A Formal Approach For Engineering Resilient Car Crash Management System

Yasir Imtiaz Khan
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg

TR-LASSY-12-05

February 2012

Contents

Date	Version	Description	Author(s)
2012/02/22	0.1	Initial draft of this document	Yasir Imtiaz Khan
2012/03/07	0.2	Second draft of this document	Yasir Imtiaz Khan

Table 1: Revision History

1 Introduction

"The dependability of a computer system abstractly characterises its trustworthiness" [Guelfi, 2011]. The trustworthiness basically means the degree of user confidence that system will operate as they expect and system will not fail in its normal use. Informally dependability concepts are organized into three categories, which are attributes (availability, reliability, safety, confidentiality, integrity, maintainability), threats (faults, failures, errors) and means (fault prevention, fault tolerance, fault removal, fault forecasting) [Avizienis et al., 2004]. Resilience in information and communication technological systems was introduced around the seventies and has been most intensively used within the research community in the very few last years. By reviewing the important references, we notice that word resilience, is used with a variety of definitions and at different levels [Black et al., 1997, Mostert et al., 1995, Svobodova, 1984].
We have proposed a formal framework called DREF designed to ease the development of dependable systems from a software engineering perspective. This framework provides a mathematical definition of resilience and related concepts. The intention is also that framework should allow for being refined such that more detailed definitions of its concepts using different mathematical structures may be introduced in a consistent way with the framework [Guelfi, 2011].
In DREF the fundamental concepts are: entities, properties, satisfiability functions, nominal satisfiability, tolerance threshold and evolution. Entities are anything that is of interest to be considered. It might be e.g. a program, a database, a person, a hardware element, a development process, or a requirement document. Properties are the basic concepts to be used to characterise entities. It might be e.g. an informal requirement, a mathematical property or any entity that aims at being interpreted over entities. The fact that a property is satisfied by an entity is defined by a satisfiability function having the real numbers as co-domain. In our context, we want to consider entities whose existence (i.e. definition) may vary. Thus change is the difference between two definitions of two entities distributed over a common evolution axis. The intention is to allow for comparison of entities relatively to evolution axes. At the modeling level DREF model is to be composed of 3+1 categories of models. The first three are dedicated to the nominal view, tolerance view, the fail view and the fourth provide satisfiability view.
Intuitively, the general concept of resilience is as a property of an evolution process that is considered to improve capabilities thus avoiding failures and reducing degradations. Roughly it is the existence of a change toward improvement that reduces failures and tolerance needs. In this work, we have prime focus on building a resilient system based on DREF concepts. The generic problem we address in this report is to find a solution that allows for flexible handling of dependability or resilience in a scientific framework supported by software engineering tools and techniques. In order to achieve object, we are using case study of Car Crash management system. In particular entities are system models defined in term of Algebraic Petri Nets (APN) [Reisig, 1991] and properties are defined in terms of safety properties i.e. invariants regarding places in the APN as defined by the model checker AlPiNA [Buchs et al., 2010].

As shown in the figure 1, we are using AlPiNA model checker for the verification of interesting properties with respect to entities. At first we formally specify Car Crash system to Algebraic Petri Nets and interested properties to AlPiNA property language. The AlPiNA model checker uses as models Algebraic Petri Nets. Specifications in AlPiNA are composed of two parts: an

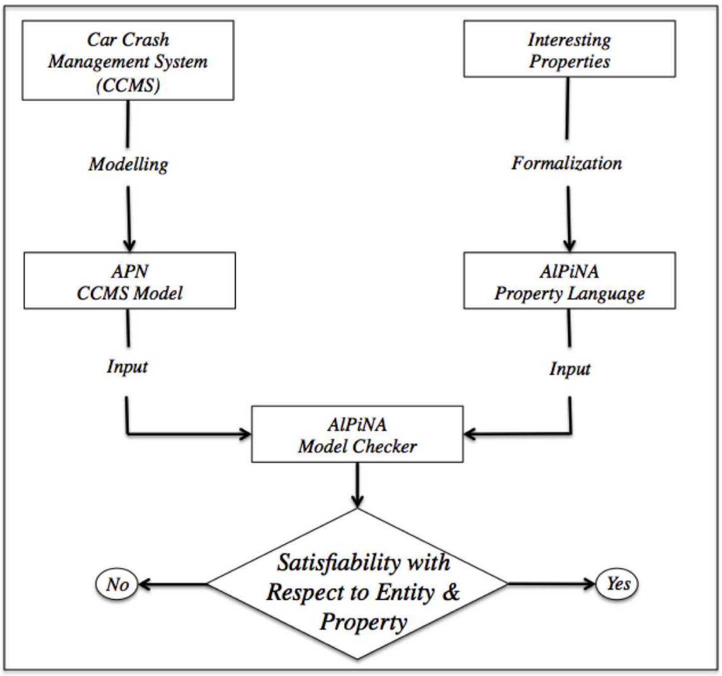

Figure 1: DREF satisfiability model

algebraic specification, which is a set of abstract definitions of sorts and associated operations; a Petri Net, which is represented graphically. AlPiNA and is able to decide on the satisfaction of invariant properties on those nets. The invariants are expressed as conditions on the tokens contained by places in the net at any state of the nets semantics. Invariants are built using first order logic, the operations defined in the algebraic specification and additional functions and predicates on the number of tokens contained by places.
The report is structured as follows: in section 2 we have discussed related work regarding dependability and resilience. In this section, we have discussed definitions regarding resilience or dependability according to various authors. Section 3 contains description about our case study. Use case formalism is uded for the behavioral description of case study. In sub sections algebraic petrinet models are discussed for different versions of Car Crash management system. In section 4, we have provided evolution satisfiability function; intention is to allow for comparison of entities or properties relative to an evolution axis. In section 5 we discuss open questions that arises from this research. Appendix is provided in section 6. In appendix section, we have provided all the algebraic specifications for the carCrash management system and screen shots from AlPiNA property checking results.

