A Hierarchical Formal Framework for Adaptive N-variant Programs in Multi-core Systems

Li Tan
Washington State University
Richland, WA 99354-1641
litan@wsu.edu

Axel Krings
University of Idaho
Moscow, ID 83844-1010
krings@uidaho.edu

Abstract

We propose a formal framework for designing and developing adaptive N-variant programs. The framework supports multiple levels of fault detection, masking, and recovery though reconfiguration. Our approach is two-fold: we introduce an Adaptive Functional Capability Model *(AFCM) to define levels of functional capabilities for each service provided by the system. The AFCM specifies how, once a fault is detected, a system shall scale back its functional capabilities while still maintaining essential services. Next, we propose a* Multi-layered Assured Architecture Design *(MAAD) to implement reconfiguration requirements specified by AFCMs. The layered design improves system resilience in two dimensions: (1) unlike traditional fault-tolerant architectures that treat functional requirements uniformly, each layer of the assured architecture implements a level of functional capability defined in AFCM. The architecture design uses lower-layer functionalities (which are simpler and more reliable) as reference to monitor high-layer functionalities. The layered design also facilitates an orderly system reconfiguration (graceful degradation) while maintaining essential system services. (2) each layer of the assured architecture uses N-variant techniques to improve fault detection. The degree of redundancy introduced by N-variant implementation determines the mix of faults that can be tolerated at each layer. Our hybrid fault model allows us to consider fault types ranging from benign faults to Byzantine faults. Last but not the least, multi-layers combined with N-variant implementations are especially suitable for multi-core systems.*

1 Introduction

Adaptive software has attracted much research interests in recent years. Two key features of an adaptive software are (1) the ability to monitor its own execution and (2) the ability to reconfigure itself based on the result of runtime monitoring [6]. Self adaptation is essential to improve system survivability for a range of applications from safety-critical embedded software to mission-critical web services that shall be resilient to malicious attacks.

Developing adaptive software also raises some challenging questions. First, in many cases self adaptation adds one more dimension of complexity to often already complicated dependable system designs. A question is how one can specify requirements for adaptiveness and implement them in a way that facilities orderly and verifiable system reconfiguration. Second, a system may be subject to a variety of faults. So a challenge is how one could compartmentalize and diversify system design so the system can be resilient to different types of faults. This may be especially relevant in safety-critical applications. Finally, runtime monitoring requires additional computation power. Thus, it is important that our design can make efficient use of the underlying hardware architecture to minimize overhead.

To address the first challenge, we introduce a formal model to specify requirements for self adaptation and then propose a multi-layered assured architecture to realize requirements expressed in the formal model. The Adaptive Functional Capability Model (AFCM) defines levels of capabilities for each system functionality. AFCM specifies how a system shall reconfigure itself and scale down its functional capabilities while still maintaining essential services and guaranteeing information assurance. Each level of functional capability in AFCM will then be implemented as a layer in the proposed *Multi-layered Assured Architecture Design* MAAD. Note that we use the term *level* in the context of the AFCM and the term *layer* in the context of the MAAD. The architecture design embeds a *Monitoring and Reconfiguration Module* (MRM) that uses lower-layer functionalities as reference to monitor high-layer functionalities and detect faults. The layered design also implements requirements for reconfiguration defined in AFCM and provides information assurance: in case a fault is detected, the system reconfigures itself by disabling affected layers, while lower layers still maintain essential services.

To further improve system resilience, we use a diversified layered design based on N-variant techniques in each layer [3, 4]. The N-variant techniques use redundant executions to reduce system vulnerability to common mode faults. Redundant executions to benefit reliability have been extensively used in fault-tolerant systems design, where the evolution of redundancy schemes has gone from homogeneous redundancy to heterogeneous redundancy. The latter refers to components that are functionally equivalent but implemented dissimilarly. The expectation is that redundant but dissimilar implementations reduce or eliminate common mode faults. Dissimilarity is typically discussed in the context of N-version programming [1] dating back to the late 70s. In N-version programming it is assumed that several software development groups derive programs based on the same specification in isolation. The expectation is that this helps to reduce common mode faults. An approach inspired by N-version software is N-variant or multi-variant software, where different variants are generated in a more automated fashion. Again, the expectation is that a fault affecting one variant will not affect another. In both cases a fault is detected if a difference is detected between outputs generated by two versions or variants.

