6-2014

A Constrained, Possibilistic Logical Approach for Software System Survivability Evaluation

Yanjun Zuo
University of North Dakota, yanjun.zuo@und.edu

A CONSTRAINED, POSSIBILISTIC LOGICAL APPROACH FOR SOFTWARE SYSTEM SURVIVABILITY EVALUATION

Yanjun Zuo

University of North Dakota
Grand Forks, ND, USA
yanjun.zuo@und.edu

Received (October 10, 2013)
Revised (March 12, 2014)
Accepted (April 4, 2014)

In this paper, we present a logical framework to facilitate users in assessing a software system in terms of the required survivability features. Survivability evaluation is essential in linking foreign software components to an existing system or obtaining software systems from external sources. It is important to make sure that any foreign components/systems will not compromise the current system's survivability properties. Given the increasing large scope and complexity of modern software systems, there is a need for an evaluation framework to accommodate uncertain, vague, or even ill-known knowledge for a robust evaluation based on multi-dimensional criteria. Our framework incorporates user-defined constrains on survivability requirements. Necessity-based possibilistic uncertainty and user survivability requirement constraints are effectively linked to logic reasoning. A proof-of-concept system has been developed to validate the proposed approach. To our best knowledge, our work is the first attempt to incorporate vague, imprecise information into software system survivability evaluation.

Keywords: Survivability; software system; logic; constraint; possibility; evaluation.

1. Introduction

The security and reliability of software systems have become more important than ever as those systems are continuously used in various high security and high integrity settings, such as healthcare, national defense, financial services, telecommunications, and utility infrastructure. Given the critical functions that those systems provide, they must be reliable and dependable. Unfortunately, critical software systems are often the targets of malicious attacks due to the important roles that they play. Furthermore, those systems are often subject to functional and operational failures. As components, architectures, and networks become ever more complex, there are simply more things that could go wrong. In a study by the National Institute of Standards and Technology (NIST), it was reported that software systems deficiency in security and reliability alone cost the US economy $59.5 billion annually in breakdowns and repairs [1]. This does not include the loss of productivity as employees spent time trying to clean up after attacks. Hence, we must make sure that a critical software system has the ability to survive malicious attacks and system failures while still providing adequate levels of services to support mission critical applications. Consequently, evaluating the survivability features of a system becomes important.

Survivability is a multi-dimensional notation, which covers not only security but also reliability and other aspects of a software system. While security focuses on prevention, defense, and recovery from intentional attacks, reliability focuses on system operational fault tolerance, functional damage masking, and performing robustness. Rather than trying to achieve a completely attack-free system, survivability focuses on provisioning of an acceptable level of services even in the presence of malicious attacks [2]. Hence, survivability aims at a higher level of assurance that a system can survive malicious attacks and internal failures even if part of the system has been damaged. It includes not only security but also other important features, such as reliability, adaptation, re-configuration, and damage recovery.

For many mission-critical systems, survivability is an important system property. Survivability considerations have to be designed into a system, rather than in an add-on fashion [3]. Any systems/components acquired from external sources must meet a user's criteria to ascertain that those systems will not compromise the survivability properties of the existing systems. This assentation is essential when the systems/components are to be applied to support mission-critical functions. The research work in this paper is applied to such situations as linking software components/modules dynamically to the existing systems or obtaining external software systems from third-party providers. Software linking and acquisition have been widely used in component-based software development where individual software modules/components are composed to form a larger-scale software system. In those cases, any foreign software systems/components must meet the users' survivability requirements.

In this paper, we propose a logical approach for evaluating the survivability features of a software system/component. The user's survivability requirements are represented in a logic, called $\hat{\mathcal{C}}$-$\mathcal{P}$, with application specific operators and inference rules. A software system's compliance with those requirements are checked through a logic reasoning process. Applying a formal, logic-based approach provides a rigorous verification and guarantee of some system properties in a well-structured reasoning process. When the scope of software systems becomes large and their complexities continuously grow, there is a pressing need for formal methods to analyze, evaluate, and verify important system properties for the purposes of security, reliability, and survivability. Based on solid mathematical structures and proven theories, a logic-based approach offers some key advantages, such as rigorous reasoning, systematic analysis, and sound methodologies.

A use scenario of the framework Figure 1 shows a generic use scenario of our proposed $\hat{\mathcal{C}}$-$\mathcal{P}$ framework and the major steps to conduct survivability evaluation for a software system/component. In step 1, a user determines their survivability requirements towards the software system/component to be acquired. Those requirements are specified as the user's survivability policy. To evaluate that the software system/component complies with the policy, the user's survivability evaluation agent first collects evidence from some trusted evaluators who can verify certain survivability features of the system/component (step 2). The evidence is encoded as formulas in the $\hat{\mathcal{C}}$-$\mathcal{P}$ logic. The framework has an evaluator server, which certifies the trusted evaluators for different survivability properties. It is likely that no evaluator can assess all the survivability features of a system. So, the evaluator server can authorize different evaluators are used for different survivability properties. To ensure data integrity, all the survivability supporting evidence is signed digitally. After the necessary evidence is collected and all the digital signatures are verified, the user applies a theorem prover program to prove/disapprove that the system/component complies with the user's survivability policy (step 3). If so, the system/component is considered as satisfied with the user's survivability requirements and can be safely acquired or linked (step 4).

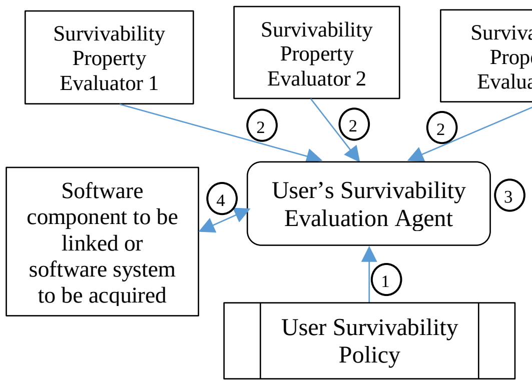
Survivability Property Evaluator 1
Survivability Property Evaluator 2
Survivability Property Evaluator *n*
2
2
2
Software component to be linked or software system to be acquired
4
User's Survivability Evaluation Agent
3
1
User Survivability Policy

Fig. 1: A Generic Use Scenario of the Survivability Evaluation Framework

<u>*Design principles*</u> In designing the $\hat{\mathcal{C}}$-$\mathcal{P}$ survivability evaluation framework, we follow a set of principles and guidelines as shown below.

First of all, since survivability is a multi-dimensional concept, a software system's properties need to be evaluated from different aspects, including security, adaptability, robustness, and fault tolerance. We apply the survivability requirement model [4], which allows a user to customize their specific requirements by defining flexible terms from different perspectives, called *survivability characteristics* (*SC*s). A *SC* contains a set of primitive properties called *survivability primitives* (*sp*s), which further describe the system's more concrete features in the context of that *SC*. For an illustrative purpose, a set of *SC*s and *sp*s are listed in Table 1 [4].

Secondly, given the increasing large scope and complexity of modern software systems, it is virtually impossible for any user to evaluate every property of a system. Due to lack of expertise and detailed knowledge about the software system, such an evaluation may be overwhelming for a user. As we illustrated in the use scenario, to obtain an objective and more accurate assessment about a system's survivability features, third-party trusted evaluators can be used who are specialized in some particular aspects of system survivability features. More specifically, our approach supports collecting survivability property certificates from trusted evaluators, encoded as $\hat{\mathcal{C}}$-$\mathcal{P}$ logic formulas, reasoning on those individual assessments through a logic proof process, and integrating them into a complete survivability evaluation result.

Thirdly, it is often the case that even a specialized evaluator may not be very certain about a particular feature of a software system. Therefore, our approach supports logic reasoning on uncertain, imprecise, or even vague information. This uncertainty-aware reasoning is achieved by defining many-valued logic formulas and necessity-based possibilistic uncertainty, where uncertain information can be formally represented and linked to a logic reasoning process. Our framework makes it possible to express fuzzy pattern matching in formal survivability proof.

Finally, an evaluation framework should be applicable to practical case scenarios. In terms of users' system property requirements, it should have a mechanism to represent and reason about constraints on the required survivability features of a software system. Some system properties may take others as their pre-requisite conditions. For example, the system's self-healing ability depends on an accurate and timely damage assessment. As another example, the capability of a system to reasonably predict the causes of system faults and take the corresponding corrective actions to recover from damage is closely related to the system's ability to control vulnerability. Therefore, both of those two properties may be required for the system. Incorporating those constraints and allowing an efficient connection between a constraint domain and a logic reasoning process is critical for a survivability evaluation framework to be practical. As will be discussed later, our $\hat{C}$-$\mathcal{P}$ framework supports constrained logic reasoning to accommodate these and other types of constraints.

Table 1. Survivability Characteristics (*SC*s) and Survivability Primitives (*sp*s)

Survivability Characteristics (*dimensions*)	**Survivability Primitives** (*Survivability properties within each dimension*)	
Reliability and resilience (***SC***$_1$)	Service availability (sp_1)	Service consistency (sp_2)
	Endurability (sp_3)	Predictability (sp_4)
	Strong authentication/authorization (sp_5)	Vulnerability control (sp_6)
	Monitoring and control (sp_7)	Behavioral adjustment (sp_8)
Adaptability (***SC***$_2$)	Reusability (sp_9)	Component reconfiguration (sp_{10})
	Process migration (sp_{11})	Connectivity and interoperability (sp_{12})
Recoverability (***SC***$_3$)	System self-healing (sp_{13})	Prompt damage assessment (sp_{14})
	Service/component resynchronization (sp_{15})	Fault traceability (sp_{16})
	Malice immunization (sp_{17})	
Fault tolerance (***SC***$_4$)	Redundancy based fault masking (sp_{18})	Fault isolation (sp_{19})
	Tolerance through backup (sp_{20})	Proxy-based fault tolerance (sp_{21})

<u>*Research contributions*</u> Our major contributions in this paper include the following: (1) a threshold-based, multi-dimensional framework for software system survivability evaluation; (2) a formal approach to represent possibilistic uncertainty on many-valued formulas annotated with principals' belief (evaluation) degrees; (3) a systematic mechanism to incorporate uncertain, imprecise knowledge and user-defined constraints to logic reasoning; (4) a specification of the soundness of the proposed $\hat{C}$-$\mathcal{P}$ logic; and (5) a proof-of-concept prover system to validate the logic.

