How to early integrate operational diagnosis objectives in model-driven engineering processes: A methodological proposal based on fault and behavior trees

Nikolena Christofi, Claude Baron, Xavier Pucel, Marc Pantel, David Canu, Christophe Ducamp

Received: 17 April 2022 | Revised: 31 March 2023 | Accepted: 27 November 2023

DOI: 10.1002/sys.21740

REGULAR ARTICLE

WILEY

How to early integrate operational diagnosis objectives in model-driven engineering processes: A methodological proposal based on fault and behavior trees

Nikolena Christofi[1] | Claude Baron[2] | Xavier Pucel[3] | Marc Pantel[4] | David Canu[5] | Christophe Ducamp[5]

[1]IRT Saint Exupéry, LAAS-CNRS, Airbus Defence and Space, INSA Toulouse, Université de Toulouse, Toulouse, France

[2]INSA Toulouse, LAAS-CNRS, Université de Toulouse, Toulouse, France

[3]Office National d'Etudes et de Recherches Aérospatiales (ONERA), Artificial and Natural Intelligence Toulouse Institute (ANITI), Université de Toulouse, Toulouse, France

[4]IRIT, INP Toulouse - ENSEEIHT, Université de Toulouse, Toulouse, France

[5]Airbus Defence and Space, Toulouse, France

Correspondence
Claude Baron, Computer and Electrical Engineering department, INSA Toulouse, and Researcher, System Engineering and Integration team, LAAS-CNRS, 7 avenue du Colonel Roche, 31031 Toulouse cedex 4, France.
Email: claude.baron@laas.fr

Funding information
Agence Nationale de la Recherche, Grant/Award Number: 10-AIRT-0001

Abstract

To help operators perform their diagnosis tasks more efficiently, the authors put forward a novel methodology introducing a new type of model, dedicated to operations, co-created in parallel with the design models, in the preliminary stages of system development, using the language semantics of behavior trees. In this paper, the authors present the need for this new model type as expressed by the industry and justify their choice for adopting behavior trees, while illustrating in detail the proposed methodology.

KEYWORDS
behavior trees, complex systems, fault trees, MBSA, MBSE, operational diagnosis, space operations, space systems, system modeling, system monitoring

1 | INTRODUCTION

In the field of complex systems such as satellites, operational performance requirements impose important constraints to system design. System operation risk assessments are based on RAMS performance factors that is, Reliability, Availability, Maintainability, and Safety.[1] However, while the methodologies associated to the evaluation of system Reliability and Safety are well defined, no official methodology is established for Availability and Maintainability. For the case of satellites, maintenance is performed by their operators in distance, since no intervention for in-situ reparation is possible. The only way ground operators can resolve issues on-board the satellite is by initially detecting and isolating the faults occurred, with the eventual goal to restore the system to its nominal (or acceptably functional) state, as soon as possible. These activities constitute key elements of Fault Detection and Diagnosis (FDD) activities, an evolution of the Fault Detection and Isolation (FDI) domain.[2]

However, although the onboard Fault Detection, Isolation and Recovery (FDIR) capabilities of modern-day satellites are very advanced, that is not the case for their respective Earth segments. The

latter are expected to operate with a lower level of autonomy, whilst on-the-spot intervention for physical repairs is possible. In fact, physical repairs in space operational systems are only possible onboard manned orbiting spacecrafts, such as the International Space Station.

System operators have various simulation tools in their disposal. Nevertheless, their activities are bound by strict time limits: narrow communication windows (the time interval within which an operator can exchange data and commands with the satellite) last no longer than 10 min in the case of Low-Earth-Orbit satellites. Working towards the improvement of diagnosis during operations is imperative, since downtime costs are significant for every second the satellite is unavailable that is, satellite unreachable or not able to perform its mission. Based on feedback received by Airbus Defense and Space and the French Aerospace Lab ONERA, the authors have confirmed the space industry's urgent need for models and tools dedicated to system monitoring, aiming to helping operators in their diagnostics tasks.

The long-term objectives of international space agencies such as the National Aeronautics and Space Administration (NASA) and the European Space Agency (ESA), also confirm the need to improve operations by increasing the use of model-driven design techniques. More specifically, according to NASA's 2022 Strategic Plan,[3] one of the four strategic goals of the agency is to "enhance capabilities and operations to catalyze current and future mission success". In addition, and as stated in its previous strategic plan review, the consolidation and improvement of operations already consisted an important pillar within NASA's long-term objectives, the end goal being to "balance risks across services and activities to provide a safe and reliable infrastructure".[4] In an effort to reduce costs, while moving towards an interdependence model between facilities, tools and services, NASA aims to a "more focused investment on condition-based maintenance and reliability-centered maintenance". In line with the effort for the digitalisation of data, tools, services, and their continuity, ESA aims to reduce the recurring cost of operations, and to invest in innovative spacecraft operations solutions, in order to "enable flexible, efficient and high-performant operations concepts". In particular, the agency aspires to "further position Europe as a strong competitor on the world market", as stated in its *Technology Strategy for Space19+* plan.[5]

Operators' troubleshooting tasks vary depending on the health status of the system at each given moment, as indicated by the registered data. Per contra, if a symptom –or a combination of symptoms, is unknown, that is, if system designers did not model the symptom or foresee the specific failure occurrence, operators are expected to perform troubleshooting, in order to eliminate the error, and restore functions to their nominal state (or the least degraded possible). Troubleshooting consists of the tasks of requesting for the system for additional information or/and performing manual tests/manual investigation, etc. The operators' priority is to avoid loosing the system services. Thus, unless the anomaly has an associated troubleshooting procedure for example, in a symptoms' database, the operators must take individual action that is, carrying out a dedicated analysis and updating the system's knowledge data base.

These actions are usually based on the operators' knowledge and experience. If this knowledge were to be supported by organized documentation of the system itself (architectural and behavioral, as well as functional and dysfunctional), operational diagnosis could be improved significantly. At the same time, system design models are not adapted for maintenance activities, while they are far too complex to exploit. What is more, operators do not necessarily have the needed expertise to explore these models. For this reason we envisaged the creation of an Operations-Dedicated Model (ODM) oriented toward system maintenance tasks and FDD.

As shown in Figure 1, system architecture and behavior description models, which are created throughout Systems Engineering (SE) activities, along with system dysfunctional behavior models, which are in turn produced during Safety Analysis (SA) activities, are being built in the beginning of the system lifecycle, during the system design phase. MBSE and MBSA models are used beyond the system design, particularly in the development and verification system phases, in particular for traceability reasons. The first serves in meeting the set requirements, and the latter in respecting safety objectives while complying with international standards and regulations, which are defined during the system conception phase—mostly the case for the aeronautics domain, not applicable in the aerospace sector—except for satellite End-Of-Life management. As illustrated in the schema, the activities related to system design (prior to system launch), as much as operations and maintenance (post-system deployment), are, although related and interdependent, detached.

