (Campbell, 2009) or the physical *nodes* of (wireless) sensor networks, is still comparatively harder, which has (amongst other reasons) much to do with a shortage of standardisation in this field. There is much agreement amongst many software engineering experts that only the *specialization* of well-defined *sub*-areas of software engineering will eventually lead to methodological maturity of the discipline. For comparison: there is no such thing as general 'hardware engineering'; in the domain of hardware we can find automotive engineering, electronics, railway engineering, and so on. Positive examples in our field, software engineering, are *compiler* construction, or *relational data-base* construction, which are well understood since many years and do not confront us with major difficulties any longer. In the long run, the same must become true also for other sub-areas of software engineering, such as the construction of software for sensor network applications, embedded systems, and so on. Further positive examples pointing into the right direction are the standardisation of software systems for the automobile industry (Dörr, 2008), as well as the already mentioned standardisation attempts for robotic platforms (Campbell, 2009).

The need for standardised frameworks in the area of wireless sensor network development was most recently also addressed by (Alkazemi, 2010). In that paper a general system model was outlined on the basis of several layers from the hardware level via operating system levels up to the interfaces for the user application software. Also in (Alkazemi, 2010) one can thus find a 'classical' well-understood technique being transfered into a comparatively new application domain, namely the domain of wireless sensor nodes. In such a way by separating a node's hardware from the user's application programs with intermediate layers of firmware and system-software the re-usability of such a node in different scenarios can be considerably increased (Alkazemi, 2010). The same principle of layering was also applied in the approach of (Dörr, 2008) for non-standard hardware in the domain of automobile software engineering. In all these layered approaches there is of course a trade-off between *flexibility* and *speed*:

Gruner S. (2010).
TOWARDS A GENERIC DESIGN FOR GENERAL-PURPOSE SENSOR NETWORK NODES - Position Paper.
In *Proceedings of the Fifth International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 259-264
DOI: 10.5220/0003029802590264

more layers between hardware platform and the user application program will lead to more application flexibility at the expense of computational speed (performance) whereas less intermediate layers will lead to greater speed at the expense of flexibility. At the moment, the user applications are mostly implemented very close to the hardware of a sensor node, which is due to the limited storage capacity of such devices, such that the programming of the user application code is a cumbersome task and the result of such efforts will not be characterised by high reusability. In some way we could say that wireless sensor networks with their small computational nodes have taken us back into the early days of computing when storage and computational power was limited and sophisticated operating systems did not exist. However, according to Moore's law, one might reasonably expect that this situation will change rather sooner than later, such that a layered application approach to the deployment of software on tiny sensor nodes will become feasible, too. Under this presumption also this short position paper is presented.

This short position paper outlines a generic model, at a high level of abstraction, of sensor network nodes which is supposed to serve as a basis for future standardisation efforts at the interfaces which this model defines. The finer details of construction and implementation (for example, whether communication between a node and its partners is synchronous or asynchronous, whether a node has a unique identity or is anonymous in a network, whether or not a node can change its internal state on the basis of a memory, etc.) are left deliberately un-specified, such that the model does not undermine its own reusability by being overly specific. The 'engine' part of this model is derived from (Ellis, 2008), with additional parts added to describe the interactions between a node and its environment. For each part identified by this model it should then become possible for designers and developers to create standardised interfaces and re-usable software module libraries, such that (in the end) the deployment of software-driven sensor network applications of any kind will become very much a matter of component-based development (CBD) (Lau, 2004) (Alkazemi, 2010) of well understood *devices* (Maibaum, 2008) in the fashion of the classical engineering disciplines.

2 SCHEMA OF A SENSOR NODE

A conceptual assumption is made *ab-initio*: For a network of nodes, being a distributed concurrent system (Tanenbaum, 2007), it is here assumed for the sake of simplicity that every node, as an atomic unit of the network, is *not* yet another, concurrent micro-system in itself; instead it is assumed here that every node is based on a mono-processor with internally sequential operations. This does not imply any loss of generality: Should one technically wish to construct a distributed sensor network system of which the nodes are concurrent systems (on the micro-level) based on poly-processors themselves, then we would be dealing with a distributed system of higher order, whereby each concurrent poly-processor node can also be modeled as a distributed system in the same terms of the model described in this position paper. For better understandability the model will be developed and discussed stepwise in the following paragraphs.

