Dynamics of System Evolution

Ashirul Mubin[1], Rezwanur Rahman[1] and Daniel Ray[2]
[1]*University of Alabama, Tuscaloosa, AL,*
[2]*The University of Virginia's College at Wise, Wise, VA,*
USA

1. Introduction

A system is built to serve a common purpose of an organization or a network; it usually consists of a set of operations, interfaces for inputs and outputs, and a group of users with direct or indirect interactions. Systems exist in nature as well as in virtually any conceivable area of human society (Dori, 2003). We are surrounded by systems which undergo changes over time and experience some sort of evolutionary pressure. In order to formulate a system with its specifications, a complete set of updated requirements is established before delving further into the development process. Here the presumption is that, based on the specified requirements, the system would adequately serve the underlying community within its predefined life cycle. However, like any other objects or materials, the system will gradually become outdated over an extended period of time (unless any newly emerged requirements are addressed); this is due to the changes in its surrounding environment, which includes end users, groups of people involved through meetings or other common interests, their mutual interactions with other systems or exchange of information through social networks, related auxiliary or dependent systems and its type of services to the community. In the end, the system will lose its value over a period of time and it is a common fact, but unavoidable scenario, unless explicit measures are taken for re-configuring the system with new specifications. In other words, the current expiring features need to be replaced with newly emerged requirements so that it can maintain its efficacy and remain competitive in the market. Traditional systems do not have such capabilities to address emerging requirements. These systems either need to be thoroughly re-engineered, or simply replaced with a new system. Both of these options are very expensive in all aspects. With the help of a "*Wrapper system,*" if a system can identify these upcoming requirements and able to direct necessary changes into the system itself by dynamically adjusting its specifications, then it will be in a good standing to extend its life cycle, and maintain a higher level of user satisfaction through its dynamic configurations. A wrapper system is a real-time system itself; it is a carefully formulated meta-structure to address dynamic configurability. In this setup, the target system can be termed an "*Evolvable system*" which, via its adaptability, gradually makes a valuable return of investment over an extended software lifetime.

Normally, when a system is built and deployed into a production environment, it becomes very difficult to change or upgrade it. Additionally, to take down the service for maintenance or upgrade without the complete knowledge of the problem scope, is also very expensive. Several examples of these non-evolvable traditional systems are listed below:

- *Vending Machines* -- lack the ability to track purchase rates, assess current inventory and automatically notify when it is necessary to reorder. These also lack the ability to capture the underlying changing patterns of seasonal purchasing habits for that locality.
- *Microwave Ovens* -- lack the ability to assess the weight-volume of the food, and the intensity or duration of cooking, and it cannot track heating patterns of meals in a household.
- *Elevator systems* -- lack the ability to learn and communicate with other elevators, to assess load balancing, deduce operation schedules based on usage patterns and cannot notify when it is the right time for maintenance.
- *OBD Code Readers* -- lack the ability to suggest the probable cause(s) of the problem from previous history, predict any upcoming related issues, or inform the manufacturer with the estimated fault rate of certain parts so that their next model can eliminate any such issues in the future.
- *Document Management Systems* -- lack the ability to generalize the identification of input locations of data fields in the paper-based forms with the help of OCR-texts accordingly and cannot suggest probable filing destinations in the database.
- *Security System of Buildings* -- lacks the ability to identify and track object movements or sounds generated from certain areas, and to capture occupancy patterns by observing and comparing over a period of time.

Without the options for reconfigurations to meet these shortfalls, the above mentioned systems cannot be termed as evolvable systems since it will be very difficult to make necessary changes outside of their preset functionalities. Such rigid and non-configurable systems will become outdated at some point and will need to be replaced with newer versions. Otherwise, some continued laborious support will need to be provided to go on with the current settings. To avoid this, a system should be as dynamically reconfigurable as possible. It opens up a wide range of opportunities to address many emerging requirements through fine tuning specifications from any desirable perspective.

Building a system itself is not enough; the main exhaustive part emerges from the great effort to sustain and keep pace with ongoing demands. Surveys indicate that on average as much as 70% of projects software budget is devoted to maintenance activity over the life of the software (Port, 1988; Bennet, 1990). Maintenance of any system is inevitable, either to enhance the system by altering its functionality, or to adapt the system to cope with the changes in the environment; to correct newly discovered errors, or to update the system in anticipation of any future problems. Therefore, it is becoming increasingly important to consider future system maintenance activity as it is designed and developed (Ferneley, 1998).

Our area of concentration for the study of system dynamics focuses on software-driven processes in general, because they have the capability of automatically collecting system's pre-configured meta-data from various junction points in the workflow, as well as from its surrounding environment. It can also provide real-time analytics and the flexibilities in deducing meta-models for dynamic configuration of system specifications. Having a supporting sub-system can also enable the architects, designers, developers and stakeholders to have real-time snap-shots of system states. At any instant of time they can view how well the system is performing its services, examine the current workload of the system, track the history of system usage patterns, and detemine imminent changes. Moreover, it can provide clear insight into the system and collect important feedbacks and analytics for necessary changes being applied into the system. These capabilities are the *primary constituent elements* of an evolvable system. In this way, a newly built system is expected to have a way to adapt with any additional future requirements that were imperceptible at the time of initial system design. These new requirements gradually emerge by frequent and recurring usages of the system surrounded by an operating environment over a long period of time.

The primary goal here is to be able to address these newly emerged requirements and then apply them into the system already in production so that it can continue to meet the incremental needs of the end users. To implement such versatile capabilities into a system requires a set of additional supporting components that will efficiently capture detailed system usage patterns throughout its operating workflow and, implicitly collect new system requirements from users by survey agents, automated collection of system usage patterns, and random voluntary feedback over a period of time. It is vital to look for any evolutionary changes, or indications through carefully analyzing the meta-data collected from the system. The objective is to be able to reflect in its behavior any ongoing changes in the surrounding environment so that it may continue to serve satisfactorily with a high value of return in the competitive market.

In this chapter, we discuss the development efforts to identify general terms and metrics that are necessary to track a system's upcoming evolutionary phases. We present higher-level analyses of these metrics through examples of several years of systems development track history and usage data in multiple projects in order to discover any significant implications of applying new changes towards their extended system lifecycle. Based on our rigorous observation, we also derived a preliminary methodology to formulate system dynamics towards their evolution, which can be followed or modified as needed for the purpose of building evolvable systems.

2. Evolvable systems

Evolution is often an intrinsic, feedback-driven, property of a software-based system. The meta-structure (as mentioned in Section 1), within which a system evolves, contains a number of feedback relationships (we will see more details in Section 3). The organization and environmental feedbacks transmit the evolutionary pressure to yield the continuing change in the process. The rate at which a program executes, the frequency of usage, user interactions with the operating environment, and economic and social dependencies of external processes on the system in production-all these cause its deficiencies be exposed over a period of time (Lehman, 1980). Therefore, these deficiencies to eventually become newly emerged system requirements that need be addressed to ensure the sustainability of the system.

Program maintenance is generally used to describe all changes made to a system after its deployment. With maturity of software development practices, maintainability of the resulting products (i.e. software-driven systems) has become one of the most important concerns in recent years. This is because we need systems to be evolvable to avoid any failed investments. Evolutionary behavior relates to attributes of relevant software processes, their components and relevant domains or environment. Attributes (such as system size, complexity, efforts applied, and the rate of changes) reflect aspects of its evolutionary behavior. Measurement or estimation of these attributes has provided a basis for the study of software evolution dynamics (Ramil & Lehman, 1999). By classifying programs according to their relationship to the environment in which they are executed, the sources of evolutionary pressure on computer-based applications and programs can be identified (Lehman, 1980).

The dynamic evolutionary nature of computer-based applications, the software that implements them, and the process that produces, introduce the concept of system lifecycle management as a whole. In studying system evolution, the repetitive phenomena that define a lifecycle can be observed on different time scales representing various levels of abstractions (Lehman, 1980). The laws of system evolution include: (1) continuing change, (2) increasing complexity, (3) system dynamicity (which is subject to measures of system attributes that are self-regulating with statistically determinable trends and invariance.), (4) conservation of the organizational stability, and (5) conservation of familiarity (Lehman, 1980). Since they arise from the habits and practices of users and organizations, their modification or change requires involving the surrounding environment, and cross into the realm of sociology, economics and management.

Each system can evolve differently, based on the type of its functionalities, services and the way it is used or consumed by the users. Therefore, the impact of foreseeable evolutionary changes varies with the nature of the system itself. For example recurring usage cycles (eg. yearly event), continuous roll-over usages (eg. viewlist online), or aperiodic/ad-hoc usage (eg. on-demand services).

