Exploring Entropy in Software Systems: Towards a Precise Definition and Design Rules

Herwig Mannaert, Peter De Bruyn, Jan Verelst
Department of Management Information Systems
University of Antwerp
Antwerp, Belgium
{*herwig.mannaert, peter.debruyn, jan.verelst*}*@ua.ac.be*

***Abstract*—Software systems need to be agile in order to continuously adapt to changing business requirements. Nevertheless, many organizations report difficulties while trying to adapt their software applications. Normalized Systems (NS) theory has previously been able to introduce a proven degree of evolvable modularity into software systems, based on the systems theoretic notion of stability. In this paper, we explore the applicability of this other fundamental property of systems (i.e., entropy) to the issues of software maintenance and evolvability. The underlying concepts in entropy definitions will be explained and applied to software systems and architectures. Further, the considerable complexity of running multi-tier multi-threading software systems and the relation with entropy concepts is discussed and illustrated. Finally, the concordance of design rules for controlling that entropy with previously formulated NS principles is explored.**

***Keywords*-Normalized Systems, Entropy, Systems engineering, Evolvability**

I. INTRODUCTION

Current organizations need to be able to cope with increasing change and increasing complexity in most of their aspects and dimensions. As a consequence, all constructs and artifacts of an organization have to be able to swiftly adapt to this agile and complex environment, including its business processes and organizational structure, as well as its supporting information systems. Indeed, also the software applications an organization employs, should be able to evolve at an equivalent pace as the business requirements of the organization they are embedded in.

However, many indications are present that most modular structures in software applications do not exhibit this required evolvability, flexibility, etcetera. One very early indication of this phenomenon was expressed by the formulation of Manny Lehman's Law of increasing complexity, stating that "*As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.*" [1, p. 1068]. Interpreting entropy as a measure for uncertainty or the degree of absence of structure (i.e., disorder) in a system, one could conceive Lehman's law as referring to the irreversable tendency of software applications to build up entropy (i.e., structure deterioration and degradation) troughout their lifecyle and hence become more and more complex while being less and less maintainable as time goes by. Indeed, Lehman himself initially proposed his law as an instance of the second law of thermodynamics [1]. Therefore, the law can also be interpreted as describing ever increasing entropy or disorder in the structure of software systems, unless effort is done in order to reduce the amount of entropy. In a similar way, Frederick Brooks stated that program maintenance is an inherently entropy increasing process, and that even its most skilfull execution is only able to delay the subsidence of the system into unfixable obsolescence [2]. In practice, manifestations of this ever increasing difficulty in maintaining software is for instance reflected in terms of ever growing IT departments and rising IT maintenance costs [3], [4].

Specifically focusing on these issues of software maintenance and evolvability, *Normalized Systems (NS) theory* has recently proven to introduce a degree of proven ex-ante evolvability at the level of software systems. To start with, the theory states that the implementation of functional requirements into software constructs can be considered as the *transformation* of a set of requirements $\mathcal{R}$ into a set of software primitives $\mathcal{S}$ [5], [6], [7]. In order to reach evolvable modularity, NS theory demands that this transformation should exhibit systems theoretic *stability*. Mannaert et al. have formally proven in [6] that this assumption implies that the modular software structure should strictly and systematically adhere to four design *principles* as a necessary condition. In this paper, our goal is to explore the applicability of this other fundamental property of systems, i.e. *entropy*, to software maintenance and evolvability. Therefore, we propose a more precise definition of entropy in software systems, and explore how this notion of entropy might provide guidance to the software engineering process, in order to control the amount of entropy continually built up during the lifecyle of a software application. As an initial step towards design rules, we attempt to relate this guidance to the design principles of NS theory.

The remainder of this paper is structured as follows. In section II, we briefly discuss some related work. A concise overview of Normalized Systems Theory is presented in a

 ISBN: 978-1-61208-184-7

third section. In section IV, an attempt is made to derive an unambiguous definition of entropy in software architectures. Next, the feasibility of controlling the entropy in software systems is discussed in a fifth section, and the implications for software design are related to NS design principles. Finally, we present some conclusions and future work in Section VI.