The ODM shall help the operators perform their diagnostics tasks more efficiently by reducing their response time to failure, but also facilitate access to documentation and dedicated procedures so as to optimise their troubleshooting actions. Hence the ODM shall have the capability not only to provide the current system overview, but also to be exploited by a diagnosis tool. This way it shall provide possible fault candidates, in the case of a raised alarm, or erroneous received data.

Hence this new type of system model, dedicated to system monitoring, shall be able to be exploited during operations. One way to exploit the ODM is to drive a diagnosis tool, with which the operators could interact, through a dedicated Graphical User Interface (GUI). Moreover, the ODM can be built during the design phase and along the production of other design models and documents, in a co-design manner that is through a digital continuity between design, operational safety and ODM construction.

The structure of this article is as follows. Section 1 outlines the context of our research and the problem addressed, as well as the academic and industrial gaps that motivate our study. Section 2 introduces the language semantics of BTs and justifies their choice for the ODM construction. Section 3 provides a detailed description of the standard semantics of BTs, as well as the extensions we have added to the language, to meet our FDD needs. The proposed approach, and greatest contribution of our work, is presented in Section 4. Finally, Section 5 draws conclusions and discusses future work opportunities.

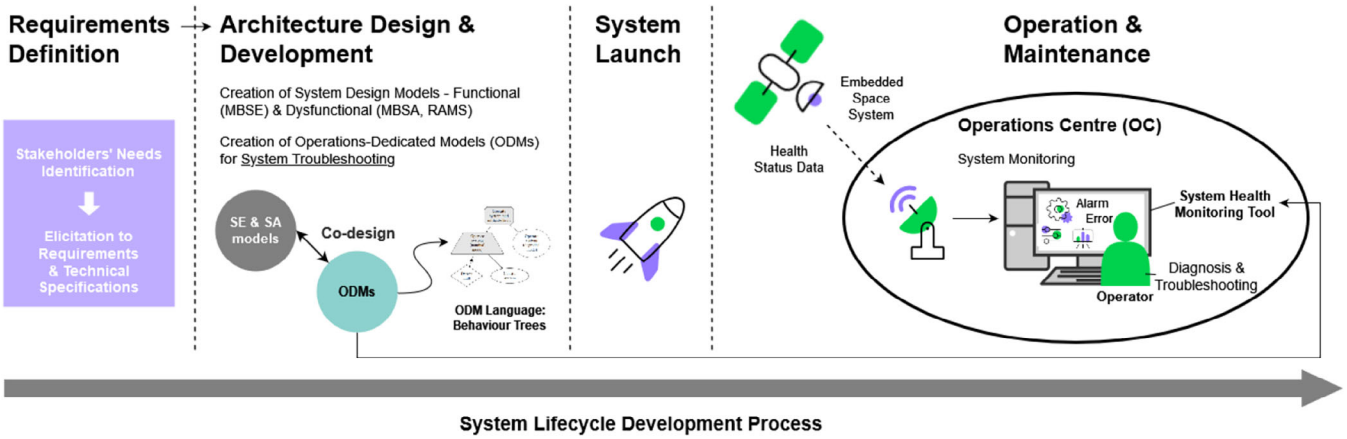


FIGURE 1 Overview of the proposed approach: ODM concurrent creation with system functional and dysfunctional models. Exploitation during operations and maintenance: integrated in system health monitoring tools, for fault detection, diagnosis and troubleshooting. System lifecycle processes based on.[6] ODM, Operations-dedicated model.

2 | CHOOSING THE ODM SEMANTICS

In order to construct the ODM, we need a formalism that can interpret both SE and SA elements into one unique model. This model shall serve a double purpose. On the one hand, it shall include necessary system monitoring information in order to perform automatic diagnosis and hint to a single failure candidate, and on the other, to be easily readable and usable by operators, in order to aid them in their troubleshooting tasks. As BTs can fulfill both these causes, we have selected them, among other candidate language semantics, as illustrated in Section 2.2), as the most suitable choice for the ODM construction.

To this end, before demonstrating the use of BTs to construct an ODM, we set their requirements in Section 2.1. We then provide a thorough comparative study amongst the candidate language semantics for the ODM construction and justify our BT choice in Section 2.2. Section 2.3 provides a summary of the section's findings.

2.1 | Setting the ODM requirements

As mentioned earlier, several languages can be considered suitable for expressing the ODM. The ability of each language semantics to meet the ODM semantic requirements was used as a criterion for their evaluation. Each criterion was assigned a weight factor, indicating its importance for the ODM creation and exploitation, on a scale of one to three—one having the lowest and three the highest impact, as shown in Table 1. The criteria were defined based on the:

- ability of the language to represent specific system elements, notably:
  - hierarchical structural decomposition (component breakdown), as hierarchical modeling capacity is essential for FDD purposes;
  - hierarchical behavioral decomposition/functional description (processes/tasks/activities/functions' breakdown);
  - functional *and* dysfunctional behavior (how the system behaves in nominal and non-nominal conditions), necessary for operational diagnostics;
  - represent system supervision information / account for feared events, faults, and mitigation means such as troubleshooting and repair that is, system diagnostic capacity;
- ability of the language to integrate textual information and model data derived by SE and SA activities;
- ability of the language to model operational activities;
- models' executability;
- ability of the executable models to return all system states;
- ease (level of semantic complexity) and intuitiveness of modeling, so as to reduce the skills required by the ODM modeler;
- ease (level of semantic complexity) and intuitiveness of model exploitation, in order to maximise user-friendliness (we consider operators as the end-users);
- maintainability of the created models, in the sense that high semantic complexity increases the difficulty of understanding by future modelers hence making the models hard to maintain;
- ability of the language to support of functional & dysfunctional system analyses, for the amelioration of system design; not part of the comparison since it consists future work, but was taken into consideration;
- ability of the language to be integrated within a diagnostic tool equipped with a GUI; not part of the comparison since it consists future work, but was taken into consideration.

2.2 | Language semantics comparison

Based on their popularity, extensive usage and numerosity of applications, we have selected seven semantic approaches to evaluate. A brief description of each and an assessment of their potential application to the operation of aerospace systems is to follow.

TABLE 1 Comparison amongst candidate language semantics for the creation and exploitation of Operations-dedicated models.