2.1 Prerequisites of evolvable system

Like the dynamic evolutionary nature of complex social networks or economic systems and the processes of their subsistence, any software-driven system should have enabling processes to sustain itself over a longer period of time. For a system to be evolvable, it needs to be more flexible in interaction with not only the end users, but also various self-contained meta-data collecting agents or data-loggers (Lehman, 1986). These might include automated survey agents, probing points, or task request history (Mubin & Luo, 2010a). Table 1 lists some of the desirable characteristics of an evolvable system.

The characteristics mentioned in Table 1 strongly suggest that there should be a wrapper system responsible for both collecting meta-data, as well as applying the desired changes. Such a wrapper system should be built in parallel to the system itself (Mubin & Luo, 2010b) with equal emphasis.

System Characteristics	Descriptions
Encourage a high-degree of user involvements	Involve users in various phases of system development life cycle (including post development activities) and deployment activities; distribute significant real-time system usage statistics to the users while using the system; present analytical summary reports at the relevant areas in the system workflow to enable users to become more aware of the dynamicity of the system components and their interactions.
Actively participate in system feedbacks	Users are informed with ongoing development and new upcoming changes, and infrequent requests for feedbacks or suggestions from them. It can also implicitly encourage them to drop few responses to very simple questions at their will, while traversing through the system workflow.
Experience of uniqueness	System is able to personalize each user's experience based on the system usage patterns from activity log; this would help a user to feel as if it is molded towards his/her own expectations from the service provider.
Ability to provide current system state	The meta-structure should be able to provide instant pictures of the system from various viewpoints. This dynamicity is also a form of self-advertisement, as it tends to attract all user groups, developers and stakeholders where they are able to view real-time results of interests; and thereby help all in the community to further promote the services in a more guided way.
Flexibility in the incorporation of new features	User is both aware of necessary changes and is engaged with the system's flexibility and openness for incorporating new adjustments in the future.

Table 1. Desirable Characteristics of an Evolvable System

3. Operational environment

Conceptual models are the means by which software intensive systems are conceived, architected, designed, and built (Dori, 2003). To realize such a conceptual model, we introduce a wrapper system that interacts with the system itself and its surrounding environment. This interaction is a great source of valuable indications about the system dynamics. A system cannot be successful unless it can effectively communicate with its environment. Therefore, it is important to establish a controlled link between the system and the environment (Lehman, 1986) and collect the interaction behavior as completely and efficiently as possible. Data should be collected from any sort of direct/indirect feedbacks, inquiries, error logs, usage patterns, and real-time analytics – all with timestamps saved into knowledgebase so that the underlying meta-structure is able to produce meaningful insights about the system's current state, performance, and any noticeable changes in the usage patterns. Cultural changes, over a period of time (in the surrounding environment) also affect how users view and perform their works and interact with other groups of users and service providers.

3.1 System reconfiguration

Systems and models are intimately related. Modeling is a set of abstract artifacts representing systems (Dori, 2003). Therefore, we can develop a meta-model while designing a system that will abstract out the system specifications by carrying over the underlying adjustable parameters. This mechanism may impose additional tasks at the developers end, and will incur further expenses; however it will open up a wide door for any future adjustments without the need for refactoring or overhauling the system. Thus, it is a valuable investment to the more contolled software maintenance activities. Such a wrapper system provides direct interactions among the system users, surrounding environment, and the service providers. This link (or communication path) enables the environmental responses on new changes that need to be applied as feedback into the system. Figure 1 shows an example of such placement of a wrapper system. A set of automated survey-agents and probing stations (Mubin & Luo, 2010a) are responsible for capturing various usage factors from the system itself (such as, access rates, feature ranks, utilization factors, path traversals), from its operational environment (such as, system users, work habits, workstations, workspaces, user feedbacks, new system requirements, user satisfaction index), and from the external environment (with influential factors).

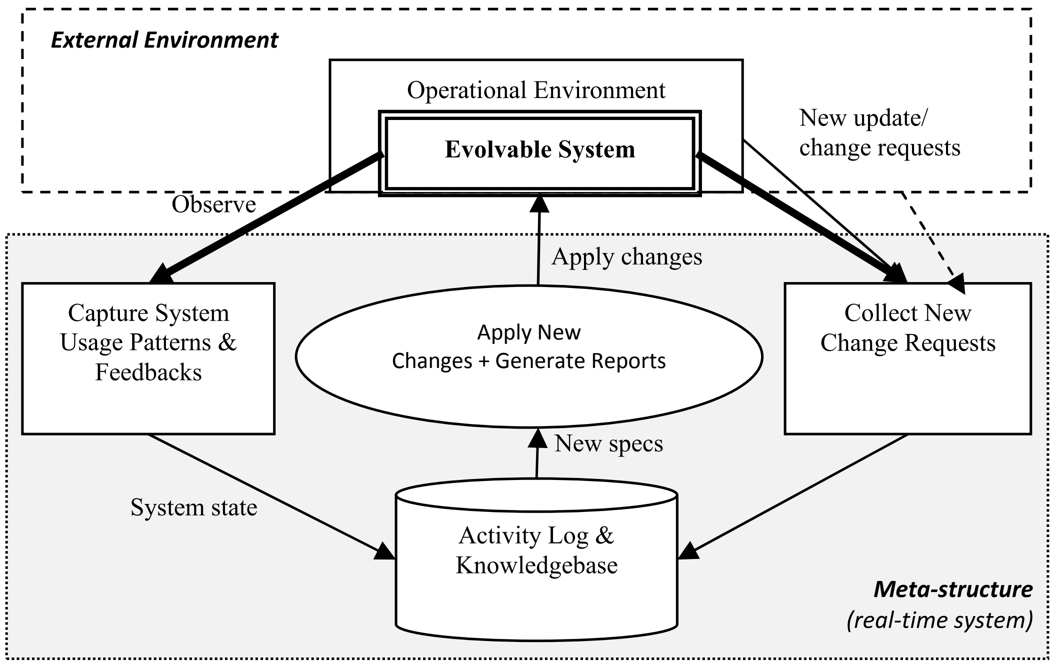
External Environment
Operational Environment
Evolvable System
New update/ change requests
Observe
Apply changes
Capture System Usage Patterns & Feedbacks
Apply New Changes + Generate Reports
Collect New Change Requests
New specs
System state
Activity Log & Knowledgebase
Meta-structure (real-time system)

Fig. 1. Setup of a Real-Time Meta-Structure for a System to become Evolvable

3.2 Real time feedback

The feedback concept is at the heart of the system dynamics approach (System Dynamics Society); it emphasizes a continuous view that strives to look beyond regular services to see the underlying dynamic patterns. In our setup, there are four feedback loops in the proposed operational environment through the meta-structure: (1) system usage activities are observed, and quantified into utilization factors, satisfaction indices and system state; and the knowledge base identifies whether there are any noticeable changes, and instructs the application of this change into the system, if necessary. Similarly, any change or update requests from the (2) system (3) operational environment, and (4) external influences are also applied through the service queue. Thus, over a period of time, outcomes from these activities will result in the corresponding *loop dominance*, which are then traced back into the knowledge base. The responses of such a meta-structure not only depend on logical inferences, but also on the physical instant from which the outputs are produced.

4. System dynamics

The central concept of system dynamics is the idea of two-way causation or feedback (Meadows and Robinson, 2002); it is assumed that social or individual decisions are made on the basis of the *flow* of information about the system state or environment surrounding the decision makers. The decisions lead to actions that are intended to change the state of the system. New information about the system state then produces further decisions and changes. Each such closed chain of causal relationships forms a *feedback loop*. System dynamics models are made up of such loops linked together (as shown in figure 1).

In other words, system dynamics include a set of concepts, representational techniques, and beliefs that make it into a definite modeling paradigm. It emphasizes a *continuous view* (Richardson, 1999) and looks for *causality* that underlies the longer-term patterns of change in systems. Here, the focus is on the general dynamic tendencies; under what conditions the system as a whole is stable or unstable, oscillating, growing, declining, self-correcting, or in equilibrium state (Meadows and Robinson, 2002). Properties of dynamic problems contain quantities that vary over time; such variability is described by the *causality*, which influences a closed system with *feedback loops*. Thus, dynamic systems are characterized by interdependence, mutual interaction, information feedback and circular causality. Every system experiences a number of changes over time and only in keeping detailed track-records of these behavioral changes can visualize the system dynamics. Therefore, the core objective is to tool the wrapper system so that we can capture dynamic system behavior, track any change history and discover correlations, as well as analyze and act upon it for better service for end users and thereby extend the system lifecycle.