II. Related Work

Entropy as expressed in the second law of thermodynamics, is considered to be a fundamental principle. There are many versions of this second law, but they all have the same intent, which is to explain the phenomenon of irreversibility in nature [8]. Moreover, mathematical derivations of the principle of entropy start in general from a formula describing the number of possible combinations. In statistical thermodynamics, entropy was defined by Boltzmann in 1872 as the number of possible microstates corresponding to the same macrostate [9]. In information theory, entropy was defined in 1948 by Shannon as the number of possible combinations or uncertainty associated with a random variable [10].

It has been attempted in many areas to apply and operationalize the concept of entropy, both inside and outside engineering sciences. In [11], Janow has studied organizations and productivity based on entropy. Janow concluded that entropy offered an interesting means to explain why organizations tend to become gradually more slow in their decisionmaking processes, as well as lose productivy and speed as they become larger over time. In the computing area, entropy is defined as the randomness collected by an operating system or application for use in cryptography or other uses that require random data [12], [13]. This randomness is often collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators [14].

In software engineering, earlier attempts have been made to apply entropy concepts to software. For example, Harrison argued for an entropy-based metric for measuring the complexity of software applications, based on the information theory perspective on entropy [15]. Based on an analysis of existing complexity metrics for software, Bianchi et al. [16] propose a class of metrics aimed at assessing the amount of software degradation as an effect of continuous change.

Further, Manny Lehman considered his law of increasing complexity as an instance of the second law of thermodynamics [1]. Therefore, the law can also be interpreted as describing ever increasing entropy or disorder in the structure of software systems. This disorder or structure degradation hampers future adaptations of the software system, unless effort is done in order to reduce the amount of entropy [1], [17]. In a similar way, Frederick Brooks related the severe complexities of software engineering to the concept of entropy, and stated that program maintenance is an inherently entropy increasing process [2]. Consequently, as the theory on Normalized Systems [5], [6], [7] is aimed at understanding and controlling the law of increasing complexity, it can also be interpreted as an approach to controlling entropy, as will be further highlighted below.

III. Normalized Systems Theory

Specifically focusing on the issues of software maintenance and evolvability, *Normalized Systems (NS) theory* has recently proven to introduce a degree of proven ex-ante evolvability at the level of software systems. To start with, the theory states that the implementation of functional requirements into software constructs could be regarded as a *transformation* of a set of requirements $\mathcal{R}$ into a set of software primitives $\mathcal{S}$ [5], [6], [7]:

$$\{\mathcal{S}\} = \mathcal{I}\{\mathcal{R}\}$$

In order to limit the complexity of the evolvability analysis, it is argued in [6] that it is not a conceptual limitation to limit the software constructs to those of procedural programming languages, distinguishing data structures S_m and processing functions F_n. However, in order to study the evolvability, an additional variable needs to be introduced to represent the version of the programming constructs, both for data structures $S_{m,i}$ and processing functions $F_{n,j}$.

Further, in order to obtain evolvable modularity, NS theory demands that this transformation should exhibit systems theoretic stability, meaning that a bounded input function (i.e., bounded set of requirement changes) should result in a bounded output values (i.e., a bounded impact or effort) even if an unlimited time period and systems evolution is considered (in which the number of primitives and their dependencies becomes unbounded). Applied to information systems, this means that the impact of a change can only be dependent on the nature of a change itself. Alternatively, changes having impacts also depending on the size of the system are called *combinatorial effects* and thus should be avoided in order to obtain a stable software architecture. In fact, one could observe that the behavior of combinatorial effects seems to be very similar to the ever increasing complexity issue as coined by Lehman: a continuously growing number of changes including combinatorial effects, each of them exhibiting an ever increasing impact of N, would contribute to the ever increasing complexity. As such, Mannaert et al. [6] have formally proven that this implies that the modular software structure should strictly and systematically adhere to the following *principles* as a necessary condition:

- *Separation of Concerns*, enforcing that each change driver becomes seperated;
- *Data Version Transparency*, enforcing that communication between data is performed in a version transparant way;

 ISBN: 978-1-61208-184-7

- *Action Version Transparency*, requiring that action components can be updated without impacting their calling components;
- *Separation of States*, enforcing that each action of a workflow becomes seperated from other actions in time, by keeping state after every action.

