Using Managed Communication Channels in Software Components

E. A. Stoyanov  
University Of Stuttgart

M. A. Wischy  
Siemens CT, SE 2

Prof. D. Roller  
University of Stuttgart

Emil.Stoyanov@informatik-uni-stuttgart.de  
Markus.Wischy@siemens.com  
Dieter.Roller@informatik.uni-stuttgart.de

Abstract

The paper discusses the potential usage of principles from General System Theory (GST) and Cybernetics for design of Autonomic Software. Motivated by the characteristics of open systems and benefits of software communication management, we introduce the abstraction of Managed Communication Channels and propose a general purpose architecture for composition and activation of communication channels. We illustrate examples of their application in different aspects of component oriented design for increase of overall system stability. Prototype of framework for autonomic component communication using the discussed principles is presented.

1. Introduction

The growing of IT systems and respectively the needed human power for their management have lead to the idea of autonomic computing - an approach to design of software systems which are able to manage themselves in four aspects: self-configuration, self-protection, self-optimization and self-healing [36]. Aimed to manage the heterogeneity and complexity of the IT, the range of application of autonomic systems expands from mobile devices [4, 40] to data-centers [1, 9] and grid networks [20, 22, 24, 27]. During the last two years large efforts were spent for development of architectures [2, 3, 13, 39, 35] to enable self-management in existing and new systems, however few of them face the fundamental problem of variety in software from its roots – software evolution. This paper presents an approach to complexity management which targets directly the problem of stability in heterogeneous and complex software systems during their life-cycle. The main concept employs usage of principles from theories of open systems and control [43, 26], information theory[10] and software evolution[30, 31, 32, 33]. We state that the process of software development has characteristics, similar to the open systems' nature, such as adaptation, homeostasis and goal-directedness and show how the software architectures can profit from exploiting the Cybernetic laws of Requisite Variety (RV) and Requisite Knowledge (RK) in order to reach higher stability. To achieve this we introduce the abstraction of communication channel manager and observable software variety together with practical examples of their applications for construction of autonomic control loops and usage in common architectural approaches for component-oriented software.
2. Motivation

The complexity of IT systems has grown constantly and the maintenance throughout their life-cycle becomes more and more difficult. Because of the complexity behind the relations between different parts of a software system any single modification in the system model, implementation or configuration parameters may cause undesired overall instability in the system behavior. A common approach to system verification is to apply behavior models with automated test case generation and regression tests [19, 44] which aim to discover wrong system behavior after certain modification in the system. However these approaches consume enormous time and resources because of the large sets of test cases to be performed to reach satisfying degree of assurance that the system is stable. The main reason for that in most of the cases is due to the state explosion effect [25] which may be produced even in relatively small software programs. Different approaches to overcoming of this effect were developed, mainly using model minimization techniques and selective retesting algorithms [16, 17]. With the Internet revolution and growing popularity of distributed applications based on web standards [15, 45], pervasive environments [40] and on-demand resource provision it is already clear that difficulties using current methodologies for software stability assurance will raise to an unmanageable level. It becomes difficult to design a clear and fixed system behavior which can be used together with the above mentioned techniques. In relation to the Autonomic Computing initiative and having in mind the current issues with preserving overall software stability we were inspired to take a look at the development of modern IT systems from a different angle and focus on the problem from the roots of its development – the lack of a concrete model, which treats the software systems as open, adapting and evolving sets of recursively referencing and dependent open subsystems [30] including software components, human resources and business entities. Such a model should support stability of autonomic systems in two aspects: runtime support for adaptable business process execution and development support for stable extension and adaptation to new business requirements. We divided the way towards accurate specification of such model into three steps:

- Selection of relevant theoretical base and implications of the theories principles for concrete use cases
- Formal specification of system entities and relations between them
- Implementation and testing of real world scenarios using the proposed system model

In the remaining part of this document we will discuss the first point from this sequence (selection of relevant theoretical base). To understanding the dynamics of software systems evolutions we explore the results delivered by the research done by M. Lehman [31, 32, 33] regarding Software evolution and the theory on E-type and S-type systems which together with the achievements in control theory and cybernetics form a reasonable theoretic toolbox. We shall discuss implications of these theories in software development referencing proved design models for extensible and manageable software and introduce the first two essential elements of our model – observable variety and the communication channel manager.
3. Open Systems and Software

Important standpoint in our research is the similarity between the characteristics of nowadays software systems and the open systems as viewed by the General System Theory and Cybernetics – well established and proved by the time concepts related to understanding and control of biological and social systems[6, 7, 37].

