Towards Formal Design of FDIR Components with AI*

Marco Bozzano Alessandro Cimatti Marco Cristoforetti Alberto Griggio
Piergiorgio Svaizer Stefano Tonetta

Fondazione Bruno Kessler
{bozzano,cimatti,mcristofo,griggio,svaizer,tonettas}@fbk.eu

The development of accurate, reliable and effective FDIR (Fault Detection, Identification and Recovery) components is essential in several application domains, to meet the dependability constraints and to accomplish the higher degree of autonomy required in future missions.

In this work, we report on an ongoing activity that addresses the formal design, development and validation of FDIR integrating rule-based components with components based on Machine Learning (ML) and Deep Learning (DL). We show that the integration of symbolic and AI techniques can substantially improve the effectiveness and efficiency of FDIR management functions, while formal tool-supported verification and validation can provide a formal guarantee of the quality of the FDIR systems before they are implemented and deployed. This activity is being carried out within the AIFDIR study, funded by the Italian Space Agency (ASI) under the "Innovative Space Technologies" initiative.

The AIFDIR methodology will be implemented and demonstrated using TASTE, a tool developed by the European Space Agency (ESA), which follows the MBSE (Model-Based System Engineering) approach. TASTE has been recently extended to enable the modeling of HW components and their possible failures, and the verification and validation using automated techniques based on model checking. TASTE will be further extended to allow for modeling and verification of systems including both symbolic and ML/DL-based components, and to support the deployment on the target hardware. A further contribution of the project is the development of a reference architecture for AIFDIR and its demonstration on case studies of interest.

1 Introduction

The operation of complex critical systems relies on the capability to detect and tolerate runtime faults. This is more and more the case for future systems, which are required to have an increasingly higher level of autonomy and reliability, and must be fail-operational in order to accomplish the objectives of their mission. The Fault Detection, Identification and Recovery (FDIR) modules are responsible to timely detect the occurrence of faults, localize them, contain their possible propagation to other parts of the system, and take appropriate measures, e.g., system re-configuration, to avoid system failures.

*The AIFDIR study is funded by ASI under contract n. 2024-14-I.0, CUP F63C24000110001, CIG B05506C97C.

This study was carried out within the Interconnected Nord-Est Innovation Ecosystem (iNEST) and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) - MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.5 - D.D. 1058 23/06/2022, ECS00000043). This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

We acknowledge the support of the MUR PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU.

Furthermore, we acknowledge the support of the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

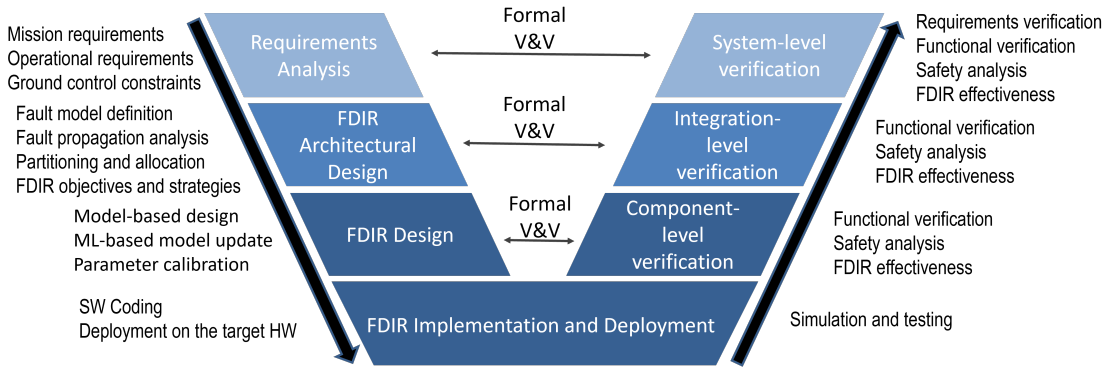


Figure 1: FDIR formal development.

Fault detection and identification (FDI) components usually run in closed loop with the system, they take observations generated by the system (e.g., by sensors) and use this information to infer the occurrence of anomalies and identify the source of the problem (e.g., a component fault). In response, they may generate a set of alarms, which in turn may trigger reconfiguration actions.

FDIR components are typically arranged based on the system architecture and possibly categorized in several levels, depending on the hierarchy of the sub-systems, the level of criticality of the potential fault effects, the level of automation or the characteristics of the recovery strategies (e.g., local recovery versus system-level recovery strategy).

Existing methodologies for designing FDIR sub-systems often leverage past experience of domain experts and are based on adaptations and extensions of existing designs. However, the increasing complexity of systems, both at SW and HW level, calls for more formal *model-based* methodologies to develop FDIR sub-systems, verify and validate their design before they are implemented, and thoroughly test them before they are deployed [12, 3], as illustrated in Fig 1. A notable design environment in the space domain is TASTE [17, 24], a model-based design and development environment for embedded, real-time systems, which has been actively developed by ESA since 2008. It comprises several tools such as graphical editors, visualizers and code generators that support the development of embedded systems within a MBSE (Model Based Systems Engineering) methodology, offering capabilities such as automatic code generation, deployment and simulation. TASTE has been applied for system development and deployment in several projects, both in aerospace and in other domains, see e.g. [1, 21, 22].

Recently, the COMPASTA study [6, 7, 4, 5] has extended TASTE with formal modeling and verification capabilities from the COMPASS toolset [8, 9, 10], based on model checking. In particular, the extended version of TASTE [11] offers functionalities for requirements analysis, contract-based design, modeling of HW components and their faults, automated fault injection, functional verification, safety assessment, modeling and verification of FDIR components (see Fig. 2). TASTE is based on different languages, in particular AADL [23] (for modeling the system architecture) and SDL [18] (for modeling the behavior of SW components). COMPASTA extends TASTE by enabling the specification of the behavior of HW components in SLIM (a dialect of AADL), the library-based specification of HW faults, and their automatic injection into the models. In a nutshell, COMPASTA enables the early verification and assessment of a design model that comprises, and specifies the behavior of, both HW and SW components, and reuses TASTE functionality to perform code generation and deployment.

