An observation metamodel for dependability tools

Laura Carnevali
University of Florence
Italy

Stefania Cerboni
University of Florence
Italy

Benedetta Picano
University of Florence
Italy

Leonardo Scommegna
University of Florence
Italy

Enrico Vicario
University of Florence
Italy

***Abstract*—FaultFlow is a library for modeling and evaluation of dependability of component-based systems. It represents duration to occurrence and propagation of faults across the hierarchy of components through non-Markovian distributions, facilitating fitting of observed data and design assumptions. Additionally, FaultFlow can be extended to simulate the system behavior and generate synthetic time series encoding occurrences of faults and failures and results of diagnostic tests. Time series can in turn be employed to train and test data-driven methods aimed at various tasks, notably failure prediction. As a first step in this direction, we define a flexible and extensible observation metamodel for FaultFlow, representing type and time of observations of the system behavior, and facilitating definition of monitoring policies.**

***Index Terms*—Component-based system, timestamped observations, Model-Driven Engineering, software tools and libraries.**

I. INTRODUCTION

Motivation. In the design of component-based systems [4], quantitative models of the system failure logic [21] represent propagation of faults and failures across the hierarchy of components and support evaluation of dependability attributes [3]. Quantitative models of dependability can be used to simulate the system behavior and generate time series encoding occurrences of faults and failures and results of diagnostic tests. In turn, time series can be exploited to train and test data-driven methods with various aims, notably failure prediction [30]. To this end, it is necessary that dependability models support representation of observations of the system behavior.

Related works. Various tools support specification and evaluation of dependability models by leveraging Model-Driven Engineering (MDE) [9], [32], notably Model-to-Model (M2M) transformations, with differences in the model expressivity and in the scope of solution methods. Specifically, FaultFlow [26] is an open-source library [12] with custom metamodel, encoding fault propagations having non-Markovian (expolynomial [37]) duration distribution, between directly and indirectly connected components (*direct* and *indirect couplings*, respectively). FautFlow models repeated events in the failure logic but not repairs. Its metamodel instances can be derived from SysML Block Definition Diagrams (BDDs) [24] and Stochastic Fault Trees (SFTs), and translated into Stochastic Time Petri Nets (STPNs), deriving the time-to-failure distribution through semi-symbolic analysis based on stochastic state classes [17]. FaultFlow metamodel instances can be translated also into an extension of UML statecharts [16] if there are no repeated events, deriving fault importance measures by efficient numerical analysis [6]. CHESS [5], [8], [22] is a partially open-source tool based on the custom language CHESS-ML, containing tailored subsets of SysML and the UML profile for Modeling and Analysis of Real-Time Embedded systems (MARTE) [23], representing selected non-Markovian duration distributions, both direct and indirect couplings, repeated events, and repairs. CHESS implements translation first to the Intermediate Dependability Model (IDM) [22] and then to Continuous Time Markov Chains (CTMCs) and Stochastic Petri Nets (SPNs) [7], exploiting fault tree analysis and simulation to compute reliability and availability measures. OSATE [11], [13] is an open-source tool mainly exploiting the Architecture Analysis & Design Language (AADL) [14] Annexes on Error Model [29] and Safety [36], supporting representation of Exponential duration distributions and repair activities while not modeling indirect couplings and repeated events. OSATE implements translation into Discrete Time Markov Chain (DTMCs) and CTMCs analyzed by PRISM [20], exploiting numerical analysis for computation of failure probabilities. Although no longer supported, ADAPT [27] implemented derivation of Generalized Stochastic Petri Nets (GSPNs) [1] from AADL models annotated with dependability attributes, and ASTRO [33] supported dependability modeling and analysis by exploiting Reliability Block Diagrams (RBDs), CTMCs, and SPNs. Other general-purpose tools for quantitative evaluation of stochastic models could be used for dependability modeling and analysis, though requiring users to have significant modeling expertise, e.g., SHARPE [37], ORIS [25], GreatSPN [2], TimeNET [40], PRISM [19], Möbius [10], CPN IDE [18], [38], Mercury [34].