Initially we know only that we have a *unit*, which is a sensor node, which somehow *interacts* with its environment. The environment typically comprises all sorts of things, including other units, but the inner details of the unit and its forms of interactions are not yet specified. This situation is shown in Figure 1.

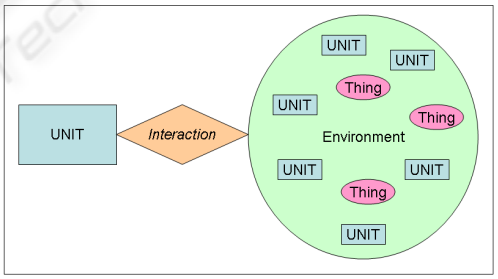

Figure 1: Unit interacting with a diverse environment.

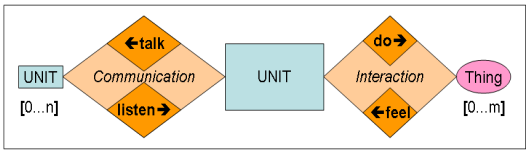

Figure 2: Classification of different types of interactions in various directions.

Thus we realize that different modes of environment engagement will be needed: one for the units (of the same or similar type) with which our first unit forms a *society*, and one for the other influential or influencable entities or agents or 'things' which do *not* have 'member' status in the

society of units that constitutes a wireless sensor network. This can be modeled as shown in Figure 2, with *communication* amongst units and (other) interaction between units and things, whereby in for communication we can further distinguish *talking* and *listening*, whereas in interaction we can further distinguish *doing* and *feeling*. In other words: any device which can be described in terms of the schema of Figure 2, regardless of the details of its technical construction, is a general-purpose node for deployment in a sensor network application.

We have now reached the point at which we can no longer regard our model unit as black box any longer. Feeling and doing, talking and listening are usually implemented technically by sensor(s) and actuator(s), sender(s) and receiver(s), the abstract model of which is depicted in Figure 3. The control relation, by which sender, receiver, sensor and actuator are related, is obviously also not an atomic concept: there should be some form of memory ('M' in Figure 3) to capture a unit's possibly changeable internal state, as well as the finer structures and algorithms (*not* shown in Figure 3) by which the control relation must be implemented.

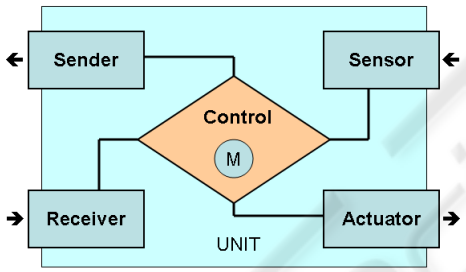

Figure 3: Coarse components of the stanndardised general-purpose sensor network node unit.

With such a purposefully abstract design schema, many technical variations, for different network applications, are possible:

- A unit could communicate with only one other unit in a master-servant-relation;
- A unit could communicate with *N* other peer units, whereby *N* is a-priori defined and fixed;
- A unit could communicate with *n* other peer units, whereby n is not a-priori defined and could even vary in time as *n*(*t*);
- A unit could be 'mute' (no talking);
- A unit could be 'deaf' (no listening);
- A unit could sense only one type of feeling (in various degrees of intensity);
- A unit could sense *N* different types of feelings (in various degrees of intensity);
- A unit could perform only one type of action to the environment;
- A unit could perform *N* different types of action to the environment;
- A unit could be 'blind' (no sensing);
- A unit could be environmentally passive (without acting).

Depending on the behavioural variations as outlined above, the following technical variations seem to be reasonable:

- Where *N* is fixed a-priori, the unit could possess a multiplicity of *N* distinguished senders, receivers, sensors, actuators as sub components, each of which would then be individually accessible by the controller.
- Where *n* is not a-priori determined, or even variable as *n*(*t*) during the passage of time, each of the unit's sub components (sender, receiver, sensor, actuator) could be implemented internally (on the fine scale), for example, as queue-buffered multiplexer with an internal scheduler, as it is well known from the standard operating systems literature (Silberschatz, 2008).

Needless to emphasise that these implementational variations will also depend on further *physial* variations, for example: whether the signals to and from the environment will come through a cable, or wirelessly via radio waves, and so on.