The traditional rule-based approach to fault detection and identification may not be possible to accomplish for complex systems, where the intricacy of the physics of the system components makes it

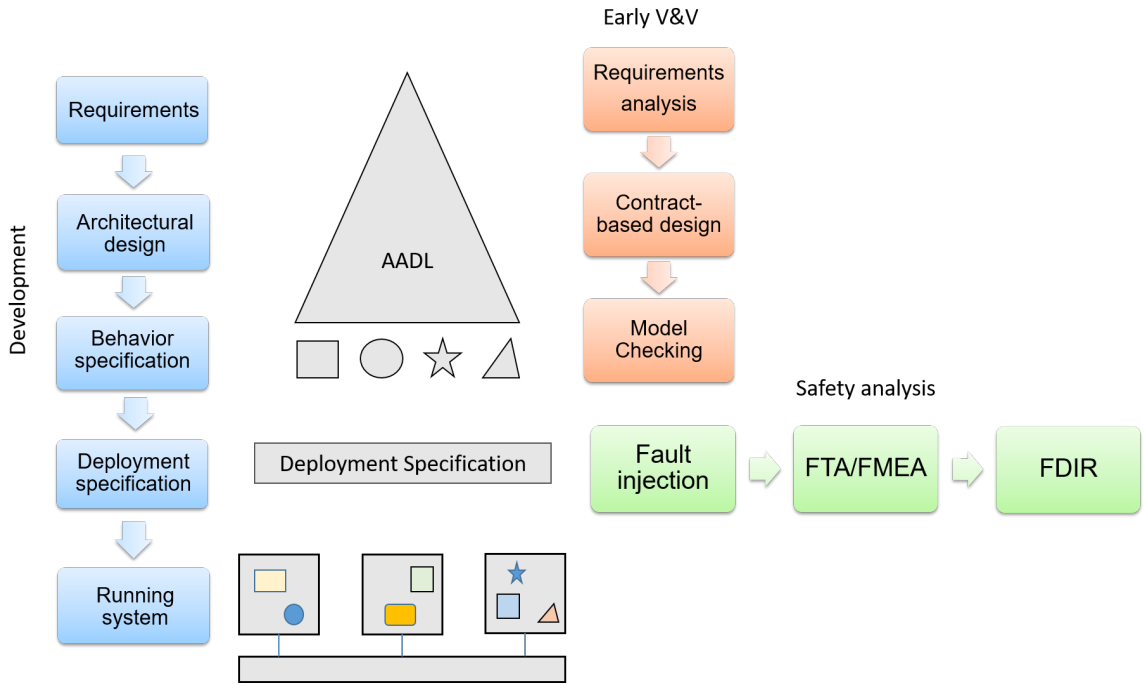


Figure 2: COMPASTA functionality.

impossible to model the components' behavior in a formal and symbolic way. In this context, FDI components based on machine- and deep-learning models are increasingly used for the tasks of anomaly detection and fault detection and identification. This makes the design and formal verification of FDI components even more challenging.

Aim of this work is to explore how to integrate AI-based capabilities into a structured FDIR flow, and to leverage the complementarities of the rule-based and AI-based approaches for the new generation of FDIR systems. In order to encompass the design of FDIR sub-systems which comprise both rule-based components, and components based on machine learning models, we will extend the functionality of COMPASTA, in the context of an ongoing study funded by the Italian Space Agency (ASI) under the "Innovative Space Technologies" initiative. The presence of ML-based components makes the design of FDIR, and the tool support in TASTE, even more challenging at all levels. First, it requires to model ML-based components as part of the system architecture, and to model their behavior. Second, it requires to identify techniques to carry out formal verification of systems including symbolic models along with ML models. Finally, it requires to extend TASTE to enable the deployment of ML-based components.

As regards formal verification of system including ML-based models, our proposal is based on the methodology we identified in the VIVAS project, an ESA-funded activity aimed at developing a generic framework tailored for system-level simulation-based verification and validation of autonomous systems with AI/ML components. The approach is based on a simulation model of the system, an abstract model that describes symbolically the system behavior, and formal methods to generate scenarios and verify the simulation executions. It permits the specification of diverse coverage criteria, thereby directing the automated creation of scenarios, and formal properties to be verified on the simulation runs. The framework has been instantiated and evaluated on both space and automotive applications [16].

The rest of this paper is structured as follows. In Sect. 2, we discuss the use of AI in FDIR design. In

Sect. 3 we discuss our approach based on integrative AI. In Sect. 4, we illustrate the envisaged workflow and tool support. Finally, in Sect. 5 we draw some conclusions and discuss future directions.

2 AI for FDIR

It is not always possible to model explicitly all the potential faults in a complex system. Moreover, rule-based approaches may fail to detect anomalies in complex systems due to their rigid nature, while anomalies may only sometimes conform to predefined rules. In these cases, rule-based FDIR can be advantageously complemented by AI-assisted methods that try to detect upcoming faults that are already in their early stages. This is achieved by means of highly flexible implicit models that, by analyzing patterns and correlations, can learn from historical data and adapt to changing conditions [15].

Machine learning techniques excel at identifying patterns and deviations from those patterns. Unsupervised learning algorithms are commonly used for anomaly detection. They range from simple clustering techniques (K-Means, Density-Based Spatial Clustering), to ensemble methods as Isolation Forests, to One-Class Support Vector Machines, up to autoencoders based on Deep Neural Networks. [13]. With an appropriate training phase, autoencoders can extract relevant features from raw data and learn a model for normal behavior. When the model is then applied to new data, it is possible to point out instances that deviate significantly.