As a common trait, none of these tools supports in any way the representation of observations of the system behavior.

Contribution. We define a flexible and extensible observation metamodel for FaultFlow, using the Reflection architectural pattern [31] and the Observation & Measurement analysis pattern [15] to facilitate customization of monitoring policies. Translation of observation metamodel instances into STPNs is defined, generating an STPN submodel where each transition represents the acquisition of an observation on some component, and has an entry-point method returning the observation value based on the state of the observed component. A technical report illustrating application of the approach to a case study is available at https://doi.org/10.5281/zenodo.10473653.

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

In the rest of the paper, we extend FaultFlow with an observation metamodel (Section II), discuss threats to validity [28] (Section III), and draw our conclusions (Section IV).

II. Observation metamodel

A. Extended FaultFlow metamodel

Fig. 1 shows the FaultFlow metamodel, representing both the *structure* of a system (classes with blue scale background) and its *failure logic* (classes with orange scale background). The metamodel implementation leverages the Reflection architectural pattern [31]. Thus, classes at the *knowledge* (abstract) level encode *types* (e.g., of component, fault, failure, etc.), while classes at the *operational* (concrete) level represent *instances* (e.g., of component, fault, failure, etc.). A detailed description of the FaultFlow metamodel is reported in [26].

We extend FaultFlow with an observation metamodel, by exploiting the Observation & Measurement analysis pattern [15], still leveraging the Reflection architectural pattern. The added classes are highlighted with yellow scale background in the metamodel shown in Fig. 1. On the one hand, at the knowledge level, we introduce the concept of observation type (represented by class `ObservationType`) which can be acquired on a type of component through some type of observation port (class `ObservationPortType`). On the other hand, at the operational level, each scenario (class `Scenario`) is associated not only with a set of components (class `Component`) but also with a set of observations (class `Observation`), and each component is associated with a set of observation ports (class `ObservationPort`). In particular, an observation is acquired on a component through an observation port based on the component state (class `State`). In turn, the component state consists of a set of timestamped events (class `Event`), each being a fault (class `Fault`), or an error (class `Error`), or a failure (class `Failure`). In doing so, the metamodel supports representation of a variety of observation policies, e.g., memoryless policies where observations depend only on the last event occurred to a component, and memoryfull policies where observations depend on the whole event history.

An observation acquired on a type of phenomenon (class `PhenomenonType`), e.g., temperature, according to a protocol (class `Protocol`) can be either a qualitative observation (class `CategoryObservation`) whose value is a category (class `Category`), e.g., high temperature, or a quantitative observation (class `Measurement`) whose value is a quantity (class `Quantity`) referred to a unit measure (class `Unit`), e.g., 20 degree celsius temperature. Moreover, an observation is associated with timing information (class `TimeRecord`) characterizing the time instant (class `TimePoint`) or the time interval (class `TimePeriod`) in which the observation is acquired (attribute `recordingTime` of class `Observation`) and is valid (attribute `validity` of class `Observation`).

It is worth noting that, thanks to the Reflection architectural pattern, *classes* at both knowledge and operational levels are independent of domain-specific concepts (e.g., specific types of components, specific decomposition of a component of a specific type into sub-components), which are likely to change, in principle even frequently. Therefore, these classes turn out to be few (i.e., as shown in Fig. 1), they are implemented by ICT experts, and they are expected to change or increase in number only if the conceptual elements need to be changed or extended (e.g., if repair policies need to be represented). Conversely, specific types (e.g., laptop type X1, fault type F1 of laptop type X1) are implemented by domain experts as *object*s of classes at the knowledge level, and specific instances of specific types (e.g., laptop of type X1 with serial number 1234, fault of type F1 occurred at time T to laptop of type X1 with serial number 1234) are implemented by domain professionals as *objects* of classes at the operational level. Thus, definition of structure, failure logic, and observation policies of a system amounts to the implementation of objects and not classes, facilitating usage, maintenance, and extendibility.