Redundant executions exercised by multiple variants and extra work of runtime monitoring requires additional computational power. To reduce overhead, our N-variant-based implementation takes advantage of recent advances in multi-core hardware. Most new general-purpose computers incorporate dual or quad-core processors and higher numbers of cores are already implemented in graphics processing units. Whereas in theory the computational capabilities increase with the number of cores, it becomes difficult to exploit sufficient parallelism to keep all cores utilized. Most common applications still allow little parallelism and it is likely that cores may be underutilized or running idle. In our approach, unused or underutilized cores are exploited to increase reliability, security, and survivability. Specifically, multiple variants execute on different cores, and if they can execute on idle cores, this overhead can be largely absorbed. This was also shown in [8] where multi-variant executed in multi-core systems. Our approach extends this by making extensive use of N-variant implementation at each layer of functional capability. In general, the lower a layer, the more variants it may have in order to provide a higher degree of resilience and information assurance for essential service. Nevertheless, exact number of variants and their configurations required at each layer depend on the type and number of faults that are to be detected or masked.

Background and Motivation N-variant executions have been used in order to detect and mask transient faults [4] and security related faults [3, 7, 8]. The different executions are considered to be functionally equivalent. For example, in [4] the replicas are managed dynamically by a hypervisor (a virtual machine monitor) inserted between the hardware and the operating system. The redundant functionalities execute on replica partitions, where the number of partitions is dictated by the fault model considered. In [8] multi-variant executions have very high probability of exposing buffer overflows, e.g., as would be experienced during a buffer overflow attack. Here the dissimilarity is mainly affecting the way memory is allocated. Again the functionalities do not differ with respect to their functional specifications. The same holds for the work in [3]. In fact the application of the principle of N-variant execution is based on functional equivalence of the executions.

The research presented here departs from this equivalency assumption. Whereas we still see the system as being composed of functionalities, we assume that these functionalities may have different levels of functional capabilities implemented at respective layers. Intuitively, by applying the principle of "Occam's razor" we make the assumption that lower levels of functionality (and thus capability) will ultimately result in lower probability of failure, as will be described in the context of Figure 1.

Fault Model The system is subjected to diverse fault types arising from diverse fault sources. Faults have been described in the context of hybrid fault models [2, 9]. The hybrid fault model in [9] considers three fault types, *benign* faults, which are globally diagnosable; *symmetric* faults, which imply that values are wrong, but equally perceived by all components that receive the values; and *asymmetric* faults, which have no assumption on the fault behavior. The latter is often called Byzantine fault. Within the context of this research we are mostly concerned with the error produced by the fault, rather than fault sources or types. For example, a buffer overflow may result in observable differences in memory management. This in turn can lead to detection and/or correction.

2 Specification Model & Architecture Design

In this section we extend and generalize the model described in [5], which is a special case of the research below.

2.1 Adaptive Functional Capability Model

We propose a formal model to specify multiple functionalities with adjustable levels of capability. The model, i.e., the Adaptive Functional Capability Model, attaches each functionality to layers of capability. The AFCM is used as part of requirement specification. During requirement elicitation, a development team works with stake holders of a project to identify not just functionalities, but also capability levels for each functionality. These capability levels specify graceful degradation in case of faults or when under attack.