	Business Process Model and Notation (BPMN)	Finite State Machines (FSMs)	Markov Chains (MCs)	Petri Nets (PNs)	Decision Trees (DTs)	Fault Trees (FTs)	Dynamic Fault Trees (DFTs)	behavior Trees (BTs)	Weight (1–3)
Original Application Field	Human Task Planning	System Logic modeling	Event Probabilities	Formal Verification	Rational Decision Making	Safety Analysis	Safety Analysis	Gaming, AI, Robotics	-
Can represent system structure	No	No	No	Yes	No	Yes	Yes	Yes	3
Can represent system behavior	Yes	No	Yes	Yes	No	Yes	Yes	Yes	3
Can represent operational sequences & fault mitigation mechanisms	Yes	No	No	Yes	No	No	No	Yes	3
Models' executability	No	Yes	Yes	Yes	Yes	No	No	Yes	3
Can return status of all system states	No	Yes	Yes	Yes	No	Yes	Yes	Yes	3
Supports hierarchical structural modeling	No	No	No	No	No	No	Yes	Yes	3
Supports hierarchical behavioral modeling	Yes	Yes	No	Yes	No	No	No	Yes	3
Supports functional and dysfunctional input data	No	Yes	Yes	Yes	No	Yes	Yes	Yes	3
Can intuitively illustrate current system states (system states overview)	Yes	Yes	No	No	Yes	No	No	Yes	2
Can integrate system design data	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	2
Low modeling semantic complexity	Yes	No	No	No	Yes	Yes	Yes	Yes	2
Level of modeling intuitiveness	High	Medium	Low	Low	High	High	High	High	2
Level of user understanding intuitiveness	High	High	Low	Low	High	High	High	High	3
Level of model maintainability	High	Low	Low	Low	High	Low	Low	High	1

First we tackle the *Business Process Model and Notation (BPMN)*. BPMN is a standard for business process modeling[7] that provides a graphical notation for specifying business processes. BPMN models comprise of *Business Process Diagrams (BPDs)*.[8] The latter are based on the flowcharting technique—similar to UML Activity Diagrams.[9] The objective of BPMN is to support business processes' management. It also provides a mapping between the graphics of the notation and the underlying constructs of execution languages, particularly *Business Process Execution Language (BPEL)*. BPMN is made to represent processes, and not complex embedded systems. Although it provides a notation that is intuitive to both business and technical users, while being able to represent complex process semantics, it does not meet the ODM objectives. Although with BPMN it is possible to model the behavior of system processes, it is not possible to model the system structure. The same goes for hierarchy: sub-processes can be modeled inside other processes, but sub-activities cannot be decomposed into sub-activities. Some research work on risk analysis and MBDA has implicated the use of BPMN, but only for visualisation purposes that is, methodological process modeling, not executable behavior modeling.[10,11]

We then turn to *Finite State Machines (FSMs)* —otherwise called Finite State Automata (FSA). FSMs consist the simplest modeling description of Discrete Event Systems, by providing a graphical representation of finite relations between the constituting system states. Each FSM can be in exactly one of a finite number of states at any given time *t*, based on its input and state transition function.[12] Some example model diagrams that use FSMs are *SysML Activity Diagrams* or *Harel Statecharts*. Although FSMs are based on a strong mathematical computation model, they fail to represent system structure and

behavior. Moreover, FSM diagrams can get quite overwhelming when dealing with large models. As mentioned in ref. [13], FSM-based models "involve exhaustive searches or simulation of system behavior and are especially impractical for large and complex systems". What is more, they provide only one way (finite state automaton) to represent a system element for example, function and component. They hence lack modeling expressiveness. What is more, automata are not good with parallelism and numbers, that cause the number of states to explode.

Defining system structure is possible by using variants of FSMs, such as synchronous state machines. The latter allows the splitting of machines into components, however the tool is difficultly deployed and the system hard to model. Another example of FSM variants are *Markov Chains*. A Markov chain describes a process in which the transition to a state at time $t + 1$ depends only on the state at time t. The main difference to FSMs is that the transitions in Markov chains are probabilistic rather than deterministic. Also, Markov chains must be synchronized to be able to represent system structure. Thus they display the same limitations as FSMs, as regards to ODM design and execution.

Petri Nets (PNs) offer a richer language to describe FSMs. In contrast to FSMs, they offer more convenient means to complex system modeling in that large systems can be represented in a more compact manner. As stated in ref. [13], PNs "offer a much more compact state space than finite automata and are better suited to model systems with repeated structure"; a reason why they are still used in industrial research and development. PNs also allow representing system elements in many different ways, hence providing more modeling freedom than FSMs.

Nevertheless, PNs were shown to be inferior to FSM in terms of response time performance,[13] for some problems. Moreover, since PNs do not have a tree structure, there can be multiple shared places and transitions between multiple nets, leading to a potential spaghetti code problem. That is in the sense that tree structures tend to be easily maintainable.

Last but not least, in PNs or automata-based methods (PN transitions are similar to event firings in FSA), only the start and finish of each simulated event is considered in the modeling, resulting to neglecting the variable dynamics that take place in between.[14] For all these reasons, we also determined PNs to be unsuitable for ODM construction and exploitation.

Next, we contemplate the use of *Decision Trees* as the language semantics to create and exploit ODMs. Although Decision Trees have very similar logic to BTs, they present a main disadvantage. That is that they represent stateless functions, that is, functions whose output at time t depends on their input at time t. It is still possible to use them to represent a transition function, where the Decision Tree input represents the system state at time $t - 1$, and the system input at time t, and the Decision Tree output represents the system state and output at time t.

Although Decision Trees are good at expressing complex functions in a compact and explainable manner, they lack the notion of *component*. Their "tree" aspect cannot be used to model a component or function hierarchy, because it is used to represent the different possible choices that a function involves. This assuredly argues that Decision Trees are incompatible with the ODM requirements, and are thus rejected. What is more, attempting to force the notion of state into a Decision Trees is not possible. In fact, BTs are known to generalize Decision Trees.[15]

Finally we evaluate the sole use of *Fault Trees (FTs)* as an ODM alternative. FTs are considered as the most used technique in complex systems dependability assessment. Albeit their many advantages, they present many drawbacks when it comes to systems modeling with strong temporal component dependencies. That is, in FTs, there is a structural dependence between components, but this dependence is not dynamic (temporal). Hence the modeled faults are considered to occur asynchronously (asynchronous faults-sequence).

Moreover, as observed in ref. [16], "The assumption of components independence is precisely what makes FTs so powerful, but this assumption is extremely restrictive, and may prove to be totally unrealistic and lead to grossly erroneous results for some kinds of systems. To be able to model component dependencies, one has to recur to dynamic models. The most popular are Markov processes, because of their numerous nice mathematical properties." However, and as discussed above, we have already concluded that Markov chains do not consist adequate ODM candidates.

In addition to the lack of FTs' dynamic and synchronous properties, FTs fail to support hierarchical modeling and operational sequences representation. What is more, FTs are not easily maintainable, since a small change in the respective SE model can lead to big modification efforts in the FT. That is, unless we consider an automatic solution to generate FTs from design models, by defining a cross matching between an MBSE meta-model, an FT meta-model and a dysfunctional meta-model (failure mode). However, this kind of solutions are not yet mature, while they are based on assumptions regarding the design model structure. For this and all the above reasons, we can safely conclude that BTs consist a more optimal solution, than FTs, for the ODM semantics.