B. Extended FaultFlow-to-Sirio transformation

FaultFlow metamodel instances can be translated into Sirio metamodel instances [35] representing STPNs [39]. STPNs consist of transitions (depicted as bars), tokens within places (dots in circles), and directed arcs (directed arrows) from input places to transitions and from transitions to output places. Transitions model stochastic durations of activities; directed arcs model precedence relations among activities; and, tokens model the system discrete logical state. A transition is enabled by a marking (i.e., assignment of tokens to places) if each of its input places contains at least one token and its enabling function evaluates to true. Upon enabling, each transition samples a time-to-fire from its distribution, which can be Exponential or non-Exponential. In the latter case, if the transition time-to-fire takes a deterministic value τ (i.e., it follows the generalized distribution of a Dirac Delta function), the transition is termed deterministic (DET) if $\tau \neq 0$ and immediate (IMM) if $\tau = 0$. The transition with minimum time-to-fire fires, removing one token from each of its input places and adding one token to each of its output places. Ties among transitions with equal time-to-fire are solved by a random switch determined by probabilistic weights of transitions.

FaultFlow-to-Sirio transformation [26] yields an STPN of the failure logic. Specifically, each internal fault (modeled by class `InternalFaultMode` in Fig. 1) is translated into a transition modeling the time-to-fault duration (transition `internalFaultOccurrence` in Fig. 2(a)), with input place containing one token and modeling activation of the fault process, and output place modeling the fault occurrence. Each fault-to-failure propagation (modeled by classes `ErrorMode` and `FailureMode` in Fig. 1) is translated into a transition modeling the fault-to-failure duration (transition `faultToFailure` in Fig. 2(b)), with enabling function being true if the condition on activation of faults is true, input place containing one token and modeling activation of the fault-to-failure process, and output place modeling failure occurrence. Each failure-to-fault propagation (modeled by classes `ExternalFaultMode` and `PropagationPort-Type` in Fig. 1) is translated into an IMM transition (transition `failureToExternalFault` in Fig. 2(b)), with input place

Fig. 1. Extended FaultFlow metamodel: blue scale background highlights classes modeling the system structure, orange scale background highlights classes modeling the system failure logic, and yellow scale background highlights classes modeling the system observation policies.

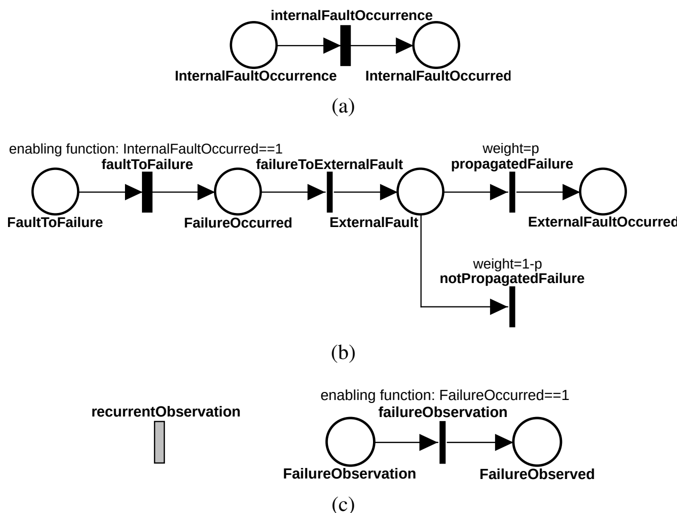

Fig. 2. STPN fragments modeling: (a) occurrence of internal faults; (b) fault-to-failure and failure-to-fault propagations; and, (c) observation processes.

modeling failure occurrence, and output place being the input place of two IMM transitions with weight p and $1 - p$, modeling fault propagation and non-propagation, respectively (transitions `propagatedFailure` and `notPropagated-Failure` in Fig. 2(b), respectively).