The following characteristics apply to the currently developed business software:

- Software systems are aiming certain business requirements, which occur to change in time
- Systems are functionally adapted to meet these requirements, by being reconfigured, updated, or redesigned[18, 46]
- They are kept stable while they exist, by means of reliability and availability tests, iterative problem localization and correction [34]
- They have hierarchical organization [5] of functionally separated modules, e.g. resources, components, services, business process, etc.

Looking into the open systems definitions we notice the following characteristics:

- Goal-directedness. Systems have certain goals and follow them throughout their existence.
- Self-adaptation. Systems adapt to the environment and evolve in order to survive and reach their goals
- Homeostasis. Systems keep important parameters in certain range to assure their existence. (e.g. human body temperature)
- Hierarchical organization. Systems are organized in functionally separated levels to absorb as much as possible variety coming from external disturbances.

The mapped set of these characteristics of the open systems definition given by Cybernetics is shown in Table 1.

<table>
<thead>
<tr>
<th>Software systems</th>
<th>General Open Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Requirements</td>
<td>Goals-directedness</td>
</tr>
<tr>
<td>Configuration, Update, Redesign</td>
<td>Self-Adaptation</td>
</tr>
<tr>
<td>Stability Assurance</td>
<td>Homeostasis</td>
</tr>
<tr>
<td>Hierarchical organization</td>
<td>Hierarchical organization</td>
</tr>
</tbody>
</table>

Table 1. General Open Systems – Software systems mapping

Looking at the items in the right column of the table it is noticeable that they are primarily responsibility of the human in his role of the control force which pushes the system to evolve. As stated[36], human fault is the main reason for malfunctioning or failures in today IT Systems. Having Cybernetics as a tool to understand software as open systems we use the notion of homeostasis and define requirements which a software system should complete in order to acquire
self-stabilization characteristics and reduce the human-error factor. Our understanding of open software system requires exchange of information between related subsystems using shared conceptualization which gives the opportunity of external for the system entities to use it and become part of the whole system. This is valid for the development and runtime phases. In its development phase a system should be supported by an adaptive management which responds on time to the requirements of the environment in which the system will run. Extensive research on this level is provided by the E-Type theory of Lehman[31,32]. Figure 2 illustrates the life-cycle of a software program on the level of creation, adaptation and verification.

**Figure 1. Software development phase**

Important statement in this theory is that “e-type evolution processes are multi-level, multi-loop, multi-agent feedback systems”, which was formulated as the 8th law of Software Evolution[30]. Figure 2 shows this process by means of control loops.

**Figure 2. Software development phase control loops**

The software is a product and dependent part of a system which includes higher levels giving the needed requirements. On its behalf the software system is a standalone unit and its behavior and internal state variety are result from user input or interactions with other systems, sharing the same operating environment. In its runtime phase an autonomic system should be able to control its activity, reflecting the events coming from external for the system units. We assign to this class of systems the current distributed systems based on open standards, operating in networked environment. To introduce homeostatic software behavior[21], prior to defining the process of adaptation we need to take into consideration the related to the notion of homeostasis terms of variety and entropy [6,7, 10] and outline their respective meaning in software systems.
4. Observable Variety

The general definition of variety is *the number of states a system may have in its state space* [26]. Examples of variety in the context of open systems are the existing number communication protocols [12, 23, 41], management interfaces, event messages and event situations [11], process [42] and policy [8] definitions, sharing and execution specifications. We call the variety of that type, which is specified and accessible to third parties, *observable variety*. It is a quantitative measure for the size of the state space which a certain system element may have. A direct application of this definition is seen in the following scenarios:

- **Client-server interaction** - when a client module accesses a remote object on a server the system has to assure that the two communicating parts will complete their task. This model usually implies communication protocols on the lower level combined with access to remote interfaces on the higher level [12], which form the observable variety of the communication interfaces shown in Figure 3.