AI models can handle complex, high-dimensional data (e.g., sensor readings, system states, logs) and detect subtle anomalies that can be difficult to have under control with rule-based methods. In systems with complex physics (e.g., aerospace, energy grids, manufacturing), explicitly modeling all the interactions can be a challenging task [14, 2, 20]. AI methods leverage data-driven models that don't require explicit knowledge of the underlying physics. Instead, they can learn from data and capture complex relationships, which can be exploited also to achieve symbolic regression [19], an emerging machine learning method that combines data-driven insights to learn concise, interpretable symbolic models that accurately represent a system's behavior.

Deep learning models (i.e., models obtained by training deep neural networks) have the ability to approximate complex non-linear functions processing multimodal data, and learning interactions between sub-systems without relying on rule-based descriptions. Properly trained neural networks can contribute both to detect anomalies and to classify the types of fault. A further advantage of AI-based models is their capability to achieve continuous learning: these models can, in principle, update their parameters as new data arrive, making them suitable for predictions on dynamic systems and estimation of time-to-fault (residual useful lifetime).

3 Integrative AI

We address the problem of FDIR design with integrative AI. Namely, we target the design of FDIR subsystems which integrate rule-based components along components based on machine learning models. In this section, we discuss the open problems and research directions that we want to address in the AIFDIR study.

The goal of integrative AI is to encompass the specification of FDIR modules for components which are out of reach for traditional methodologies, due to the lack of formal, symbolic models describing their physical behavior. For such components, we aim at use models based on machine learning to target the tasks of anomaly detection and FDI. Moreover, we claim that the use of integrative AI may bring

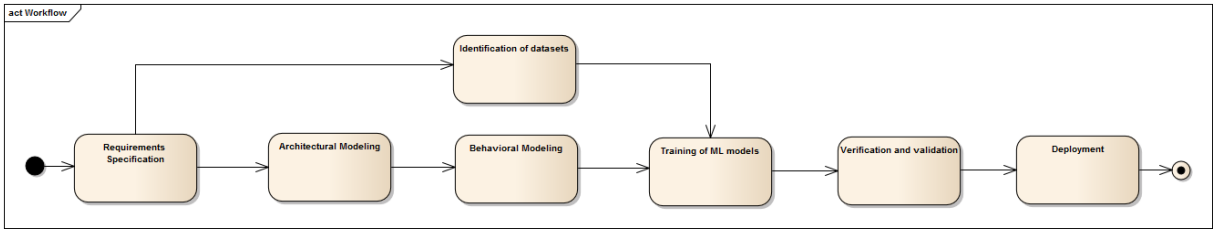


Figure 3: AIFDIR Workflow.

significant advantages in terms of effectiveness and precision of detection, with the ultimate goal of increasing the autonomy and reliability of systems.

In the AIFDIR project, we plan to investigate different paradigms combining symbolic models with ML-based ones. Specifically, we will explore two main paradigms of integration:

1. **Symbolic as an aid for data-driven methods**: In this paradigm, we exploit the knowledge embedded within design models and other information artifacts produced in the development of the monitored system. Here, symbolic models are complementary to data, generating additional features or labels to enhance the performance of ML models, with the goal to produce more robust and insightful predictions.

2. **Data-driven synthesis of symbolic model parameters**: Contrary to the first paradigm, this approach focuses on utilizing data-driven techniques to synthesize the parameters of symbolic models. Symbolic models often rely on manually crafted rules or predefined structures, using various constants to define for example safety margins or the equations of physical dynamics. In some cases, such constants cannot be derived from the telemetry but can be estimated based on historical data.

The use of formal methodologies for FDIR design aims at developing FDIR components that are effective and reliable by construction, and whose properties are formally proved at design time. To achieve this goal, in the AIFDIR project we will extend the VIVAS methodology described in Sect. 1 to encompass the formal verification and simulation of ML-based components. Overall, in AIFDIR we want to deliver a platform, based on TASTE, which supports formal modeling of HW, SW, HW faults, automated fault injection, formal V&V, safety assessment, FDIR effectiveness analysis, automated code generation and deployment.

4 Workflow

The AIFDIR workflow is illustrated in Fig. 3. It starts with the definition of the FDIR requirements and the modeling of the architecture of the FDIR sub-system. Then, it continues with the specification of the behavior of the components, including both SW and HW. In parallel, it is required to identify the datasets for the training of ML-based FDIR components, which is done next. Finally, the system is verified and validated, and it is deployed to the target architecture.

In the rest of this section, we illustrate the intended workflow on a (simple) running example, namely a model of a redundant power supply system for an engine starter (the system architecture is illustrated in Fig. 5 using TASTE). The example is simple for illustration purposes, and is not tied to a specific application domain.

```
system implementation SystemBlock.others
properties
  GenericProperties => (
    [
      Name => "Both starters working";
      Formula => "Starter_1.mode != mode:init and Starter_2.mode != mode:init and
                  Starter_1.is_ok.ok and Starter_2.is_ok.ok";
    ]
  );
end SystemBlock.others;

system implementation SystemBlock.others
properties
  Requirements => (
    [
      PropertyId => "Both starters working";
      Description => "[SystemBlock.others] Both starters working";
  );
end SystemBlock.others;
```

Figure 4: Requirements specification.