4.1 Background of dynamic system behavior

As mentioned earlier, each dynamic system should consist of one or more loops running over a period of time as a part of its lifecycle. The items that affect other items in the system but are not themselves affected by anything in the system are called exogenous items. These exogenous disturbances are seen as the triggers of system behavior. *Positive loops* tend to amplify any disturbance and to produce exponential growth: if the cause increases, the effect increases and if the cause decreases, the effect decreases. *Negative loops* tend to counteract any disturbance and to move the system toward an equilibrium point: if the cause increases, the effect decreases and vice versa. Stable conditions will exist when negative loops dominate positive loops. Non-linear relationships can cause feedback loops to vary in strength, depending on the state of the rest of the system. The dominating loop might also shift over time; linked non-linear feedback loops form the patterns of shifting loop dominance. Under some conditions one part of the system is very active, and under other conditions, another set of relationships takes control and shifts the entire system behavior; i.e. one particular loop is always responsible for the overall behavior of that system.

The timing of system behavior depends on the presence of system elements that create inertia or delays. These inertial elements are referred to as state variables (see section 4.2). Each state is an accumulation (*stock*) of information. System elements representing the decision, action, or change in a state variable are indicated by a *flow* of information to/from a state variable. Also, there could be time-delays in information flows, and we need to look for any lagged relationships in the system.

4.2 Transition of system states

A system state can be defined as a report on system status from different perspectives for an instant of time. For example, the current performance rate of the system, how well the system is doing in terms of its user satisfaction or system value, or what is the current condition at the workflow nodes in the system graph (i.e. utilization factors) and so on. By observing system states and their transitions over a period of time, we can analyze the causes behind each transition, re-assign edge probabilities in the workflow to fine tune each transition and formulate system behavior accordingly.

As explained in figure 1, after running through one or more recurring cycles during the initialization phase of a system in production, the underlying knowledge base will begin to identify the system's state of transition, in terms of its variations in system value. Based on the specific conditions or causes (such as task requests arrival rates, task completion rates, changes in user satisfaction index, or system access rates) this, in turn, will trigger a state transition and indicate whether any newly captured requirements need to be placed in the service queue. After application of these new changes into the system by the server (a process that performs the new service requests, not necessarily automatic), the wrapper-system will continue to observe the new state and will mark any changes in the system value or in the user satisfaction levels by repeating the process.

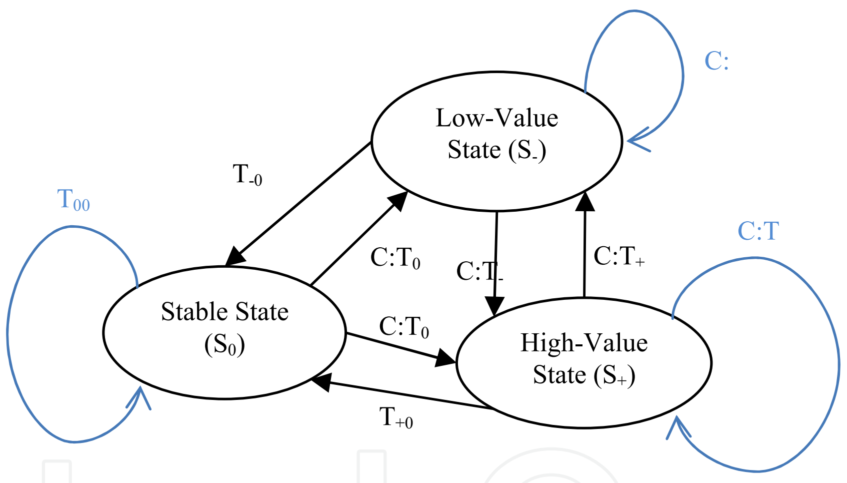

Fig. 2. Schematic diagram of an Evolvable System based on concepts of Finite State Machines

To maintain the system in a stable state (and avoid any reduction in system-value), a state-machine, as shown in figure 2, can be established to build an Ordinal Response Model (such as -2, -1, 0, +1, +2 to indicate system status and new transition) so that a state transition can be triggered to notify any inception phase of an upcoming evolution, the end of an evolutionary change or positive/negative loop dominances. There are three states of relative system value: stable state (S_0), downward or lower state (S_-), and upward or higher state (S_+). Based on this, we can deduce nine possible state transitions, labeled with combinations of $T_{[0,\ -,\ +]}$, as described in figure 2. Each transition is associated with one or more causes ($C_{T0..Tn}$). These may include task request arrival rates, task completion rates, delay in addressing requested changes due to complexity, applying new features into the system, changes in system access rates, variations in feature rank, utilization factors, etc.

The primary assumption is that the persistent dynamic tendencies of any system arise from its internal causal structure. With no causes or activities, the system will come back to a stable state. One or more causes will initiate a trigger, which in turn may cause another trigger from that point. Based on the collected data from survey-agents, probing stations and other means (Mubin & Luo, 2010a) the knowledgebase is responsible for identifying a cause in $\{C_{T0}..C_{Tn}\}$, where C_{T0} being no cause (nothing significant happened). The influential cause(s) of the transition, as well as the factor(s) in the operational environment, are archived into the knowledgebase for contextual analyses at a later point. Based on further statistical analyses of the current system and past track records each transition is associated with a *transition* probability. This may help the system avoid overshoot issues against a preset threshold value. Out of three different states, there are nine possible state transitions (see table 2) which can be mapped into an "acceptability" matrix, where each cell is a response-indicator for the current system dynamics of an evolutionary phase.

Transition due to occurrences of a set of causes, in $\{C_{T0}..C_{Tn}\}$	***Destination State***		
Begin State	Stable (S_0)	High-Value State (S_+)	Low-Value State (S_-)
Stable (S_0)	T_{00}: Okay (nothing significant happened)	T_{0+}: Good (High-yield)	T_{0-}: Bad (trigger)
High-Value State (S_+)	T_{+0}: Okay (new stable state)	T_{++}: Good	T_{+-}: Bad (trigger)
Low-Value State (S_-)	T_{-0}: Good (new stable state)	T_{-+}: Good (Ideal case)	T_{--}: Bad (trigger)

Table 2. Evaluation of State Transition Matrix for Acceptability Measure

Ideally, we would expect the system to maintain either at a stable state or at a higher-value state. However, due to environmetal factors, we cannot ensure such elevated states. Rather, the system would often experience low-value state over a period of time. Based on all system related feedbacks, new task requests, and applying new changes if a system experiences low-value state for a time period, we may need to reconfigure the meta-structure to generate a trigger. System architects, analysts and stakeholders would observe that the system runs for a while and then fine tune these speficications according to their expectations.

5. Measurement metrics for system evolution

We need to be able to instrument a system so that it becomes sensitive to any changes in software characteristics. Any appropriate measurement is a catalyst for improvement, it helps in system advancements, and improves the system's bottom line (Dekkers & McQuaid, 2002). The users' behaviors and contributions towards using a system's service(s) should be the integral measurement collected. A software-driven system's characteristics, usage and task request history, and its surrounding environment are all useful in measuring the quality and maintainability of that system (Coleman, et el., 1994). Measurements of these characteristics can be incorporated into system dynamics which can then be properly utilized as input to the upcoming evolutionary phases. It is always beneficial to outline the purpose and scope of the measurements within the context of system evolution. In addition, the metrics concerned with the measurement of an evolvable system's attributes will themselves be evolved, as needed. In addition to measuring the software itself, we would like to measure how the system impacts its users, as well as how the incremental additions to the system configurations influence the performance over a longer period of time. In summary, the set of measurement metrics are not fixed; instead it is up to the system's architect and stakeholders to pick and define any necessary metrics deemed useful upon careful observations or experimentations.

5.1 Classification of metrics

A system that meets the needs of its users will reinforce satisfaction with the major elements of that system (Ives et el., 1983). Otherwise, the system value will continue to degrade. Metrics can be used to specify what we want, to predict what we can expect to get, to measure what we've got, and to control variations between desired and actually attained values of various attributes of software products (Sherif et el., 1988). Metrics-based meta-structures help in fact-finding and process-selection decisions. At the component level, these models can be used to monitor any changes to the system as they occur, then fine tune their values. Also these help to predict fault-prone components (Coleman, et el., 1994). In other words, we can also analyze the values of metrics to generate a trigger that will indicate an inception phase of an upcoming system evolution.