Assume the system is comprised of functionalities $F_1 \cdots F_m$. Figure 1 shows the AFCM for two sample functionalities F_1 and F_2. The requirements for F_1 define three levels of capabilities: F_1^1 defines the set of core operations that are mission-critical, F_1^2 includes F_1^1 and some non-critical but value-added operations, and F_1^3 adds some more

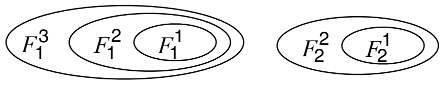

Figure 1. AFCM for functionality F_1 and F_2

value-added operations. We write $F_1^1 \preceq F_1^2 \preceq F_1^3$, where $\preceq$ is a preorder on the capability levels. The semantics of $\preceq$ is defined and interpreted based on application context. For instance, in a transaction-based asynchronous system, a functionality F can be specified as a set of sequences of operations $T(F)$ (in requirement elicitation, often referred to as scenarios or flow of events). We represent a sequence of operations as $(p_0(I_0), O_0), (p_1(I_1), O_1), \cdots$, where p_i is the operation at step i, I_i is the input data set, and O_i is the output dataset. By default, $T(F)$ also includes a null sequence τ. In such a system, we can define that $F^j \prec F^{j+1}$ if and only if $T(F^j) \sqsubseteq T(F^{j+1})$, where the piecewise inclusion relation $\sqsubseteq$ is defined as follows:

(i) $T(F^j) \subseteq T(F^{j+1})$; and,

(ii) For each $(p_0(I_0), O_0), \quad (p_1(I_1), O_1), \quad \cdots \in T(F^{j+1})$, there is a sequence of operations $(p'_0(I'_0), O'_0), (p'_1(I'_1), O'_1), \cdots \in T(F^j)$ and a non-decreasing function g such that $g(0) = 0$, $p_k = p'_{g(k)}$, $I_k \supseteq I'_{g(k)}$, and $O_k \supseteq O'_{g(k)}$.

Note that (i) states that every sequence of operations defined at capability F^j shall also be included at capability F^{j+1}. Furthermore, (ii) states that each sequence of operations in F^{j+1} extends a sequence of operations in F^j. Note that (ii) doesn't prohibit the introduction of a sequence of completely new operations in $T(F^{j+1})$. In such a case, the sequence of new operations can be seen as an extension of the null sequence $\tau \in T(F^j)$.

As an example, consider an adaptive N-variant implementation of a secured database system D. Each record $d = \{d_1, d_2\}$ in D contains two sets of data. Set d_1 contains mission-critical data and d_2 is a set of non-mission-critical but value-added data. For a registered client, D stores its private key K'_D and each registered user's public key K_U. A registered user keeps his/her private key K'_U and D's public key K_D. Communication between D and a registered user is encrypted using the public/private key pairs. For simplicity, we assume that the encryption algorithm in use is deterministic. Let's consider a functionality F of D, which allow a registered user to retrieve a record by its record ID (RID). F is defined with two levels of capabilities. At level F^1, a client can retrieve the mission-critical data associated with a record, and at level F^2, a client may also retrieve valued-added data associate with the record.

Using our framework, the functional capabilities F^1 and F^2 are implemented by two layers L^1 and L^2. Each