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3, we introduce a base logic. Our constrained, probabilistic logic is based on this logic. Section 4 presents the details of our $\hat{C}$-$\mathcal{P}$ logical framework. Section 5 discusses experiments and performance evaluation. Section 6 concludes this paper.

2. Related Work

Several research works have been conducted on survivability evaluation. Shen, *et al.* [5] propose a mechanisms of survivability evaluation for attacked Wireless Sensor Network. Based on the classical reliability theory, the evaluation is composed of the reliability, survival lifetime, and availability in the steady state. Yang, *et al.* [6] analyze and evaluate the survivability of three virtual machine-based architectures, i.e., load balance server architecture, isolated component server architecture, and Byzantine fault tolerant server architecture. Different architectures are modeled with Continuous Time Markov Chain. The transient behaviors and steady states of those architectures are analyzed. Wang, *et al.* [7] propose a network survivability analysis and evaluation model. Network survivability is abstracted as a dynamic game process among network attacker, network defender, and normal user. An evolutionary game model is established and analysis algorithm is proposed based on the game model. Ghasemieh, *et al.* [8] develop a model checking algorithm to evaluate the survivability properties of fluid critical infrastructures. The Stochastic Time Logic is introduced to express intricate state-based and until-based properties for hybrid Petri nets. The feasibility of the proposed approach is illustrated using a case study to evaluate the survivability of a water refinery and distribution plant. Fu, *et al.* [9] evaluate the survivability schemes in hybrid wireless optical broadband access network (WOBAN) from a point view of Quality of Recovery (QoR). The evaluation is based on availability, recovery time, redundancy, and bandwidth of a backup path. To verify the performance of the survivability schemes, extensive simulations are conducted under different WOBAN configurations. Wang, *et al.* [10] present a framework for network survivability testing and evaluation. While the testing of network survivability is performed based on specific network survivability measurement models, the evaluation of network survivability is based on the quantification results of network survivability metrics. The experimental results demonstrate the generality and practicability of the proposed framework.

Current research works are very specific – they focus on a particular system's survivability evaluation. Our research aims at a more general, systematic survivability evaluation framework for a wide range of critical software systems. Furthermore, we apply a formal, logic-based approach for survivability evaluation. In the literature, the PCS framework [4] is the closest to the work presented in this paper. However, there are several fundamental differences between our $\hat{C}$-$\mathcal{P}$ approach and the PCS framework. First, PCS focuses on proof-carrying mechanisms, and the $\hat{C}$-$\mathcal{P}$ framework targets a general, constrained and possibilistic evaluation model. Secondly, PCS cannot reason on incomplete, vague or ill-known knowledge. For instance, PCS cannot represent the case when an evaluator is only able to assess a system's property with a partial certainty (confidence), say 0.75, instead of a full level of guarantee. Furthermore, PCS cannot support that an evaluator signs a statement with a partial truth value about a particular survivability property. Technically, the truth of each formula in PCS is binary (either completely true or completely false), and an evaluator has no way to express a more subtle and gradual ranking towards a system property. Nevertheless, those features are highly desirable in an evaluation framework, given the increasing complexity of modern systems. Thirdly, the PCS framework cannot represent the inherent relationships among the properties of a system. As we have discussed earlier, appropriate constraints on those and other types of restrictions are both practically necessary and theoretically important. A mechanism to incorporate user survivability requirement constraints to logic reasoning is presented in [11]. However, that mechanism cannot represent uncertain or ill-known knowledge in logic reasoning and does not support fuzzy reasoning. Our proposed $\hat{C}$-$\mathcal{P}$ framework effectively supports both necessity-based possibilistic uncertainty and user survivability requirement constraints in logic reasoning. To our best knowledge, our work is the first attempt to incorporate vague, imprecise information into software system property evaluation.

The proposed $\hat{C}$-$\mathcal{P}$ framework uses necessity measuring as defined in possibilistic logic (PL) [12-13] to describe judgment uncertainty and to include fuzzy information in logical pattern matching. PL provides formal methodology to reason about incomplete or vague values, built upon classical first-order logic. However, our $\hat{C}$-$\mathcal{P}$ logic is more general to include higher-order formulas. Dubois, *et al.* [14] propose an extension of PL, dealing with fuzzy constants and fuzzily restricted quantifiers (called PLFC). Alsinet, *et al.* [15] provide PLFC with a formal semantics and a sound resolution-style calculus. Like many other works, the proof method for PLFC is based on refutation through rules, such as resolution, fusion, and merging. $\hat{C}$-$\mathcal{P}$ also incorporates variable weights and fuzzy constants, but the proof method is goal-oriented and performed in a bottom-up manner through a set of inference rules. Benferhat, *et al.* [16] propose a flexible representation of uncertain information where the weight in a possibilisic formula is associated with an interval instead of a unit number. The semantics of the interval-based PL is compatible with the standard PL bases, and it effectively extends PL whenever all the intervals are singletons. Alsinet, *et al.* [17] present a new logic programming language, called Possibilistic Defeasible Logic Programming, which combines features from argumentation theory and logic programming, incorporating the treatment of possibilistic uncertainty. Such features are formalized on the basis of PGL, a possibilistic logic, based on Godel fuzzy logic.

In the literature, there are several research works on constraint specification and verification in formal logics. Jia and Walker [18] present an intuitionistic linear logic with constraints which combines the logic with a classical constraint domain. Classical formulas are isolated by a modal operator rather than binary connectives. Bartak [19] propose to use constraint satisfaction to describe which variables are unified in the hypotheses to specify the structure of the hypothesis in inductive logic programming. A constrain model is presented with index variables accompanied by a Boolean model to strengthen inference and hence improve efficiency. Saranli and Pfenning [20] propose a constrained intuitionistic linear logic for hybrid robotic planning problems. The logic incorporates domain-dependent constraints to linear logical reasoning (e.g., some resources are consumable and hence can only be used once) by defining two new connectives (used in a similar way to the pre- and post-condition operators in the $\hat{C}$-$\mathcal{P}$ logic) that effectively link the constraints to a sequent calculus statement. Constraint solving and specification are out of the scope of the logic and should be conducted in an application-specific constraint domain. Finally, Casali, *et al.* [21] present a logical framework to represent and reasoning about gradual desires and intentions. Additional constraints can be addressed that an agent can set about the kind of preferences he/she is dealing with. The framework is expressive to describe how desires, together with other information, can lead agents to intentions.

Constraint specification and solving in $\hat{C}$-$\mathcal{P}$ are different from the existing approaches. The necessity constraint domain is based on the well-developed PL logic; hence, its soundness and completeness can be proved. This proof increases the confidence for users to use the logic. For the survivability requirement constraint domain, formal constraint types and rules are defined. As we will see, the constraint solver is implemented as a logical program. Therefore, the $\hat{C}$-$\mathcal{P}$ framework can directly integrate constraint specification and checking into a logical search process. This integration is different from all the existing approaches where the constraint reasoning is separated from the logical reasoning itself. Closely tying constraint expression and verification to logical operations has an advantage that constraint information can be used as part of the logic reasoning, and its correctness can be guaranteed, using logic instruments.

3. A Base Logic

To better present our $\hat{C}$-$\mathcal{P}$ logical framework for software system evaluation, we specify a base logic to build our logic constructs for constrained, probabilistic logic reasoning. Instead of developing a completely new logic, the PCS logic [4] is chosen since it supports the above-mentioned threshold, multi-dimensional evaluation guidelines. However, the proposed techniques in this paper can be applied to other logic models. We next briefly discuss the predicates and inference rules of the PCS logic.

The PCS predicates can be classified into two categories:

a) Survivability property certification predicates.

- *ensures*(*Ev*, *S*): entity *Ev* (e.g., an evaluator) endorses that statement *S* (e.g., *sat*(*sp*)) is true. For example, if the statement *S* is digitally signed by *Ev*, then *Ev* ensures *S*;
- *sat*(*sp*): the system under evaluation satisfies the user's requirements in terms of a survivability primitive *sp*;
- *signed*(*K*, *S*): statement *S* is signed by an entity with a public key *K*;
- *keyBind*(*K*, *E*): entity *E* is bound to a public key *K*. Therefore, any statement signed by the private key paired with *K* should be considered as a statement ensured by *E*;
- *cerAuth*(*CA*): entity *CA* is a key certificate authority, i.e., if *CA* confirms a key binding between a public key *K* and an entity *E*, the predicate *keyBind*(*K*, *E*) is true;
- *evaluator*(*Ev*): entity *Ev* is a trusted evaluator on a certain survivability property *sp*. Therefore, if *Ev* says that a system satisfies the property *sp*, then it is believed true.

b) Threshold selection predicates.

- *ltso*(*Sys, n, SPlist*): a low bound threshold selection operator. For a given survivability characteristic *SC*, *ltso*(*Sys, n, SPlist*) specifies that a system *Sys* must satisfy at least *n* out of sp_1, sp_2, …, sp_m survivability primitives in *SPlist* in order to be considered as satisfying the user's requirements in terms of *SC*. *n* is called the degree of *ltso*(*Sys, n, SPlist*);
- *cltso*(*Sys*, *n*, *mL*, *SPlist*): a conditional low bound threshold operator. It indicates that a system *Sys* must satisfy at least *n sp*s in *SPlist*, among which all the *sp*s in *mL* ($mL \subseteq SPlist$) must be satisfied.