The model features several HW components, namely two generators powering two batteries through a switch, with the two batteries in turn powering two engine starters via an additional switch [1]. The connections between the HW components and the switches are redunded, so that the power supply (generator to batteries, or batteries to engine starters) can be maintained in case of failure of a generator or a battery. Generator and batteries may fail, in that case they provide zero output, and engine starters may also fail. The main system requirement specifies that the power supply system must be functional, i.e., at least one starter must be powered and working. The re-configuration of the system is taken care of by two (SW) FDIR components (one dedicated to the generators and one to the batteries) which redirect the generators/batteries output in case of faults (e.g., in case of a fault of one generator, the remaining generator will be used to power both batteries). The behavior of FDIR components is based on a logical specification (finite state machine). However, the FDIR component for the batteries relies on two ML-based components (health monitors) in order to detect possible battery faults. A health monitor takes as input multiple parameters of a battery (output voltage, internal resistance, peak current, etc.) and uses a machine learning model to detect the presence of a fault.

4.1 Requirements Specification

COMPASTA enables the specification of properties and requirements. A requirement is defined with reference to a given property. As an example, Fig. 4 (upper code snippet) shows the definition of a property stating that both starters are working. The corresponding formal definition is given as a formula in SLIM syntax. Fig. 4 (lower code snippet) show the system requirement stating that both starters must

[1] For completeness, the model includes a HW processor hosting the code for the FDIR components, and a monitor component used to state some properties at system level. We do not discuss these components further.

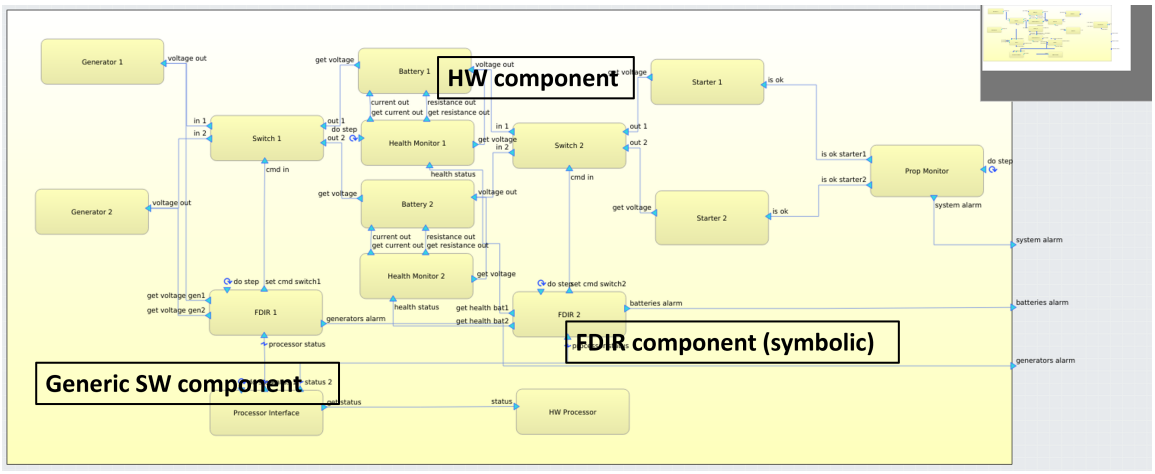


Figure 5: Architectural modeling.

be working. The requirement refers to the previously defined property.

Specification of requirements in COMPASTA is limited to functional and dependability requirements. It is an open issue, to be investigated in AIFDIR, how to express "meta-requirements" (compare Fig. 1) about FDIR, e.g. requirements about the FDIR architecture, the recovery strategies, etc. A starting point for this analysis is the methodology proposed in [12, 3].

4.2 Architectural modeling

Fig. 5 illustrates the system architecture as modeled in TASTE. The system is composed of different blocks, including HW components and SW components. The latter include ML-based SW components (the health monitors). Connections between components are used to feed data (e.g., the output voltage of a generator is an input to an FDIR component), and to route commands (e.g., reconfigurations commands from an FDIR component to the corresponding switch component).

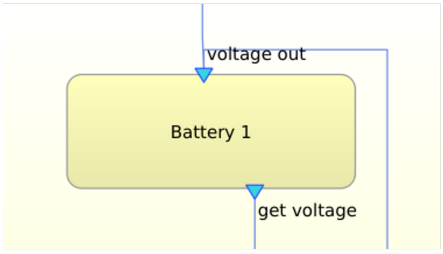
In AIFDIR, we need to extend COMPASTA to enable supporting the modeling of ML-based components in the architecture. We envisage that this can be done by creating an additional component type (ML) in TASTE. For the specification of the behavior of ML-based components, we refer to Sect. 4.6.

4.3 Modeling of HW components

In COMPASTA, we use the SLIM language (a dialect of AADL [23]) to model HW components. As an example, in Fig. 6 we show a fragment of a SLIM specification for a battery. Namely, the fragment shows a transition where the output voltage of battery decreases, due to an insufficient input power supply.

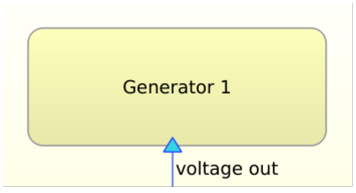
4.4 Fault injection

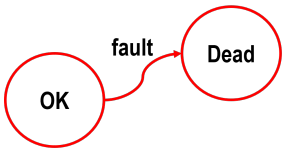
Faults are modeled in COMPASTA, with reference to a predefined library of faults, specifying the effects and dynamics of the injected fault. As an example, in Fig. 7, we show the specification of a permanent fault of a generator, which causes its output voltage to be stuck at zero. Namely, the following information is specified: name of the fault, fault mode (which refers to an item in the fault effects library),




```
system implementation Battery_1.Imp
subcomponents
 delay : data clock;
states
   base: initial state while (delay <= 1);
transitions
   base -[when
             delay >= 1 and get_voltage.voltage < 10 and
             voltage_out.voltage >= 1
           then
             delay :=0 and
             voltage_out.voltage := voltage_out.voltage – 1
          ]-> base;
   …
end Battery_1.Imp;
```

Figure 6: Modeling of HW components.






```
system implementation Generator_1.others
properties
 FaultInjections => (
  [
   Description => Dead;
   Fault_Model => StuckAtByValue_I;
   Fault_Dynamics => Permanent;
   Probability => 1.e-7;
   DataInput => voltage_out.voltage;
   DataVarout => voltage_out.voltage;
   DataTerm => 0;
  ]
 );
end Generator_1.others;
```

Figure 7: Fault injection.

fault dynamics (which refers to an item in the fault dynamics library), fault probability, affected variables (input and output) and a term (the zero constant, in this case).