For the purpose of studying system dynamics, we need to make the measurement a part of the overall development process and investigate three possible categories of metrics: (1) metrics of workflow graph, (2) metrics of system usage, and (3) metrics of development activities. The following sections discuss these classes of metrics in detail. Later in the chapter we will provide case studies on applying some of these metrics and their outcomes.

5.1.1 Metrics of workflow graph

Given any system or process, we can draw its information flow graph. The major elements in the information flow analysis can be determined at the system design time. The availability of this quantitative measure early in the system development process allows the system structure to be corrected with the least cost.

Also, by observing the communication patterns among the system components, we can define measurements for complexity, module coupling, level interactions, and stress points (Henry & Kafura, 1981). Figure 3 gives a simple example of mapping a system workflow into an equivalent graph model, with nodes (based on corresponding processes or decision points) and edges (based on the control flow paths). Hence, a general rule for mapping workflow processes, decision points and control paths can be defined by the system management team with the main objective that covers all areas with a set of nodes and edges of a graph model and apply various metrics related to graph models (such as McCabe’s Cyclomatic index). Then at the junction points in the workflow, we may place data-loggers (Lehman, 1986) or, more specifically, probing stations or survey agents (Mubin & Luo, 2010a).

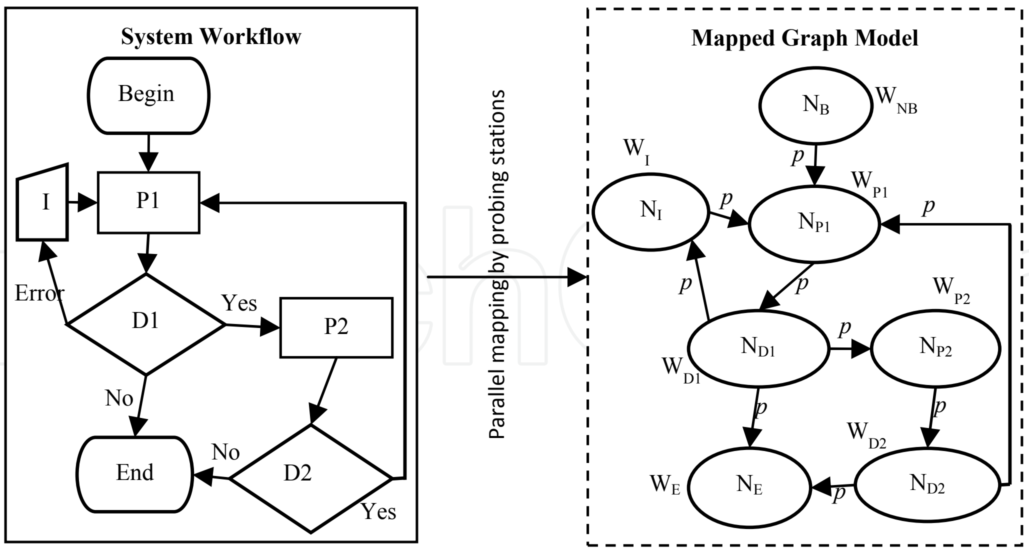

Fig. 3. Mapping of system workflow into an equivalent graph model with transition probabilities and process weights

5.1.2 Metrics of system usage

From the mapped model, we can now introduce a set of system usage metrics to help assess system dynamics. As denoted in the figure 3, each node (N_i) is assigned with a weight (W_{Ni}) calculated from its constituent set of features. A node may or may not contain components with features of concerns. If not, the weight will be zero. The chance of traversing through an edge from one node to another is based on the desired usage patterns and branching factors from which the initial probabilities (P_E) are assigned. The functionality of a process may be assigned a Feature Rank (FR_i), which is based on its significance of contribution into the service provided by the system as defined by the stakeholders and its expected usage frequency. Feature Utilization (FU_i) is the number of times a feature is actually used within a predefined usage cycle. A feature may be more under-utilized than expected, or over-utilized. The User Satisfaction Index or USI (Mubin et el., 2009) is a quantitative indicator of the overall satisfaction of using the system during a predefined time frame. With the combination of USI's for all groups of users, we may deduce a quantitative system value and compare it against other metrics for overall assessment of system evolution. As mentioned earlier, for an evolvable system measurement metrics should be configured in a way so that they can be re-adjusted or fine tuned as needed.

Suitable weights for nodes (i.e. different components or characteristics) can be assigned based on how important it is in the overall system evaluation metrics. For example, the functionality, reliability, usability, and efficiency of a system can be assigned weights 4, 4, 3 and 2, respectively. Here, weight value measures from 1 (poor) to 4 (high) (Jeanrenaud & Romanazzi, 1994). In a similar way, for any desired set of metrics we may use ordinal values for generating indexes. We need to identify whether a feature is being over-utilized or under-utilized than expected by the system architect or analyst. Increased value of a Feature Rank may be due to its over-utilization. Thus, it will impact on USI and on system's overall value. Similarly, decreased value of a Feature Rank could be from the under-utilization and will impact on the USI and system value, as well.

Class	List of Metrics	Description
1. Metrics from Graph Model	Weight of Edge	Probability that the sustem user or control flow will visit this path while using the system, E_N = p (path traversal)
	Weight of Node	$W_N = \sum_{i=0..m} \{ (FR_{Ni} * FU_{Ni}) / (\sum FU_{0..N})\}$, where *m* is number of features in this node's (N) equivalent process in the system
	Work-flow Nodes & Interconnections (Mubin & Luo, 2010a)	• Decision Nodes $\{D_i\}$ have one or more output branches that end up in a sub-process $\{P_i\}$ • Probing Stations $\{PS_i\}$ mark each branching areas and collect contextual data • Feature Rank $\{FR_i\}$ is an index and can be calculated in a way that is suitable for the architects to assess the utility of a certain feature or component of the system • Interconnection rules: $D_i \rightarrow \{PS_{1..n}\} \rightarrow P_j$ ……… … (1) $P_i \rightarrow \{FR_i\} \rightarrow D_i$ ……… … (2) • Branching Factor (%), based on rule (1) and (2) mentioned above • Locality of Change-requests: Map of change request areas in the system work-flow
2. System Usage Metrics	Feature Utilization	FU is the usage frequency of system features in the workflow tree
	Feature Rank	$FR_{i+1} = FR_i + \{(FU_{i+1} - FU_i) / FU_i\}$, where, FR_0 is assigned as mentioned in table 1, and $FU_0 = 0$; *i* indicates previous run cycle.
	System's overall value, SV	$SV = \sum_{i=1..n} \{(USI_i * f_i) / \sum_{j=1..n} f_j \}$, where *n* is the number of users in the system, and f_j is the number of cycles completed by user j.
3. Development Activity Metrics	Task Requests, TR	Average Inter-arrival time Frequency of Task Arrivals Task Request Priority Tasks in Multi-project environment (out-of-scope)
	Task Completion (or Service) Time, TC	Frequency of Task Completions Wait-time/ Idle-Time/ Latent-Time Service-Time & Deployment-Time Allocated time for a task in multi-project environment (out-of-scope)
	Task Complexity	Complexity = *f*(service-time, priority, USI-change, man-hour, nature of task)
	System State	A system's state can be viewed from different perspectives; and can give absolute, abstract or recurring metrics.

Table 3. System metrics necessary for managing evolvable systems

5.1.3 Metrics of development activities

In addition to system usage metrics, we also need to focus on the metrics that will help track all sorts of system development activities such as; new change requests, requests to add new features in a system component, and task completion rates based on the environment setup described in Section 3. Depending on the system's functionalities, we need to deduce a specific set of metrics to drive the meta-structure (wrapper-system) in the surrounding environment of the system so that its knowledgebase will be able to generate necessary supporting configurations to guide the subsequent evolutionary phases and provide appropriate suggestions for applying new changes and analyzing the changing patterns throughout the system lifecycle. Table 3 lists a summary of these sets of metrics.

In order to find comparative measurements of metrics so that we can visualize the impact of system dynamics we apply normalization and then plot the normalized measurements against each other.

5.2 Indicator metrics

Mapped nodes in the graph model are either processing nodes or decision nodes (figure 3). Decision paths reflect how the users are making choices and thus directly impacts on "Edge Probabilities", based on the variations in Edge Probabilities the analysts may decide to rearrange the workflow in favor of currently used paths thus changing the branching factors as well as the underlying graph model.

Probing stations can collect node visit frequencies which will identify the popularity of workflow locations and, more specifically, certain features of the system. Feature Utilization (FU) frequency directly impacts the feature's rank. Variations in the rank are major indicators in identifying any redundant components (for elimination), under-utilized features (may need advertisements) and over-utilized features (that need to be efficient). With the changing rank in features each node's weight will vary. For certain nodes with no significant features the weight will eventually be reducded to zero and will have no contributions in the system value.