Lastly, we slightly shift our focus from FTs to *Dynamic FTs (DFTs)*, which were created to overcome FTs' non-temporal constraints, by allowing the modeling of sequence/time-dependent failure system behavior. Although DFTs extend FTs' expressiveness in terms of events' execution priority, redundancy and functional dependency to other trigger events, they tend to get very large and complex when dealing with real-size systems. Not only a high level of expertise is required for their modeling, but also additional processing is necessary for their qualitative and quantitative analysis.[17] By keeping in mind that our main objective is to have an easily accessible overview of the system functions' state for the operators, and at the right level of abstraction so as to facilitate their troubleshooting tasks, we can safely conclude that DFTs do not either fulfill ODMs' requirements.

2.3 | Discussion

Although our inclination towards the use of BTs is justified, the use of other language semantics would add some benefits to our approach. For example, FTs are far more known and popular than BTs in the industrial world, and can be easily interpreted by non-experts (useful for

troubleshooting activities). Other than their familiarity, FTs can model a flexible number of feared events, while they are also applicable to positive events. However, BTs can also model both positive and negative events, and with no limitations to the size of the tree that is, number of modeled nodes. Still, the industrial world will need to be convinced for adopting a new modeling language.

The main reason why we propose an activity behavior model as the ODM, and BTs to represent it, is mainly the fact that we must remain, at all times, system agnostic. This eliminates the choice of any domain-specific models for example, Decision Trees, FSA, etc., as well as of model templates where expected system behavior is categorized by preset dysfunctional tasks.[18] Furthermore, an activity behavior model helps with coordinating diagnosis activities' actors, since some diagnosis activities are automated, and some other manual.

Furthermore, the ODM is supposed to help operators decide what should their next action be—and elaborate an automatic diagnosis as far as possible, which is a matter of activity, rather than structure, or interactions. Not choosing a structure or interaction diagram—as opposed to a behavior diagram, assumes having a single "Operator" entity communicating with a single "System" entity. In a system with a more complex structure, initial work is required to narrow down "Operator-system" pairs to which our methodology can be applied.

Incidentally, our methodology proposes a semi-automated model transformation, in contrast to a fully automated approach, considering that some expert knowledge is still required to link all the complementary information coming from the SE activities in the right "place" in the new model. Moreover, using FTs as the transformation starting point means that the system dysfunctional analysis is completed by an an independent team of RAMS/SA/MBSA experts. So by having FTs as input, we ensure receiving a full system FMECA description. This means that we do not need to be concerned about fault omissions, neither about introducing wrong dysfunctional elements in the system model. Errors during operational diagnosis, for example due to a bad BT model, will result in inappropriate actions from the operators. This kind of fault is typically addressed in FTA. In that sense, our methodology does not intervene in the work of SA.

3 | BT LANGUAGE SEMANTICS

Before demonstrating the use of BTs to construct ODMs, an introduction to BT theory is necessary. This section is dedicated to the BT language semantics description.

BTs have shown a lot of potential in the last decade,[19,20] mainly with their application to robotics and artificial intelligence.[21–23] They were initially developed within the gaming community to replace FSMs with more user friendly models.[24]

According to Colledanchise and Ögren, "a Behavior Tree is a way to structure the switching between different tasks (assuming that an activity can somehow be broken down into reusable sub-activities called tasks) in an autonomous agent, such as a robot or a virtual entity in a computer game".[25] According to García et al., a Behavior Tree is "a mathematical model of plan execution that allows composing tasks in a modular fashion through a set of nodes representing tasks and connections among them".[26]

The underlying formalism and semantics of BTs support top-down elicitation, thus allowing modular modeling. By this it is meant that, throughout the system development, some subsystems can be thoroughly developed and some others remain as "blackboxes", depending on the information available at the time. This way a single BT is easier to elicit, since it can be exploitable at any level of development. The model can hence stay abstract or be developed in detail. Moreover, new knowledge can be integrated in the BT when this knowledge becomes available from the elicitation of other models, namely the SE and SA models.

In this section, we present the classic formulation of BTs as described in ref. [25] by Colledanchise and Ögren, where BTs can be considered as a form of directed tree, where the flow amongst its nodes and edges is sequential. BTs can be confused with Decision Trees but both are fundamentally different. On the one hand, a Decision Tree, in its classical formulation, implements a function from $\mathbb{B}^n$ to $\mathbb{B}$ –where $\mathbb{B}$ is the set of Boolean values $\{0, 1\}$. On the other hand, a BT implements a way to orchestrate the execution of a set of processes (called behaviors), to support both sequential and concurrent compositions. Most BTs libraries use concepts of "Success" and "Failure" of a behavior to perform conditional branching. We present here the formulation of ref. [25], illustrated in Figure 2.

A BT represents and implements a way to control the execution of a set of concurrent processes. Each concurrent process is represented by a leaf node. Parent nodes for example, Fallback, Sequence, Parallel, can start and interrupt their children nodes, and query their status. The BT itself is another process that executes periodically. The root node has no parent nodes, and leaf nodes have no children. A more complete description can be found in ref. [25].

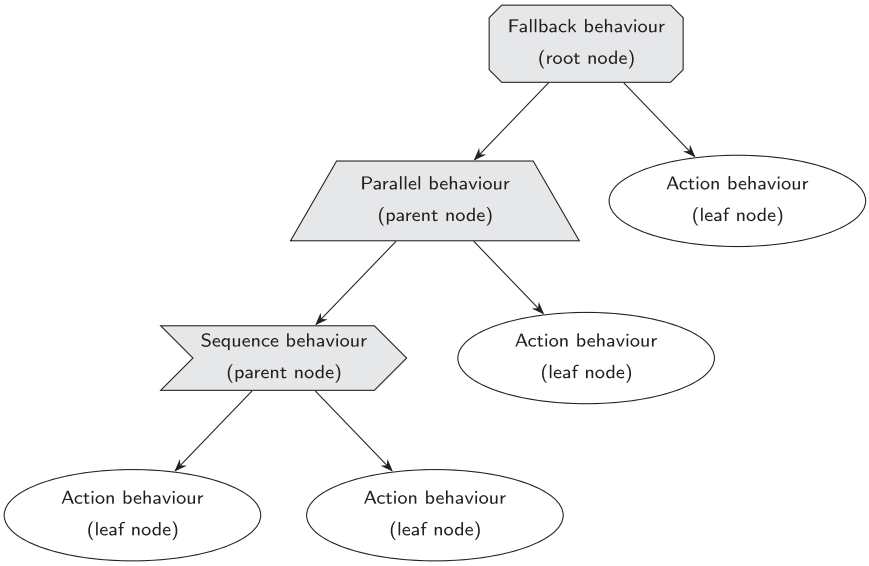
During execution, at regular time intervals, the BT "ticks" its nodes. The Root is ticked first, and each composite node ticks some (or all) of its children, in a top-down, left-right execution priority, as shown in Figure 3. The returned status of each node can be one of the three: "Success", "Failure", or "Running". Fallback and Sequence nodes tick their children sequentially (from left to right), while Parallel nodes tick all of their children in parallel.