We extend the transformation to represent observation policies. Each observation process (classes `ObservationType` and `ObservationPortType` in Fig. 1) is translated into a transition modeling observation acquisition, either recurrent acquisition (transition `recurrentObservation` in Fig. 2(c)) or one-shot acquisition upon event occurrence (transition `failure-Observation`). The acquisition pattern of each observation port is not directly represented in the metamodel, and can be easily implemented by domain experts through the time-to-fire PDF of the corresponding transition.

To collect observations during STPN simulation, each transition in the observation submodel is associated with: a new feature termed `ComponentFea-ture` identifying the observed component; and, a new feature termed `Observation-Feature` with entry-point method (implemented by domain experts) returning the observation value based on the state of the observed component. Upon each transition firing in the STPN observation submodel, the entry-point method of the fired transition is executed, and the returned value is stored as

an `Observation` instance in the metamodel instance.

III. Threats to validity

Construct validity. Feasibility of the approach is shown by its application to an example (see the technical report available at https://doi.org/10.5281/zenodo.10473653), manually modeling observation policies in metamodel instances and translating them to STPNs, which may be costly for non-experts. To mitigate this issue during approach implementation, metamodel-to-STPN transformation is formalized so as to be unambiguous.

Internal validity. The approach requires specification of stochastic parameters, e.g., time-to-fault PDF. To mitigate this threat, in the example, these parameters are derived from a database collecting historical data on component failures.

External validity. Application to domains other than the one considered in the example may be not completely easy. To mitigate this threat, the FaultFlow metamodel represents common system features, independent of application domains.

IV. Conclusions

This paper devises a methodology to define a flexible and extensible observation metamodel for dependability evaluation tools. Future implementation will enable simulation of STPN models representing failure logic and monitoring policies of component-based systems, and generation of synthetic time series of observations of the system behavior, making FaultFlow an agile workbench to train and test data-driven methods with various purposes, e.g., failure prediction. Future work also includes representation of new concepts in the metamodel, e.g., sensitivity and specificity of diagnostic tests.

References

[1] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. on Computer Systems*, 2(2):93–122, 1984.

[2] E. G. Amparore, G. Balbo, M. Beccuti, S. Donatelli, and G. Franceschinis. 30 years of GreatSPN. *Principles of performance and reliability modeling and evaluation*, pages 227–254, 2016.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] J. Boardman and B. Sauser. System of systems-the meaning of of. In *IEEE/SMC Int Conf. on SoS Eng.*, pages 6–pp. IEEE, 2006.

[5] L. Bressan, A. L. de Oliveira, L. Montecchi, and B. Gallina. A systematic process for applying the CHESS methodology in the creation of certifiable evidence. In *EDCC*, pages 49–56. IEEE, 2018.

[6] L. Carnevali, R. German, F. Santoni, and E. Vicario. Compositional Analysis of Hierarchical UML Statecharts. *IEEE Trans. on Soft. Eng.*, 48(12):4762–4788, 2021.

[7] H. Choi, V. G. Kulkarni, and K. S. Trivedi. Markov regenerative stochastic petri nets. *Performance evaluation*, 20(1-3):337–357, 1994.

[8] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *Int. Conf. Aut. Soft. Eng.*, pages 362–365, 2012.

[9] A. R. Da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, 2015.

[10] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Mobius framework and its implementation. *IEEE Tran. Soft. Eng.*, 28(10):956–969, 2002.

[11] J. Delange, P. Feiler, D. P. Gluch, and J. Hudak. AADL fault modeling and analysis within an ARP4761 safety assessment. Technical report, Carnegie-Mellon Univ. Soft. Eng. Inst., 2014.

[12] FaultFlow library. https://github.com/oris-tool/faultflow, 2023.

[13] P. Feiler. The open source AADL tool environment (OSATE). Technical report, Carnegie Mellon Univ. Soft. Eng. Inst., 2019.