User satisfaction index (USI) reflects the outcome experience of a user or a group of users. A declining USI is a driving force to initiate major revisions into the system to meet newly emerged requirements. Overall system value is something that the stakeholders, architects, analysts and developers keep their eyes on. Based on the type of services provided they will derive a composite formula or index that best describes a system's overall performance at a particular instant of time. For example, a highly significant feature may yield a lesser system value during low usage activity for that cycle time.

5.3 Dependencies of metrics

It is beneficial to categorize system metrics into dependent and independent metrics. There should be clear precedence relationships among the dependent metrics in terms of their logical ordering of causes, events or development activities. If necessary, such dependencies can be artificially manipulated by associating one or more attributes such as priority level, task complexity, urgency, service queue re-ordering, etc. For example, priority of a certain activity can be put to a halt if a new higher priority activity arrives in the service queue. In general, we can establish a set of primary measures from which other attributes could be derived. That is, having recorded the primary measures associated with a given process, one could reconstruct evolutionary behavior depicted by them and by secondary measures where the later could be obtained by some combinations of the primary measures (Ramil & Lehman, 1999).

6. Evolutionary phases

An evolutionary phase is a transition from the current state of the system configuration to the next stage with newly system features applied into the system. There could be major changes or simple minor adjustments. System architects and stakeholders may pre-set the threshold on defining the degree of "evolutionary phases" specific to a system's functionalities or services provided to the end users. As a simple example of a one-dimensional measure the receipt of X or greater number of "feature-adding" task-requests within one cycle will trigger an inception phase and prepare for an upcoming evolution. The number X can vary based on the type and service of the underlying system. Such measures can be made composite to incorporate other necessary measurement metrics.

6.1 System evaluation

A solid framework should be established that collects all necessary metrics for the purpose of system evaluation as a prerequisite to the study of system evolution. According to the ISO 14598 (1999) "Information Technology – Software Product Evaluation," there are four phases that the evaluation process should follow: (1) requirements, (2) specification, (3) design and (4) execution. We can classify and rearrange system metrics to cover these areas in the right order and then the underlying meta-structure will collect and generate system evaluations for any instant of time, as described in figure 1. System evaluation should have the flexibility to include new measures as it passes through evolutionary phases.

6.2 Evolutionary factors

It is critical to observe the system within the operational environment over a period of time to identify and derive possible causes or key evolutionary factors covering the system dynamics. For this purpose we need to capture emerging requirements, shifts of usage patterns, changing service retention policies, changes in satisfaction levels (or composite system values), changing influences from the surrounding environment, and so on. Collectively, the comparative analysis of the metrics within each cycle with the knowledge base will suggest necessary system evolution. It is the responsibility of the system architect or analysts (along with stakeholders) to devise their own policies and threshold measures for determining the trigger for the next evolutionary phase. Over a period of time the knowledgebase will have sufficient historical data for generating a system's dynamic specifications (in terms of configuration values, development stages, changing patterns, and timing) to address the requests for newly desired features in the system. The process of system evolution itself may be generalized for the repeating evolutionary phases observed over a period of time. This will help to *predict* further evolutions.

7. Implications of updating or applying new changes

Empirical evidence has suggested a relationship between the application of design measurements and future software maintainability (Grady, 1994). Establishing a meta-structure with a repository of historical measures of various metrics provides in depth analytics for software maintainability. Such a meta-structure serves as a link between the system users' needs and the developer team's activities for both system development and process improvements. Understanding the reasons for variability in the data provides a powerful decision tool. Naturally, real data will have variations. Efforts need to be given to understand the causes of variations in the collected data sets. Since any positive process improvement changes deliver better positive results the objective here is to improve the process and thereby maintain system value at higher level.

Newly added desirable features requested by system users or added by the developer team should nominally add value to the system. Therefore, such activities raise the level of overall satisfaction in user expectations. Evaluation of the track history of development activities within a system usage cycle can be captured through the changing rates of accumulating task completions compared with the cumulative system values. These cumulative system values may a composite estimation that is based on the accumulation of changes in identified USIs, superimposed with system access rates, for example

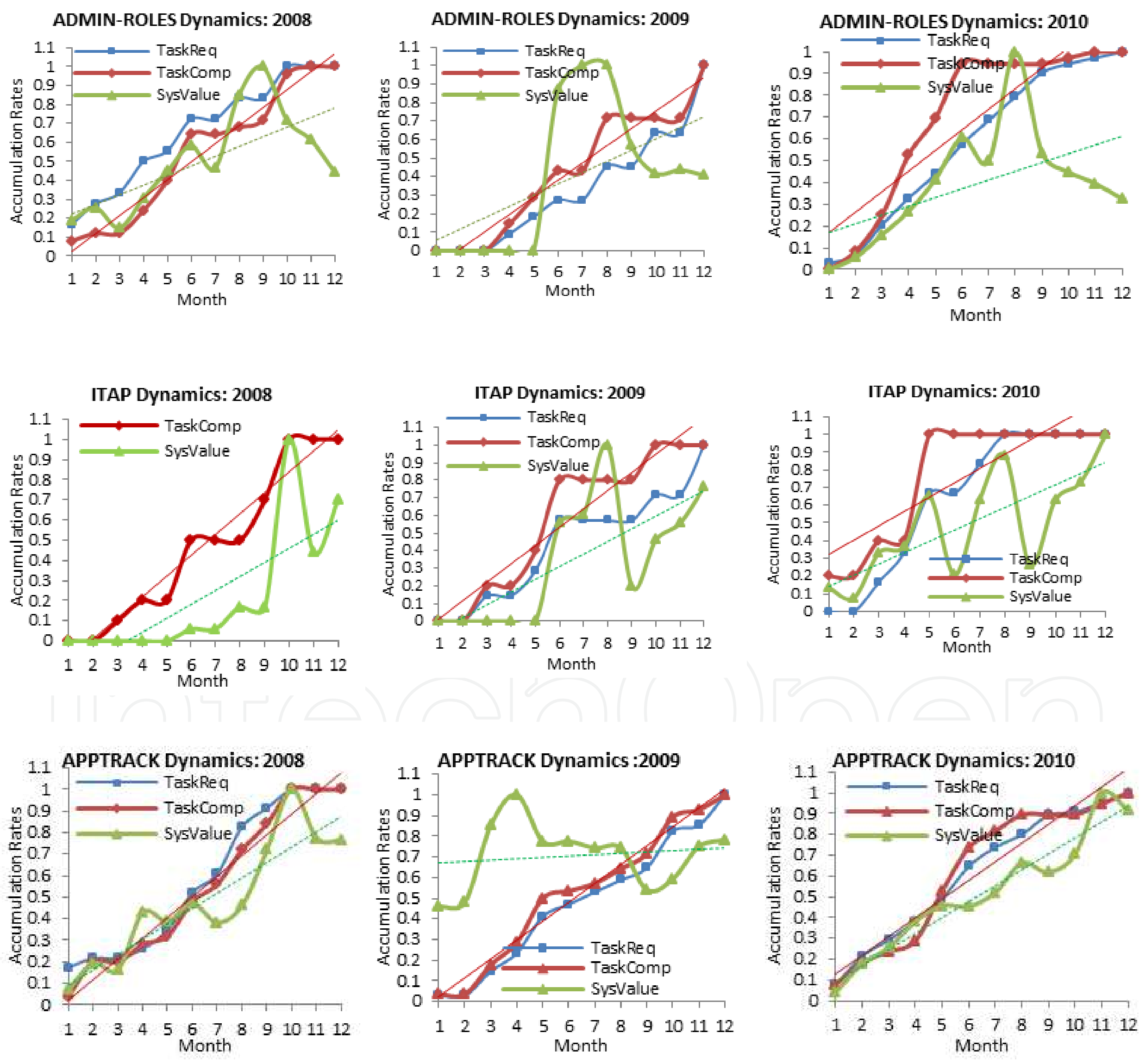

Fig. 4. Identifying possible state transitions in various systems from intersecting points within a yearly cycle

For example, figure 4 shows the *normalized* data plots (for side-by-side comparison) of the last several years for three development projects for a university administrative office. These are illustrated by different services: **ADMIN-ROLES** (to manage graduate administrative roles across the campus), **ITAP** (registration and scoring system for international teaching assistant program) and **APPTRACK** (a sub-system to manage graduate applications data online). ADMIN-ROLES and APPTRACK are moderate sized projects that run throughout the year. ITAP is a comparatively smaller sized project with seasonal usage patterns.