4.5 Modeling of SW components

We rely on TASTE and the SDL language [18] to specify traditional SW components. As an example, in Fig. 8 we show the SDL model for the FDIR 1 component. The corresponding finite state machine has three main states, called *primary*, *secondary1* and *secondary2*. In primary mode, the two generators are powering the corresponding batteries. The secondary states are entered when the FDIR detect a fault. For instance, if a fault of the upper generator is detected, FDIR sends a commands to the switch and reconfigure the connections so that both batteries are powered by the lower generator. The execution of the FDIR component is controlled by the *do_step* signal, which is cyclic port which is triggered periodically at fixed intervals of time.

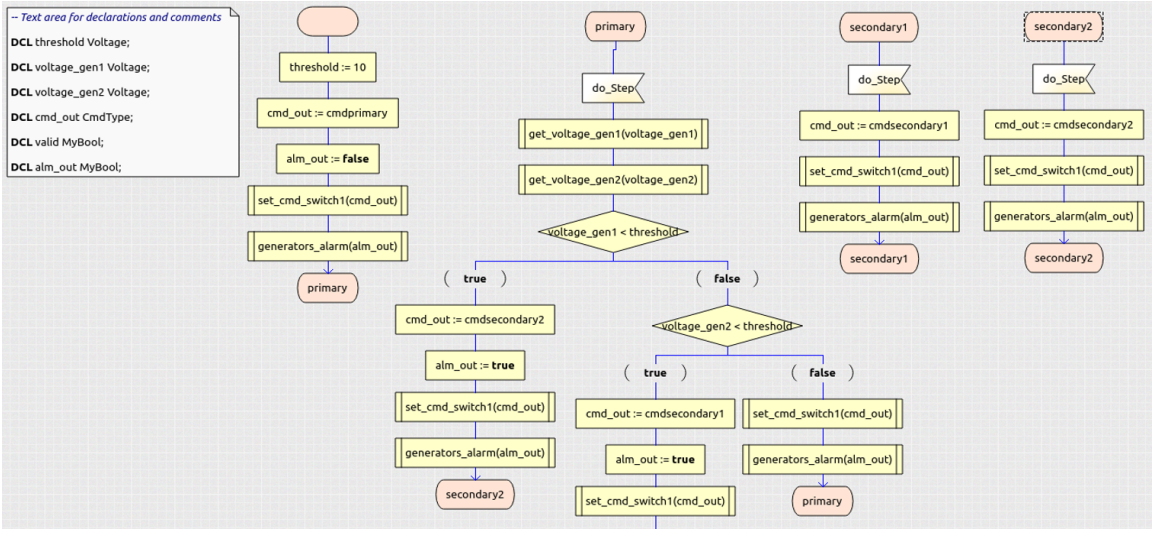


Figure 8: Modeling of SW components.

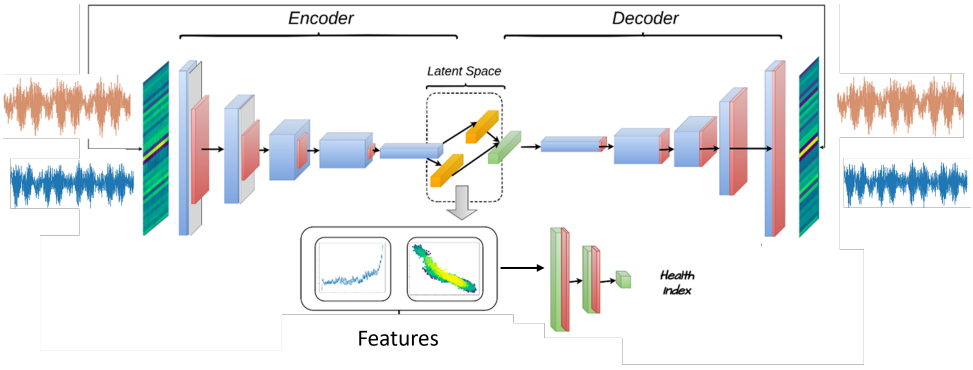


Figure 9: Modeling of ML-based components.

4.6 Modeling of ML-based Components

Detecting anomalous behaviors or changes in trends starting from the analysis of time series data is a typical task for unsupervised machine learning. In the last years, most state-of-the-art solutions have been based on Deep Neural Networks. These algorithms show better performances and the flexibility to include heterogeneous data sources in the model. Autoencoders are a typical example of network architecture adopted for this type of problem and will be considered in the AIFDIR project.

Autoencoders learn to compress input data into a lower-dimensional latent representation and then reconstruct the original data from this representation. An autoencoder consists of two main components: an Encoder that learns the input data's compressed representation (latent space), and a Decoder that reconstructs the original data from the latent representation.

Autoencoders can be used for anomaly detection by leveraging reconstruction errors. During training, the autoencoder learns to minimize the difference between the input and its reconstructed output. Anomalies result in higher reconstruction errors because they deviate significantly from the learned patterns.