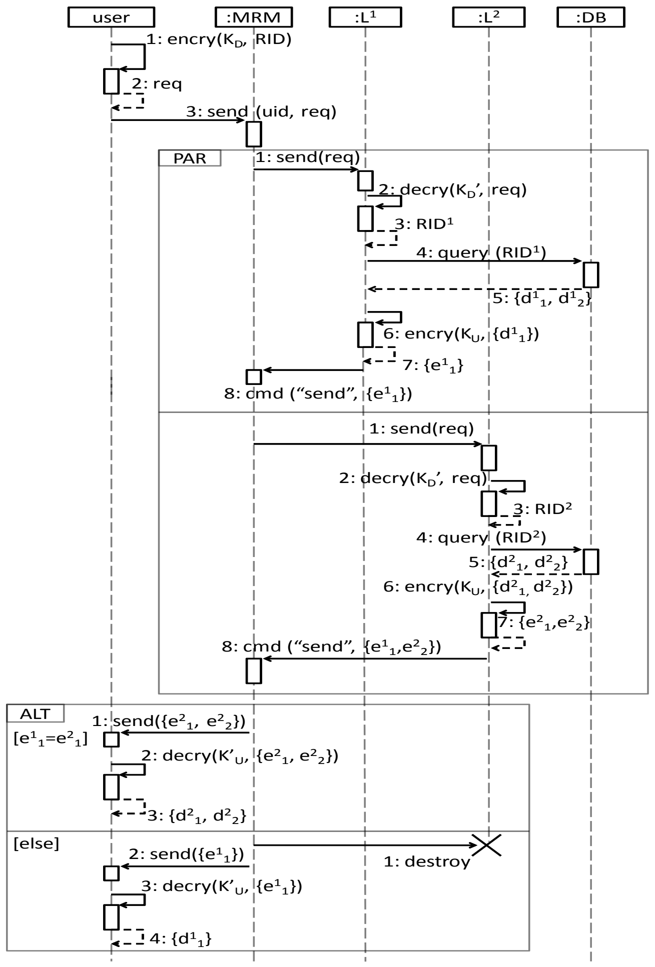

Figure 2. Sequence diagram for an adaptive secured database system D

layer consists of N-variant modules for required reliability and security. The Monitoring and Reconfiguration Module (MRM) decides the operational status of L^1 and L^2. It also serves as the interface between a user and D. The underlying database contains actual records and it can only be accessed by L^1 and L^2. The details of architecture design in our framework will be discussed in Section 2.2. The UML sequence diagram in Figure 2 shows interactions between a registered user and the database system D. The system's behavior at capability level F^i is defined by the set of successful interactions $T(F^i)$ among a user, MRM, L^i, and the underlying database DB. If both L^1 and L^2 operate correctly, then $d_1^1 = d_1^2$ and hence $\{d_1^1\} \subseteq \{d_1^2, d_2^2\}$. Therefore, $T(F^1) \sqsubseteq T(F^2)$ and $F^1 \preceq F^2$, i.e., the system D implements the preorder on capability levels F^1 and F^2.

The purpose of AFCM is to specify not only functional requirement, but also requirements for reconfiguration and adaptiveness. It has two features to serve its purpose:

First, the model associates each functionality with capability levels, which specify reconfiguration requirements for the functionality. It states that, in the event of a fault, e.g., the system has been compromised, a system shall scale back its services in an orderly manner by following the capability levels defined in AFCM, e.g., recovery to a lower level implemented in next layer down.

Second, the definition of capability levels also facilitates reconfigurable design. For example, consider the piece-wise inclusion relation $\sqsubseteq$ we proposed for transaction-based systems. It requires that system behavior at a higher capability level shall be an extension of behavior at a lower capability level. Hence, we can use behavior at a lower capability level as a reference for monitoring behavior at the higher capability level, and an implementation for capability levels provides a path for a system to scale back itself.

It shall be noted that AFCM does not require that all the functionalities in it have the same hierarchy. For example, F_1 and F_2 in Figure 1 have different levels of capabilities. Each functionality in an AFCM has its own hierarchy of capability levels reflecting its requirement for adaptiveness.

2.2 Layered N-variant Architecture

To implement the reconfiguration requirements specified in the AFCM using different levels, we propose a layered adaptive architecture design. Each layer realizes its corresponding AFCM level using N-variant techniques such as shown in [3, 8]. Figure 3 shows an example of the adaptive N-variant architecture for two functionalities F^1 and F^2. The architecture has a layered structure. Each horizontal layer realizes a capability level, i.e., it implements sequences of operations specified for its capability level. A layer may be disabled if it is found not functioning correctly.Fault detection is the result of redundancy management at the specific layer, or the layer beneath it, which has the monitoring abilities of its capabilities at the layer above it. The capability level of the entire functionality is decided by its highest enabled layer. Each layer is a collection of variants implementing its capability level. Variants are systematically diversified so that it is unlikely that that a common mode fault can occur [3, 8], e.g., for a given fault model, an attacker can not compromise all the variants without being detected and/or the fault being masked. One could argue that a lower layer should have more variants to improve resilience of the functionality. Each functionality may have a different layered structure that reflects its adaptability requirement. For example, in Figure 3 F_1 and F_2 are implemented by (L_1^1, L_1^2, L_1^3) and (L_2^1, L_2^2), respectively.