Using *ltso*(.) and *cltso*(.), a survivability requirement policy is shown below where the predicate *ok_To_Satisfy* indicates that the system *Sys* satisfies the user's survivability requirement policy. Since the system *Sys* to which it is implicitly referred in a *ltso*(.) or *cltso*(.) predicate, we omit *Sys* in the following discussions. As we can see, to prove *ok_To_Satisfy*, it only needs to prove an appropriate set of *ltso*(.) and *cltso*(.).

$$cltso(4, \{sp_5\}, \{sp_1, sp_2, sp_3, sp_4, sp_5, sp_6\}) \wedge (cltso(4, \{sp_7, sp_{12}\}, \{sp_7, sp_8, sp_9, sp_{10}, sp_{11}, sp_{12}\}) \vee (ltso(3, \{sp_{13}, sp_{14}, sp_{15}, sp_{16}, sp_{17}\}) \wedge ltso(2, \{sp_{18}, sp_{19}, sp_{20}, sp_{21}\}))) \rightarrow \quad ok\_To\_Satisfy$$

The PCS inference rules relevant to our discussions are shown in Figure 2. The first four rules reason about a system satisfying a *sp*, i.e., *sat*(*sp*). The rule "*ensures_i*" says that if a statement *S* (e.g., *sat*(*sp*)) is signed by a private key corresponding to the public key *K* and an entity *E* is bound to *K*, then we can conclude that *E* ensures that *S* is true. The rule "*ensures_e*" says that if an evaluator server *Es* says that *Ev* is a trusted evaluator, then it is believed so. The "*keyBind_i*" rule says that if a certificate authority *CA* ensures that a key binding statement between *K* and *E* is true, then we believe that *K* is the public key of entity *E*. The "*sat_i*" rule says that if an evaluator *Ev* ensures a statement *sat*(*sp*), then it is believed that the statement is true, i.e., the system satisfies the user's requirements in terms of that survivability primitive *sp*.

$$\frac{signed(K, S)\quad keyBind(K, E)}{ensures(E, S)}\ (ensures\_i) \qquad \frac{cerAuth(CA)\quad ensures(CA, keyBind(K, E))}{keyBind(K, E)}\ (keyBind\_i)$$

$$\frac{ev\_server(Es)\quad ensures(Es, evaluator(Ev))}{evaluator(Ev)}\ (evaluator\_i) \qquad \frac{n = 0}{ltso(n, SPlist)}\ (ltso\_b)$$

$$\frac{evaluator(Ev)\quad ensures(Ev, sat(sp))}{sat(sp)}\ (sat\_i) \qquad \frac{sat(sp)\quad ltso(n - 1, SPlist)}{ltso(n, mk\_Set(SPlist, sp))}\ (ltso\_i_1)$$

$$\frac{ltso(n, SPlist)}{ltso(n, mk\_Set(SPlist, sp))}\ (ltso\_i_2) \qquad \frac{ltso(|mL|, mL)\quad ltso(n - |mL|, set\_Sub(SPlist, mL))}{cltso(n, mL, SPlist))}\ (cltso\_e)$$

Fig. 2. PCS Logic Inference Rules

The next four rules reason about a system satisfying a user's requirements in terms of a survivability characteristic *SC* as represented in a low-bound threshold selection structure *ltso*(*n, SPlist*). The "*ltso_b*" rule represents the base case: if the degree $n \le 0$, then *ltso*(*n, SPlist*) is true. The "$ltso\_i_1$" and "$ltso\_i_2$" rules represent the recursive evaluation of *ltso*(*n, SPlist*). If *sp* is satisfied, then *ltso*(*n*, [*sp*|*SPlist*]) will be reduced to *ltso*(*n-1, SPlist*). Otherwise, *ltso*(*n, SPlist*) must be satisfied. Finally, the rule "*cltso_e*" converts the proof of a *cltso*(.) to the equivalent *ltso*(.).

We next use an example to briefly describe how to apply the PCS logic predicates and inference rules to compile a proof for *sat*(*sp*). For more details, please refer to [4].

Step 1: $signed(K_{CA}, keyBind(K, EServer)) \wedge keyBind(K_{CA}, CA) \rightarrow ensures(CA, keyBind(K, EServer))$

By applying the *ensures_i* rule, if a statement *keyBind*(*K, EServer*) is signed by an entity with a public key K_{CA} and K_{CA} is bound to entity *CA*, then it is believed that *CA* ensures that the statement *keyBind*(*K, EServer*) is true.

Step 2: $ensures(CA, keyBind(K, EServer)) \wedge cerAuth(CA) \rightarrow keyBind(K, ESever)$

By applying the *keyBind_i* rule, if an entity *CA* ensures a key binding between an entity *EServer* and a public key *K* and *CA* is an authorized certificate authority, then it is believed that *K* is the public key of *ESever*.

Step 3: $ensures(EServer, Evaluator(Ev)) \wedge ev\_server(ESever) \rightarrow evaluator(Ev)$

By applying the *evaluator_i* rule, if an entity *ESever* ensures that *Ev* is a trusted evaluator and we know that *EServer* is an authorized evaluator server, then it can be believed that the endorsed statement *evaluator(Ev)* is true, i.e., entity *Ev* is indeed a trusted evaluator.

Step 4: $evaluator(Ev) \wedge ensures(Ev, sat(sp)) \rightarrow sat(sp)$

By applying the *sat_i* rule, if entity *Ev* ensures a statement *sat*(*sp*) and *Ev* is a trusted evaluator, then it is believed that *sat*(*sp*) is true, i.e., the system under evaluation satisfies the user's requirement in terms of survivability primitive *sp*.

A PCS inference rule can be specified as a higher-order Horn clause "$p_0 \vee \neg p_1 \vee \ldots \vee \neg p_n$", where each p_i is a positive atomic predicate representing a term, such as *keyBind*(*K*, *E*) and *sat*(*sp*). More conveniently, the rule is denoted as "$p_1 \wedge \ldots \wedge p_n \rightarrow p_0$". For example, the $ltso\_i_1$ rule is written as "*sat*(*sp*) ∧ *ltso*(*n*-1, *Splist*) → *ltso*(*n*, *mk_Set*(*SPlist*, *sp*))". We next discuss our constrained, possibilistic $\hat{C}$-$\mathcal{P}$ logic built on top of the PCS logic.

4. The $\hat{C}$-$\mathcal{P}$ Logical Framework for Software System Survivability Evaluation

This section presents the details of our logical framework for software system survivability evaluation. Its core part is a constrained, possibilistic logic, called $\hat{C}$-$\mathcal{P}$, based on the PCS logic. The $\hat{C}$-$\mathcal{P}$ logic supports fuzzy pattern matching for survivability evaluation uncertainty reasoning and user requirement constraint specification and verification. In the following discussions, we first introduce many-valued formulas with annotated necessity measures. Then, we present a mechanism to represent user-defined survivability requirement constraints. Thirdly, we specify the $\hat{C}$-$\mathcal{P}$ inference rules in a sequent calculus. Fourthly, we discuss proof search given a survivability evaluation goal statement by applying a set of inference rules subject to a set of constraints. Finally, we show the soundness of the $\hat{C}$-$\mathcal{P}$ logic.

4.1. *Many-Valued Formulas with Annotated Necessity Measures*

4.1.1. *Truth necessity measure of a formula*

As we mentioned earlier, the $\hat{C}$-$\mathcal{P}$ logic supports expression and reasoning on uncertain information, such as partial belief of an evaluator about his/her evaluation on a survivability feature of a system, the uncertainty that an entity is a trusted evaluator, and the partial truth of a statement, such as *ltso*(*n*, *SPlist*). To represent the possibilistic uncertainty of a logic formula, we use the approach of annotating – formulas are labeled with truth necessities (or *necessities* for short) [12, 22]. In the $\hat{C}$-$\mathcal{P}$ logic, a necessity-annotated formula is denoted as a weighted pair $<F, \alpha>$, where F represents an atomic formula (e.g., *sat*("*self-healing*"), *ltso*(*n*, *SPlist*)) or a compound formula (e.g., *ev_Server*(*Es*) ∧ *ensures*(*Es*, *evaluator*(*Ev*)) → *evaluator*(*Ev*)); and α represents a lower bound on the degree of necessity for F to be true. α is evaluated to an element of a totally-ordered, bounded scale Ł, such as a lattice. For the sake of simplicity, in this paper we use the ordinary scale {0, 1} for the domain of α.

A necessity-annotated formula $<F,\alpha>$ in $\hat{C}$-$\mathcal{P}$ is interpreted as $N(F) \geq \alpha$, where $N(.)$ is a truth necessity function, mapping a set of logical formulas to Ł ([15, 23]). $N(F) \geq \alpha$ means that F is true at least α–certain. For example, <*sat*("*self-healing*"), 0.8> indicates that an evaluator certifies that a software system satisfies the user's survivability requirement in terms of the property "*self-healing*" to a degree of at least 0.8. The necessity function $N(.)$ in the possibilistic logic [12, 22] is specified in terms of a *possibility distribution* π over a set of interpretations ρ on the formulas in the language, i.e., $\pi: \rho \rightarrow \{0,1\}$. Essentially, π models the belief states on the set of interpretations ρ. Each interpretation $I \in \rho$ represents one belief state view, mapping each atomic formula to a truth value. $\pi(I)$ represents the degree of compatibility of interpretation I with available information. $\pi(I) = 0$ means that the interpretation I is impossible, given the current information, while $\pi(I) = 1$ means that I is totally possible. Given π, the possibility measure Π and necessity measure N for a formula F are defined as follows [12, 13]: $\Pi(F) = max\{\pi(I): I \in \rho; I \models F\}$ and $N(F) = 1 - \Pi(\neg F)$, where $I \models F$ represents the interpretation I entails formula F. π satisfies a formula $<F, \alpha>$, denoted as $\pi \models <F,\alpha>$, iff $N(F) \geq \alpha$.