Referencing the state diagram mentioned in figure 2, we can compare the variations in the system value (as the state transitions T_i in the matrix) throughout the yearly cycle. We use the changing values of accumulating Task Requests or Task Completion rates as the underlying cause, because Task completion rates have a more direct impact on the users and overall system values. We have also generated the trend lines to observe the significance of the intersecting points (see figure 4), as well as their regression analyses in table 4.

ADMIN-ROLES					
Year	R2	sERR	sigF	p-V	Correl.
2000	0.507	0.194	0.009	0.009	0.713
2009	0.369	0.333	0.036	0.035	0.608
2010	0.593	0.179	0.003	0.003	0.769
ITAP					
Year	R2	sERR	sigF	p-V	Correl.
2000	0.707	0.185	0.001	0.001	0. 841
2009	0.656	0.218	0.001	0.001	0.810
2010	0.459	0.233	0.015	0.015	0.677
APPTRACK					
Year	R2	sERR	sigF	p-V	Correl.
2000	0.877	0.103	0.000	0.000	0.936
2009	0.027	0.165	0.606	0.606	0.165
2010	0.837	0.121	0.000	0.000	0.915

Table 4. Regression Analyses: Task Completion (TC) vs. System Value (SV)

7.1 System assessment

By selecting significant metrics, developers may derive their own system value that best reflects the current condition of the system state based on the type of services and its influence in the surrounding environment. We calculated the system value (SV) as a composite metric of system access rates with the accumulating perceived USI changes that emerged due to task completions over a yearly cycle. Higher demands for system usage should put more values in SV. Ideally, we'd expect a transition of T_{++} due to expected positive impact of any incremental development activities. However, the scenario is often not the same for real data sets.

In comparing the accumulating rates of task completions (TC) with the rate of variations in system value (SV) in normalized plots (as shown in fig. 4), first we are interested in the intersecting points indicating that TC and SV have the same rates. Second, for the segments between the intersections, our focus is on the transition of SV. As described in the following sections we see wide variations in the patterns for different projects in respective plots.

7.1.1 Assessment of ADMIN-ROLES

Although ADMIN-ROLES is a yearly usage system (with roll-over), higher activities are concentrated during the months of August and September. We noticed that there were four intersecting points in 2008 where the SV transitions occurred between the segments were T_{+-}, T_{++}, T_{-+} and T_{+-}, respectively. The causes $C_{1..n}$ include the access patterns and the perceived changes in USI values over a year. At the intersecting points, the rates are the same and ideally we would expect the same rate of change in both TC and SV. However, due to various environmental factors both of these metrics vary and their patterns change during the segmented time frame. Moreover, the projected trend lines intersecting in 2008 and 2009 yearly cycles indicate the instability in system state. But, in 2010 there were no intersections with increasing disparity between TC and SV. Regression analyses in table 4 shows close ties between TC and SV in 2008 and 2010 and incidcates environmental disturbances in 2009.

7.1.2 Assessment of ITAP

ITAP has seasonal usage patterns, and showed a scenario rather parallel activities. Brief intersections occured where the transition pattern was T_{+-} in that segment. During the last three years, the trend lines did not intersect at all and TC > SV was maintained across these years. The regression analyses in table 4 also support the consistent and stonger connections between TC and SV.

7.1.3 Assessment of APPTRACK

APPTRACK also has a rolling usage patterns. Again, regression analyses shows close ties between TC and SV in 2008 and 2010 and incidcates either environmental disturbances or large changes in usage patterns in 2009.

In summary, thorough analyses of a carefully designed set of relevant metrics, as well as the underlying development track history, open up many possible experiments for closely studying system dynamics over a period of time. Upon maturity, these metrics of measurements would form the foundation for developing futher statistical models to drive a set of tools for managing evolvable systems.

8. System usage patterns

One of the basic patterns that we would like to know is how frequently a system is being used or accessed over each cycle (with sufficiently long periods between cycles). Then if we can compare this for several consecutive cycles, we can capture more details on how the behavior is changing over a period of time along with possible causes. For example, figure 5 demonstrates such activities over the last four years. If we closely observe the changing patterns we can summarize that ADMIN-ROLES has had *horizontal shifts,* meaning that users have changed their habit of system usage time over the yearly cycle even though the overall access volume remained same. For APPTRACK we see that there had been *vertical shifts* over the last several years indicating that the system has become more/less popular. For GTA-WS and ITAP systems, there are clear signs of seasonal activities that repeat over each yearly cycle.

These access rates can be superimposed with the accumulated USI values that are related to applying new changes into the system. This would represent a more meaningful output of overall system values as demonstrated in figure 4. We present another example of system usage patterns that compares the actual usage versus the expected usage patterns. As proposed in figure 3, we can assign the probabilities of using certain features in the workflow paths. Figure 6 demonstrates such mapping of partial workflow paths of the system ADMIN-ROLES. Initially we would expect that under normal circumstances, a user may take one of the five possible options (so 0.2 is the probability to access one option). Then at the next level (from *Node4* to *Node4.1*), there is just one possibility with probability 1.0; and from Node4.1 there are three options with probability 0.33 probability each.