2 Related Work

In information and communication technological systems literature, resilience has been defined at various levels. In dependable computing community the most agreed definition of resilience is, the persistence of service delivery that can be justifiably be trusted, when facing changes and mainly regarded as fault tolerance [Lepri, 2008]. Informally, ability of a system to provide resilience and second, it is the ability to justify the provided resilience. Pragmatically resilient systems can be viewed as open distributed systems that have capabilities to dynamically adapt, in a predictable way, to unexpected and harmful events, including faults and errors. Engineering

such systems is a challenging issue, which implies reasoning explicitly and in a consistent way about functional and non-functional characteristics of systems.
In [Guelfi, 2011] introduced an abstract and generic terminology defined mathematically to be used when speaking about dependability and resiliency (called DREF). This formal framework is defined from a software engineering perspective, which means that we define its components such that they are useful for the development or improvement of analysis, architectural design, detailed design, implementation, verification and maintenance phases. This formal framework provides the necessary elements in accordance with a model driven engineering perspective that enable the definition of a new modeling language for dependable and resilient systems. In DREF the fundamental concepts are: entities, properties, satisfiability functions, nominal satisfiability, tolerance threshold and evolution. Entities are anything that is of interest to be considered. It might be e.g. a program, a database, a person, a hardware element, a development process, or a requirement document. Properties are the basic concepts to be used to characterise entities. It might be e.g. an informal requirement, a mathematical property or any entity that aims at being interpreted over entities. The fact that a property is satisfied by an entity is defined by a satisfiability function having the real numbers as co-domain. In DREF forseen events are ensured by the evolution of the satisfiability function while unforseen events are provided by the explicit definition of the satisfiability over the evolution axis.

3 Introduction to Crises Management System (Car Crash Case Study)

Generally Crisis management System is the process by which an organization deals with a major event that threatens to harm the organization, its stakeholders, or the general public. Crisis management involves identifying, assessing, and handling the Crisis situation. The scope of this work is limited to one particular kind of Crisis management system, which is the Car Crash Crisis management system. According to Wikipedia Car Crash is defined as:

"A car accident or car crash is an incident in which an automobile collides with anything that causes damage to the automobile, including other automobiles, telephone poles, buildings or trees, or in which the driver loses control of the vehicle and damages it in some other way, such as driving into a ditch or rolling over. Sometimes a car accident may also refer to an automobile striking a human or animal" [Kienzle et al., 2009].

In this technical report we are using a particular kind of Car Crash management system. In order to to keep the case study manageable, we shall use a simplified model of Car Crash management system. We used textual use cases formalism for discovering and recording behavioral requirements.

3.1 Use Cases carCrash

In principle use-case scenario is a story about how someone or something external to the software (known as an actor) interacts with the system. The actors involved in our case study are:
Coordinator: A person in charge of recording the Crisis information.
System Administrator: An in charge person for managing Crisis.
SuperObserver: A skilled person dispatched to the crisis scene. In our case there are two Superboservers which are fire fighter and lifter.

3.1.1 Use Case 1: Capture Crisis

Scope: Car Crash Crisis Management System Primary
Actor: Coordinator
Intention: The Coordinator intends to record a Crisis based on the information obtained from Capture data (Capture data can be a Fire on the Crisis location or Blockage of traffic).
Main Success Scenario:
Coordinator records Crisis and sends it to System.

1. Coordinator sends information to System as recorded.

Use case ends in success.

3.1.2 Use Case 2: Assign Mission

Scope: Car Crash Crisis Management System Primary
Actor: System Administrator
Intention: The System Administrator intends to assign a mission to Superobserver .
Main Success Scenario:
System Administrator assigns Superobserver to execute the mission.

1. System Administrator assigns a Crisis to Superobserver to execute Crisis mission.

Use case ends in success.

3.1.3 Use Case 3: Send Report

Scope: Car Crash Crisis Management System Primary
Actor: Superobserver
Intention: Send report to System after execution of the mission.

1. Superobserver sends report about executed Crisis mission.

Use case ends in success.

3.2 Formal Language Representation of Car Crash Management System

A petri net is a well-known mathematical modeling language for the description of distributed system, where Algebraic petri nets are evolution of petri nets. Algebraic petri nets has two aspects:
The control part, which is handled by a Petri Net.
The data part, which is handled by one or many AADTs.
In petri nets places hold resources also known as tokens and transitions are linked to places by input and output arcs, which can be weighted. Normally a petri net has a graphical concrete syntax consisting of circles for places, boxes for transitions and arrows to connect the two. The semantics of a P/T petri net involves the sequential non-deterministic firing of transitions in

the net where firing a transition means consuming tokens from the set of places linked to the input arcs of the transition and producing tokens into the set of places linked to the output arcs of the transition. We will be using concepts from the modelling world to represent and reason about the notions of entity, property and satisfaction, in particular: Algebraic Petri Nets as a modelling langage to represent entities and more generally evolving systems; the AlPiNA model checker to model decidable properties and compute their satisfaction on APN models.