Given the above notations, a necessity-annotated formulas in $\hat{C}$-$\mathcal{P}$ logic is represented as $F_c = <F, \alpha>$, such as <*keyBind*(*E*, *K*), α_1>, <*sat*("*self-healing*"), α_2>, <*ok_To_Satisfy*, α_2>. For an inference rule l: $p_1 \wedge p_2 \ldots \wedge p_n \rightarrow p$, the quantitative relationship among the necessity measures of l and its predicates is specified in the generalized modus ponens rule [24] as shown below:

$$\frac{\begin{array}{c}<(p_1 \wedge p_2 \ldots \wedge p_n \rightarrow p), \gamma> \\ <p_1, \beta_1> \quad <p_2, \beta_2> \quad \ldots \quad <p_n, \beta_n>\end{array}}{<p, min(\gamma, \beta_1, \beta_2, \ldots, \beta_n)>}$$

4.1.2. *Many-valued truth*

To incorporate uncertain and imprecise information in logic reasoning, we define fuzzy predicates in $\hat{C}$-$\mathcal{P}$ logic with variable weights and fuzzy constants [14]. Basically, a variable weight is added to a formula F so that the truth of the formula is many-valued (no longer binary) depending on a particular value that the variable will take in a fuzzy set. In our framework, this allows an evaluator to express his/her assessment of a system's survivability feature in a fine-graded scale rather than a simply binary satisfaction or unsatisfaction. Consider the formula *sat*(*sp*) in the rule "*sat*(*sp*) ∧ *ltso*(*n*-1, *Splist*) → *ltso*(*n*, *mk_Set*(*SPlist*, *sp*))". The many-valued counterpart of *sat*(*sp*) in $\hat{C}$-$\mathcal{P}$ is represented as *sat*(*sp*, *x*), which indicates that a system satisfies the user's survivability requirement in terms of *sp* with a partial truth determined by a value to be taken for the variable *x*. Therefore, the truth of the predicate *sat*(*sp*, *x*) can now take any intermediate value in {0, 1}. A variable weight is employed to express the statement such as "the more *x* is *High* (highly satisfied), the more true is *sat*(*sp*, *x*)", where *High* is defined on a fuzzy set with a membership function denoted as $\mu_{High}(x)$. Defined on a domain $Đ$, say an evaluation scale {0, 10}, $\mu_{High}(x)$ maps each value in $Đ$ to a member degree of *High*, i.e., $\forall\ x \in Đ$, $\mu_{High}(x) \rightarrow \{0, 1\}$. Essentially, a variable weight can be seen as (flexible) restrictions on a universal quantifier. Semantically, it is interpreted as "*sat*(*sp*, *x*) is true with a necessity measure at least $\mu_{High}(x)$". When *High* represents an imprecise, but non-fuzzy interval, such as {6, 7}, then *sat*(*sp*, *x*) = 1 if $x \in \{6, 7\}$ and *sat*(*sp*, *x*) = 0, otherwise.

Fuzzy constants are used to model the statement such as "an evaluator assesses that a system satisfies the users' survivability requirements in terms of *sp*, as represented by a value in an range *B*, say {3, 4}", denoted as *sat*(*sp*, *B*). Such a statement represents the evaluation certificate collected from a trusted evaluator. Exactly which value in {3, 4} will be taken is uncertain. Therefore, the statement is interpreted as "*sat*(*sp, y*) = 1 if $\exists y \in \{3, 4\}$ and *sat*(*sp, y*) = 0, otherwise". A formula *p*(*B*) with a fuzzy constant *B* can be seen as (flexible) restrictions on an existential quantifier over formula *p*.

In a logical reasoning process, the unification between a variable weight *x* (defined on a fuzzy set *A* on a domain $Đ$) and a fuzzy constant *B* is conducted using the following rule, where $N(A \mid B)$ represents the possibility of *A* given *B* and is defined as [15, 23]: $N(A \mid B)$

$= \mu_A(B) = inf_{x \in B}\mu_A(x)$, if B is a non-fuzzy, but imprecise set defined on $Đ$; or $N(A \mid B) = inf_{x \in Đ}\, max(1- \mu_B(x), \mu_A(x))$, if B is a fuzzy set.

$$\frac{<p(x) \rightarrow q,\ \alpha_1> \qquad <p(B),\ \alpha_2>}{<q,\ min(\alpha_1,\ \alpha_2,\ N(A \mid B)>}$$

Given that the truth of a formula F with variable weights and fuzzy constants is many-valued, for an interpretation I, $I(F)$ can now take any intermediate value in $\{0, 1\}$. For a possibility distribution π, a formula F no longer induces a crisp set, but rather a fuzzy set of interpretations. We use the notion $[F]$ to represent such a fuzzy set of interpretations, i.e., $[F] \subseteq \rho$ so that for each $I \in [F]$, we have $I(F) > 0$. Formula (1) [15, 23] is used to measure the uncertainty induced on a formula F by π on $[F]$. We will use this formula to prove the soundness of the $\hat{C}$-$\mathcal{P}$ logic (see Section 4.5).

$$N([F] \mid \pi) = inf_{I \in [F]}\, max(1- \pi(I), I(F)) \qquad (1)$$

4.2. *Specifying Survivability Requirement Constraints*

While Section 4.1 describes the logic constructs to support uncertain and fuzzy information in logic reasoning, this section discusses a mechanism based on [11] to incorporate user-defined survivability requirement constraints to logic reasoning. To explain how our framework specifies constraints on the *sp*s in *SPlist* in proving a formula *ltso*(*n*, *SPlist*) or *cltso*(*n*, *mL*, *SPlist*), we give some constraint examples [11] (see Figure 3) with regard to *SC*s and *sp*s represented in Table 1. Although they are only exemplary, those constraints show it is necessary for those user-desired constraints to be represented and enforced. We call such types of constraints *survivability requirement constraints*.

To clearly state constraints, there must be a formal specification to define the constraints. In the meantime, constraint verification should be easily integrated into logical reasoning. To enforce the constraints, rules must be defined and proof obligations must be created whenever the rules are applicable. To show that logical reasoning will not violate any of those constraints, some elements must be prohibited in a proof process. A constraint domain *Ç* is defined to integrate those three components [11] as shown below:

Ç.L: a survivability requirement constraint specification. It is composed of a set of constraints representing the inherent causality and dependency relationships among the *sp*s in *SPlist* in proving *ltso*(*n*, *SPlist*) as well as the corresponding constraint rules for checking and enforcing those relationships. We present those constraint rules in Table 2. The constraints shown in Figure 3 can therefore be represented as ***Constraint* 1**: sp_2 ╡║ sp_1; ***Constraint* 2**: ╔ sp_8, sp_{10} ╗; and ***Constraint* 3**: sp_{17} ┤ [$sp_{16} \vee sp_{14}$].

Ç.O: a set of proof obligations that must be satisfied. It contains the *sp*s which are obligated to be proved as a result of applying some constraint rules. As shown in Table 2, when a *sp* is provable, some proof obligations may be generated. For instance, if a constraint specifies that sp_i depends on sp_j, …, sp_k, and if $sat(sp_i)$ becomes true, then sp_j, …, sp_k are obligated to be proved. So, $\{sp_j, …, sp_k\}$ must be admitted to *Ç.O* as proof obligations. When a proof process proceeds and resources are available (e.g., when sp_j, …, sp_k are proved), those corresponding proof obligations will be discharged.

Ç.R: a set of *sp*s that must be restricted in any future poof process. Such a restriction is also created as a result of applying some constraint rules. For instance, for an exclusive rule sp_i ╫ sp_j, if sp_i is provable, then sp_j must be excluded in any future proof process. The last row of Table 2 describes this case.

As we can see, checking whether a *sp* complies with *Ç* amounts to rule-checking *sp* against each constraint rule in *Ç.L*. A rule is applicable if *sp* can unify with the premise of the rule. A proof violates the constraints if for any *sp* to be proved, (1) *sp*∈*Ç.R*; or (2) for any constraint rule *r*∈*Ç.L*, *sp* satisfies *r* and the resulting *Ç.O* ∩ *Ç.R* ≠ ∅.

***Constraint* 1**: In SC_1, the survivability primitive "*Service consistency*" (sp_2) depends on "*Service availability*" (sp_1). This dependency must be specified since if a service cannot be guaranteed to be available, then it is impossible to require service consistency.

***Constraint* 2**: In SC_2, "*Behavior adjustment*" (sp_8) and "*Component reconfiguration*" (sp_{10}) must be required to be satisfied at the same time.

***Constraint* 3**: In SC_3, "*Malice immunization*" (sp_{17}) depends on either "*Prompt damage assessment*" (sp_{14}) or "*Fault traceability*" (sp_{16}) since either of them satisfies the pre-conditions for the system to generate immunization formulas to make sure that other components will not be subject to the same type of attacks in case some components have been compromised.

Fig. 3: Survivability Requirement Constraints

Table 2. Survivability Requriement Constraints and the Corresponding Constraint Rule Semantics

Constraints	**Constraint Definitions**	**Operations on *Ç.O* & *Ç.R* if sp_i is Proved**

Atomicity: ╓ sp_i, sp_j, ..., sp_k ╖	{sp_i, sp_j, ..., sp_k} must all be satisfied if any one of them, say sp_i, is proved.	*Ç.O* ∪{sp_j, ..., sp_k}
		If sp_i ∈*Ç.O*, then *Ç.O* – {sp_i}
		If sp_i∈{sp_m, …,sp_n} and *L*∈*Ç.O*, then *Ç.O* – *L*

Dependency: sp_i ╡ {sp_j, …, sp_k}	sp_i depends on all elements in{sp_j, …, sp_k}, i.e., if sp_i is proved, then sp_j, …, sp_k are obligated to be proved.	*Ç.O* ∪{sp_j, …, sp_k}
		If sp_i ∈ *Ç.O*, then *Ç.O* – {sp_i}
		If sp_i ∈{sp_m, …, sp_n} and *L*∈*Ç.O*, then *Ç.O* – *L*

Selective dependency: sp_i ┤ [sp_j ∨ …∨ sp_k]	sp_i depends on one element in {sp_j, …, sp_k}. If sp_i is satisfied, either sp_j, …, or sp_k must be satisfied.	*Ç.O* ∪{[sp_j∨ …∨ sp_k]}
		If sp_i∈*Ç.O*, then *Ç.O* – {sp_i}
		If sp_i∈{sp_m, …, sp_n} and *L*∈*Ç.O*, then *Ç.O* – *L*

Exclusion: sp_i ╫ sp_j … ╫ sp_k	{sp_i, sp_j, …, sp_k} cannot be satisfied at the same time	If sp_i ∈*Ç.O*, then *Ç.O* – {sp_i}
		If sp_i ∈{sp_m, …, sp_n} and *L* ∈*Ç.O*, then *Ç.O* – *L*
		Ç.R ∪ {sp_j, …, sp_k}

*In Table 2, we assume sp_i is the constraint term to be checked. We define *L* = [sp_j∨ …∨ sp_k] to represent a special selective set in such a way that if any one element in {sp_j, …, sp_k} is satisfied, then the whole set [sp_j∨ …∨ sp_k] is satisfied and can be discharged from *Ç.O*.