3 Adaptive Survivability

The layered adaptive N-variant architecture improves system resilience by supporting 1) real-time fault detection

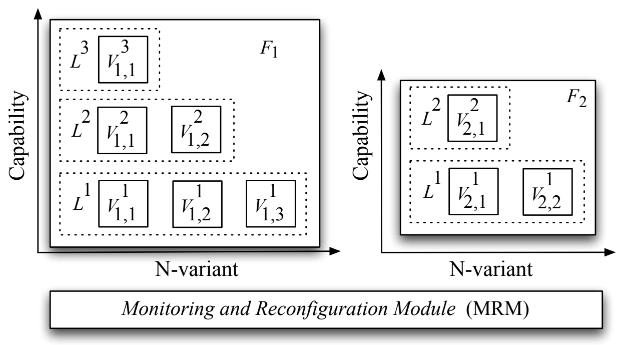

Figure 3. A layered adaptive N-variant architecture design for the AFCM of Figure 1

though redundancy management and cross-layer monitoring, 2) fault masking, and 3) system reconfiguration. The architecture design in Figure 3 includes the Monitoring and Reconfiguration Module (MRM). Critical or sensitive functionalities are implemented using the layered N-variant architecture and the MRM acts as a sentry for layered N-variant components. The MRM monitors and sanctions the communication in and out of the N-variant components. Together, the N-variant-based layers and the MRM provide runtime monitoring and real-time fault tolerance with reconfiguration, essential for an adaptive system.

Runtime monitoring by the MRM The MRM uses observable behavior of a lower layer to decide whether the layer above is compromised or not. If a fault is detected, it reconfigures the system by disabling affected layers while essential functional capabilities are still provided by the lower layer(s). In section 2.1 capability levels in the AFCM are defined in such a way that a sequence of operations specified at a higher level is an extension of some sequence of operations specified at a lower level. In our layered adaptive N-variant architecture, all the layers process incoming requests concurrently. Since a layer L^i is an implementation of an AFCM capability level F^i, a sequence of operations executed by layer L^i shall be *included* in a sequence of operations executed by layer L^{i+1}. Should this not be the case it indicates problems (i.e., a fault) in L^{i+1}. The lower layer L^i is realized using N-variants of simpler implementations and potentially a higher degree of redundancy. It is argued that lower complexity implementations together with more stringent analysis/testing at L^i is assumed to make variants in L^i more reliable than in L^{i+1}. A larger degree of N-variants also increases reliability, as it implements a *k-of-N* configuration. In general we argue that the number of components (degree of N-variant) at layer L^i should be larger than that of L^{i+1} and the fail-rates of the components at L^i are smaller than those at layer L^{i+1}, due to its simpler implementation. The result is a higher reliability at layer L^i.

Real-time reconfiguration The AFCM provides a reconfiguration plan in which a functionality can scale back its services in an orderly manner, thus providing graceful degradation. A layer L^i serves as the backup for layer L^{i+1} above it. A lower layer forgoes some functional capability in lieu of improved dependability. If the MRM detects a fault in layer L^{i+1}, it disables L^{i+1} and the system automatically scales its capability to the level implemented by L^i. For completeness shake it should be noted that capabilities can not only be decreased, but also extended should the need arise, e.g., after recovery or repair.

Consider the example of the secured database system in Figure 2. The design contains two N-variant-based layers, L^1 and L^2, that implement capability levels F^1 and F^2 respectively. Each query request is duplicated by the MRM and routed to both layers for processing. Consequently, each layer issues the same $query$ to a back-end database and encrypts the query result. The difference is that L^1 only encrypts the mission-critical portion of the query result as e_1^1 while L^2 encrypts the entire result as $\{e_1^2, e_2^2\}$. Requests to send back encrypted data from both layers are intercepted and checked by the MRM. Since we assume that the encryption algorithm is deterministic, $e_1^1 = e_1^2$ if both layers operate correctly. Otherwise, the MRM infers that layer L^2 has been compromised and hence it disables L^2. Consequently D scales back its capability to F^1, which is implemented by L^1. This action constitutes a survivability feature with respect to the functionality F.