At each tick, Fallback, Sequence and Parallel nodes return "Running" if and only if their currently active child returned "Running". Upon ticking, a Fallback node returns "Success" if at least one of its children nodes returns "Success", and "Failure" if all of its children return "Failure". On the other hand, a Sequence node returns "Success" if all of its children return "Success", and "Failure" if at least one of its nodes returns "Failure".

Parallel nodes launch all of their children in parallel. They return "Success" if all of their children nodes have returned "Success", and "Failure" if at least one of their children nodes have returned "Failure". Inverter nodes return the inverse behavior status of their child. They return "Success" if their child node has returned "Failure", and "Failure" if their child node has returned "Success".

Different libraries dedicated to BT modeling exist, such as a python implementation of BTs,[27] and a BT library in C++.[28] Moreover, there

FIGURE 2 Behavior Trees' basic elements.



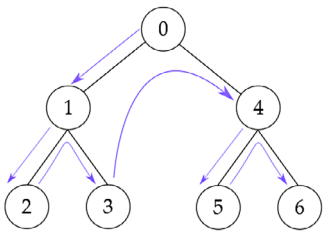


FIGURE 3 BTs' execution sequence. BT, Behavior Trees.

exist Graphical Editors to create BTs, such as "Groot",[29] compatible with "BehaviorTree.CPP". For the needs of our work we have decided to use the python BT implementation for the ODM construction. Based on our experience, after having tested both PyTrees and BehaviorTree.CPP, we concluded that PyTrees is more adapted to our requirements. We believe that PyTrees is much easier and more intuitive to use than BehaviorTree.CPP, especially for people with no particular coding skills. Hence even without the help of a graphical tool, engineers with no programming background nor development experience (consisting our target group) can build ODMs. That would assume the modelers' accompaniment with appropriate guidance and training.

BTs represent Discrete Event Systems in a way that seems promising, in our context of model elicitation by operators. Nevertheless, other representations provided by MBSE, MBSA and scientific research also do exist. To our knowledge, the use of BTs for operational or model-based diagnosis has not so far been explored elsewhere.

3.1 | BT language semantics - Standard nodes

This section provides definitions for standard BT nodes which are used in the ODM methodology. Although a terms description exists in literature, formal definitions are lacking. We have hence chosen to define our own formal descriptions for BTs as follows, as shown in Section 3.1 below. Section 3.2 provides the definitions of BT behavior nodes as extended to match the semantic needs of ODMs.

Behavior Tree (BT) A *Behavior Tree (BT)* is a tuple $\langle \mathcal{B}, root, children \rangle$ where $\mathcal{B}$ is a set of behaviors, $root \in \mathcal{B}$ is the root behavior, and $children : B \rightarrow B^*$ is a function which associates each behavior with an ordered (and possibly empty) list of children behaviors. In a BT, each behavior has exactly one parent—except the root behavior, which has no parent.

Figure 4 illustrates a BT, in which the root behavior is named "Operate system and mitigate fault". Its children are named "Operate system (nominal mode)" and "Operate system (degraded mode)"; the former has children behaviors, while the latter is a leaf behavior. Each behavior *is composed of* its children—with the exception of leaf/atomic behavior nodes.

Behavior status The set of possible *statuses* for behaviors is the finite set $\mathcal{S} = \{\text{IDLE}, \text{RUNNING}, \text{SUCCESS}, \text{FAILED}\}$. At each instant in the execution of a BT, the state of the BT is a function $state : \mathcal{B} \rightarrow \mathcal{S}$, which associates a status to each behavior in the tree.

A behavior whose status is not RUNNING can be started by its parent, and its status then becomes RUNNING. A running behavior can be interrupted by its parent; its status then becomes IDLE. A running behavior can autonomously change its status to SUCCESS or FAILED.

Leaf behaviors are used to represent the actual activities implemented by the system, under the form of concurrent processes. Their execution is orchestrated by their parent behaviors, which are usually picked among a set of predefined composite behaviors.

We use four predefined types of composite behaviors: SEQUENCE, FALLBACK, PARALLEL and the INVERTER behavior. For the needs of our study we have created a new type of parallel behavior node. We hence

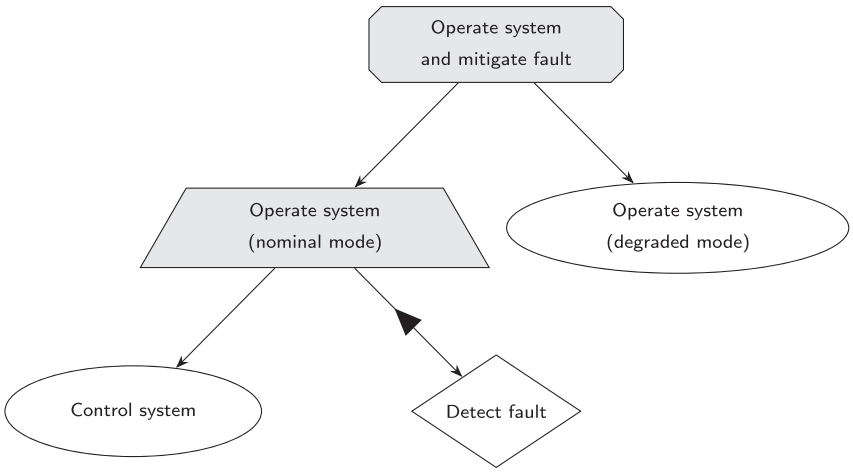


FIGURE 4 The BT for a system operation with fault detection and mitigation. Children are ordered from left to right. The BT features a new type of *Fault detection node* as presented in Definition 3.2. BT, Behavior Tree.

TABLE 2 Description Summary of Behavior Tree Standard and Extended Nodes Execution: Sequence, Fallback, ParallelAll, ParallelAny, Inverter, Fault Detection and Fault Avoidance.

	Sequence	Fallback	ParallelAll	ParallelAny	Inverter	Fault Detection	Fault Avoidance
On tick	Ticks current child	Ticks current child	Ticks all children	Ticks all children	Ticks child	Ticks child	Ticks child
One child returns "Success"	Ticks next child	Returns "Success"	Waits all children	Interrupts other children, Returns "Success"	Returns "Failure"	Returns "Success"	behavior never succeeds
Last child returns "Success"	Returns "Success"	Returns "Success"	Returns "Success"[a]	Returns "Success"[a]	–	–	–
One child returns "Failure"	Returns "Failure"	Ticks next child	Interrupts other children, returns "Failure"	Waits all children	Returns "Success"	behavior never fails	Returns "Failure"
Last child returns "Failure"	Returns "Failure"	Returns "Failure"	Returns "Failure"[a]	Returns "Failure"[a]	–	–	–
One child returns "Running"	Returns "Running"	Returns "Running"	Returns "Running"	Returns "Running"	Returns "Running"	Returns "Running"	Returns "Running"

ahere 'last child'' refers to the last non-terminated child; the child which has not yet returned its status to its parent—applicable only to nodes that tick several children in parallel.

named the behavior node standardly called Parallel, PARALLELALL, so as to make a distinction between the two Parallel behaviors. The definitions of the non-standard behavior nodes (contribution) are presented in Section 3.2. The returned behavior status of each standard BT node, based on the status behavior of their children nodes is summarized in Table 2.