[14] P. H. Feiler, B. A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. In *CACSD-CCA-ISIC*, pages 1206–1211. IEEE, 2006.

[15] M. Fowler. *Analysis patterns: reusable object models*. Addison-Wesley Professional, 1997.

[16] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[17] A. Horváth, M. Paolieri, L. Ridi, and E. Vicario. Transient analysis of non-Markovian models using stochastic state classes. *Perform. Eval.*, 69(7-8):315–335, July 2012.

[18] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. Journal on Soft. Tools for Technology Transfer*, 9:213–254, 2007.

[19] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Int. Conf. on Modelling Techniques and Tools for Computer Perf. Eval.*, pages 200–204. Springer, 2002.

[20] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[21] O. Lisagor. *Failure logic modelling: a pragmatic approach*. PhD thesis, University of York, 2010.

[22] L. Montecchi, P. Lollini, and A. Bondavalli. Towards a mde transformation workflow for dependability analysis. In *IEEE Int. Conf. on Eng. of Complex Comp. Sys.*, pages 157–166. IEEE, 2011.

[23] Object Management Group. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems v1.0*, 2009.

[24] Object Management Group, www.omg.org/spec/SysML/1.6/. *OMG Systems Modeling Language 1.6 (OMG SysML)*, 2019.

[25] M. Paolieri, M. Biagi, L. Carnevali, and E. Vicario. The ORIS Tool: Quantitative Evaluation of Non-Markovian Systems. *IEEE Transactions on Software Engineering*, 47(6):1211–1225, 2021.

[26] J. Parri, S. Sampietro, and E. Vicario. Faultflow: a tool supporting an mde approach for timed failure logic analysis. In *European Dependable Computing Conference*, pages 25–32. IEEE, 2021.

[27] A.-E. Rugina, K. Kanoun, and M. Kaâniche. The ADAPT tool: From AADL architectural models to stochastic petri nets through model transformation. In *EDCC*, pages 85–90. IEEE, 2008.

[28] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, 2009.

[29] SAE International. *AADL Error Model Annex, (Standards Document AS5506/1*, 2006.

[30] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Computing Surveys*, 42(3):1–42, 2010.

[31] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-oriented software architecture, volume 1: a system of patterns, 1996.

[32] D. C. Schmidt. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.

[33] B. Silva, G. Callou, E. Tavares, P. Maciel, J. Figueiredo, E. Sousa, C. Araujo, F. Magnani, and F. Neves. Astro: An integrated environment for dependability and sustainability evaluation. *Sustainable computing: informatics and systems*, 3(1):1–17, 2013.

[34] B. Silva, R. Matos, G. Callou, J. Figueiredo, D. Oliveira, J. Ferreira, J. Dantas, A. Lobo, V. Alves, and P. Maciel. Mercury: An integrated environment for performance and dependability evaluation of general systems. In *IEEE/IFIP Int. Conf. Depend. Sys. and Net.*, 2015.

[35] SIRIO Library. https://github.com/oris-tool/sirio, 2022.

[36] D. Stewart, J. J. Liu, M. Heimdahl, D. Cofer, and M. Peterson. Safety annex for the architecture analysis and design language. 2018.

[37] K. S. Trivedi and R. Sahner. Sharpe at the age of twenty two. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):52–57, 2009.

[38] E. Verbeek and D. Fahland. CPN IDE: An Extensible Replacement for CPN Tools That Uses Access/CPN. In *Int. Conf. on Process Mining Doctoral Consortium and Demo Track*, pages 29–30, 2021.

[39] E. Vicario, L. Sassoli, and L. Carnevali. Using stochastic state classes in quantitative evaluation of dense-time reactive systems. *IEEE Trans. on Software Engineering*, 35(5):703–719, 2009.

[40] A. Zimmermann. Modelling and performance evaluation with TimeNET 4.4. In *Int. Conf. on Quant. Eval. of Sys.*, pages 300–303. Springer, 2017.