3.3 Interesting Properties

In order to prove a system to be dependable we have following set of properties, which are availability, reliability, safety, confidentiality, integrity, maintainability [Avizienis et al., 2004].
In formal verification, we verify that a system meets a desired property by checking that a mathematical model of the system meets a formal specification that describes the property. In general property is safety property, which assert that the system always stays within some allowed region. Intuitively, a property is a safety property if every violation occurs after a finite execution of the system. We can use this fact in order to base model checking of safety properties on a search for finite bad prefixes. Such a search can be performed using a simple forward or backward symbolic reachability check [Kupferman et al., 1999]. From the dependability perspective safety property is defined as "the ability of the system to operate without catastrophic failure" [Sommerville, 2001]. Informally, safety property of a system is a judgment of how likely it is that the system will cause harm to environment or people.
In AlPiNA model checking all properties that are verified are reachability properties: a property is a Boolean expression that can be valid or not on a single state of the Petri Net state space. If the property does not hold for at least one state in the state space, we say that the property is invalid, and one counterexample is returned. AlPiNA does not return trace of the transitions fired to reach the counterexample, only counterexample is returned. In AlPiNA, an important aspect to note is, reachability properties do not include properties defined in temporal logics, as CTL or LTL. AlPiNA property language is inspired from language used in Helena; property language is mainly composed of expressions. There are three types of expressions, which are Boolean expressions, Natural expressions and Term expressions. AlPiNA property language is equivalent to first order logic [Buchs et al., 2010].
In our case, we are interested in the safety property, i.e.
"The Crisis can exist only if it has validated Crisis reporting and Superobserver"

An important safety threat, which we will take into an account in this case study, is that invalid Crisis reporting and Superobservers can be hazardous. Invalid Crisis reporting is the situation that results from wrongly reported Crisis. Execution of Crisis mission based on wrong reporting can waste both human and physical resources. In principle it is essential to validate Crisis that it is reported correctly. Practically Superobservers are assigned to execute Crisis mission according to their skills, for example, if there is a fire situation then it is obligatory to assign a fire fighter. It is critical to validate Superobserver for the safe execution of mission. What we mean from validate Superobserver is to assign a particular Superobserver(according to his skill) to the Crisis. System should prevent from the situation where it has any one of them is invalid.
We have safety property in the following composed property:

- ⋄ *Crisis can exist only if it has valid Crisis reporting*
- ⋄ *Crisis can exist only if it has valid Superobserver*

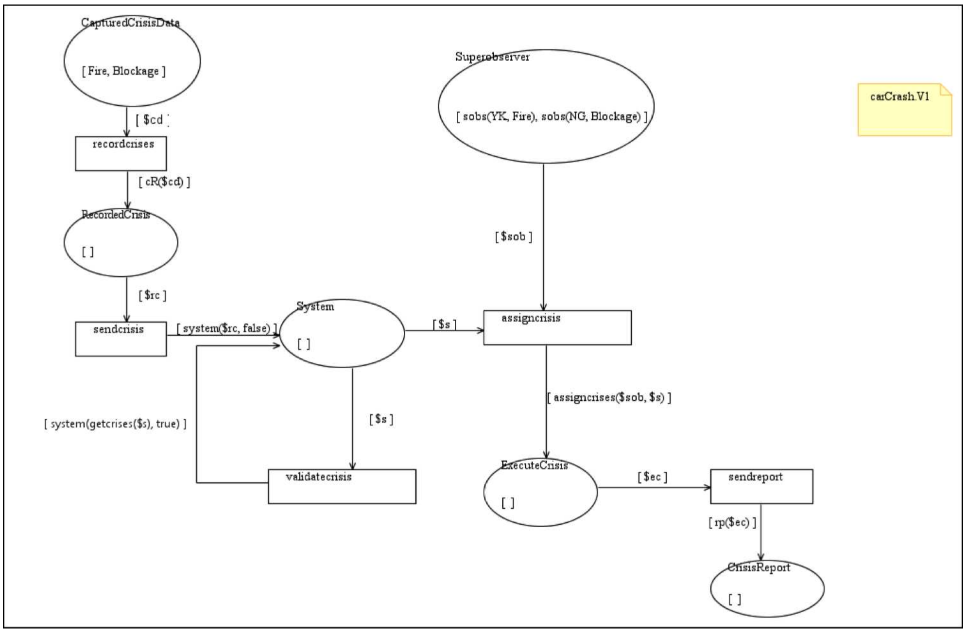

Figure 2: carCrash.V1

3.4 Entities

In order to illustrate a resilient system we will present APN models for three entities representing an evolving system which models of safe Car Crash management system. Formally we have set of entities that are:

Set of Entities

$Ent = \{carCrash.V1, carCrash.V2, carCrash.V3\}$

3.5 APN Model carCrash.V1

We start with the first entity, which we will call carCrash.V1. The APN model can be observed in figure 2 and represents the semantics of the operation of a first version of Car Crash management system. This behavioral model contains labeled places and transitions. In the carCrash.V1 figure 2, in place CapturedCrisisData there can be two tokens of type Fire and Blockage. These tokens are used to mention which type of data has been captured. The output arc of recordcrises contains variable $cd of sort crisesR. Place named RecordedCrises in our model takes this variable as its token with term cR($cd). The transition recordcrises enables to record Crisis based on the capture data. The transition sendcrises takes $rc variable as an input arc from Recordcrises place and the output arc contains term system ($rc,false) of sort sys. Initially every Crisis is set to false with its captured data. The sendcrises transition pass recorded Crisis to system for further operations.