These design principles show that current software constructs, such as for example functions and classes, by themselves offer no real mechanisms to accomodate anticipated changes in a stable way. Moreover, as the systematic application of these principles results in very fine-grained modular structures, NS theory proposes to build information systems based on the aggregation of instantiations of five higher-level software *elements*, i.e., action elements, data elements, workflow elements, trigger elements and connector elements [5], [6], [7]. The internal structure of every of these elements has been described in a very fine-grained way, proven to be free of combinatorial effects. Hence, by building normalized software applications based on an aggregation of instances of the different elements, the stable software structure of an application can be expanded, based on the internal structure of the elements.

While NS theory thus originally originated from applying the concept of systems theoretic stability on the transformation of (elementary) functional requirements into constructional software primitives, it seems reasonable to expect that the NS theory concepts can also be related to entropy, and that the principles can be interpreted as a means of controlling the amount of entropy built up during the lifecyle of a software application.

IV. Towards a Definition of Software Entropy

In this section we will first discuss some underlying concepts in general entropy definitions. Next, we will apply these concepts on software systems by proposing a definition for their macrostates and microstates.

A. Underlying Concepts in Entropy Definitions

Consider in more detail the statistical perspective on entropy as introduced by the Austrian physicist Ludwig Boltzmann. In statistical thermodynamics, the aim is to understand and to interpret the measurable macroscopic properties of materials – the macrostate – in terms of the properties of their constituent parts – the microstates – and the interactions between them. In Boltzmann's definition, entropy is a measure of the number of possible microstates of a system, consistent with its macrostate. It is basically the number of possible combinations of individual microstates that yield the same macrostate [18]. This notion of entropy can be seen as a measure of our lack of knowledge about a system. Consider for example a set of 100 coins, each of which is either heads up or tails up. The macrostates are specified by the total number of heads and tails, whereas the microstates are specified by the facings of each individual coin. For the macrostate of 100 heads of 100 tails, there is exactly one possible configuration, so our knowledge of the system is complete. At the opposite extreme, the macrostate which gives us the least knowledge about the system consists of 50 heads and 50 tails in any order, for which there are 10^{29} possible microstates. It is clear that the entropy is extremely large in the latter case because we have no knowledge of the internals of the system. An obvious mechanism to decrease the entropy or complexity, is to increase the *structure* and therefore knowledge of the internals. Suppose we would have 10 groups of 10 coins, each with 5 heads and 5 tails, the number of possible combinations or microstates would only be 2520 [18]. In summary, structure can be used to control entropy, in the sense that the microstates are known, which leads to less uncertainty and therefore, a kind of determinism.

Another example which is often suggested is that of a container with a boundary in the middle and containing a gas in the left part of the container. The macrostate is characterized by observable variables such as pressure and temperature, while the microstate is the union of the position, velocity, and energy of all gas molecules in the container. Once the boundary division is removed, the gas molecules will spread out over the entire container, increasing the number of possible microstates and therefore the amount of entropy. In summary, entropy increase is typically related to a process of *spreading out*, and intermediate boundary structures are a typical mechanism to avoid the increase of entropy.

B. Macrostates and Microstates in Software Systems

Starting from the generic definition of entropy as *the number of microstates for a given macrostate*, we first need to propose a definition for the concepts of macrostate and microstates. As a *macrostate* is in general an observable state of the system, and all source code is observable in a transparent way, it seems more promising to apply the concept of entropy to the run-time analysis of a software system, than to the compile-time analysis of the source code. Therefore, the observable macrostate seems to be related to information that is visible in output streams such as loggings, or observable system states such as database entries.