The layered adaptive N-variant architecture is designed for improving system survivability for mission-critical applications. By implementing functionalities in layers the capability of shifting to a lower layer (upon detection of a fault) provides a contingent plan that allows a system to scale back its services towards essential services as the result of faults or malicious attacks. However, the layered architecture is also designed to support information security. As can be seen in the example above w.r.t. confidentiality, the MRM ensures that sensitive information in d_2 will not be leaked by a higher layer, even if the latter is compromised by an attacker. In the example, the detection of a fault due to discrepancies of results in L^1 and L^2, i.e., if $e_1^1 \neq e_1^2$, will result in MRM blocking the release of d_2.

4 Reliability and Resilience

If we look at the multi-variant approach within a single layer or our architecture, we can see that the N-variant approaches described in [3, 4, 7, 8] are actually special cases, i.e., these approaches can be adopted at any layer within our architecture. It should be noted that they all have specifications and implementation at the same level and layer respectively. This means that the approaches deal with fault detection and possible treatment dependent on the degree of redundancy. However, adaptability and graceful degradation as described above is not supported. For example, the multi-variant scheme described in [8] uses two variants of memory referencing. Both variants implement the same functional capability. The model in [3] has the similar limitation.

Fault masking using N-variant approaches is actually more effective than typically observed in redundant systems, e.g., *k-of-N* or NMR. For example, in a triple modular redundant systems two faulty modules can produce the same result and consequently the TMR would vote on the incorrect value in the majority vote. Given the schemes described in [8] and [7] it is statistically very unlikely that two modules produce the same fault. This is very advantageous when trying to determine thresholds for non-faulty values and to reduce the degree of N-variants at each layer.

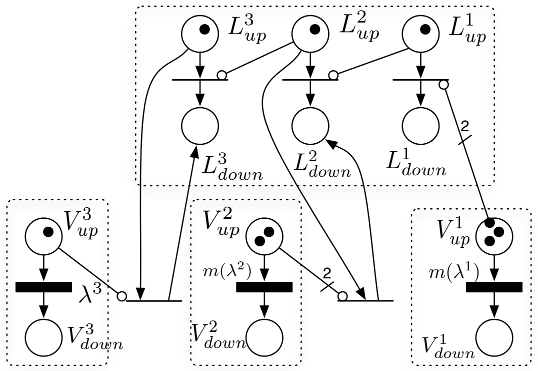

Figure 4. Petri-net for F_1 in Figure 3

The analysis of the multi-level and multi-layer approach is described using the example of functionality F_1 shown in Figure 3. F_1 has three levels, F_1^1, F_1^2 and F_1^3, and their respective layers L^1, L^2 and L^3 use 3-variant, 2-variant, and simplex implementations. Given the levels of redundancy at each layer one can note that at L^1 one can mask one value fault, at L^2 one can only detect (but not mask) one fault and L^3 has neither detection nor correction potential. The masking and detection capabilities of functionality F^1 is modeled in the Petri net shown in Figure 4. Note that this net does not reflect inter-layer monitoring, which will be addressed separately. The upper subnet models the reliability of the layers and is controlled by the Petri nets of the triplex, duplex and simples of layer L^1, L^2 and L^3 respectively. Note that only the timed transitions of the triplex and duplex depend on the markings of their input places, reflected by the marking functions $m(\lambda^1)$ and $m(\lambda^2)$ respectively. The simplex at layer L^3 has a fail rate of simply λ^3. Furthermore note that unreliability of individual layers are the probability of a token in the places L^i_{down}.

Adaptability, and thus fault treatment, are modeled in the upper part of the net. For example, layer L^3 fails if either it fails due to the firing of the transition between places V^3_{up} and V^3_{down}, or if it is "shut down" due to a failure of a lower layer, i.e., should a layer at level i fail, it will automatically shut down layer $i+1$, implemented by inhibitory arcs in the upper part of the Petri net.

It is simple to establish the exact reliability of F_1 when the fail-rates are known. However, in the presence of malicious faults, e.g., hacking attacks or exploits, the assumption of constant fail-rates does not hold anymore. In that case, the Petri net stays the same, whereas the formal analysis of the net becomes much more complicated. The reason is that the constant fail rates of the timed transitions have to be replaced by time-dependent hazard functions.