4.3. *Sequent Calculus Rules*

The $\hat{C}$-$\mathcal{P}$ framework provides a mechanism to unify hybrid constraint domains with a logical reasoning process. The interplay between constraint checking/verification and logic reasoning is through a set of inference rules. We present the $\hat{C}$-$\mathcal{P}$ logic inference rules using sequent calculus.

4.3.1. *A sequent*

To describe the high level design of $\hat{C}$-$\mathcal{P}$ logic reasoning, we start from a sequent (hypothetical judgment), which is the fundamental construct for reasoning from assumptions. A sequent has a format "$\sum$; Ψ; Γ => F_c", where $\sum$ represents a context, Γ represents a set of hypothetical formulas, F_c represents a conclusion formula, and Ψ contains a set of constraints. The sequent is interpreted as: given all the appropriately sorted variables in $\sum$, if the constraints in Ψ are all satisfiable, then we can prove the goal formula F_c given the hypothesis in Γ.

<u>*The Context*</u> $\sum$ defines the vocabulary of the $\hat{C}$-$\mathcal{P}$ logic language, including the following specifications:

(1) the sort (or type) of each term or variable appearing in a formula. Basic sorts include characters, strings, integers, sets, principals, survivability requirement parameters, necessity measures, and constraint types,
(2) the variable weights and fuzzy constants in a formula. For each variable weight, the corresponding fuzzy set A is specified with its membership function $\mu_A(x)$. For a fuzzy constant, the corresponding fuzzy set or the imprecise but non-fuzzy interval is defined,
(3) a mapping of each basic sort to a domain of values,
(4) a mapping of each necessity measure term (e.g., constant, or valuation expression) to an element of a totally-ordered, bounded scale.

<u>*The constraint* Ψ</u> Consider a $\hat{C}$-$\mathcal{P}$ rule "$<p_1, \beta_1> \wedge <p_2, \beta_2> \ldots \wedge <p_n, \beta_n> \rightarrow <p, \alpha>$". The equality $\alpha = min(\beta_1, \beta_2, \ldots, \beta_n)$ must hold. We consider this is another type of constraint and call it a *necessity-based possibilistic uncertainty constraint* (or *necessity constraint* for short), since it represents the semantics of an assertion about the quantitative relationship among the necessity values of the formulas of an inference rule. We represent this type of constraint using $\tilde{N}$ in order to distinguish it from the survivability requirement constraints Ç as we discussed earlier. Therefore, the entire constraint domain Ψ in $\hat{C}$-$\mathcal{P}$ logic is composed of these two hybrid constraint worlds represented by $\tilde{N}$ and Ç. We denote these two sub-domains as $\Psi.\tilde{N}$ and Ψ.Ç, respectively.

In constraint solving, a *constraint term* (denoted as C) refers to a formula to be checked with regard to a constraint sub-domain or a variable to be solved (to a ground term) given other constraints in that particular sub-domain. More specifically, a constraint term in Ψ.Ç may be a *sp* to be checked (e.g., to see whether proof of *sp* would potentially violate any constraint rules) or Ψ.Ç.O (e.g., to see whether the proof obligations have all been discharged). In $\Psi.\tilde{N}$, C represents either an assertion about the relationship among the necessity measures of a set of formulas (e.g., $\alpha = min(\beta_1, \beta_2, \ldots, \beta_n)$) or a necessity variable to be solved.

A constraint solver is defined for each of Ψ.Ç and $\Psi.\tilde{N}$. The two major tasks of a constraint solver are:

(1) to determine whether a constraint term C can be admitted to the constraint sub-domain, denoted as $admit(\Psi.\tilde{N}, C)$ or *admit*(Ψ.Ç, C). The former is to verify whether C can be assumed given the existing necessity values and the necessity relationships in $\Psi.\tilde{N}$. Let $\Psi.\tilde{N}$ contain a set of necessity assertions $C_1, C_2, \ldots, C_n$. C is admissible to $\Psi.\tilde{N}$ if there is no conflict in assigning values to the free variables in $C_1, C_2, \ldots, C_n$ given the quantitative constraints expressed by C. If indeed there is no conflict, $\Psi.\tilde{N}$ is updated by adding C. Otherwise, $admit(\Psi.\tilde{N}, C)$ returns false.

The function of *admit*($\Psi.Ç$, C) is to check that the admission of an *sp* (represented by C) will not violate any constraint rules defined in $\Psi.Ç.L$ given the current constraint state. Similar to the case for $\Psi.\tilde{N}$, *admit*($\Psi.Ç$, C) returns false if C is not admissible to $\Psi.Ç$. As we discussed earlier, a failure case occurs when (1) $sp \in \Psi.Ç.R$; or (2) for any constraint rule $r \in \Psi.Ç.L$, *sp* satisfies r and the resulting $\Psi.Ç.O \cap \Psi.Ç.R \neq \emptyset$. Otherwise, $\Psi.Ç$ is updated by adding *sp* to $\Psi.Ç.O$ and performing other actions as shown in Table 2.

(2) to check whether a constraint term C can be solved given the existing constraints in $\Psi.\tilde{N}$ or $\Psi.Ç$ (denoted as $\Psi.\tilde{N} \vdash C$ or $\Psi.Ç \vdash C$), i.e., whether a constraint set Ψ entails C. If C represents an *sp*, $\Psi.Ç \vdash C$ is true iff *admit*($\Psi.Ç$, C) is updated without violating any constraint. Otherwise, if C represents a constraint obligation set O to be checked, $\Psi.Ç \vdash C$ returns true if $\Psi.Ç.O$ is empty (i.e., all the proof obligations have been fulfilled). Constraint solving in $\Psi.\tilde{N}$ means solving a necessity variable (as represented by C) to a ground term given the existing necessity constraints in $\Psi.\tilde{N}$. This essentially reduces to a multi-equation/inequality solving problem.

We next discuss some properties of the constraint solvers, which can be easily proved:

(1) (***Hypothesis***): $\Psi.Ç \vdash C$ is true for a constraint term C not unifiable with any rule $r \in \Psi.Ç.L$;

(2) (***Truth universal rule***): if C is a ground term, then for a constraint domain $\Psi.\tilde{N}$, we have $\Psi.\tilde{N} \vdash C$.

(3) (***Cut***): if $\Psi.\tilde{N} \vdash C$ and $\Psi.\tilde{N}, C \vdash C'$, then $\Psi.\tilde{N} \vdash C'$;

(4) (***Weakening***): if $\Psi.\tilde{N} \vdash C$, then $\Psi.\tilde{N}, nc \vdash C$ for any necessity constraint *nc*, given *admit*($\Psi.\tilde{N}$, *nc*).

4.3.2. *Ĉ-Ƥ logic inference rules*

We introduce two more connectives in *Ĉ-Ƥ*: a constraint implication operator "$\gg_c$" and a constraint conjunction operator "$\ll_c$". While "$C \gg_c F_c$" introduces a pre-condition C to formula F_c, "$F_c \ll_c C$" asserts the validity of a post-condition C of F_c. These operators are used to connect the constraint checking with logic reasoning. Their semantics are represented by the inference rules that use them.

The *Ĉ-Ƥ* inference rules are represented in Figure 4, where F represents an unconstrained formula, $p(B)$ and $p(x)$ represents formulas with fuzzy constant B and variable weight x, φ is an arbitrary formula, $\sum \vdash v{:}c$ indicates that variable v has a sort c, and $\sum \vdash var(C)$ checks that the variables in formula C all have appropriate sorts. Each inference rule is read bottom up. As an example, the "$\ll_c L$" left rule indicates that the proof of $\Gamma, F_c \ll_c C \Rightarrow \varphi$ in a constraint domain Ψ can be reduced to the proof of $\Gamma, F_c \Rightarrow \varphi$ in a new constraint domain determined by *admit*($\Psi.\tilde{N}$, C). As we discussed earlier, a constraint term C is admissible to $\Psi.\tilde{N}$ if the addition of C will not cause any inconsistency between C and the existing constraints. As another example, the $initial_3$ rule represents that a semantic unification of fuzzy events is performed through variable weights and fuzzy constants. To prove Γ, $(p(B), \alpha') \Rightarrow (p(x), \alpha)$ given the variable weight x (defined on a fuzzy set A) and the fuzzy constant B, it is only necessary to prove Γ, $(p(B)) \Rightarrow (p(x))$ with the post-condition that the necessity constraint $\alpha = min(\alpha', N(A \mid B))$ is admissible to $\Psi.\tilde{N}$ (where $N(A \mid B)$ is defined in Section 4.1). The last example is about the $initial_4$ rule. It says that to prove $\Gamma \Rightarrow (ltso(0, SPlist), \alpha)$, we need to show that the necessity variable α can be set to 1 in $\Psi.\tilde{N}$ (i.e., $admit(\Psi.\tilde{N}, \alpha = 1)$) and that the obligation set $\Psi.\text{Ç}.O$ is empty. Other rules can be interpreted similarly.