Concerning *microstates*, they could be defined in terms of the elementary instructions and data registers of the processor. However, as our purpose is to establish principles to guide the software engineering and architecture process, we propose to define the concept of microstates at the level of the constructs of the programming languages. As the correct and faultless execution of software programs is in general considered to be a worthwhile pursuit, we propose to define microstates as binary values representing the correct or erroneous execution of a construct of a programming language. However, it is important to clearly specify the *molecules* at this level, and to define clear boundaries between them. For instance, the run-time execution of a

 ISBN: 978-1-61208-184-7

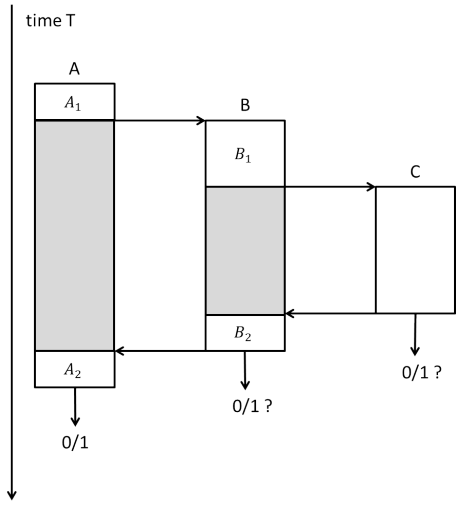

Figure 1. Synchronous pipelines.

procedural function can be defined as the execution of a specific procedural function operating on specific instances of the argument data structures, and resulting in a specific instance of an output data structure.

Moreover, these boundary divisions are not only needed between the various programming constructs at a certain point in time, but should also avoid coupling or leakage in the temporal dimension. For instance, the operations of a specific function could also be influenced by the values of global variables that each have received their specific value from other functions, or influence other functions by assigning values to such variables. Only when avoiding these effects, this function will not influence any other software molecule in any other way than through the specified output data arguments. In an object-oriented programming environment, all class member variables behave like global variables for the various methods of the class.

Consider Figure 1, representing an action entity A calling action entities B and (indirectly) C in a stateless way. Each of the seperate action entities A, B and C will generate a correct (0) or erroneous (1) outcome. In our representation, the seperate action entities can be regarded as the 'software molecules'. The correct or erroneous outcome of each of the subparts is defined as their microstate. The final outcome of action entity A, after executing B and C, can be regarded as the macrostate. Interestingly, one can note that the boundary division between the individual modules exhibits leakage in the temporal dimension. Indeed, action entity B (A) can only be succesfully completed after the execution of action entity C (B) and are hence interdependent. This inherently results in an increase in the amount of entropy: suppose for example that an error has occured as final outcome of A. This situation generates an uncertainty effect regarding where the actual error did occur. Did the returned final value originate in one of the two genuine parts of A, one of the two genuine parts of B, or C (or possibly a combination of them)? In other words, multiple combinations of microstates (correct or erroneous execution of the different (sub)activities) might generate the same macrostate (the resulting error as a final outcome of activity A), causing an increased degree of entropy or uncertainty.

In accordance with the evolvability analysis in [6], we only distinguish data structures $S_{m,i}$ and processing functions $F_{n,j}$. However, in order to represent the run-time state of the system, an additional variable k needs to be introduced representing the actual instance or value of the data structure $S_{m,i,k}$, and a variable l representing the program thread that is executing the processing function $F_{n,j,l}$. In order to control the complexity, which is widely accepted to be related to the concept of entropy, it seems reasonable to adhere to the straighforward extension of this model to an object-oriented programming environment, as proposed in [6], [7]. This means that a dedicated class is defined for every processing function, which is then the central method of this class, and for every data structure, which fields are the member variables of this class. Moreover, it is proven in [6], [7], in accordance with the *Separation of Concerns* principle, that no other functionalities should be added to those classes.

V. Towards Isentropic Software Architectures

In the previous section we discussed by means of a pedagogical example how entropy (and the according micro and macrostates) can be defined in the context of software systems. In this section, we will extend our analysis by illustrating the degree of possible entropy in real-life running software systems and which design rules might be formulated in order to control this entropy.

A. Entropy in a Running Software System

In order to illustrate the complexity of running software systems and its relation to our attempts in defining entropy, we start from the programming patterns for basic data entry operations in a multi-tier JEE (Java Enterprise Edition) architecture, as described and elaborated in [19], [20], [7]. Every data element or table `Obj` uses the same simple patterns for the various operations manipulating data, such as create, update, retrieve, delete, and search. For instance, in order to create a new instance, the webtier MVC (Model View Controller) framework calls the method `act` on an `ObjEnterer` class. The call is related to the application tier through a method `create` in an `ObjAgent` class, calling remotely through RMI (Remote Method Invocation) the `create` method in an `ObjBean` class residing in the application server. The various calls pass to each other instances of a serializable `ObjDetails` class, that contains the actual values of the various fields or attributes.