- **Logging and tracing** - an essential part of the process of debugging and analysis of IT systems, where different sources produce different debugging messages, which form the observable variety of the log and trace base shown in Figure 4.

- **Policy Management** – the managers monitoring and acting selectively. The system responds to *events*, reasons through *rules* and performs *operations* using actuators, which together form the observable variety of the policy manager shown in Figure 5.

- **Extensible Frameworks** – models for implementation of dynamically plugged and configured components [14]. The model provides a set of accessible control points together with context objects passing mechanism through which the system can be monitored or controlled. They form the observable variety of the framework model shown in Figure 6.

Variety shows the freedom of a system to take a certain state, but we are interested in system organization and how we can control it in software systems. We introduce the term *Entropy*.

---

**Figure 3. Observable Variety in Remote Method Invocation**

**Figure 4. Observable Variety in Log and Trace Bases**
5. Decreasing System Entropy

The Entropy of a system expresses quantitatively our uncertainty about the state a system may have and respectively measures the information that we have about the system. The expression of Entropy $H$ was originally given by Shannon in his Theory of Communication[10] and has the following form:

$$H = -[p_1 \log p_1 + p_2 \log p_2 + \ldots + p_n \log p_n]$$  \hspace{1cm} (1)$$

where $n$ is the number of independent states of a system and $p$ is the probability of choice of state number $i$. This definition of Entropy is closely related to the definition given by Thermodynamics concerning the decreasing organization or increase of chaos in closed systems. Equation 1 shows that the $H$ takes negative values. If entropy reaches value $H=0$ we have a probability 1 for a certain state while the rest of the states have probability 0, which fully describes the state of the system. Essential difference between closed and open systems is the change of entropy in time. The entropy tends to decrease in open systems which means increase of the internal system organization in accordance to system's goals and external disturbances. As stated by the First, Second and Seventh Laws of Software Evolution software has a tendency of increasing complexity and decreasing quality unless something is done to maintain this.

Decrease of entropy is a subject of research in the cybernetic community which reached results in the study of circularity and self-reference as key concepts to self-organization. The most clear example of a model exploiting the cybernetic principles for description of organizational behavior is the Viable Systems Model (VSM) proposed by Stanford Beer in 1970s and being polished in the next thirty years of his life. Designed to define an efficient way of knowledge management in enterprises, it is applicable to a wide range of fields such as social and organic behavior modeling. There are already ideas to apply this model to software architectures[21], however the problem of management of variety in communicating software functional groups and levels is still untouched, thus preventing
direct practical application of the included in the model principles. Keeping entropy in a low level requires on-time
control of the dynamics of the existing subsystems for the completeness of the viability requirements. Having the above
definition of observable variety we can use the rules of viable organization to specify homeostatic behavior in a
software system. The aim of the rest of this paper is to explain possible practical application of Beer’s principles for
development of autonomic components and component frameworks.

6. Requirements for Software Homeostasis

Along with a structural representation Beer gave a system of requirements which enable a system to be viable.
Requirements for homeostasis are put on the first place and can be explained with a minimized chart of system
organization, such as one presented on Figure 7.

![Diagram of Operation-Management-Environment](attachment:image)

**Figure 7. Operation-Management-Environment**

The diagram shows the variety channels between the related groups of management, operation and environment. To be
able to have a homeostatic behavior the operation needs to match its variety to the variety of the environment in which
it acts. In order to manage the operation group, the management has to match its variety to that of the operation. This is
the Beer's First Rule of Organization: “All communication channels between the functional groups of a system need
requisite variety” This requirement references the cybernetic Law of Requisite Variety which states: Control can be
obtained only if the variety of the controller is at least as great as the variety of the situation to be controlled. This
definition is expressed by the following equation:

\[ V(E) \geq V(D) - V(R) - K \] (2)