The transition assigncrisis contains two input arcs with $sob & $s variables and the output arc contains term assigncrisis ($sob, $s) of sort crises. The output arc of transition sendreport contains term rp ($ec). This enables to send report about the executed Crisis mission.

Entity	Properties	Sat
carCrash.V1	*validCr*	0
	validSob	0

$$gSat = \frac{\sum_{<p,e>\in dom(sat)} Sat(Prop, Ent)}{|p|}$$

$$gSat_{ccv1} = \frac{0+0}{2} = 0$$

Figure 3: Table Property satisfaction for the carCrash.V1 System

3.6 DREF Satisfiability Model

DREF satisfiability model with respect to Car Crash management system can be formally defined as:

Set of Entities

$Ent = \{carCrash.V1, carCrash.V2, carCrash.V3\}$

Set of Properties

Prop = {*"The Crisis can exist only if it has validated Crisis reporting and Superobserver"*}

In our case entities are logical structures and properties are logical formula. Satisfiablity function would be:

Sat: Lstruct × *Lspec* → ℝ ∪ {⊥}
s.t. Sat(lstruct,p)=
1 if lstruct ⊨ *p*
0 if lstruct ⊭ *p*
⊥ *else*
where
lstruct ∈ *Lstruct* ∧ *p* ∈ *Lspec*

We have formal representation of our first entity in APN as described in section 3.4. We can now express above defined properties in AlPiNA property language for model checking. The syntax to express properties in

AlPiNA is:

Fist Property Syntax:
Prop1 : forall($sy in ExecuteCrises : isvalidcrisis($sy)= true);

Second Property Syntax:
Prop2 : forall($sy in ExecuteCrises : isvalidsobs($sy)=true);

In figure 3, we have shown table for the property satisfaction for the carCrash.V1 and its measure with respect to DREF satisfiability function.

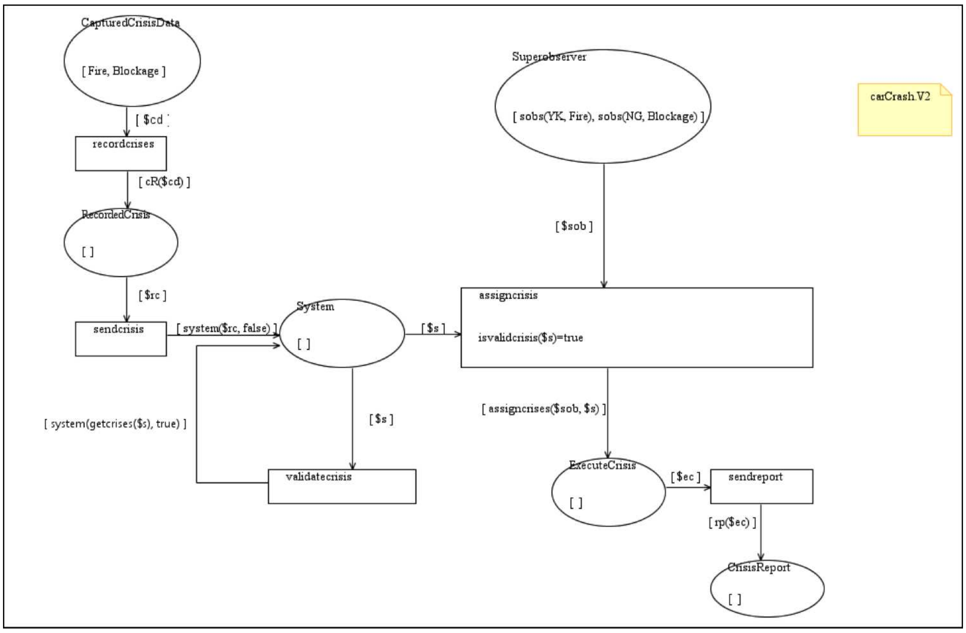

Figure 4: carCrash.V2

3.7 APN Model crashCrash.V2

The second entity which we will call carCrash.V2 is an evolution of the first version carCrash.V2 in figure. In this version, we have added guard condition on the transition assigncrises, which is:

isvalidcrisis($s)=true

In the carCrash.V2 figure 4, in place CapturedCrisisData there can be two tokens of type Fire and Blockage. These tokens are used to mention which type of data has been captured. The output arc of recordcrises contains variable $cd of sort crisesR. Place named RecordedCrises in our model takes this variable as its token with term cR($cd). The transition recordcrises enables to record Crisis based on the capture data. The transition sendcrisis takes $rc variable as an input arc from RecordedCrisis place and the output arc contains term system ($rc, false) of sort sys. The sendcrises transition pass recorded Crisis to system for further operations. Initially every Crisis is set to false with its captured data. The output arc of validatecrises contains system (getcrisis ($s), true) term which sends validated Crisis to system. The transition assigncrises has guard isvalidcrisis ($s)=true which enables to block invalid Crisiss reporting to be executed for the mission. Transition assigncrisis contains two input arcs with $sob & $s variables and the output arc contains term assigncrisis ($sob, $s) of sort crises. The output arc of transition sendreport contains term rp (&ec). This enables to send report about the executed Crisis mission. In figure 5, we have shown table for the property satisfaction for the carCrash.V2 and its measure with respect to DREF satisfiablity function.