 ISBN: 978-1-61208-184-7

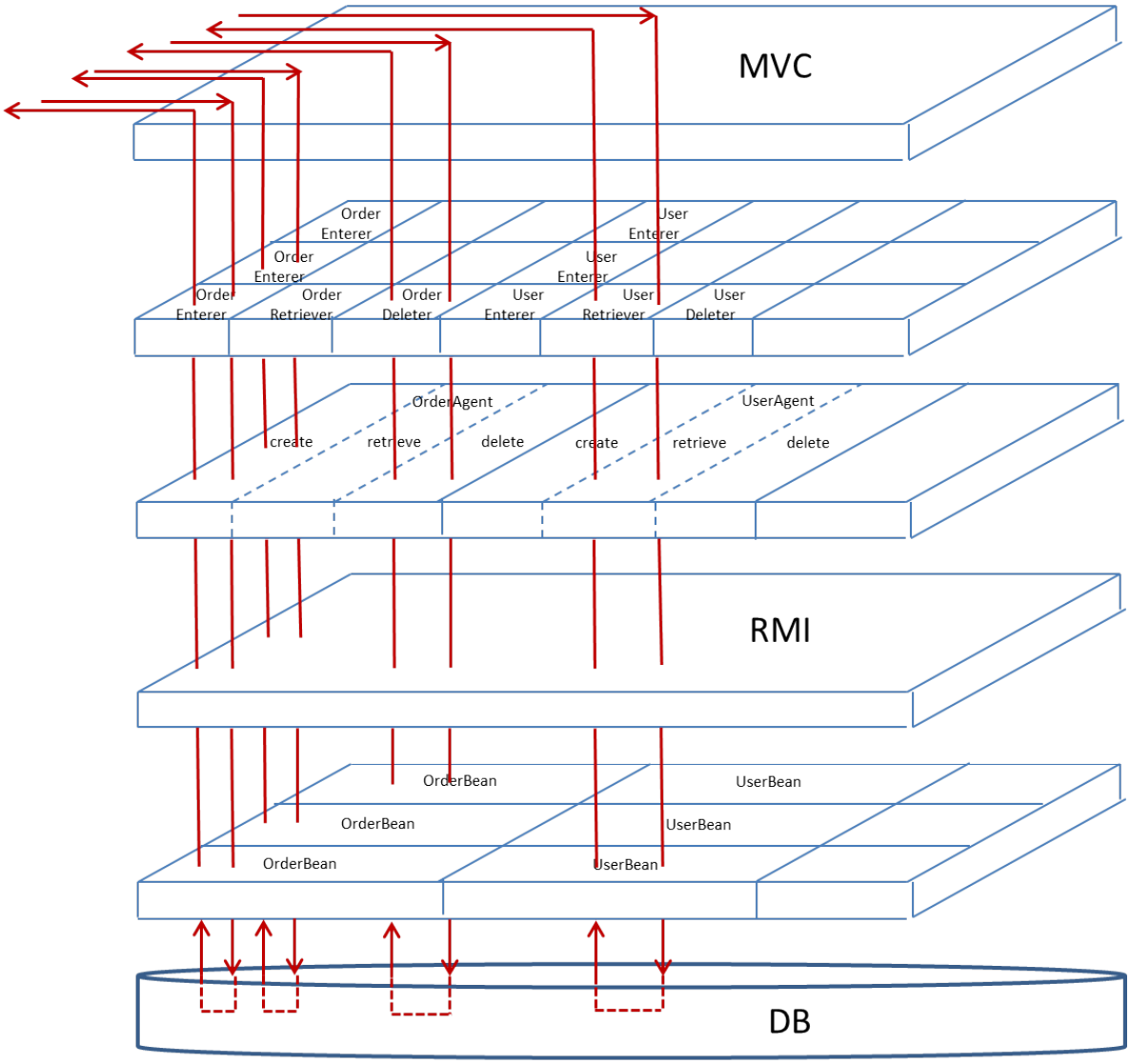

Figure 2. 3D visualization of a *state space* representing a running data entry system.