$$\frac{\Sigma; \Psi; \Gamma \Rightarrow F_c \quad \Sigma; \Psi; \Gamma, F_c' \Rightarrow \varphi}{\Sigma; \Psi; \Gamma, F_c \rightarrow F_c' \Rightarrow \varphi} \quad (\rightarrow L \text{ rule}) \qquad \frac{\Sigma; \Psi; \Gamma, F_c \Rightarrow F_c'}{\Sigma; \Psi; \Gamma \Rightarrow F_c \rightarrow F_c'} \quad (\rightarrow R \text{ rule})$$

$$\frac{\Sigma; \Psi; \Gamma, F_c, F_c' \Rightarrow \varphi}{\Sigma; \Psi; \Gamma, F_c \wedge F_c' \Rightarrow \varphi} \quad (\wedge L \text{ rule}) \qquad \frac{\Sigma; \Psi; \Gamma \Rightarrow F_c \quad \Sigma; \Psi; \Gamma \Rightarrow F_c'}{\Sigma; \Psi; \Gamma \Rightarrow F_c \wedge F_c'} \quad (\wedge R \text{ rule})$$

$$\frac{\Sigma \vdash var(C) \quad \Sigma; \Psi.\text{Ç} \vdash C \quad \Sigma; \Psi; \Gamma, F_c \Rightarrow \varphi}{\Sigma; \Psi; \Gamma, C \gg_c F_c \Rightarrow \varphi} \quad (\gg_c L \text{ rule}) \qquad \frac{\Sigma \vdash var(C) \quad \Sigma; admit(\Psi.\text{Ç}, C); \Gamma \Rightarrow F_c}{\Sigma; \Psi; \Gamma \Rightarrow C \gg_c F_c} \quad (\gg_c R \text{ rule})$$

$$\frac{\Sigma \vdash var(C) \quad \Sigma; admit(\Psi.\tilde{N}, C); \Gamma, F_c \Rightarrow \varphi}{\Sigma; \Psi; \Gamma, F_c \ll_c C \Rightarrow \varphi} \quad (\ll_c L \text{ rule}) \qquad \frac{\Sigma \vdash var(C) \quad \Sigma; \Psi.\tilde{N} \vdash C \quad \Sigma; \Psi; \Gamma \Rightarrow F_c}{\Sigma; \Psi; \Gamma \Rightarrow F_c \ll_c C} \quad (\ll_c R \text{ rule})$$

$$\frac{\Sigma \vdash v{:}c \quad \Sigma; \Psi; \Gamma, [v/x]F_c \Rightarrow \varphi}{\Sigma; \Psi; \Gamma, \forall x{:}c.F_c \Rightarrow \varphi} \quad (\forall L \text{ rule}) \qquad \frac{\Sigma \vdash v{:}c \quad \Sigma; \Psi; \Gamma \Rightarrow [v/x]F_c}{\Sigma; \Psi; \Gamma \Rightarrow \forall x{:}c.F_c} \quad (\forall R \text{ rule})$$

$$\frac{}{\Sigma; \Psi; \Gamma, F \Rightarrow F} \quad (initial_1 \text{ rule}) \qquad \frac{\Sigma; admit(\Psi.\tilde{N}, (\alpha_1 = \alpha_2)); \Gamma, F \Rightarrow F}{\Sigma; \Psi; \Gamma, <F, \alpha_1> \Rightarrow <F, \alpha_2>} \quad (initial_2 \text{ rule})$$

$$\frac{\Sigma \vdash x{:}variable\ weight\ A; B{:}fuzzy\ constant \quad \Sigma; admit(\Psi.\tilde{N}, (\alpha = \min(\alpha', \mu_A(B)))}{\Sigma; \Psi; \Gamma, (p(B), \alpha') \Rightarrow (p(x), \alpha)} \quad (initial_3 \text{ rule})$$

$$\frac{admit(\Psi.\tilde{N}, \alpha = 1) \quad \Psi.\text{Ç} \vdash \Psi.\text{Ç}.O}{\Sigma; \Psi; \Gamma \Rightarrow (ltso(0, SPlist), \alpha)} \quad (initial_4 \text{ rule})$$

Fig. 4: $\hat{C}$-$\mathcal{P}$ Logic Inference Rules

4.3.3. *The hypothesis set Γ*

Γ contains two categories of hypotheses: (1) A_{rules} – a set of necessity-annotated inference rule formulas used for proof derivation (see Table 3); and (2) A_{axioms} – a set of axioms assumed to be true. As we can see, the necessity constraint on each inference rule in A_{rules} is represented using the post-condition operator $\ll_c$ and solved in the constraint domain $\Psi.\tilde{N}$. As an example, "$(<signed(K, P), \alpha_1> \wedge <keyBind(K, E), \alpha_2> \rightarrow <ensures(E, P), \alpha>) \ll_c (\alpha = \min(\alpha_1,\alpha_2))$" indicates that the logical inference "$signed(K, P) \wedge keyBind(K, E) \rightarrow ensures(E, P)$" can be applied in a proof with a post-condition that the necessity constraint $\alpha = min(\alpha_1,\alpha_2)$ holds in $\Psi.\tilde{N}$.

Table 3. $\hat{C}$-$\mathcal{P}$ Inference Rule Formulas in A_{rules}

$(<signed(K, P), \alpha_1> \wedge <keyBind(K, E), \alpha_2> \rightarrow <ensures(E, P), \alpha>) \ll_c (\alpha = min(\alpha_1,\alpha_2))$
$(<cerAuth(CA), \alpha_1> \wedge <ensures(CA, keyBind(K, E)), \alpha_2> \rightarrow <keyBind(K, E), \alpha>) \ll_c (\alpha = min(\alpha_1,\alpha_2))$
$(<ev\_Server(Es),\alpha_1> \wedge <ensures(Es, evaluator(Ev)), \alpha_2> \rightarrow <evaluator(Ev), \alpha>) \ll_c (\alpha = min(\alpha_1,\alpha_2))$
$<ltso(0, SPlist), \alpha> \ll_c (\alpha = 1)$
$(<evaluator(Ev), \alpha_1> \wedge <ensures(Ev, sat(sp, x)), \alpha_2> \rightarrow <sat(sp, x), \alpha>) \ll_c (\alpha = min(\alpha_1,\alpha_2))$, where x is a variable weight defined on a fuzzy set A with a membership function $\mu_A(x)$
$(<sat(sp, x), \alpha_1> \wedge <ltso(n\text{-}1, SPlist), \alpha_2) \rightarrow <ltso(n, \{sp\} \cup SPlist), \alpha>) \ll_c (\alpha = min(\alpha_1,\alpha_2))$
$(<ltso(n', mL), \alpha_1> \wedge <ltso(n - n', SPlist - mL), \alpha_2> \rightarrow <cltso(n, mL, SPlist), \alpha>) \ll_c (\alpha = min(\alpha_1,\alpha_2))$

The axioms in A_{axioms} as shown in Table 4 represent some proof assumptions (e.g., *Emily* is a key certificate authority, public key K_{200} is bound to *Emily*). For a formula representing an evaluator's judgment on a system's survivability property, we use $check_{(\Psi.Ç)}(sp)$ to represent the constraint term *sp* as a pre-condition, which is to check whether the proof of sp_i satisfies the constraints defined in *Ψ.Ç*. In addition, a fuzzy constant *B* and a belief necessity measure α are specified in a constrained formula such as <*signed*(K_{100}, *sat*("*service migration*", {3, 4})), *0.9*>.

Table 4. *Ĉ-Ƥ* axiom assumpitons in A_{axiom}

$check_{(\Psi.Ç)}$(*sp*: "*service migration*") »$_c$ <*signed*(K_{100}, *sat*("*service migration*", {3,4})), 0.9>
<*cerAuth*(*Emily*), 1>
<*keyBind*(K_{200}, *Emily*), 0.98>
<*signed*(K_{200}, *keyBind*(K_{100}, *Alice*)), 0.95>
<*ev_Server*(*Bob*), 1>

4.4. *Proof Search*

Proof search is a process for identifying a derivation of a goal statement given a set of assumptions *Γ* by applying a set of *Ĉ-Ƥ* inference rules subject to a set of constraints. A proof derivation is logically viewed as a tree rooted by the conclusion sequent (e.g., ∑; *Ψ*; *Γ* => <*ltso*(3, {"*service migration*", …}), *α*>). The leaf sequents of the derivation tree are all axioms and each non-leaf sequent is derived from its premise sequents by an inference rule application. A goal statement such as <*ltso*(*n*, *SPlist*), *α*> is provable if all the following are true: (1) *ltso*(*n*, *SPlist*) is provable; (2) no constraint is violated in either *Ψ.Ç* or *Ψ.Ñ*; and (3) *α* is solved to a ground term.

A theorem prover generates a proof (or disproof) given a survivability requirement policy. In a proof process, each application of an inference rule reduces a sequent matching the conclusion of the rule to its premises. A branch of the proof is successfully terminated when the formula to be proved unifies with a formula in the hypothesis set *Γ* and no constraint is violated. The resulting unifier is propagated to the next remaining premise and the process is repeated. Proof search follows the following rules:

(1) if every leaf node is an instance of an axiom (i.e., ∑; *Ψ*; *Γ*, F_c => F_c or ∑; *Ψ*; *Γ* => <*ltso*(*0, SPlist*), *α*>) and the corresponding constraints are solvable, the proof search has terminated successfully;
(2) if some leaf contains no logical connectives, but is not an instance of any axiom, then the search has terminated unsuccessfully;
(3) if a leaf contains some logical connectives, a search step may choose one connective and apply the corresponding inference rule to reduce the proof of the conclusion to its premises.

4.5. *Soundness of Ĉ-Ƥ Logic*

A logic is said to be sound if a formula can be proved syntactically from a set of assumptions *Γ*, then under any model, the formula must be true if the assumptions in *Γ* are true. To prove the soundness of a logic, there should be a model that gives meaning to the logic. We next show the soundness of the *Ĉ-Ƥ* logic.

Theorem (*Soundness*). Any constrained formula F_c proved in $\hat{C}$-$\mathcal{P}$ is sound with respect to the underlying higher-order logic and the constraint models, i.e., the possibility distribution model [15, 23] and the survivability requirement model (see Section 4.2).