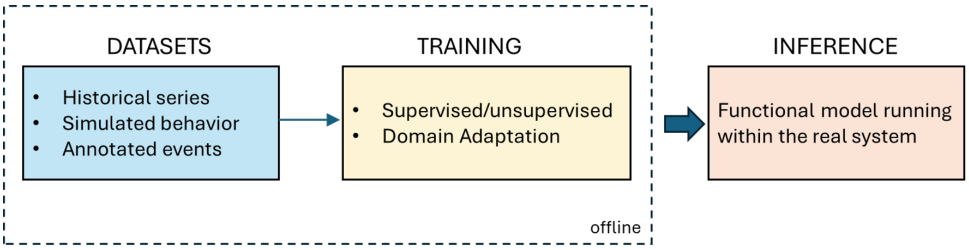


Figure 10: Training of ML models.

4.7 Training of ML models

Autoencoders can effectively utilize heterogeneous input data to detect the anomalous behavior of complex systems. These include sensor readings (temperature, pressure, voltage, etc.), logical variables, time series, feature vectors (e.g., statistical features, domain-specific metrics), images (real or computed, e.g., spectrograms), and text data (system logs, error messages, etc.). In addition to the reconstruction error, the compressed representation of the latent space can be used to evaluate the system's current health state and detect anomalous behavior. An additional neural network, trained to identify known classes of anomalies from the latent variables, enables a "health index" estimation and the prediction of the useful residual lifetime of specific components.

Under this perspective, determining the appropriate dimensionality of the latent space is crucial. While a low-dimensional latent space may not capture all relevant features, leading to information loss, a high-dimensional latent space may overfit the training data and fail to generalize. Transfer learning, domain adaptation, or fine-tuning on new data can help improve generalization.

Capturing temporal dependencies is crucial for time-series data. Recurrent autoencoders (RNN-based, e.g. LSTM) or 1D convolutional autoencoders are appropriate when the temporal context is relevant, i.e., when trends, periodicities, and patterns of sequences are correlated with the correct operation of the system.

Data-driven models require a suitable training set, as the quality and representativeness of the training set significantly impact the model's performance. A model trained on diverse and representative datasets will more likely generalize well to unseen data. When accessible historical data are limited in quantity, additional data may be obtained by a realistic system simulator, if available. Having annotated data usable for model training offers several advantages in machine learning and data-driven tasks. In particular, annotated data (labels, tags, or target values) enable supervised learning, improved loss calculation, and better generalization. Moreover, annotated data can be exploited to apply data augmentation techniques, resulting in increased robustness and regularization of the models.

4.8 Verification and Validation

Regarding the verification and validation (V&V) methodology, our goal is that to extend the capabilities of TASTE in terms of simulation and test case generation so that it can handle AI/ML-based FDIR components. The general idea of our approach is to make use of an approximation (abstraction) of AI models of FDIR components obtained through symbolic models that can be used for the generation of test cases by simulation, in combination with similar abstract symbolic models for the execution environment and possible failure scenarios to consider.

More specifically, we will integrate in TASTE the VIVAS approach, which provides a generic V&V

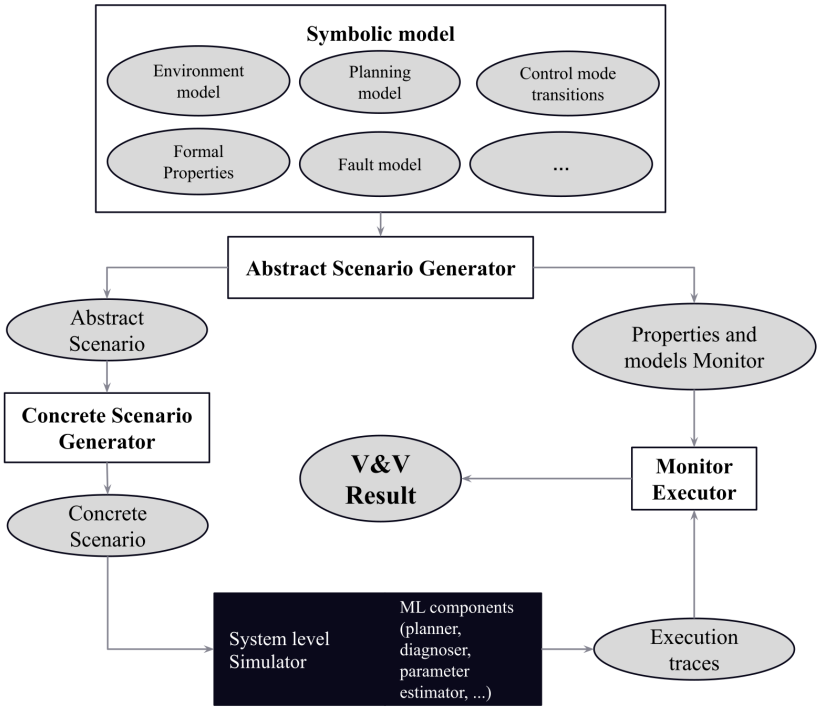


Figure 11: Verification and validation.

framework for generating test cases for systems containing AI/ML components via a combination of system-level simulation and symbolic model checking. VIVAS makes use of formal, symbolic models of the environment and system components to generate abstract test scenarios for the system of interest using model checking techniques. The abstract test scenarios are then instantiated by the concretization of the abstract parameters to provide concrete scenarios to be executed on a system-level simulator encompassing AI models, to obtain execution traces that are in turn analyzed by an automatically generated monitor. The output of the framework is a V&V result consisting of coverage statistics of the executed traces with respect to the symbolic models and quantitative and qualitative information for each use case. The overview of the architecture can be seen in Figure 11 above, which depicts the main parts of VIVAS. These are the abstract scenario generator, the concrete scenario generator, the simulator, and the executor monitor.