While this programming pattern is quite straightforward at compile-time, it is much more complicated at run-time in a multi-tier and multi-threaded system. Figure 2 shows a 3D visualization of the state space of such a running system, distinguishing three dimensions:

- The *functional dimension* horizontally, representing the various data objects (Order, User, ...) and operations (create, retrieve, delete, ...).
- The *multi-tier dimension* vertically, representing the various tiers: the webtier controlled by the MVC, the application tier called through RMI, and the data tier.
- The *multi-thread dimension* in the depth, representing the various threads of the incoming calls.

The drawing also represents the thread instantiation pattern of this simple software pattern for data entry. Every incoming thread leads to the creation of a new and separate instance of the appropriate action class, like `OrderEnterer`. The agents, like `OrderAgent`, are designed as singletons, and are a single point of access for that data element to relay the calls to the application tier. In the application tier, every call leads again to the creation of a new and separate instance of a bean class, like `OrderBean`, though the same class groups the various operations on a single data element or table.

The aim of this section in general and this drawing in particular is not to present a high quality pattern that should be preferred over other existing or conceivable patterns. Its goal is to explain that the run-time state of a software system in general and a programming pattern in particular, has many facets that are in general not explicitated in a compile-

 ISBN: 978-1-61208-184-7

time software pattern. And it is this complicated run-time statespace that needs to be analyzed and mastered, in order to control the number of possible microstates, and therefore the entropy of the software system.

B. Design Rules for Controlling Entropy

As our aim is to use the concept of entropy as guidance for software engineering and architecture, we have proposed to use the instantiations of programming constructs, such as functions or class methods, as the molecules or basic building blocks representing the microstates of the system. Knowing that local variables cease to exist after the processing function has been completed, and assuming that we do not allow hidden coupling or information leakage through all sorts of global variables, the microstate of such a function — being a correct or erroneous execution — can be studied in isolation from the rest of the system.

The synchronous pipelines that exist in Figure 2 in relaying the calls through the various tiers of the architecture, seem to pose a problem. Indeed, as the `act` method of the `OrderEnterer` is only completed after the calls to the other tiers, its microstate cannot be studied in isolation from these other processing methods. Splitting the method in its two parts, before and after the remote call, would introduce coupling between those parts through all the local variables of the method. Therefore, the *Separation of States* principle, as derived in [6], [7], seems to be in accordance with the concept of software entropy control. Indeed, if seperation of states is not adhered to (e.g., synchronous pipelines), it is not clear or uncertain where exactly a certain error or exception has occured in the modular structure. In such architectures, the macrostate (i.e., an error has occured) can be explained through many possible microstates (i.e., many possible causes related to many possible atomic tasks which are not seperated by keeping state) and by definition the amount of entropy in the software system increases. Of course, the use of synchronous pipelines in this example is quite innocent, as it is limited to relaying a simple data entry call through the various tiers of the architecture.

Also, because the essence of controlling entropy is the reduction of uncertainty, and the operations of a software system are described in terms of the various *tasks* or *concerns*, the *Separation of Concerns* principle (as derived in [6], [7]) seems to be in full accordance with the proposed concept of software entropy as well. Indeed, encapsulating every task or concern in a separate programming construct, would allow us in general to externalize the state of every task. Exporting every such microstate — through detailed loggings for instance — to the observable macrostate, would avoid the creation of multiple microstates for the same macrostate, and would by definition avoid the creation of entropy. Finally, while from a stability point of view the identification of concerns should be based on so-called change drivers, the identification of concerns from an entropy point of view should be based on so-called uncertainty drivers. While theoretically, concerns identified from one point of view or the other might turn out to be different, we anticipate that generally both perspectives will lead to the identification of the same concerns in practice.

Two other principles derived in [6], [7], *Data Version Transparency* and *Action Version Transparency* manifest themselves at compile-time, and therefore seem less relevant in this run-time analysis. However, the isentropic requirement demanding the ability to study every microstate of a processing function in isolation of the rest of the system, calls for the export of all microstate details to an observable macrostate. More specifically, this implies the following rules.