Proof (*sketch*). The soundness of the $\hat{C}$-$\mathcal{P}$ logic can be proved from the following three perspectives.

(1) The $\hat{C}$-$\mathcal{P}$ logic is based on PCS, which is in turn based on the higher-order logic - each unconstrained operator in $\hat{C}$-$\mathcal{P}$ is defined in the higher-order logic terms. Since each unconstrained term of $\hat{C}$-$\mathcal{P}$ is eventually represented by the higher-order logic terms, the soundness of $\hat{C}$-$\mathcal{P}$ is determined by the underlying higher-order logic which has been proved as sound [25].

(2) The soundness of the $\hat{C}$-$\mathcal{P}$ in terms of necessity constraints is proved with respect to the possibility distribution model [15, 23].

As we discussed in Section 4.1, a normalized possibility distribution π models the belief states on the set of interpretations ρ defined as $\pi: \rho \rightarrow \{0, 1\}$. π satisfies a constrained formula $<F, \alpha>$, written $\pi \models <F, \alpha>$ iff $N([F] \mid \pi) \geq \alpha$. Now let Γ be a set of constrained formulas. Γ entails $<F, \alpha>$, denoted as $\Gamma \models <F, \alpha>$ iff every possibility distribution π that satisfies all the constrained formulas in Γ also satisfies $<F, \alpha>$. In this sense, we say $<F, \alpha>$ is the semantic consequence of Γ.

From the syntactic reasoning perspective, if a set of possibilistic formulas Γ induces a constrained formula $<F, \alpha>$, denoted as $\Gamma \vdash <F, \alpha>$, we say that $<F, \alpha>$ is the syntactic consequence of Γ.

The soundness of $\hat{C}$-$\mathcal{P}$ is proved by showing: if $\Gamma \vdash <F, \alpha>$, then $\Gamma \models <F, \alpha>$. In other words, if we can syntactically prove $<F, \alpha>$ from a set of constrained formulas Γ, then this formula must be semantically correct in the possibility distribution model given Γ.

As we have seen, the generalized modus ponens rule (in Section 4.1) is the only rule used in $\hat{C}$-$\mathcal{P}$ for possibilistic uncertainty reasoning. Given $\Gamma \vdash <F, \alpha>$, to show the proof of $\Gamma \models <F, \alpha>$ is reduced to check, for each possibilistic distribution $\pi: \rho \rightarrow \{0, 1\}$, if $N([(p_1 \wedge p_2 \ldots \wedge p_n \rightarrow p)] \mid \pi) \geq \gamma$, $N([p_1] \mid \pi) \geq \beta_1$, $N([p_2] \mid \pi) \geq \beta_2$, …, $N([p_n] \mid \pi) \geq \beta_n$, we have $N([p] \mid \pi) \geq min(\gamma, \beta_1, \beta_2, \ldots, \beta_n)$. Following Formula (1) in Section 4.1, the first n+1 conditions amount to, for each interpretation $I \in \rho$, $max(1\text{-}\ \pi(I), I(p_1 \wedge p_2 \ldots \wedge p_n \rightarrow p)) \geq \gamma$, $max(1\text{-}\ \pi(I), I(p_1)) \geq \beta_1$, $max(1\text{-}\ \pi(I), I(p_2)) \geq \beta_2$, …, and $max(1\text{-}\ \pi(I), I(p_n)) \geq \beta_n$. The conclusion $N([p] \mid \pi) \geq min(\gamma, \beta_1, \beta_2, \ldots, \beta_n)$ amounts to $max(1\text{-}\ \pi(I), I(p)) \geq min(\gamma, \beta_1, \beta_2, \ldots, \beta_n)$. According to the Godel semantics, we have $I(p_1 \wedge p_2 \ldots \wedge p_n) = min(I(p_1), I(p_2), \ldots, I(p_n)) = min(\beta_1, \beta_2, \ldots, \beta_n)$. Furthermore, $I(\varphi \rightarrow \omega) = 1$ if $I(\varphi) \leq I(\omega)$ and $I(\varphi \rightarrow \omega) = I(\varphi)$, otherwise. Here we have $I(p_1 \wedge p_2 \ldots \wedge p_n \rightarrow p) = 1$ (since each inference rule in PCS logic has been proved and therefore plausible, we have $\gamma = 1$). Consequently, we know $min(I(p_1), I(p_2), \ldots, I(p_n)) \leq I(p)$. Given $\gamma = 1$, we have $I(p) \geq min(\gamma, I(p_1), I(p_2), \ldots, I(p_n)) = min(\gamma, \beta_1, \beta_2, \ldots, \beta_n)$. Therefore, $max(1\text{-}\ \pi(I), I(p)) \geq min(\gamma, \beta_1, \beta_2, \ldots, \beta_n)$ holds for each interpretation I, and thus $N([p] \mid \pi) \geq min(\gamma, \beta_1, \beta_2, \ldots, \beta_n)$.

(3) The soundness of the $\hat{C}$-$\mathcal{P}$ logic with respect to the survivability requirement model is evidenced by the rules defined in Table 2. Since the semantics of those rules are defined by users and strictly enforced during a proof process, their

soundness is guaranteed as long as the constraint solver correctly implements those rules.

5. Experiments

To make sure that the $\hat{C}$-$\mathcal{P}$ logic inference rules are correct, we have developed a prototyping theorem prover implemented in JProlog (http://www.ugosweb.com/jiprolog/index.aspx). The $\hat{C}$-$\mathcal{P}$ logic engine is encoded in Prolog with 43 rules. We have conducted a set of experiments for system survivability evaluation on a PC with i5-3570 CPU @3.40GHZ and 3.48GB RAM. All the $\hat{C}$-$\mathcal{P}$ inference rules have been validated.

Theoretically, the time for the $\hat{C}$-$\mathcal{P}$ theorem prover to generate a proof given a goal statement (e.g., <*ok_To_Satisfy*, α>) depends on the complexity of the user's survivability requirement policy and the available assumptions that support the proof process. Given other factors being the same, the more $\vee$ logical connectives in a survivability requirement policy and the more proof options under each $\vee$ connective, the larger the search space is. Logically, each $\vee$ logical connectives represents a binary choice in determining which set of *SC*s to choose to prove. Consequently, a proof choice represents all the connective set of *SC*s that must be proved in order to satisfy the final goal statement. Taking the survivability requirement policy in Section 3 as an example, there are two proof choices from which to choose:

(1) $<cltso(4, \{sp_5\}, \{sp_1, sp_2, sp_3, sp_4, sp_5, sp_6\}), \alpha_1> \wedge <cltso(4,\{sp_7, sp_{12}\}, \{sp_7, sp_8, sp_9, sp_{10}, sp_{11}, sp_{12}\}), \alpha_2>$;

(2) $<cltso(4, \{sp_5\}, \{sp_1, sp_2, sp_3, sp_4, sp_5, sp_6\}), \alpha_1> \wedge (<ltso(3, \{sp_{13}, sp_{14}, sp_{15}, sp_{16}, sp_{17}\}), \alpha_3> \wedge <ltso(2, \{sp_{18}, sp_{19}, sp_{20}, sp_{21}\}) , \alpha_4>)$.

In the worst case scenario, the theorem prover needs to try all the proof choices to finally either prove or disapprove the final goal statement. Since the proof time for a proof choice is the sum of the times to prove all of the <*ltso*(.), α> in that proof choice, we will only show the proof time for one <*ltso*(.), α> in the following discussions. Furthermore, for each proof of <*ltso*(.), α> subject to a set of constraints, the proof time is the sum of the time required for the unconstrained logic reasoning and the time required for constraint checking and verification. Constraint solving in both of the necessity constraint domain $\Psi.\tilde{N}$ and user survivability requirement domain $\Psi.Ç$ has been implemented as logic rules. We have verified that the $\hat{C}$-$\mathcal{P}$ logical framework can represent and enforce all the types of constraints discussed in this paper.

To measure the performance of the $\hat{C}$-$\mathcal{P}$ evaluation engine, we set up the following specifications to represent an evaluation environment for a moderately complicated system/component in term of one survivability characteristic *SC* (as discussed in Section 3, each *ltso*(*n*, *SPlist*) corresponds to one *SC*):

(1) The *SC* contains 25 *sp*s, i.e., |*SPlist*| = 25,

(2) There are three certificate authorities in the system. A certificate authority is fully trusted and, therefore, a formula representing such an entity, e.g., *cerAuth*(*Emily*) has a necessity degree of 1. Each key binding certificate issued by a certificate authority has a partial trust value in a range {0, 1}, e.g., <*keyBind*(100, *Alice*), 0.97>,

(3) There are three evaluator servers, which verify that 11 entities are authorized evaluators. Like a certificate authority, an evaluator server is fully trusted, e.g.,

eServer(*John*) has a necessity degree of 1. A certificate issued by an evaluator server indicating that an entity is an authorized evaluator and therefore, has a necessity degree in a range {0, 1}, e.g., <*evaluator*(*Bob*), 0.99>;

(4) About half of *sps* in *SPlist* have been evaluated by the evaluators. An evaluator assesses the survivability features of a software system in a scale of {0, 10} with a belief degree in a range {0, 1}, e.g., <*sat*("*service migration*", {3, 4}), 0.9>,

(5) The degree of *ltso(n*, *SPlist)*, i.e., *n*, varies from 1 to 10,

(6) Each survivability requirement constraint is specified with 2-4 domain variables, e.g., ╓ sp_1, sp_{16}, sp_{17} ╖, sp_2 ╣ {sp_{10}, sp_{22}, sp_{25}}, and sp_1 ╫ sp_3╫ sp_8 ╫sp_{11},

(7) Each variable weight associated in a $\hat{\mathcal{C}}$-$\mathcal{P}$ rule has a fuzzy member function defined as a trapezoid (*a*, *b*, *c*, *d*) in a domain {0, 10}.