Abstract Scenario Generation. Scenario generation is the first step of the approach. The starting point is a formal, symbolic model of the system, which provides an abstract view of both the environment and the components under test (including AI parts). ML components are defined in a declarative manner, approximated in terms of input-output mapping. Abstract test scenarios are generated from the formal system model using symbolic model checking techniques by the abstract scenario generator. Abstract scenarios are defined as combinations of values of predicates describing interesting behaviors of the abstract system. From the technical point of view, each abstract scenario is encoded as a formal property that is expected to be violated by the system (i.e. a property specifying that "the scenario cannot occur in the abstract system"). For each such property defined by the abstract scenario generator, a model

checker will be executed on the system model, with the goal of finding a counterexample to the property. By construction, each such counterexample corresponds to an execution trace witnessing the realization of the abstract scenario of interest.

In order to focus scenario generation on a set of behaviors relevant to the stimulation of FDIR components, we will use abstract models for the failure scenarios and for the FDIR components under analysis, in order to constrain the environment model to prohibit circumstances that are not relevant to the components under analysis. For example, the symbolic fault model by the on-board fault detection components can be used to prevent the generation of faults that are not considered by the on-board system. Similarly, the symbolic models of AI-based components can be used to further constrain the abstract environment and make the simulation more focused.

Concrete Scenario Generation. Each of the traces produced by the model checker is then refined into a (set of) concrete scenarios that can be used to perform simulations using the TASTE platform. Due to uncertainties and abstractions in the abstract scenarios, a one-to-many mapping is defined where a single abstract scenario can be instantiated to many, possibly infinite, concrete scenarios. This is done by defining a mapping between the abstract values and the set of concrete values that they represent and then appropriately sampling from the sets. This set of possible concrete scenarios is then sampled in order to extract a subset of it for execution on the TASTE-based simulator, producing a set of execution traces.

Simulation. The task of the system-level simulator is to run a simulation of the target asset under the requested conditions, by configuring the system, its environment, and its inputs as specified in the concrete scenario produced by VIVAS. Upon completion, the simulator provides the corresponding execution trace of the system, containing all the necessary details to evaluate the properties of interest.

Execution Monitor. Each concrete scenario produced is executed by the simulator, which generates a corresponding concrete execution trace. This trace is then used to determine whether:

1. the concrete execution of the system satisfies the property of interest, and
2. the concrete execution of the system complies with the input abstract scenario (which defines the situation of interest for the current test).

This is done by formally evaluating the trace with a runtime monitor that is automatically generated from the formal specification of the property and the abstract system model. The trace evaluation can have four possible outcomes:

1. The trace complies with the abstract scenario (defining the situation under test), and it also satisfies the property: the test execution is relevant and the test passes.
2. The trace complies with the abstract scenario, but it does not satisfy the property: this corresponds to a test failure on a relevant scenario, and it should be reported to the user.
3. The trace satisfies the property, but it does not comply with the abstract scenario: this corresponds to a (good) execution in an unexpected situation, in which some of the assumptions defining the scenario might be violated. This might be due to imprecisions/abstractions in the symbolic model and in the concretizer, which might prevent the realization of the abstract scenario under analysis. This situation might be reported to the user, as it might suggest that a revision/refinement of the symbolic model might be needed.

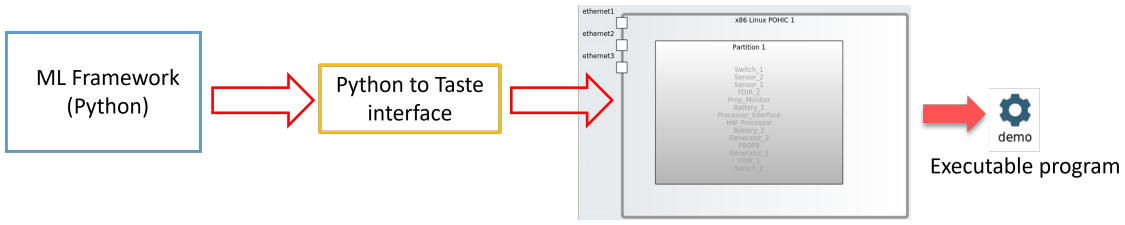


Figure 12: Deployment.

4. The trace violates the property and it does not comply with the abstract scenario: this corresponds to a test failure in an unexpected situation. Similarly to the above, it might be a warning that the symbolic model of the system is not precise enough to capture the situations of interest defined by the abstract scenario.

4.9 Deployment

A valuable feature of TASTE is the possibility to specify the deployment of a SW system onto a specific target architecture and partition, and decouple the deployment from the the system functional architecture. TASTE supports automatic generation of code (e.g., C code from SDL specifications) for different target architectures. The end result is an executable program.

In AIFDIR we plan to extend TASTE to enable the deployment of ML-based components, as illustrated in Fig. 12. Many existing tools and frameworks enable to deploy the pretrained models into for example C++ app, e.g., ONNX runtime for edge and server, NCNN and MNN for mobile device, and TensorRT for Nvidia embedded platforms. In the project we will explore how to embed this tools in the TASTE context.

5 Conclusions and Future Work

In this work we have proposed a novel approach, and a tool-supported methodology and process for the development of accurate and reliable FDIR sub-systems, which integrates rule-based FDIR components with ML-based ones. The approach is being investigated and implemented as part of the AIFDIR study, funded by the Italian Space Agency. In this work we have discussed the open issues and the most promising research directions that we plan to investigate. The AIFDIR study will be evaluated on a set of case studies of interest.