3.8 APN ModelcrashCrash.V3

The third entity which we will call carCrash.V3 is an evolution of the version carCrash.V2. In this evolution, we have added another guard to block invalid Superobservers on the transition

Entity	Properties	Sat
carCrash.V2	*validCr*	1
	validSob	0

$$gSat = \frac{\sum_{<p,e>\in dom(sat)} Sat(\Pr op, Ent)}{|\,p\,|}$$

$$gSat_{ccv2} = \frac{1+0}{2} = 0.5$$

Figure 5: Table Property satisfaction for the carCrash.V2 System

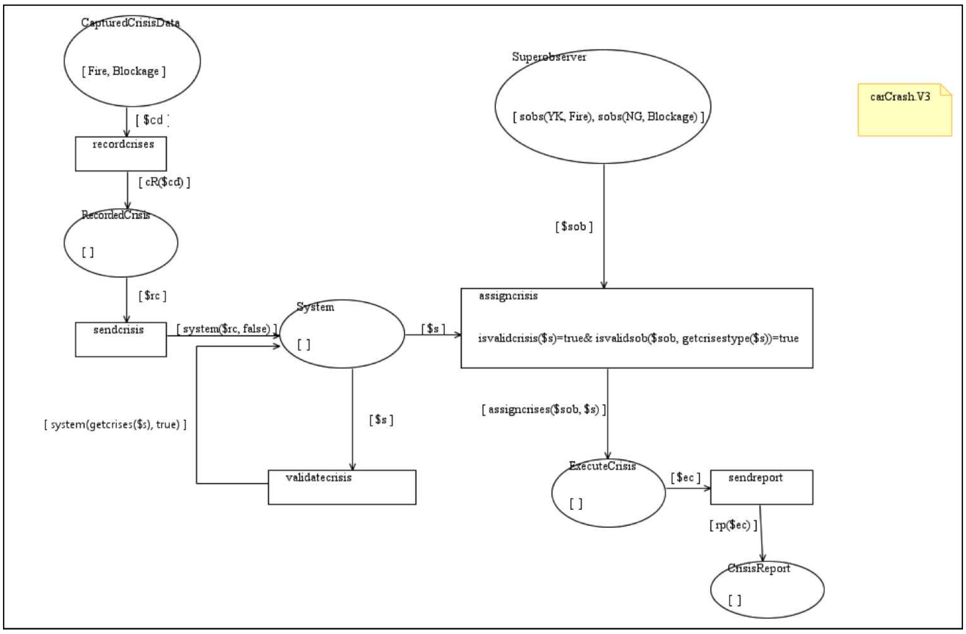

Figure 6: carCrash.V3

assigncrises, which is:

isvalidsob($sob, getcrisestype($s))=true

In the carCrash.V3 figure 6, in place CaptureData there can be two tokens of type Fire and Blockage. These tokens are used to mention which type of data has been captured. The output arc of recordcrises contains variable $cd of sort crisesR. Place named RecordedCrises in our model takes this variable as its token with term cR($cd). The transition record crises enables to record Crisis based on the capture data. The transition sendcrisis takes $rc variable as an input arc from Recordcrises place and the output arc contains term system ($rc, false) of sort sys. The sendcrisis transition pass recorded Crisis to system for further operations. Initially every Crisis is set to false with its captured‘ data. The output arc of validatecrises contains system (getcrisis ($s), true) term which sends validated Crisis to system. The transition assigncrises has two guards; first one is isvalidcrisis ($s)=true which enables to block invalid Crises reporting to be executed for the mission and the second one is isvalidsob($sob, getcrisestype($s))=true which is uded to block invalid Superobservers to execute Crisis mission. In prinicple, if Superobserver is

Entity	Properties	Sat
carCrash.V3	*validCr*	1
	validSob	1

$$gSat = \frac{\sum_{<p,e>\in dom(sat)} Sat(Prop, Ent)}{|p|}$$

$$gSat_{ccv3} = \frac{1+1}{2} = 1$$

Figure 7: Table Property satisfaction for the carCrash.V3 System

\$YK then it is mandatory to assign it to Fire. Transition assigncrisis contains two input arcs with \$sob & \$s variables and the output arc contains term assigncrises (\$sob, \$s) of sort crises. The output arc of transition sendreport contains term rp (\$ec). This enables to send report about the executed Crisis mission.
In figure 7, we have shown table for the property satisfaction for the carCrash.V3 and its measure with respect to DREF satisfiablity function.