where \( V \) is the quantitative measure of variety of a system element, \( E \) is the set of the essential for system variables, \( D \)
is a disturbance applied to regulator \( R \). The parameter \( K \) represents available buffer for initial suppression of external
disturbances. At a first glance the law feels trivial, common and needless to prove, however its application in software
takes an important place which can be demonstrated with the examples on Figures 3-6. The client proxy needs to have
the variety that the remote object provides (Fig. 3); the component manager has to have the variety which the log base
and the managed component provide (Fig. 4); the policy engine has to handle the variety of the business process
parameters and the managed element (Fig.5). The system on Fig. 7 supposes existence of controlling unit and controlled
plant.
This set of simple rules and their direct relation to open software systems led us to the idea to specify an abstraction of a communication channel manager to enable verification of the First Principle of Organization, respectively The Law of Requisite Variety in the process of establishing a connection or relation between two software entities intending to exchange observable variety as defined in Chapter 4.

7. The Managed Communication Channel (MCC)

As shown on Fig. 3, 4, 5 and 6 distributed and extensible software systems put the question of compatibility and variety handling. Speaking generally of distributed software we assume communication between geographically separated instances of managed communicating components. In this context, the change in software variety is dependent on several factors: component evolution, introduction of new components or system re-engineering [18, 46]. Any of these factors may be a result from the change of the overall system requirements (system goals) as the evolution process in the development phase continues. However, here we focus on the runtime phase and examine handling of changing observable variety. Changing variety puts the following question to software architects:

- How to estimate and express the variation of components?
- How to compare evolving components and check for compatibility
- How to automate this process?

7.1 The MCC Approach

Different approaches exist to the answers of the first two questions but the lack of general understanding of software variety and its handling hinders the process of automation and that is what an IT system needs to achieve self-control in a dynamic environment. To refer this lack of general variety handling we use the accepted notion of communication channel and add active management components for evaluation of the observable variety to meet the requirements of homeostasis as discussed earlier. Communication Channel as defined by the Shannon's Theory of Communication is “the medium used to transmit a signal from the transmitter to the receiver”. It is necessary to note that direct application of Shannon's Theory in software is impossible due to the fact that although there is software variety it cannot be estimated using a generalized quantitative measure (bit). Looking at Fig. 3,4,5,6 we can easily outline handling of variety and where a communication channel may take place. As a common example of component communication we take the client-server situation and communication between a remote client and component container:
The transmitted software variety through this communication channel is in the form of events, interface calls or lower level communication protocol messages. The communication channel abstraction would free us from the actual way of communication but still focus on the variation of the transmitted information. We take a look at another useful example for the need of communication channel control and help us define its properties. On Fig. 9 is shown the familiar Extension Interface Pattern[14] which helps in handling components with evolving access interfaces.

The requirement for the client is to have:

1. Support and knowledge about the existing components
2. Support and knowledge of the respective component extension interfaces.

For a successful communication between the two endpoints A and B, a channel connecting them should be checked for provided and handled variety and return one of the possible states it can have:

1. Both endpoints are compatible and will have accurate two way communication
2. Both endpoints are incompatible and will not be able to transmit accurately variety
3. Endpoint A will receive variety accurately, but B not (A to B accurate).
4. Endpoint B will receive variety accurately, but A not (B to A accurate).
7.2 MCC Architecture

These states show the first required properties of such managed communication channel – communication endpoints, and channel manager.

![Figure 10. Communication Channel Manager](image)

The communication endpoints are associated with the Communication Channel Manager which responsibility is to notify the communicating pairs for the status of the channel's requirement for Requisite Variety. The transducer element in the architecture represents the mediation point which Endpoint1 uses to reference Endpoint2. It is responsible for filtering of the emitted variety or adding necessary variety (amplification) in a way to keep the channel consistent. The channel manager performs a control loop to verify consistency and if possible to suggest proper transducer to satisfy the requirement for proper communication.

![Figure 11. Application of MCC with software components](image)

Depending on the type of variety (component versions, deployment descriptors, etc), the manager is allowed to execute different strategies for requisite variety check and select the most appropriate in the time of loop activation. This
architecture is applicable in a number of software applications. Figure 11 presents the application of MCC with the types of observable software variety from Fig. 3-6. It illustrates the idea of common management of elements in a system and the way they depend on variety transformation. The right and the left columns represent communicating or dependable endpoints while the middle column is the component that might cause inconsistency or wrong variety communication (transducer).