In the following we still need to look at the finer details of the control model (symbolized by the orange `diamond' shapes in Figure 3). Following (Ellis, 2008) one can distinguish basically two types (modulo finer variations) of *cybernetic feedback loops* for such controlers, namely the simple, basic, *non-adaptive* feedback control process, and the considerably smarter *adaptive* selection process. Both of them are suitable schemas also for the units of sensor networks in our context, as shown and discussed in the following paragraphs. Thereby it should be clear from the start that those loops have to be immediately supported by the hardware of the unit, as they are always the same; but the particular activities within such a cycle can vary from case to case and from application to application and must thus be implemented in software for the sake of flexibility and hardware-independent programming of such units.

The basic feedback loop according to (Ellis, 2008) is depicted in Figure 4. The meaning of this picture is that a system tries to *stabilize* its internal state by repeatedly comparing its internal state with a set of pre-defined goals. In the case of discrepancy

the controller attempts to change the system's internal state such as to better approximate the pre-defined goals. These goals can either be *implicitly* encoded ('hardwired') into the system, or –which whould be preferable from a software engineering perspective for the sake of flexibility and wider applicability– they can also be *explicitly* provided in and by a separate module (in program code). Also note that nothing is stipulated about how frequently this cycle has to be executed in real time – this is an important issue in the 'real world' of sensor network applications in which nodes with little resources in battery capacity have to 'fall asleep' from time to time in order to save their own energy.

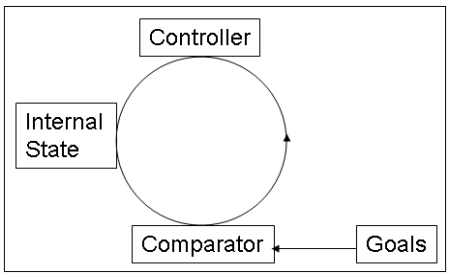

Figure 4: Simple non-adaptive feedback loop (Ellis 2008).

An example for such a simple feedback loop would be a simple thermostat: Let the goal temperature in an air-conditioned room be 21°C. If the actual room temperature raises above that value, the controller would start cooling; if the actual room temperature falls below that value, the controller would start heating. To avoid hectic (and costly) 'vibrations' of the system around a 'point' goal (such as 21°C) one could also define a 'range' goal (such as 20–22°C), such that cooling would start only above 22°C, and heating would start only below 20°C; in this way the system needs to trigger actions less frequently and could thus save some of its own battery power. On the basis of such a simple design schema, a number of variations are possible, for example:

- The number of goals can vary from 1 to *N*; (it is however fixed a-priori in this design).
- In the case of only 1 goal, the goal could be trivial ('true', always fulfilled), in which case the comparator sub-component would become obsolete; (which is basically equivalent to having no 'goal' at all: take, for example, a broadcaster unit which only forwards received messages to other units in a network).
- The system could also be 'stateless', i.e. with only 1 internal state that cannot vary over time; such a system would not need any freely programmable random access memory (RAM) – though this is probably not a very realistic variant in our context of *software-driven* sensor networks.

However, this basic model of (Ellis, 2008) needs to be somewhat modified in our context, particularly since the primary *purpose* of a unit in a sensor network is *not* to 'survive' by self-stabilisation in its environment; the purpose of a unit in our context is to communicate and to interact (for which 'survival' is only an existential precondition). Moreover, the modified process cycle (on the basis of Ellis's) must also take our four sub components, which interface with the unit's environment (i.e. sender, receiver, sensor, actuator), into consideration. The according schema is depicted in Figure 5. Where by the chosen interactions with the environment (feel, do, talk, listen) will then also depend on the actual internal state M as parameter, (thus, strictly speaking: feel_M, do_M, talk_M, and listen_M). This we could also call the different *modes* of interactions, which can be described as

IF(boolean_condition(M)) THEN {...} ELSE {...}

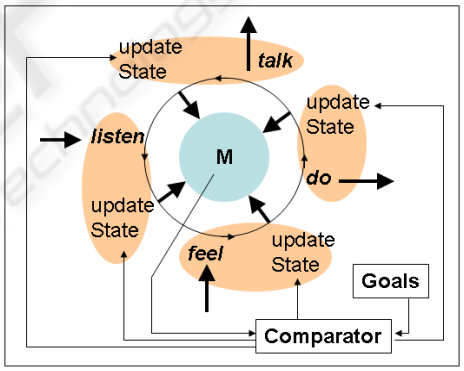

Figure 5: Node design on the basis of the simple non-adaptive feedback loop.