Sequence behavior When a *Sequence* behavior starts, it starts its first child. When the currently running child succeeds, the Sequence behavior starts its next child, or succeeds if it is the last child. If any child behavior fails, the Sequence behavior fails at the same instant. We use gray signal shapes ▻ to represent Sequence behaviors.

Fallback behavior When a *Fallback* behavior starts, it starts its first child. When the currently running child fails, the Fallback behavior starts its next child, or fails if it is the last child. If any child behavior succeeds, the Fallback behavior succeeds at the same instant. We use gray octagon shapes ⬡ to represent Fallback behaviors.

Sequence and Fallback behaviors call their children sequentially, with execution order from left to right. They both have at most one running child at each instant.

ParallelAll behavior When a *ParallelAll* behavior starts, it starts all its children in parallel. It succeeds if and only if all its children have succeeded. If one child fails, the ParallelAll behavior fails and interrupts the rest of the children. We use gray ∧-shaped trapezia to represent ParallelAll behaviors. *Note: in common literature ParallelAll behaviors are referred to as Parallel behaviors.*

Inverter behavior An *Inverter* behavior has exactly one child. It starts its child when it starts, and is running when its child is running; succeeds when its child fails, and fails when its child succeeds. We represent Inverters by triangle arrow decorations.

3.2 | BT language semantics - Extended nodes

Here are provided three new BT node types, which we defined throughout our work towards the ODM definition and creation. They are an extension of the standard BT behaviors, as introduced in Section 3.1. These new behavior nodes were created so as to achieve the representation of redundant system behaviors—that is, ParallelAny, as well as fault detection and avoidance system mechanisms in the BT language. They also facilitate the automatic model transformation from FTs to BTs. The returned behavior status of each node, based on the status behavior of their children nodes is summarized in Table 2.

ParallelAny behavior When a *ParallelAny* behavior starts, it starts all its children in parallel. It fails once all its children have failed. If one child succeeds, ParallelAny behavior succeeds and interrupts the rest of the children. We represent ParallelAny behaviors by ∨-shaped gray trapezia.

Fault detection behavior A *Detect* behavior is an atomic behavior dedicated to detecting a fault. In this behavior, success means that it has detected the fault. Otherwise, it stays in the running mode and never fails. Fault detection behaviors are represented by diamonds.

The semantics of the BT depicted in Figure 4 are as follows. The top-most behavior is a fallback, that is, it tries to run its first child, and in case of failure, falls back to its next child. In this instance, the first executed behavior is "Operate system (nominal mode)". The nominal mode is implemented by a ParallelAll behavior, that runs the inverted "Detect fault" and "Control system" behaviors in parallel. When a fault is detected, the "Detect fault" succeeds, so its inverter fails, and thus the whole nominal mode behavior fails. This interrupts the "Control system" behavior. Similarly, if the "Control system" fails for some internal reason, the nominal mode fails and interrupts the "Detect fault" behavior as a result. When the nominal mode fails, the root fallback behavior starts the "Operate system (degraded mode)" behavior. Finally, the ODM methodology uses a type of behaviors that can be either atomic or composite, but their semantic is defined with respect to a specific fault event.

Fault event avoidance behavior A *Fault event avoidance* behavior is a behavior that should fail when the fault event occurs, and never succeed. It can be atomic or composite. All behaviors whose names start with "Avoid" are Fault event avoidance behaviors.

Fault event avoidance behaviors are always associated with a fault event from a FT. These fault events represent the failure of a function or of a fault detection mechanism. Therefore they do not always make sense from an operational point of view. Fault event avoidance behaviors are merely an artifact used as temporary translation between FTs and ODMs. Moreover, in a composite Fault event avoidance behavior, its children must be compatible with the fault avoidance specification, otherwise the BT is invalid, and its semantics hence undefined.

4 | METHODOLOGICAL PROPOSAL FOR BUILDING ODMs

The ODM methodological approach consists in defining a model that describes the system's operational procedures, with particular emphasis on fault diagnosis activities. In this section we provide a step-by-step description of the proposed methodology. Moreover, we define a variant of the BT language, created for the needs of the ODM method. The aim of this extension is to provide BT language semantics with the tools to express ODMs using FTs as input artifacts.

As illustrated in Figure 5, ODMs serve a double purpose. On the one hand, since they are concurrently created with the SE & SA/RAMS models, they provide feedback to system designers, specific to health monitoring and FDIR aspects. ODMs consequently contribute to improving the system's structural and behavioral design, as well as its and maintenance aspects.

The left side of Figure 5 portrays these interactions, between system architects, safety experts, the ODM design team (left side) and the operators (right side). Note that, the left part of the Figure portrays a group of *recursive processes*, which are initiated by the preliminary system architecture proposal, followed by a series of iterative activities, eventually leading to the final system design definition.

On the other hand, following system deployment, the ODM can immediately be used for system supervision and diagnosis. BTs are executable, hence the ODM can facilitate the design of test scenarios, distribute diagnosis objectives across the various activities, and ensure