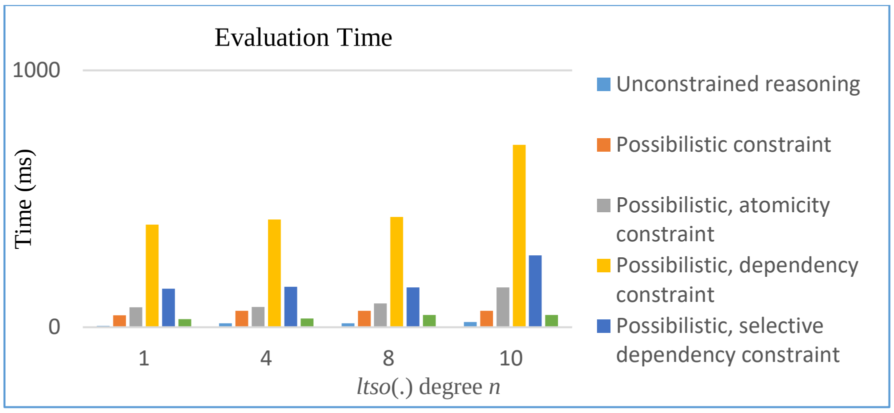

Fig. 5: $\hat{\mathcal{C}}$-$\mathcal{P}$ Performance Evaluation

To have a better understanding of how constraint checking and verification affect the performance of logic reasoning, we have measured the times required to prove a *ltso*(.) in the following cases:

(1) Unconstrained logic reasoning with formulas *ltso*(.), *sat*(*SP*), *keyBind*(*E*, *P*), etc.,

(2) Logic reasoning with necessity constraints only with formulas <*ltso*(.), *α*>, <*sat*(*SP*, *B*), β_1>, <*keyBind*(*E*, *P*), β_2>, etc.,

(3) Logic reasoning with necessity constraints with formulas <*ltso*(.), *α*>, <*sat*(*sp*, *B*), *β*>, etc., subject to a set of constraints ╓ sp_i, sp_j, ..., sp_k ╖, sp_i ╣ {sp_j, …, sp_k}, and sp_i ╫ sp_j … ╫sp_k.

Figure 5 shows the performance results to prove <*ltso*(.), *α*>. The longest time taken by the $\hat{\mathcal{C}}$-$\mathcal{P}$ theorem prover is 710 ms when the degree of *ltso*(.), *α*> is 10 subject to a *dependency* constraint (see Table 2). As we can see, in general the evaluation times in all the cases increase slightly when *n* increases. More specifically, the necessity constraint only adds a small amount overhead in terms of proof time (roughly 30-50 ms) to an unconstrained logic reasoning. However, different survivability requirement constraints (except the *exclusion* rule) result in different levels of overheads to prove the <*ltso*(.)*, α*>. In our simulations, the *dependency* constraint rule results in the highest overhead time (roughly 300-550 ms), while the *exclusion* constraint causes the least amount of overhead time. This difference can be explained by the nature of those constraint rules: a *dependency* rule essentially requires to prove additional *sp*s since the *sp* to be proved depends on other *sp*s while an *exclusion* rule exclusively prevents some *sp*s from being proved, therefore, reducing the search space. In general, we can see that the constraint checking and verification adds a moderate level of overhead to the unconstrained logic reasoning, making it realistic to be applied to an online survivability evaluation framework.

6. Conclusion

We present a new mechanism to incorporate survivability requirement constraints and possibilistic uncertainty to software system survivability evaluation. A logical framework has been developed to represent and to reason with uncertain and imprecise information under a set of user-defined constraints on system survivability requirements. A formal design is presented to link the hybrid worlds of constraint domains to logic reasoning. The interplay between the constraint checking and logic reasoning is supported by a set of logic inference rules. Applying a logic-based formal approach provides rigorous verification and guarantee of system properties in a well-structured reasoning process. When the scope of software systems becomes large and their complexities continuously grow, there is a pressing need for formal methods to analyze, evaluate, and verify important system properties.

Acknowledgement

This material is based upon work supported by the US Air Force Office of Scientific Research (AFOSR) under Awards FA9550-12-1-0131 and FA9550-09-1-0215. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of AFOSR.

References

[1] NIST, Software Errors Cost U.S. Economy $59.5 Billion Annually (NIST 2002-10), 2002, http://www.nist.gov/public_affairs/releases/n02-10.htm.

[2] C. Queiroz, A. Mahmood and Z. Tari, A Probabilistic Model to Predict the Survivability of SCADA Systems, *IEEE Transactions on Industrial Informatics* (9)4 (2013) 1975-1985.

[3] A. Serageldin, A. Krings and A. Abdel-Rahim, A Survivable Critical Infrastructure Control Application, in *Proc. of 8th Cyber Security and Information Intelligency Research Workshop*, Oak Ridge, TN, USA, 2012.

[4] Y. Zuo and S. Lande, A Logical Framework of Proof-Carrying Survivability", in *Proc. 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Changsha, China, 2011, pp. 472-481.

[5] S. Shen, R. Han, L. Guo, W. Li and Q. Cao, Survivability Evaluation Towards Attacked WSNs Based on Stochastic Game and Continuous-Time Markov Chain, *Applied Soft Computing* (12)5 (2012) 1467-1476.

[6] Y. Yang, Y. Zhang, A. H. Wang, M. Yu, W. Zang, P. Liu and S. Jajodia, Quantitative Survivability Evaluation of Three Virtual Machine-based Server Architectures, *Journals of Network and Computer Applications* (36)2 (2013) 781-790.

[7] C. Wang, Q. Miao, L. Fang, D. Wang, L. Ming and Y. Dai, A Novel Network Survivability Analysis and Evaluation Model, in *Proc. of 2nd International Symposium on Computer, Communication, Control and Automation,* Singapore, 2013, pp. 772-776.

[8] H. Ghasemieh, A. Remke and B. Haverkort, Survivability Evaluation of Fluid Critical Infrastructure Using Hybrid Petri Nets, in *Proc. of 19th IEEE Pacific Rim International Symposium on Dependable Computing*, Vancouver, Canada, 2013.

[9] M. Fu, M. He, Z. Le, W. Wang and B. Quan, Performance Evaluation of the Survivability Schemes in WOBAN: a Quality of Recovery (QoR) Method, *International Journal of Communication Systems*, 2013, DOI: 10.1002/dac.2707.

[10] C. Wang, L. Ming, J. Zhao and D. Wang, A General Framework for Network Survivability Testing and Evaluation, *Journal of Networks* (6)6 (2011) 831-841.

[11] Y. Zuo, Incorporating Constraints to Software System Survivability Specification and Proof, in *Proc. of 6th IEEE International Symposium on Theoretical Aspects of Software Engineering*, Beijing, China, 2012, pp. 67-74.

[12] D. Dubois, J. Lang and H. Prade, Possibilistic Logic. *In Handbooks of Logic in Artificial Intelligence and Logic Programming* (3) (1994) 439-513.

[13] D. Dubois, J. Lang and H. Prade, Automated Reasoning using Possibilistic Logic: Semantics, Belief Revision and Variable Certainty Weight, *IEEE Transactions on Data Knowledge Engineering* 6(1) (1994) 64-71.

[14] D. Dubois, H. Prade and S. Sandri, Possibilistic Logic with Fuzzy Constants and Fuzzily Restricted Quantifiers, in *Logic Programming and Software Computing, Chapter 4* (1998) pp. 69-90. Research studies press.

[15] T. Alsinet, L. Godo and S. Sandri, On the Semantics and Automated Deduction for PLFC, a Logic of Possibilistic Uncertainty and Fuzziness, in *Proc. 15th Conference on Uncertainty in Artificial Intelligence*, Stockholm, Sweden, 1999, pp. 3-10.

[16] S. Benferhat, J. Hue, S. Lagrue and J. Rossit, Interval-based Possibilistic Logic, in *Proc. of the Twenty-second International Joint Conference on Artificial Intelligence*, Barcelona, Spain, 2011, Vol. 2, pp. 750-755.

[17] T. Alsinet, C. Chesnevar, L. Godo and G. Simari, A Logic Programming Framework for Possibilistic Argumentation: Formalization and Logical Properties, *Fuzzy Sets and Systems* (159)10 (2008) 1208-1228.

[18] L. Jia and D. Walker, ILC: A Foundation for Automated Reasoning about Pointer Programs, in *Proc. of the 15th European Symposium on Programming Languages and Systems*, Vienna, Austria, 2006, pp. 131-145.

[19] R. Bartak, Constraint Models for Reasoning on Unification in Inductive Logic Programming, in *Proc. of the 14th International Conference on Artificial Intelligence: Methodology, Systems, and Application*, Varna, Bulgaria, 2010, pp. 101-110.

[20] U. Saranli and F. Pfenning, Using Constrained Intuitionistic Linear Logic for Hybrid Robotic Planning Problems, in *Proc. of IEEE International Conference on Robotics and Automation*, Rome Italy, 2007, pp. 3705-3710.

[21] A. Casali, L. Godo and C. Sierra, A Logical Framework to Represent and Reason about Graded Preferences and Intentions, in *Proc. of 11th International Conference on Principles of Knowledge Representation and Reasoning*, Sydney, Australia, 2008, pp. 27-37.

[22] D. Dubios and H. Prade, Possibilistic Logic: a Retrospective and Prospective View, *Fuzzy Sets and Systems* (144) (2004) 3-23.

[23] T. Alsinet and L. Godo, A Complete Calculus for Possibilistic Logic Programming with Fuzzy Propositional Variables, in *Proc. 16th Conference on Uncertainty in Artificial Intelligence*, Stanford, CA, USA, 2000, pp. 1-10.

[24] T. Alsinet, C. Chesnevar and L. Godo, A Level-based Approach to Computing Warranted Arguments in Possibilistic Defeasible Logic Programming, in *Proc. Conference on Computational Models of Argument*, 2003, pp. 1-12.

[25] A. Church, A Formulation of Simple Theory of Types. *Journal of Symbolic Logic* (5), 1940, pp. 56-68.