### 7.3 Channel Activation

As discussed earlier the CMM targets management of communication between loosely coupled and geographically distant components. This aim demands a proper sequence for binding together the channel components and their proper activation. The channel composition process must be common and suitable for any purpose the communication channel may serve. Figure 12 demonstrates the proposed sequence of channel composition on a higher level.

The Registration Monitor is notified by components E1, E2 requesting to establish communication. It notifies the Transducer Selector for the type of input and output software variety T(E1, E2). The Transducer Selector makes a query to the knowledgebase containing available transducers for input/output variety E1 and E2. Once a proper transducer is selected, the Channel Factory is notified to prepare a channel for Endpoints E1 and E2 with Transducer T.

![Figure 12. Channel Preparation](image)

An essential component in this process is the transducer artifact knowledgebase. It contains description of available software variety transducers (protocol and interface adapters, event translators, etc) and provides services for querying and storage of description entries. It serves as a base for reasoning and coupling of the channel components. The process is managed by the Communication Channel Manager as seen on Fig. 13.

Because of the extensive research in the negotiation theory and multi-agent systems it was proposed to select a common and standard protocol for channel activation. Fig. 13 shows an adaptation of the Recruiter Interaction Protocol (Recruiter IP)[47]. It was adapted to suit the needs for proper autonomic manager loop and consequent channel activation.

The activation protocol has meaningful end-states corresponding to the possible manager conclusions for channel integrity pointed in chapter 7.1.
As seen on Fig. 12 the CMM Manager phase- transition follows the traditional for the autonomic computing Monitor-Analyze-Plan-Execute sequence. In this way the activation protocol can be implemented in already existing frameworks based on the Autonomic Computing Blueprints[2]. In the “Monitor” phase the manager receives the variety descriptors of the components and extracts the needed fields. In the “Analyze” phase it applies strategy for variety match and outputs recommendations which provide information regarding selection of transducer for transformation/translation of protocols or interfaces. The recommendations are taken into the “Planning” phase which prepares the needed transducer/proxy and in the “Execution” state performs the actual observable software variety transformation.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Manager State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>E1 SET</td>
<td>Accepts E1 variety metadata</td>
</tr>
<tr>
<td></td>
<td>E2 GET</td>
<td>Query E2</td>
</tr>
<tr>
<td></td>
<td>E2 SET</td>
<td>Accepts E2 variety metadata</td>
</tr>
<tr>
<td>Analyze</td>
<td>FIND TRANSDUCER</td>
<td>Search transducer</td>
</tr>
<tr>
<td></td>
<td>VAR NOT SUPPORTED</td>
<td>Cannot parse variety metadata</td>
</tr>
<tr>
<td></td>
<td>VAR SUPPORTED</td>
<td>Metadata supported</td>
</tr>
<tr>
<td></td>
<td>NO TRANSDUCER</td>
<td>No matching transducer was found</td>
</tr>
<tr>
<td></td>
<td>TRANSDUCER FOUND</td>
<td>Matching transducer found</td>
</tr>
<tr>
<td>Plan</td>
<td>SET TRANSDUCER</td>
<td>Init transducer</td>
</tr>
<tr>
<td></td>
<td>E1 TRANS ERROR</td>
<td>Transducer failed to convert</td>
</tr>
<tr>
<td></td>
<td>E1 TRANS SUCCESS</td>
<td>Transducer initialized</td>
</tr>
<tr>
<td></td>
<td>E2 TRANS SUCCESS</td>
<td>Notify E2</td>
</tr>
<tr>
<td>Execute</td>
<td>CHANNEL ACTIVE</td>
<td>Channel is active and</td>
</tr>
<tr>
<td></td>
<td>E1 START</td>
<td>Initiate send/receive on E1</td>
</tr>
<tr>
<td></td>
<td>E2 START</td>
<td>Initiate send/receive on E2</td>
</tr>
</tbody>
</table>

Table 2. MCC Manager States
The protocol and the behavior of the autonomic channel manager can be implemented as a finite state machine with a set of states shown on Table 2. These states ensure proper CMM composition, thus reliable communication between components. A real world example and mapping of these states to concrete situations is discussed in Chapter 8.