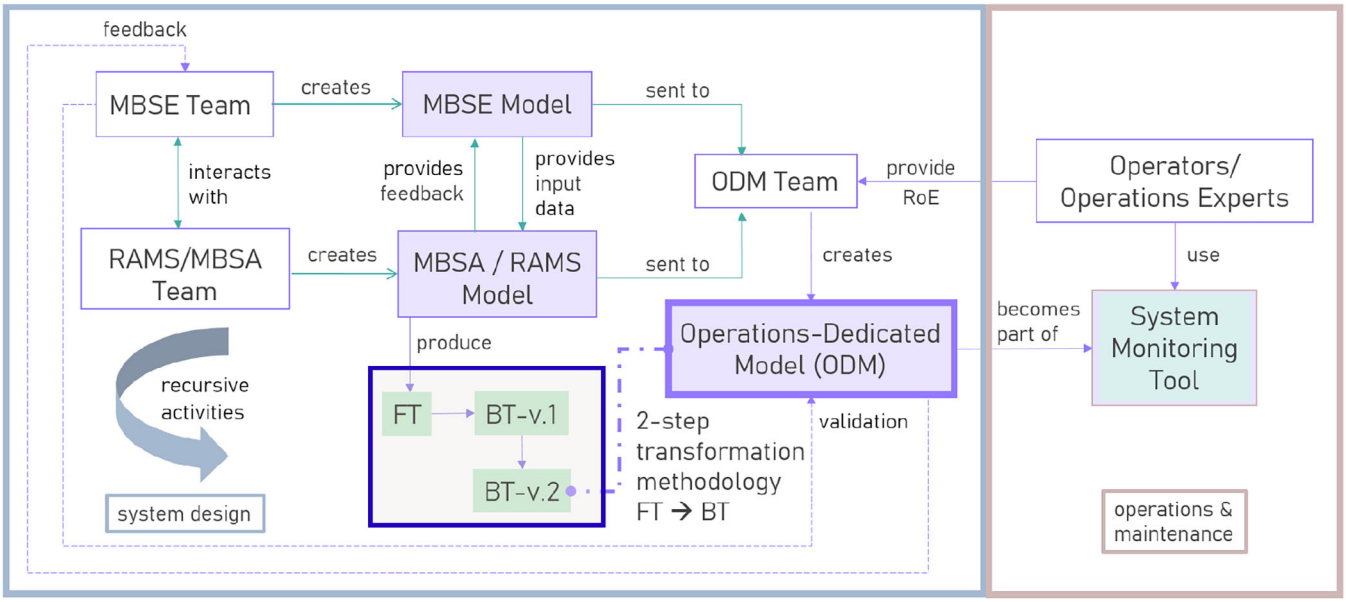


FIGURE 5 Overview of the proposed methodology for the ODM construction and exploitation. ODM, operations-dedicated model.

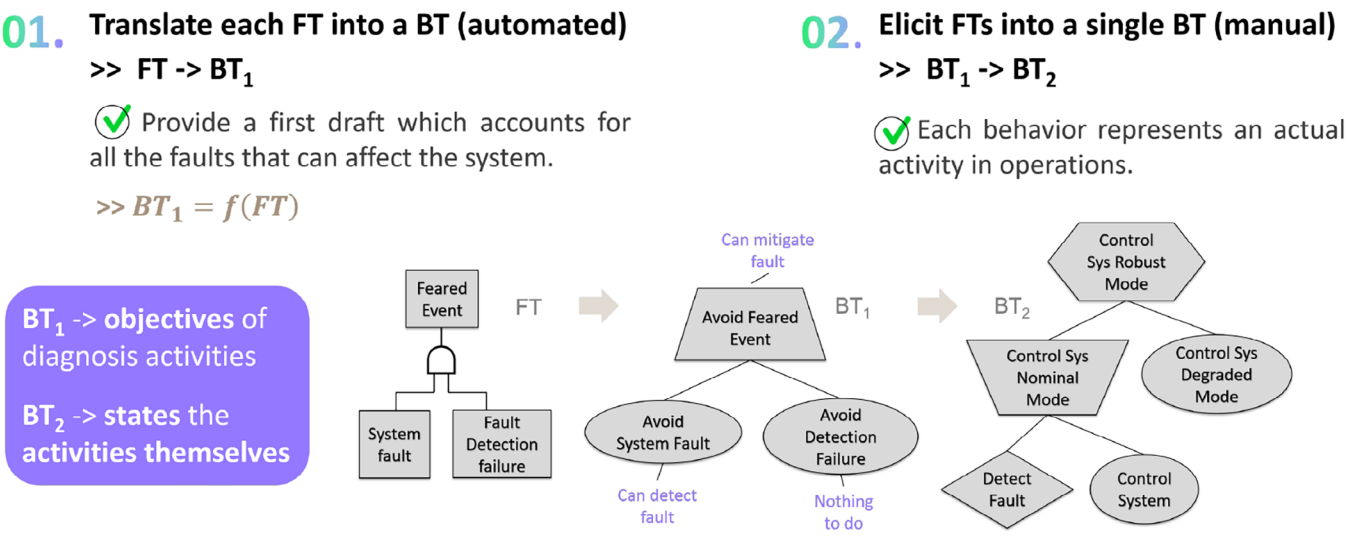


FIGURE 6 ODM creation method demonstration. ODM, Operations-dedicated model.

that operators are given realistic tasks. This process is illustrated at the right side of Figure 5.

As illustrated in Figure 6, our methodology for obtaining an ODM from a FT is composed of *two steps*:

1. Translate each FT into a BT (*BT-v.1*). This step is automated; the purpose is to provide a first draft which accounts for all the faults that can affect the system.
2. Elicit all BTs into a single BT (*BT-v.2*), in which each behavior represents an actual activity in operations. This step is done manually by a person with modeling skills, but more importantly operational experience.

For illustrative purposes, we assume that all fault events can be detected or mitigated, hence be included in the ODM as their corresponding operational activities. However, in general FTs may contain faults that cannot be detected nor mitigated (e.g., automatic monitoring not implemented), and therefore cannot be translated into any behavior. In this sense, FTs can contain information irrelevant to operations.

This is why we propose to construct ODMs from BTs through two steps, since: (i) an FT cannot consist an ODM as is, while (ii) additional new information must be elicited—in combination with FT data, to create a coherent and useful ODM.

During the *first step*, the FT is transformed into a BT as follows.

Fault tree transform *FT2BT* The transform of a FT into a BT is implemented by the function *FT2BT* defined on FT nodes as follows.

1. If *FTN* is a basic event named "Fault event *X*", then:
   FT2BT(*FTN*) is an atomic fault avoidance behaviour named "Avoid fault event *X*".
2. If *FTN* is an AND gate labelled "Fault event *X*" with children nodes FTN_1, FTN_2,..., then *FT2BT*(*FTN*) is a ParallelAny behaviour named "Avoid fault event *X*", with children behaviours $FT2BT(FTN_1)$, $FT2BT(FTN_2)$,...
3. If *FTN* is an OR gate labelled "Fault event *X*" with children nodes FTN_1, FTN_2,..., then *FT2BT*(*FTN*) is a ParallelAll behaviour named "Avoid fault event *X*", with children behaviours $FT2BT(FTN_1)$, $FT2BT(FTN_2)$,...

A pseudo-code for the *FT2BT* function is proposed below.

Fault tree transform *FT2BT* **pseudo-code**

```
def FT2BT(ft_node){
    bt_node_name = "Avoid " + ft_node.name

    if (ft_node.is_basic_event())
        return AtomicBehaviour(bt_node_name)

    bt_children = list()
    for (ft_child in ft_node.get_children())
        bt_children.append(FT2BT(ft_child))

    if (ft_node.is_AND_node())
        return ParallelAny(bt_node_name,
          bt_children)
    if (ft_node.is_OR_node())
        return ParallelAll(bt_node_name,
          bt_children)

    raise Error("Unknown FT node type")
}
```

During the *second step*, firstly, *Fault avoidance* behaviors are elicited to *"Operate"*-type behaviors. That is because BT-v.2 represents an executable system model. Then, *critical* Fault avoidance behaviors are elicited to a parent Sequence *"Operate without critical faults"* behavior, with children an Inverted *Fault detection* that is, *"Detect critical faults"* behavior, and an *"Operate with degraded faults"* behavior.