4 Evolution Satisfiability Function

The intention is to allow for comparison of entities or properties relative to an evolution axis. Concerning ICT systems, the commonly used axes are the time axis (that can be considered as discrete or continuous) related to systems versioning or related to system status.
If we consider the Car Crash management framework (entity carCrash), we have three versions of the framework. While the first is that described in 3.4, in the second version described in 3.7, we have introduced some transitions and guard conditions. In the final version described in 3.8, we have added only a guard condition.
Now we tend to introduce an evolution axis representing the three successive versions of the carCrash framework. The evolution axis is then the set $\{carCrash.V1, carCrash.V2, carCrash.V3\}$ and it concerns the entity carCrash. Here as shown in Graph 1 evolution function is with respect to overall properties. From the Graph 1, it is clear that system under observation is improving its capabilities with successive versions.
Here in Graph 2, evolutions have been shown with respect to individual properties. For the property validCr there is an improvement in the carCrash.V2 and it is preserved in the carCrash.v3. There is no improvement for the validSob property in the first two versions but in carCrash.V3 it shows an improvement.

5 Discussion

In this work we have presented a pragmatic way to evaluate satisfiability of resilient system with respect to certain decidable properties. In order to make a system resilient, one has to measure its satisfactiability at its current state with respect to interesting properties and then based on this evaluation modification can be done to make it resilient. What we mean from evaluation is to check whether system satisfy interesting properties or not. One of the possible ways to

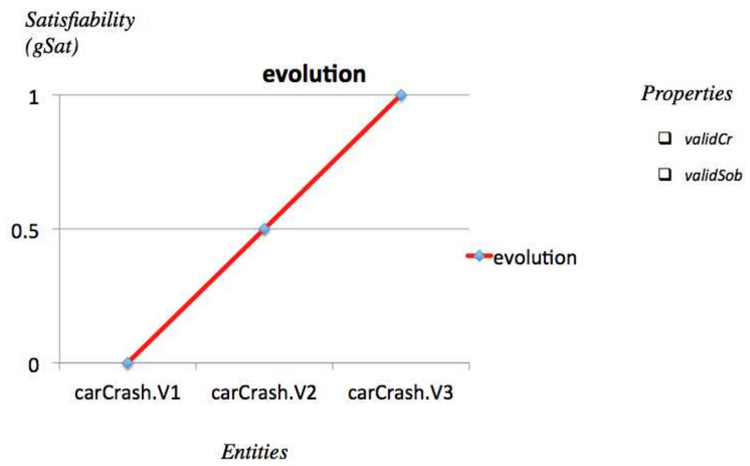

Figure 8: Graph Evolution Satisfiability Function with respect to overall Properties

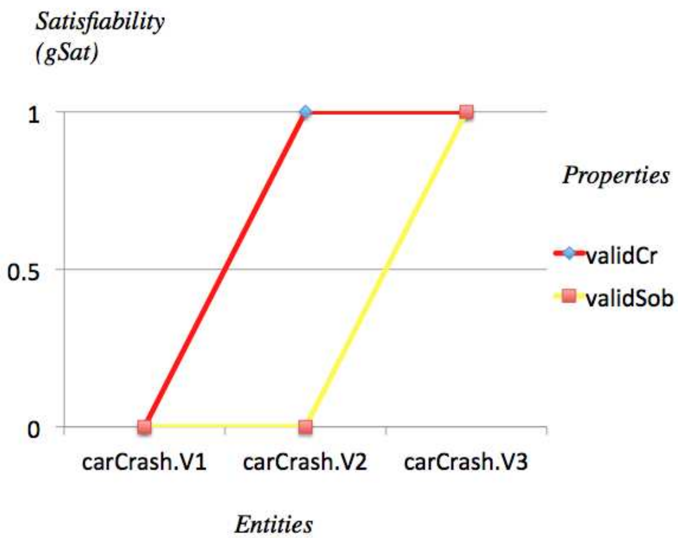

Figure 9: Graph Evolution Satisfiability Function with respect to individual Properties

evaluate is by model checking. Now based on the results, which we obtain from model checking, modification is performed. If a system is unable to satisfy interesting property one has to modify it by putting or removing some features, assertions or guard conditions etc. In principal this process will be iterated until we get our interesting properties satisfied.
This proposal raises many questions. We will start by discussing the question that it is undesirable to check repeatedly interesting properties, which have been verified in earlier versions of system. In order to overcome this situation there must be property preservation mechanism. There is work regarding this property preservation issue, for example in [Guelev, 2004] the authors described a method for checking whether a system with a feature continues to satisfy a property that held of a base system. On the other way, given a discrete evolution (ent_k, ent_{k+1}), we can impose a particular kind of structural inclusion of ent_k into ent_{k+1}. This structural inclusion ensures that the behavior of ent_{k+1} regarding safety properties expressed over ent_k is included in the behavior of ent_k.
Another important question arises is that what would be an appropriate modification or refinement to an evolving system so that it satisfies properties. This issue can be interlinked with the property preservation mechanism. Because it is important to know before modification that what you will gain and what you will loose in terms of properties.
Finally, the research we present in this work is oriented at: pragmatically defining the evaluation of satisfiability of resilient system with respect to certain decidable properties. We have selected APN (Algebraic Petri Nets) for the behavioral description of our system. For the evaluation of interesting properties, we use AlPiNA model checker. AlPiNA property language is equivalent to first order logic. With respect to modification of given discrete evolution (ent_k, ent_{k+1}) we used guard conditions, new places to net etc. Complete description is provided in sections 3.2 to 3.8.