8. Enabling Autonomic Communication in Component Frameworks – An OSGi Scenario

In order to verify and test applicability of the MCC concept we apply the MCC concept to a scenario for management of the Open Services Gateway Initiative (OSGi)[48] platform. The primary goal of this effort is to deliver a healthy environment for OSGi applications that use services and bundles that might have been deployed multiple times and have different versions. A prototype for this particular place is being developed.

We examine a scenario with the following settings and prerequisites:

- BundleX is dependent on service provided by BundleY
- BundleY has more than 1 deployed version
- BundleX requests service from BundleY
- BundleX communicates with the right version of the BundleY

In such environment is very important for the service client to resolve properly the correct service version. Otherwise the retrieved service reference may point to a wrong implementation, thus invalid type casting which may lead to undesired system behavior. In the current OSGi V3 Specification this behavior is missing but necessary for a smooth migration to new components versions. For better understanding of the direct application of the channel composition and its activation sequence Table 3 shows a mapping of MCC Manager’s states which were already explained in Table 2 with the managed type casting of OSGi bundles using the MCC method.

For the implementation of the CMM management concept in OSGI we follow the architecture shown on Fig. 12. There are three components for establishing reliable communication between by BundleX and BundleY.

- Channel Management Service Bundle – this is the low-level channel framework implementing the MCC Composition (see 7.2) and MCC Activation Protocol (see 7.3)
- Bundle Deployment Tracker – monitors OSGi events for registration, update and deregistration of bundles. Acts as a sensor for system events.
- Channel Resolver – an OSGi service using the lower level channel management to locate and resolve the proper service/bundle version. It is used as an alternative for the standard service locator.
Manager State | Scenario Situation
--- | ---
E1_SET | BundleX requests service from BundleY with version requirements
E2_GET | Manager queries for deployed instances of BundleY
E2_SET | Manager gets BundleY’s version
FIND_TRANSUCER | Manager queries knowledgebase
VAR_NOT_SUPPORTED | Manager is unable to parse the metadata of either BundleX or BundleY
VAR_SUPPORTED | Manager has parsed successfully metadata
NO_TRANSUCER | It was impossible to narrow the requested service reference.
TRANSUCER_FOUND | Manager found either a proper version of BundleY or a helper-service to adapt BundleY
SET_TRANSUCER | Request BundleY
E1 TRANS ERROR | Adapter failed to convert
E1 TRANS SUCCESS | Adapter initialized
E2 TRANS SUCCESS | Notify E2
CHANNEL ACTIVE | Channel is active and
E1 START | Initiate send/receive on E1
E2 START | Initiate send/receive on E2

| Table 3. Manager States - Scenario Mapping |

Figure 14. OSGi MCC Bundles

The arrows on Fig. 15 show the event flow between the components. As it is seen system events are sensed by the Bundle Deployment Tracker and passed to the Channel Framework as a declaration of the new variety injected in the system. Bundle X uses the Channel Resolver to locate and later invoke a service from Bundle Y. The Channel Resolver composes and activates the channel using the Channel Management Service. The channel remains active only during the process of locating and narrowing the service reference. After that the two components communicate directly. The prototype is still under development and concrete performance tests are to be done at a later stage for measuring the delay in the process of service locating and impact on the overall system performance.

9. Conclusion

In this article we introduced an interpretation and adaptation of a set of GST and Cybernetic principles and discussed their relevance to development of adaptive software systems in the context of increasing their reliability. The abstraction
of Observable Variety and Managed Communication Channel were presented as a concept and model for building reliable autonomic software systems. An example for real-world application of the concept was discussed as a demonstration of the concept of autonomic variety communication. Future research extends the application of the concept for integration of software components and their relation to dynamic and adaptive business processes.

10. References


[36] P. Horn, Autonomic Computing: IBM's Perspective on the State of Information Technology, IBM Corporation (October 15, 2001);