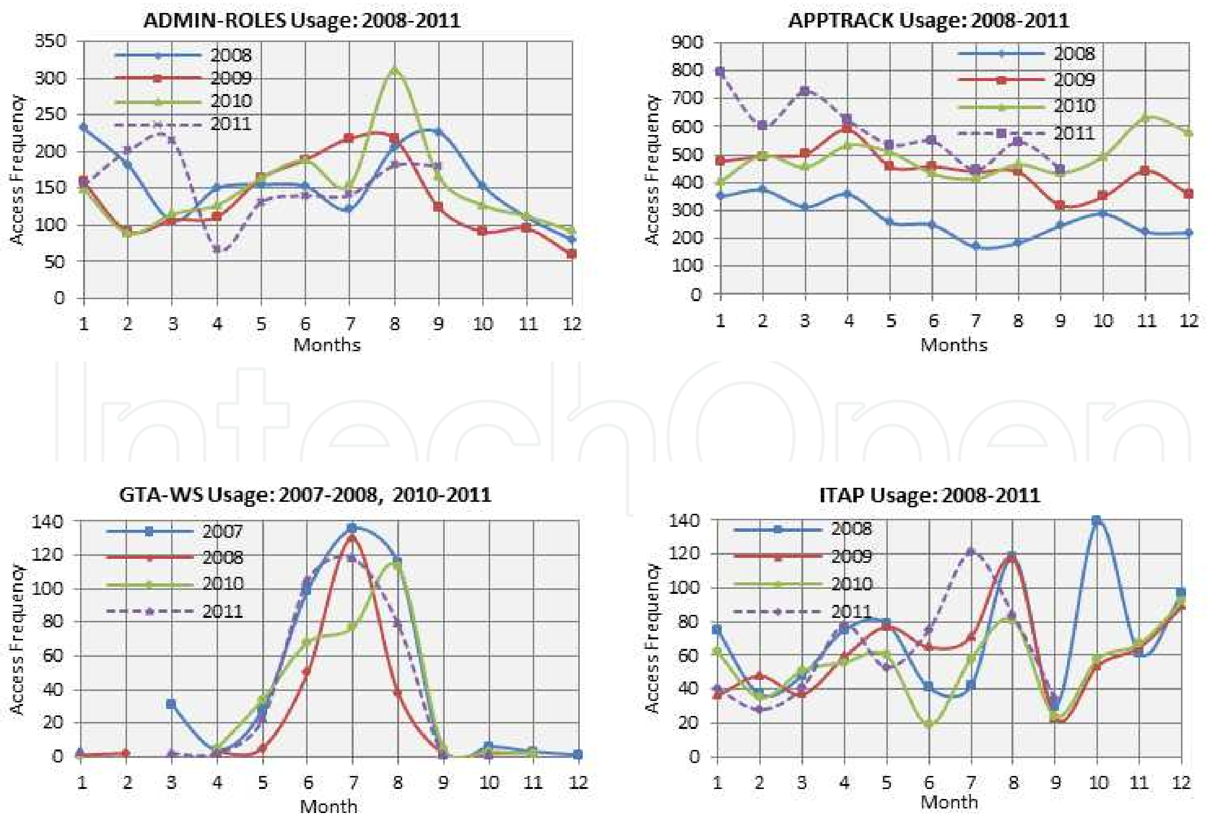

Fig. 5. Various patterns of system access frequencies

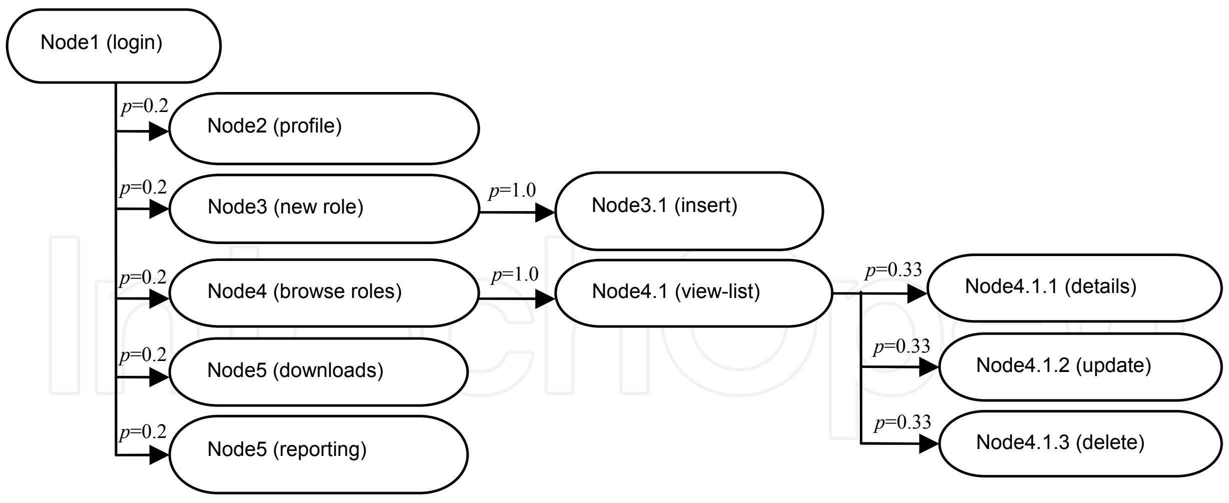

Fig. 6. Example of a mapped graph of partial workflow nodes in ADMIN-ROLES

Therefore, the chance of utilizing feature of *Node4.1.2* will be, p=0.2 x 1.0 x 0.33 = 0.066. Figure 7 compares the scenario for three activities; ACT_INS, ACT_UPD and ACT_DEL.

As we can see from the regression analyses (see table 5), there are close correlations between the actual usage patterns and the expected usage patterns. This experiment reveals that our expected usage pattern of ACT_UPD is almost 8.7 times higher, although the relative usage rates coincide with each other. (Notice the R^2, *p*-Value and Correlation for this activity in table 5). For ACT_INS and ACT_DEL the outputs look correct in terms of estimations. Therefore, the result suggests that in order to achieve the expected system behavior we need to adjust the pre-assigned probabilities to these three leaf nodes in the graph.

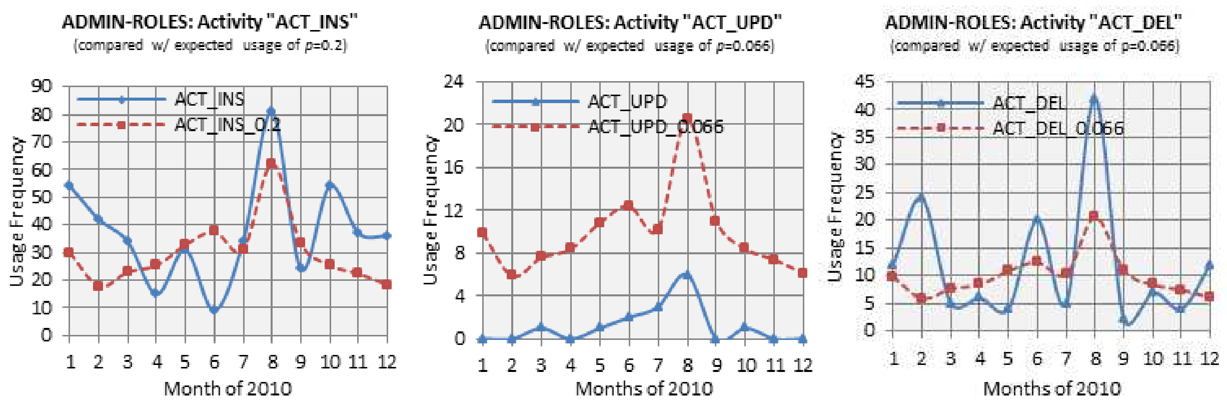

Fig. 7. Usage activities of specific features comparing to estimated probabilities

ADMIN-ROLES	R^2	sERR	sigF	p-Value	Correlation
ACT_INS	0.2066	0.1745	0.1375	0.0243	0.4546
ACT_UPD	0.7529	0.3415	0.0002	0.0000	0.8677
ACT_DEL	0.4149	0.0812	0.0237	0.0002	0.6441

Table 5. Regression Analyses three ADMIN-ROLES Features

In summary, we may follow these experiments to calibrate the processes, thereby adapting the values of multiplicative constants, probabilities, etc. to the empirical data for the environment of our particular system. However, over a longer period of time the environment itself may change significantly. Therefore, these calibrated values need to be reviewed and re-adjusted recurringly as the system evolves over time.

9. Formulating system dynamics as a process

Implementing an effective measurement program as a meta-structure of its parent system is a big challenge. And overcoming these challenges is worthwhile, because such measures provide insights into the system behavior. Such implementation also improves the ability to plan and track systems evolutionary progress and needed for addressing risks/problems much earlier (Clark, 2002). Therefore, we need to have an underlying methodology that can be applied to formulate a process for evolvable systems. Typically this would include the study and observation of selected set of metrics, development track history, changes in usage track history, and so on. In the previous sections of this chapter, we attempted to explain the necessary elements of such a process in detail. Although the resulting evolution of software mostly appears to be driven and controlled by human decisions, managerial edict and developer's judgement, yet a high-level generalized pseudo code for such a process can be developed, as presented in fig. 8. This is in conformance with the system dynamics approach suggested by Richardson (Richardson, 1999).

10. Benefits & limitations

System dynamics methods recognize that the real-world is a non-linear web of feedback processes with significant delays and is usually not in equilibrium. We can seek the causes of problems by the interactions of real-world feedback loops. System dynamics provides a way-of-thinking (a paradigm) and associated communication and software driven tools to help design policies that provide durable solutions to pressing problems. Capturing a system's dynamic behavior has a number of significant benefits that include a clear insight into the system, increased control over maintaining and upgrading the system, the capability for better and more timely services, lower maintenance cost, and the ability to progressively maintain satisfactory system outcomes. Therefore, the user retention rates are often high. In addition, the supporting meta-structure also tends to provide significantly more controlled administration and higher manageability towards building reconfigurable systems. Any additional data that can be generated by the meta-system can be utilized further toward supporting or providing credence for additional experts' opinion. It can help us better understand how a problem grew up over time and assist us in finding a long-lasting solutions to the problem.

The vital observations of system update requests and usage activities, as described in this chapter, depend on a number of factors. These include: the operational environment, groups of users performing similar functions within a somewhat common time frame, any relevant external influences on the work environment, and the functionality of the system itself.

```
Status SystemDynamics(SystemSpecifications-Initial)
BEGIN
          WrapperSystem ← SetupMetaStructure(SystemSpecifications-Initial)
          ThisSystem ← SystemConfiguration(WrapperSystem) // dynamically configure the system
          System_Initialization(ThisSystem)
          BEGIN
                    // in tabular format, collect the complete system workflow paths
                    SystemControlFlowPath ← GetControlFlow(ThisSystem)
                    // prepare equivalent graph model from the system with initial weights
                    SystemGraph ← GenerateGraphModel(SystemControlFlowPath, Node_Weights,
                    Edge_Values)
                    // environment setup: System Usage
                    Setup_Probing_Stations(SystemControlFlowPath, SystemGraph)
                    // environment setup: collect update/change requests
                    Setup_TaskActivity_Stations(SystemControlFlowPath, TaskRequest_Template,
                    TaskCompletion_Template)
          END
          // interconnect the flow with feedback loops (circular causality)
          WHILE (System is running)
          BEGIN
             FOR EACH [TIME-CYCLE] IN System-Lifecycle
             BEGIN
                    // get dynamic system behavior and state (feedback loops)
                    sysState ← GetSystemState(TimestampNow)
                    sysUsage ← GetSystemUsage(TimestampNow)
                    sysTasks ← GetSystemTasks(TimestampNow)
                    // formulate model: identify stock (or accumulations) & info flows (rates)
                    Knowledgebase ← Record_System_Dynamics(sysState, sysUsage, sysTasks)
                    // get system status for evaluation
                    SystemStatus ← SystemEvaluation(Knowledgebase, TimeStampNow)
                    SWITCH (SystemStatus)
                    BEGIN
                              CASE "EVOLUTION":
                                        PrevSpecs ← SystemSpecifications-Initial
                                        // derive policy insights
                                        NewSpecs ← GenerateNewSpec(Knowledgebase)
                                        Apply_New_Changes(PrevSpecs, NewSpecs)
                                        // save changes resulting from model-based updates
                                        Knowledgebase ← Record_New_Changes(PrevSpecs, NewSpecs)
                                        // recursive call
                                        RETURN SystemDynamics(NewSpecs)
                              CASE "INCEPTION":
                                        Prep_Inception_Phase(Knowledgebase, ThisSystem)
                              CASE "SYSVALUE_INC": // save data on incremental SV
                                        thisINC ← Generate_Change_Report(Knowledgebase,
                                        prevCHANGE,   TimeStampNow)
                                        thisCHANGE ← thisINC
                              CASE "SYSVALUE_DEC": // save data on declining SV
                                        thisDEC ← Generate_Change_Report(Knowledgebase,
                                        prevCHANGE, TimeStampNow)
                                        thisCHANGE ← thisDEC
                              CASE "NOCHANGE": continue loop
                    END SWITCH
                    Prep_Next_Iteration(Knowledgebase)
             END FOR
          Prep_Next_Cycle(ThisSystem, SystemSpecifications-Revised)
          END WHILE
          RETURN SystemStatus
END
```