The AIFDIR study is the first step towards the formal design of FDIR components, integrating symbolic models and machine learning (the final product to be delivered in the study has TRL 4). Future work includes bringing the results of AIFDIR to a higher level of maturity, and demonstrate their use in a relevant environment on more complex case studies, in space or in other domains.

References

[1] ADE: Autonomous Decision Making in Very Long Traverses, `https://www.h2020-ade.eu`

[2] Amruthnath, N., Gupta, T.: A research study on unsupervised machine learning algorithms for early fault detection in predictive maintenance. In: 2018 5th international conference on industrial engineering and applications (ICIEA). pp. 355–361. IEEE (2018)

[3] Bittner, B., Bozzano, M., Cimatti, A., de Ferluc, R., Gario, M., Guiotto, A., Yushtein, Y.: An Integrated Process for FDIR Design in Aerospace. In: Proc. IMBSA 2014. LNCS, vol. 8822, pp. 82–95 (2014). https://doi.org/10.1007/978-3-319-12214-4_7

[4] Bombardelli, A., Bonizzi, A., Bozzano, M., Cavada, R., Cimatti, A., Griggio, A., Nazaria, M., Nicolodi, E., Tonetta, S., Zampedri, G.: COMPASTA: Integrating COMPASS Functionality into TASTE. Ada User Journal **44**(1) (2023)

[5] Bombardelli, A., Bonizzi, A., Bozzano, M., Cavada, R., Cimatti, A., Griggio, A., Nicolodi, E., Tonetta, S., Zampedri, G.: COMPASTA = COMPASS + TASTE. CEAS Space Journal **16**(2), 169–181 (2024)

[6] Bombardelli, A., Bozzano, M., Cavada, R., Cimatti, A., Griggio, A., Nazaria, M., Nicolodi, E., Tonetta, S.: COMPASTA: Extending TASTE with Formal Design and Verification Functionality. In: Proc. IMBSA 2022. LNCS, vol. 13525, pp. 21–27 (2022)

[7] Bombardelli, A., Bozzano, M., Cavada, R., Cimatti, A., Griggio, A., Nazaria, M., Nicolodi, E., Tonetta, S.: COMPASTA: Extending TASTE with formal design and verification functionality. In: Proc. MBSE 2022, ESA-ESTEC (2022)

[8] Bozzano, M., Bruintjes, H., Cimatti, A., Katoen, J.P., Noll, T., Tonetta, S.: COMPASS 3.0. In: Proc. TACAS 2019 (2019)

[9] Bozzano, M., Cimatti, A., Katoen, J.P., Katsaros, P., Mokos, K., Nguyen, V., Noll, T., Postma, B., Roveri, M.: Spacecraft Early Design Validation using Formal Methods. Reliability Engineering & System Safety **132**, 20–35 (2014)

[10] Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V., Noll, T., Roveri, M.: Safety, Dependability and Performance Analysis of Extended AADL Models. Computer Journal **54**(5), 754–775 (2011)

[11] COMPASTA wiki page, `https://taste.tuxfamily.org/wiki/index.php?title=COMPASTA:_Integration_of_the_TASTE_and_COMPASS_toolsets`

[12] European Space Agency: Statement of Work: FDIR Development and Verification & Validation Process (2011), Appendix to ESTEC ITT AO/1-6992/11/NL/JK

[13] Fernandes, M., Corchado, J.M., Marreiros, G.: Machine learning techniques applied to mechanical fault diagnosis and fault prognosis in the context of real industrial manufacturing use-cases: a systematic literature review. Applied Intelligence **52**(12), 14246–14280 (2022)

[14] Ferrante, N., Giuffrida, G., Nannipieri, P., Bechini, A., Fanucci, L.: Fault detection exploiting artificial intelligence in satellite systems. In: International Conference on Applied Intelligence and Informatics. pp. 151–166. Springer (2022)

[15] Furano, G., Meoni, G., Dunne, A., Moloney, D., Ferlet-Cavrois, V., Tavoularis, A., Byrne, J., Buckley, L., Psarakis, M., Voss, K.O., et al.: Towards the use of artificial intelligence on the edge in space systems: Challenges and opportunities. IEEE Aerospace and Electronic Systems Magazine **35**(12), 44–56 (2020)

[16] Goyal, S., Griggio, A., Kimblad, J., Tonetta, S.: Automatic Generation of Scenarios for System-level Simulation-based Verification of Autonomous Driving Systems. In: FMAS@iFM. EPTCS, vol. 395, pp. 113–129 (2023)

[17] Hugues, J., Pautet, L., Zalila, B., Dissaux, P., Perrotin, M.: Using AADL to build critical real-time systems: Experiments in the IST-ASSERT project. In: Proc. ERTS (2008)

[18] International Telecommunication Union: ITU-T Z.100. Specification and Description Language - Overview of SDL-2010 (2021)

[19] Makke, N., Chawla, S.: Interpretable scientific discovery with symbolic regression: a review. Artificial Intelligence Review **57**(2) (2024)

[20] Mohd Amiruddin, A.A.A., Zabiri, H., Taqvi, S.A.A., Tufa, L.D.: Neural network applications in fault diagnosis and detection: an overview of implementations in engineering-related systems. Neural Computing and Applications **32**(2), 447–472 (2020)

[21] MOSAR: Modular Spacecraft Assembly and Reconfiguration, `https://www.h2020-mosar.eu`

[22] R. Cavada and A. Cimatti and L. Crema, and M. Roccabruna and S. Tonetta: Model-Based Design of an Energy-System Embedded Controller Using Taste. In: Proc. FM 2016. LNCS, vol. 9995, pp. 741–747 (2016)

[23] SAE: Architecture Analysis & Design Language (AADL) (2022), SAE document AS5506D

[24] TASTE web page, `https://taste.tools/`