Acknowledgments

The author would like to thank Prof. Nicolas Guelfi, Laboratory for Advance Software Systems, University of Luxembourg, for his support and guidance.
The author would like to thank Dr. Matteo Risoldi , Laboratory for Advance Software Systems, University of Luxembourg, for his kind assistance and motivation.
This work was sponsored by the FNR CORE Project MOVERE, ref.C09/IS/02

References

[Guelfi, 2011] Guelfi N., (2011). "A Formal Framework for Dependability and Resilience from a Software Engineering Persepective". Cent.Eur.J.Comp.Sci ISBN: 1(3) 2011 294-328.

[Lucio et al., 2011] Lcio L., Guelfi N., "A precise definition of operational resilience". Tech. Rep. TR-LASSY-11-02, Laboratory for Advanced Software Systems, University of Luxembourg, 2011.

[Komatsubara, 2006] Komatsubara A., (2006). "When resilience does not work. Resilience Engineering Concepts and Precepts. Ashgate.

[Hollnagel et al., 20006] Hollnagel L., Woods D., Leveson N.,(2006). "Resilience Engineering in Nutshel". Resilience Engineering Concepts and Precepts. Ashgate.

[Avizienis et al., 2001] Avizienis A., Laprie J.C., Randell B., (2001). "Fundamental concepts of dependability". Tech. rep., Computer Science Department, University of California, Los Angeles, USA, 2001.

[Avizienis et al., 2004] Avizienis A., Laprie J.C., Randell B., Landwehr C.E., (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing". IEEE Trans. Dependable Sec. Comput., 2004, 1(1), 11-33.

[Black et al., 1997] Black P.E., Windley P.J., (1997). "Verifying resilient software". 1997, 262-266.

[Mostert et al., 1995] Mostert D.N.J., von Solms S.H., (1995)."A technique to include computer security, safety, and resilience requirements as part of the requirements specification". J. Syst. Softw., 1995, 31(1), 45-53.

[Svobodova, 1984] Svobodova L., (1984). "Resilient distributed computing". IEEE Trans. Software Eng., 1984, 10(3), 257-268.

[Buchs et al., 2010] Buchs D., Hostettler S., Marechal A., Risoldi M., (2010). "Alpina: A symbolic model checker". In Petri Nets, volume 6128 of Lecture Notes in Computer Science. Springer, 2010.

[Marechal et al., 2010] Marechal A., Bucs D., (2010)). "Property specification language for algebraic Petri nets". Technical report # 216 http://alpina.unige.ch.

[Reisig, 1991] Reisig W., (1991). "Petri nets and algebraic specifications". Theoretical Computer Science, 80:134, 1991.

[Lucio, 2011] Lucio L., (2011). "Model of an evolving confidential filesystem"1. http://hera.uni.lu/levi.lucio/operational resilience/evolving filesystem.zip.

[SMV, 2010] Smv Group., (2010). "Alpina model checker". http://alpina.unige.ch.2010.

[Lepri, 2008] Laprie J.C., (2008). "From dependability to resilience ". In Proceedings of the IEEE/I- FIP International Conference on Dependable Systems and Networks, DSN - Fast Abstracts. IEEE/IFIP, 2008.

[Wirsing, 1990] Wirsing M., (1990) "Algebraic specification ". In: Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics B, 675-788, 1990.

[Guelev, 2004] Guelev P D., Ryan M., Schobbens P., (2004). "Model-checking the Preservation of Temporal Properties upon Feature Integration ".In Proceedings of the Fouth International Workshop on Automated Verification of Critical Systems (CONCUR Workshop AVoCS 2004), pages 311-324.

[Kupferman et al., 1999] Kupferman B., Vardi Y M.m (1999) "Model checking of safety properties ". Computer Aided Verification, Proc. 11th Int. Conference. Lecture Notes in Computer Science Springer-Verlag, 172183.

[Sommerville, 2001] Sommerville I., (2001). "Software Engineering, 6th Edition ". Pearson Education Ltd.

[Kienzle et al., 2009] Kienzle J., Guelfi N., Mustafiz S., (2009) " Crisis Management Systems A Case Study for Aspect-Oriented Modeling ". SOCS-TR-2009.3

[Dearnley, 1976] earnley P. A., An Investigation Into Database Resilience(1976). Comput. J., 19(2), pp. 117-121, 1976.

6 Appendix

In this appendix we present algebraic specifications for the carCrash case study.