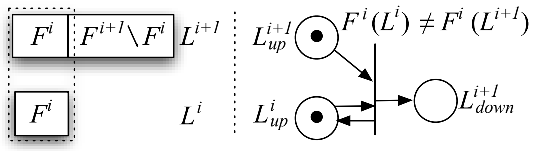

Figure 5. Cross-level monitoring

Figure 5 generalizes monitoring between two adjacent layers. The left side of the figure shows the relationship between two levels of requirements for functionality F, i.e., between F^i and F^{i+1}. Note that $F^i \prec F^{i+1}$. The respective implementations are in layers L^i and L^{i+1}. Note that layer L^{i+1} consists of the implementations of the operations of the lower layer based on F^i as well as the value-added operations specified by $F^{i+1} \setminus F^i$. Thus monitoring is limited to operations specified by F^i.

The Stochastic Activity Network (SAN) of inter-layer cross-monitoring is shown on the right side of Figure 5. The transition is activated when operations specified by F^i differ in layer i and $i+1$, i.e., if $F^i(L^i) \neq F^i(L^{i+1})$, where $F^i(L^j)$ indicates the functional specification with respect to layer L^j. Since F^{i+1} includes F^i the MRM indicates fault-free behavior if $F^i(L^i) = F^i(L^{i+1})$.

5 Conclusion

This research defined a hierarchical formal model for N-variant executions especially suitable for systems based on multi-core architectures. The model has two dimensions to support fault detection and real-time adaptation. Multiple levels of functionality are implemented in layers. At each (horizontal) layer, N-variant implementations support detection and masking of faults. Individual layers can incorporate different N-variant solutions, including existing techniques such as in [3, 4]. Adaptation is introduced in the other (vertical) dimension. Lower layers, which implement the essential subset of capabilities of the higher layers, are used to cross-monitor the higher layers. This is possible due to the inclusion relationship between functional specifications at different levels. If discrepancies are detected between layers the shut-down of the higher layer is initiated. The use of N-variant executions at individual layers has several advantages. First, lower level functionalities can effectively cross-monitor higher layers, which has positive implications for security and reliability. Second, during adaptation executions can be shifted to lower layers, which increases survivability and resilience.

References

[1] A. Avizienis , *The Methodology of N-version Programming*, Software Fault Tolerance, edited by M. Lyu, John Wiley & Sons, 1995.

[2] M.H. Azadmanesh, and R.M. Kieckhafer, *Exploiting Omissive Faults in Synchronous Approximate Agreement*, IEEE Trans. Computers, 49(10), pp. 1031-1042, Oct. 2000.

[3] B. Cox, et. al., N-Variant Systems A Secretless Framework for Security through Diversity, 15th USENIX Security Symp., Vancouver, Aug. 2006

[4] C.M. Jeffery, and J.O. Figueiredo, Towards Byzantine Fault Tolerance in Many-core Computing Platforms, 13th IEEE International Symposium on Pacific Rim Dependable Computing, 2007.

[5] A. Krings, et.al., *Resilient Multi-core Systems: A Hierarchical Formal Model for N-variant Executions*, Proc. CSIIRW09, ORNL, April 13-15, 2009.

[6] R. Laddaga, P. Robertson, and H. Shrobe, *Introduction to Self-adaptive Software: Applications*, Proc. 2nd Workshop on Self Adaptive Software, LNCS 2614, pp 275-283, May, 2001.

[7] A. Nguyen-Tuong, et. al., Security through Redundant Data Diversity, DSN, Anchorage, June 2008.

[8] B. Salamat et. al. Multi-Variant Program Execution: Using Multi-Core Systems to Defuse Buffer-Overflow Vulnerabilities, CISIS, pp. 843-848, 2008.

[9] P. Thambidurai, and Y.-K. Park, *Interactive Consistency with Multiple Failure Modes*, Proc. 7th Symp. on Reliable Distributed Systems, Columbus, OH, pp. 93-100, Oct. 1988.