In this model we have basically four sub routines (shaded in orange colour in Figure 5) for the external interactions, whereby each of them has an internal *side effect* in the form of a memory update by means of which the unit's external interactions (or the consequences thereof) can be 'logged' for future reference. For the sake of general applicability of the unit, these behavioural details of those four sub-routines should be defined in *software*, whereas the cycle itself should be directly hardware-supported for the sake of efficiency. This concept can also be found in the design of the '*Apache*' web-server, which also has a basic 'loop' with 'hooks' at which additional, case-specific functionality (modules) can be connected (Tanenbaum, 2007). Possible technical

variations of this schema include, amongst others, the following possibilities:

- The unit's designer could re-arrange the sequential order in which the internal and external events are carried out in one cycle.
- In case that a unit would be 'deaf' or 'mute' or memory-less, the according external or internal events (environmental interaction, memory update) could also be skipped.

There is also a more complex and more flexibile cybernetic feedback system which can temporarily vary and adapt itself within the limits of some higher-order *norms* which Ellis called a "value system" (Ellis 2008). Such an adaptive cybernetic system is depiced in Figure 6, (whereby I have made some minor simplifications of Ellis' original picture, for the sake of easier understanding). The picture basically tells us that (and how) an active unit, embedded in an environment, can modify its own goals in order to respond flexibly to evolutionary pressure (from the environment) for the purpose of survival in that (variable) environment.

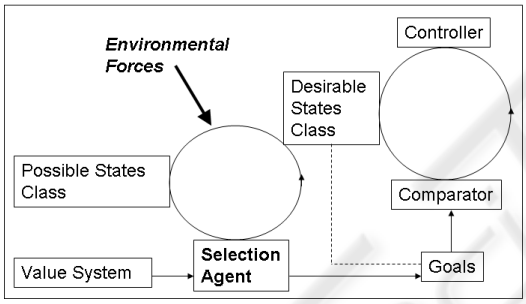

Figure 6: Cybernetic system with double feedback loop, similar to the concept from (Ellis 2008).

Also the double feedback loop can be nicely illustrated by example a thermostat that tries to keep the temperature of an air-conditioned room at the goal value of around 21°C. Cooling and heating, a large toom, however, costs a lot of fuel, and fuel can run low at times. The following environmental adaptation cycle, including a selection agent, can be added to the system: If it is winter (i.e. very cold outside) and the fuel supply (for heating) is low, then the goal temperature for the thermostat is lowered by the selection agent to around 19°C. If it is summer (i.e. very warm outside) and the fuel supply (for cooling) is low then the goal temperature is raised by the selection agent to around 23°C. Thereby, the simple thermostat cycle (in the inner loop) keeps running as usual. The system has now several possible internal states, one of which is variably regarded as desirable, depending on further external circumstances.

The question is now how to adapt this more complicated cybernetic model into a generic, general-purpose network node design in the context of this paper; again we have to attach somehow our four sub processes of interaction (listen, talk, feel, do) to these now *two* cybernetic feedback loops as they are shown in Figure 6. This could obviously be done in very many different combinations, though not every possible combination will make sense from the perspective of purposeful technical system design and software engineering. For such types of cybernetic units I suggest the following standardized schema for an 'all-purpose' type of sensor network nodes – though, as said above, many other (though similar) design schemas would equally be possible for more special purposes and applications. Once again we must also consider that the genuine purpose of our sensor network units is *not* to survive just for the sake of survival (as in Ellis' original scheme), but rather to provide some useful computational and behavioural services to the users of such a sensor network after its deployment.

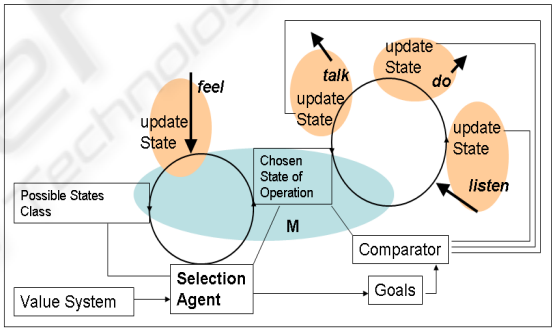

Figure 7: Schema of a sensor network node based on the double cybernetic feedback loop.

To justify the suggestion about the following design I hint at the following anthropomorphisms and basic experiences from human society – remember that a sensor network was also characterized as 'society of units' above.