- *Data Instance Traceability:* the actual version and values of every instance of a data structure serving as an argument, need to be exported to an observable macrostate.
- *Action Instance Traceability:* the actual version of every instance of a processing function and the thread it is embedded in, need to be exported to an observable macrostate.

These observable macrostates may be implemented through a wide range of possibilities, ranging from simple loggings to more elaborate database entries.

VI. Conclusion and Future Work

Triggered by the identified need for evolvable software applications, Normalized Systems theory has previously been able to introduce a proven degree of evolvable modularity into software systems, based on the systems theoretic notion of stability. In this paper, we explored the applicability of this other fundamental property of systems, i.e. entropy, to the issues of software maintenance and evolvability. First, the underlying concepts of the statical perspective on entropy were applied to and defined specifically in the context of software systems. More specifically, the macrostate was related to observable output streams of the software system, and the microstates were defined as a set of binary values representing the correct or erroneous execution of programming constructs. Next, the complexity of real-life multi-tier software applications in terms of entropy was illustrated. In order to control the amount of entropy, some design rules were explored and related to the existing design principles of NS theory. This suggested an initial concordance regarding principles for software maintainability based on applying concepts from (1) systems theoretic stability and (2) entropy from thermodynamics.

Acknowledgment

P.D.B. is supported by a Research Grant of the Agency for Innovation by Science and Technology in Flanders (IWT).

 ISBN: 978-1-61208-184-7

REFERENCES

[1] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceeding of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[3] B. P. Lientz and E. B. Swanson, "Problems in application software maintenance," *Communications of the ACM*, vol. 24, pp. 763–769, 1981.

[4] Standish Group, "The standish group report: Chaos," Tech. Rep., 1994.

[5] H. Mannaert and J. Verelst, *Normalized systems: re-creating information technology based on laws for software evolvability*. Koppa, 2009.

[6] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," *Science of Computer Programming*, vol. Article in press, 2011.

[7] ——, "Towards evolvable software architectures based on systems theoretic stability," *Software Practice and Experience*, vol. Early View, 2011.

[8] Wikipedia. (2011) Second law of thermodynamics. [Online]. Available: http://en.wikipedia.org/wiki/Second_law_of_thermodynamics

[9] L. Boltzmann, *Lectures on gas theory*. Dover Publications, 1995.

[10] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.

[11] R. Janow, "Shannon entropy and productivity: Why big organizations can seem stupid," *Journal of the Washington Academy of Sciences*, vol. 90, 2004.

[12] Wikipedia. (2011) Entropy (computing). [Online]. Available: http://en.wikipedia.org/wiki/Entropy_(computing)

[13] C. Cachin, "Entropy measures and unconditional security in cryptography," Ph.D. dissertation, Swiss Federal Institute of Technology Zürich, 1997.

[14] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the linux random number generator," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[15] W. Harrison, "An entropy-based measure of software complexity," *Software Engineering, IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 1025–1029, nov 1992.

[16] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio, "Evaluating software degradation through entropy," in *Proceedings of the 7th International Symposium on Software Metrics*, ser. METRICS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 210–219. [Online]. Available: http://dl.acm.org/citation.cfm?id=823456.823991

[17] G. Visaggio, "Assessing the maintenance process through replicated, controlled experiment," *The Journal of Systems and Software*, vol. 44, no. 3, pp. 187–197, 1999.

[18] Wikipedia. (2011) Entropy. [Online]. Available: http://en.wikipedia.org/wiki/Entropy

[19] H. Mannaert, J. Verelst, and K. Ven, "Towards rules and laws for software factories and evolvability: A case-driven approach," in *Proceedings of the First International Conference on Software Engineering Advances (ICSEA)*. IEEE Press, 2006.

[20] ——, "Exploring concepts for deterministic software engineering: Service interfaces, pattern expansion, and stability," in *Proceedings of the Second International Conference on Software Engineering Advances (ICSEA)*. IEEE Press, 2007.

 ISBN: 978-1-61208-184-7