Fig. 8. A high-level pseudo-code for a generalized process of system evolution

Although the majority of systems fall under these scenarios, there are a number of other classes of systems where a meta-structure either cannot be established or the establishment of such a structure would be cost-prohibitive. For these classes of systems, such evaluations may not support the discovery of system behavior or impact based on updated request history.

In addition, many systems will not allow suggested changes to be applied until the system is shut down for some prolonged period of time. Thus, frequent update requests will not be possible. In addition, due to time, money or manpower restrictions, many developers may not venture through setting up such a closely coupled meta-structure within the surrounding system environment.

11. Conclusion

We can think of the meta-structure attached to the system itself as a measurement program. There is a growing awareness that such systems pay off but not without some investment of both time and resources. As the benefits of applying meta-models become more evident, the establishment of such structure will become more essential for evolvable systems to retain their value to a satisfactory level and to remain competitive in the market. With the growing experiences and maturing systems, multi-dimensional data will become available for further research. This, in turn, will make it possible to develop better models. Although it is typically too costly to run carefully controlled experiments on capturing meta-data covering the entire development cycles, it is definitely worth the investment for any long-running systems. Certain meta-level system development practices must be established so that we can fully understand the system, manage it in a more cost-effective way, and readily adapt to a changing environment. Although our proposed model does not cover every aspect, it opens up a wide range of options for the system architects or stakeholders to deduce their own sets of metrics and to utilize these measures to fine-tune systems to extend their lifecycles. Eventually, the systems being instrumented to support software evolution will become more complex with higher capabilities to configure themselves with more feature-rich services. Thus, a growing proportion of resources will continue to be needed for maintenance of these systems.

12. Future research

Building evolvable systems is a considerable challenge and undertaking the issues of developing evolvable systems and then formulating methods to maintain their extended life-cycles are quite large and time consuming initiatives. Given this, some future research on the selected areas of system evolution can be outlined as follows:

- Automated ways to to update system objects, processes and codes needs to be investigated further
- In-depth studies of psychological impact on USI during each system upgrade will help refine the calculation of system value and focus on the influential evolutionary factors only. (Halstead's effort metrics and Cyclomatic complexity of workflow graph can be used to predict psychological complexity)
- Applying further statistical analyses (such as vector and velocity measures for comparing TC and SV curves, applying differentiation on the system value, etc.) on the accumulated historical data in the knowledgebase will direct accurate prediction of evolvable cycles by imposing probabilities at the state-transition points.
- Applying proper design patterns to probing stations and survey agents across the workflow will make the overall process more mature and reconfigurable.
- Because each update request needs to be addressed and implemented into the system, a corresponding queuing server model for more complex multi-project development environment can be built based on queuing theories.

- Each system will need initialization time to configure itself and fine-tune with correct parametric values – thus a novel approach needs to be sought to generalize the initiation process with necessary delay at each the inception phase of evolution.

13. Acknowledgement

We express our sincere gratitude to the Dean's Office of The Graduate School at The University of Alabama for supporting the underlying infrastructure of servers, development tools and the systems development track history records.

14. References

Bennet, K. H. (1990), *An Introduction to Software Maintenance*, Information and Software Technology, 32(4)

Clark, B. (2002). *Eight Secrets of Software Measurement*, IEEE Software, September/October 2002

Coleman, D.; Ash, D.; Lowther, B.; Oman, P. (1994), *Using Metrics to Evaluate Software System Maintenability*, IEEE Computer, August 1994.

Dekkers, C.; McQuaid, P. (2002). *The Dangers of Using Software Metrics*, IT Pro, March/April 2002, IEEE

Dori, D. (2003), *Conceptual Modeling and System Architecting*, Communications of the ACM, October 2003, 46 (10).

Ferneley, E. H. (1999), *Design Metrics as an Aid to Software Maintenance: An Empirical Study*, John Wiley & Sons, Ltd.

Grady, R. B. (1994), *Hewlett-Packard – successfully applying software metrics*, IEEE Computer, 27(9)

Henry, S.; Kafura, D. (1981). *Software Structure Metrics Based on Information Flow*, IEEE Transaction on Software Engineering, Vol. SE-7, No. 5, Sep. 1981

Ives, B.; Olson, M.; Bardoui J. (1983), *The measurement of User Information Satisfaction*, Communications of the ACM, 26(10), October 1983

Jeanrenaud, A.; Romanazzi, P. (1994). *Software Product Evaluation Metrics: A methodological approach*, Transaction on Information and Communications technologies

Lehman, M. (1980), *Programs, Life Cycles, and Laws of Software Evolution*, Proceedings of the IEEE, 68 (9), Sep. 1980

Mubin, A.; Luo, Z. (2010a), *Low Overhead and High Yield Integrated Surveys*, Proceedings of ACM SIGUCCS Fall 2010

Mubin, A.; Luo, Z. (2010b), *Building a Reconfigurable System with Integrated Meta-Model*, Proceedings of International Multi-Conference on Engineering and Technological Innovation 2010

Mubin, A.; Ray, D.; Rahman, R. (2009), *Architecting an Evolvable System by Iterative Object Process Modeling*, Proceedings of IEEE and World Congress on Computer Science and Information Engineering (CSIE), 2009.

Port, O. (1988), *The Software Trap – automate or else*, Business Week, Special Report, 9(1)

Ramil, J.; Lehman, M. (1999), *Challenges facing Data Collection for Support and Study of Software Evolution Process*, ICSE 99 Workshop on Empirical Studies of Software Development and Evolution

Richardson, G. (1999), *System Dynamics* (in Encyclopedia of Operations Research and Information Science), Kluwer Acadmics Publishers

Sherif, Y. S.; Ng, E.; Steinbacher, J. (1988), *Computer Software Development: Quality Attributes, Measurements, and Metrics*, John Wiley & Sons, Inc.

System Dynamics Society: http://www.systemdynamics.org

REAL-TIME SYSTEMS, ARCHITECTURE, SCHEDULING, AND APPLICATION
Edited by Seyed Morteza Babamir
INTECH

Real-Time Systems, Architecture, Scheduling, and Application
Edited by Dr. Seyed Morteza Babamir

ISBN 978-953-51-0510-7
Hard cover, 334 pages
Publisher InTech
Published online 11, April, 2012
Published in print edition April, 2012

This book is a rich text for introducing diverse aspects of real-time systems including architecture, specification and verification, scheduling and real world applications. It is useful for advanced graduate students and researchers in a wide range of disciplines impacted by embedded computing and software. Since the book covers the most recent advances in real-time systems and communications networks, it serves as a vehicle for technology transition within the real-time systems community of systems architects, designers, technologists, and system analysts. Real-time applications are used in daily operations, such as engine and break mechanisms in cars, traffic light and air-traffic control and heart beat and blood pressure monitoring. This book includes 15 chapters arranged in 4 sections, Architecture (chapters 1-4), Specification and Verification (chapters 5-6), Scheduling (chapters 7-9) and Real word applications (chapters 10-15).