Our ability to talk and to listen is largely independent from environmental circumstances such as the weather, whether its cold or hot etc.; therefore the communicative sub processes should be attached to the rapid, 'inner' feedback cycle (as before). A similar argument can be brought forward w.r.t. our small-scale actions and acivities. Would we now allocate our fourth sub process, namely 'feeling', to the inner feedback cycle as well, then the placeholder for 'Environmental Forces' (see Figure 6 again) would remain un-instantiated, such that the

entire 'outer' feedback loop (including the selection agent, the value system, etc.) would be useless and obsolete. Consequently, the sub process for feelings must be connected to the second (the outer, the environmental) cybernetic cycle. The resulting schema for higher-order network units designed along those considerations is sketched in Figure 7, whereby both cybernetic feedback cycles are understood to be executed in the mode of *interleaving* (pseudo-parallelism) by the unit's mono-processor and both cybernetic feedback cycles also access the same memory (RAM, 'M' in Figure 7) the contents of which defines the unit's internal states.

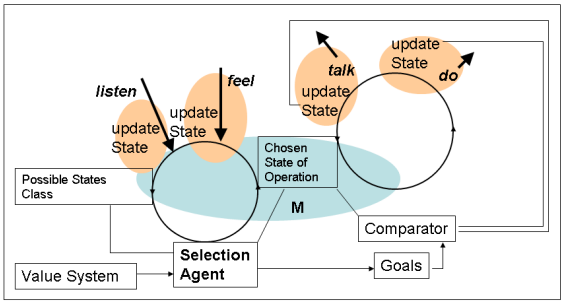

Figure 8: Variation of the double loop design, with the listening function hooked to the outer feedback loop.

Another feasible variant of this schema is shown below in Figure 8, whereby also the ability to listen is there attached to the environmental loop; consider the thermostat example of above again, in which the device would now receive the information about winter or summer via messages through the communication lines (rather than through the low-level sensor lines).

3 SUMMARY

The conjecture underlying this short position paper is that the design schemas outlined above will be suitable to serve as a standardized templates for the development of sensor network applications in hardware and software. Thereby –in the end– each interface identified and shown in those paterns could be subject to industrial normalisation, as well as the provision of micro operating systems and APIs for the benefits of the application programmer (who does not want to have to care about the unit's hardware particularities on the physical level). Interface software between sensor node platforms and the specific, particular user-application software for such nodes could then be provided in analogy to (Campbell, 2009) (Dörr, 2008). For rather simple applications (e.g. soil temperature measurements in geology) the simple design schema with only one feedback loop may probably be sufficient. For more sophisticated applications, e.g. in Robotics and Artificial Swarms, a node design on the basis of the double feedback loop schema may probably be the more intelligent and more flexible ones for the modular, component-based construction of such units.

I suggest that, based on those design schemas, sensor network applications in the future could be easily produced as a matter of hardware-independent API-programming, as we already know it today in the domain of standard devices such as desktop PCs. In summary this paper suggests that 'typical' sensor network nodes suitable for most kinds of sensor network application should become as 'orinary' as software-driven mobile phones or PCs. However, the technical development of such small-scale general-purpose devices, with APIs for the installation of scenario-specific application software on them, requires cooperation between hardware and software engineers.

ACKNOWLEDGEMENTS

This work is supported by the National Research Foundation (NRF) of the Republic of South Africa. Many thanks also to the anonymous reviewers of the ENASE'2010 conference for their helpful comments on the draft of this short position paper.

REFERENCES

Alkazemi, B.Y., Felemban, E.A., 2010. Towards a Framework for Engineering Software Development of Sensor Nodes in Wireless Sensor Networks.

Campbell, McG., 2009. Robots to get their own Operating System. *New Scientist 8*, pp. 18-19.

Dörr, H., 2008. The AUTOSAR Way of Model-Based Engineering of Automotive Systems. *LNCS 5214*, p.38

Ellis, G.E.R., 2008. On the nature of causation in Complex Systems. *Transact. of the Roy. Soc. of South Africa 63 /1*, pp. 69-84.

Lau, K.K., 2004. *Component-based Software development Case Studies.* World Scientific Publ.

Maibaum, T., 2008. Formal Methods versus Engineering. *Proc. 1st Internat. Workshop on Formal Methods in Education and Training*, Kitakyushu, Japan.

Silberschatz, A., Galvin, P.B., Gagne, G., 2008. *Operating System Concepts*. John Wiley &Sons Publ., 8th edition.

Tanenbaum, A.S., van Steen, M., 2007. *Distributed Syst. Principles and Paradigms*. Pearson/Prentice Hall Publ. 2nd edition.