6.1 Algebraic specifications for carCrash.V1

```
Adt boolean
        Sorts bool;
        Generators
                true : bool;
                false : bool;
        Operations
                not : bool -> bool;
                and : bool, bool -> bool;
                or : bool, bool -> bool;
                xor : bool, bool -> bool;
                implies : bool, bool -> bool;

        Axioms
                //not
                not(true) = false;
                not(false) = true;

                //and
                and(true, $boolVar) = $boolVar;
                and(false, $boolVar) = false;

                //or
                or(true, $boolVar) = true;
                or(false, $boolVar) = $boolVar;

                //xor
                xor(true, $boolVar) = not($boolVar);
                xor(false, $boolVar) = $boolVar;

                //implies
                implies(false, $boolVar) = true;
                implies(true, $boolVar) = $boolVar;

        Variables
                boolVar : bool;
```

```
Adt capturedata

        Sorts
                capture;

        Generators

                Fire:capture;
                Blockage:capture;
```

```
Adt observers
        Sorts
                obs;

        Generators
                YK:obs;
                NG:obs;
```

```
import"observers.adt"
import"capturedata.adt"
Adt SuperObserver
```

```
        Sorts
                Sobs;
        Generators

                sobs:  obs,capture->Sobs;
        Operations

                getsob:Sobs->obs;
        Axioms

                getsob(sobs($obs,$ct))=$obs;


        Variables
                obs:obs;
                ct:capture;
```

```
import "capturedata.adt"

Adt RecordCrises

    Sorts
        crisesR;

    Generators
        cR:     capture->crisesR;

    Operations

        getcapturetype:crisesR->capture;

    Axioms
        getcapturetype(cR($cd))=  $cd;
    Variables
        cd:capture;
```

```
import"boolean.adt"
import"RecordCrises.adt"
import"capturedata.adt"
    Adt
        system
        Sorts
                sys;
        Generators
                system:crisesR,bool->sys;
        Operations
                getcrisestype:sys->capture;
                getcrises:sys->crisesR;
        Axioms
                getcrises(system($cr,$b))=  $cr;
                getcrisestype(system($cr,$b))=  getcapturetype($cr);

        Variables
                cr:crisesR;
                b:bool;
```

```
import "RecordCrises.adt"
import"system.adt"
import"SuperObserver.adt"
import"capturedata.adt"



    Adt Executecrises

        Sorts
                crises;

        Generators
```

```
            assigncrises:Sobs,sys->crises;
        Operations
            getcrisestype:crises->capture;

        Axioms
            getcrisestype(assigncrises($sob,$sy))= getcrisestype($sy);

        Variables
            cr:crisesR;
            sob:Sobs;
            sy:sys;
```

```
import "Executecrises.adt"
import "boolean.adt"

    Adt Report

        Sorts
            report;

        Generators
            rp: crises->report;
```

6.2 Algebraic specifications for carCrash.V2 & carCrash.V3

Here we present algebraic specification for the two entities which are carCrash.V2 and carCrash.V3. We are presenting only those specifications, which are modified as compared to previous version.

```
import"boolean.adt"
Adt capturedata

    Sorts
        capture;

    Generators

        Fire:capture;
        Blockage:capture;

    Operations
        equal:capture,capture->bool;

    Axioms
        equal(Fire,Fire)=true;
        equal(Blockage,Blockage)=true;
        equal(Fire,Blockage)=false;
        equal(Blockage,Fire)=false;
```

```
import"boolean.adt"
import"observers.adt"
import"capturedata.adt"
Adt SuperObserver

        Sorts
            Sobs;
        Generators

            sobs: obs,capture->Sobs;
        Operations
            isvalidsob: Sobs,capture->bool;
            getsob:Sobs->obs;
        Axioms
            isvalidsob(sobs($obs,$ct),$ct2)= equal($ct,$ct2);
            getsob(sobs($obs,$ct))=$obs;
```

```
        Variables
                obs:obs;
                b:bool;
                ct:capture;
                ct2:capture;
```

```
import"boolean.adt"
import"RecordCrises.adt"
import"capturedata.adt"
    Adt
        system
        Sorts
                sys;
        Generators
                system:crisesR,bool->sys;
        Operations
                isvalidcrisis:sys->bool;
                getcrisestype:sys->capture;
                getcrises:sys->crisesR;
        Axioms
                isvalidcrisis(system($cr,false))= false;
                isvalidcrisis(system($cr,true))= true;
                getcrises(system($cr,$b))= $cr;
                getcrisestype(system($cr,$b))= getcapturetype($cr);

        Variables
                cr:crisesR;
                b:bool;
```

```
import "RecordCrises.adt"
import"system.adt"
import"SuperObserver.adt"
import"boolean.adt"
import"capturedata.adt"


    Adt Executecrises

        Sorts
                crises;

        Generators
                assigncrises:Sobs,sys->crises;
        Operations
                getcrisestype:crises->capture;
                isvalidcrisis:crises->bool;
        Axioms
                getcrisestype(assigncrises($sob,$sy))= getcrisestype($sy);
                isvalidcrisis(assigncrises($sob,$sy))= isvalidcrisis($sy);
        Variables
                cr:crisesR;
                sob:Sobs;
                sy:sys;
```

6.3 Algebraic specifications for Properties

In this section we are presenting algebraic specifications for the interesting properties.

```
import"carcrash.apnmm"
import"boolean.adt"
import"Handlecrises.adt"
import"system.adt"

Expressions
    Prop1: forall ($sy in ExecuteCrisis : isvalidcrisis($sy)= true );
```

```
        Check
        @Prop1 ;


Variables
        sy:crises;
```

```
import"carcrash.apnmm"
import"boolean.adt"
import"Handlecrises.adt"
import"system.adt"

Expressions
        Prop2: forall ($sy in ExecuteCrisis : isvalidsobs($sy)= true );

        Check
        @Prop2 ;


Variables
        sy:crises;
```