The second step of the ODM methodology is performed manually, whilst following several general guidelines. In many instances, a behavior named "Avoid something negative" does not represent a real activity in the system. The purpose of this step is to replace these unrealistic behaviors with other behaviors which account for:

- Fault tolerance and robust control.
- Fault mitigation activities.
- The fact that some activities occur in a predetermined sequence.
- The fact that some faults may have different observable effects depending on the system configuration, or its operational phase.

One important aspect of this step is that every transformation is documented and justified. This guarantees that every fault event considered in the FT is either directly accounted for in the BT, or handled by one or several precisely identified behaviors.

Regarding the expertise required for the building of the BT monitoring models, we propose that a dedicated team with operations background shall build the ODMs rather than the system architects or the safety analysts, so as to allow a distinct point of view. That would also ensure that the models will contain the necessary information for supervision and diagnosis only. Moreover, the construction of BTs with the PyTrees library is relatively easy and intuitive to code with. We hence believe that most system modeling engineers with no particular coding background, provided with sufficient training, guidance and documentation, could create ODMs. By not requiring particularly sophisticated skills prior to the initial training—other than system modeling and knowledge of operations, we can ensure that the monitoring model is easy to maintain throughout the years.

5 | CONCLUSIONS

In this paper we have introduced a novel approach demonstrating the potential of using BTs for system monitoring, as well as their limitations, based on the state-of-the-art and the current industrial needs. Monitoring tools for diagnosis during system operations are, to this day, still relying on knowledge acquired through previous experience as well as engineers' and operators' know-how, rather than the system design elements. For this reason we have presented a method towards the concurrent construction of a monitoring tool—along with the system design (SE and SA) models, which is tailored to the system under development. In our proposal, the way BTs are used for monitoring shall depend on the system and its operational environment, which impacts the way the monitoring tool is implemented and deployed. Our ODM approach is compatible with automated diagnosis approaches, whether model-based or data-based.

ODMs can be considered as reconstructed MBSA models built with an operational maintenance perspective, and with focus on FDD. The fact that ODM-based diagnostic tools can offer visibility on the faults' order of occurrence, is a great benefit over other traditional FDD tools, for example, tools based on FTAs, where the fault occurrence order

is not part of the modeling parameters. The latter is imperative when attempting to understand the fault propagation that took place and identify the single fault that triggered the series of events leading to the current situation.

Moreover, diagnostic tools based on ODMs can offer dedicated UI for operators, meant to facilitate their overview of the system architecture (how the system is built) and of its current state (what the system is currently doing). This can eventually reduce the response time to failure, leading to greater efficiency in diagnostic operations and increase in the satellites' Availability. Moreover, it can potentially reduce the operators' stress level during FDD activities.

On the downside, a threat associated to the ODM proposal is the non-adoption by the operators. Moreover, additional training might be considered necessary: on the on hand, for the operators, so as to learn how to use the new tools, and on the other, for the operations and diagnostics experts (future ODM team) to master the ODM methodology and BT semantics.

Regarding the types of faults ODMs can eventually integrate, that will depend on the maintenance actions. In our approach so far, faults are treated from system viewpoint. Other fault types, such as time-dependent (abrupt, incipient, intermittent) relate to a low-level, component-dependent system description, which is not the case for our modeling practises at this stage of the study.

The authors have validated the proposed approach through their collaboration with research laboratories that is, ONERA, LAAS-CNRS, aerospace companies that is, ADS and space agencies that is, CNES, as well as by the space operations community. Future work can contemplate the following perspectives: ODM method refinement; ODM integration in system monitoring/diagnosis tools; operators' training for the ODM building and exploitation; method deployment in the ongoing process of real systems' development. This would also contribute in the evaluation of the method's scalability (how large of a system the ODM can handle, and where the benefits of its usage remain significant). Another option would be the application of the ODM creation methodology in an existing, operational system, so as to compare the added value of the use of ODMs for FDD, compared to current practise.

ACKNOWLEDGMENTS

S2C Project, IRT Saint Exupéry, supported by the French National Research Agency (ANR).

CONFLICT OF INTEREST STATEMENT

The authors declare no conflict of interest.

DATA AVAILABILITY STATEMENT

Data available on request from the authors.

ORCID

Nikolena Christofi https://orcid.org/0000-0003-1249-0839
Claude Baron https://orcid.org/0000-0001-9573-7002
Xavier Pucel https://orcid.org/0000-0001-8747-0889

REFERENCES

1. Bitetti L, De Ferluc R, Mailland D, Gregoris G, Capogna F. Model based approach for RAMS analyses in the space domain with capella open-source tool. In: Papadopoulos Y, Aslansefat K, Katsaros P, Bozzano M, eds. *Model-Based Safety and Assessment*. Lecture Notes in Computer Science. Springer International Publishing; 2019: 18-31. ISBN 978-3-030-32872-6. https://doi.org/10.1007/978-3-030-32872-6_2
2. Henry D, Simani S, Patton RJ. *Fault Detection and Diagnosis for Aeronautic and Aerospace Missions*. Springer; 2010:91-128. ISBN 978-3-642-11690-2. https://doi.org/10.1007/978-3-642-11690-2_3
3. National Aeronautics and Space Administration. NASA 2022 Strategic Plan; NPD 1001.0D, Page 7. https://www.nasa.gov/ocfo/strategic-plan/
4. National Aeronautics and Space Administration. NASA 2018 Strategic Plan; Page 43. https://www.nasa.gov/ocfo/strategic-plan/
5. European Space Agency, ESA's Technology Strategy for Space19+; Version 1.2, September 2022, Page 52. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/ESA_s_Technology_Strategy_for_Space19
6. ISO15288, *ISO/IEC/IEEE 15288:2015 - Systems and Software Engineering - System Life Cycle Processes*. International Standards Organization (ISO), Geneva, Switzerland; 2015.
7. Aguilar-Savén RS. Business process modelling: review and framework. *Int J Prod Econ*. 2004;90:129-149. ISSN 0925-5273. https://doi.org/10.1016/S0925-5273(03)00102-6. Production Planning and Control.
8. Allweyer T. *BPMN 2.0: introduction to the standard for business process modeling*, 2016. ISBN-10: 383709331X.
9. Geambaşu CV. BPMN vs. UML activity diagram for business process modeling. In: *Proceedings of the 7th International Conference Accounting and Management Information Systems*. AMIS; 2012:934-945.
10. Herzner W, Sieverding S, Kacimi O, Böde E, Bauer T, Nielsen B. Expressing best practices in (risk) analysis and testing of safety-critical systems using patterns. In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE; 2014: 299-304. https://doi.org/10.1109/ISSREW.2014.24
11. Adedjouma M, Yakymets N. A framework for model-based dependability analysis of cyber-physical systems. In: *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*. IEEE; 2019: 82-89. https://doi.org/10.1109/HASE.2019.00022
12. Chen X, Jiao J. A fault propagation modeling method based on a finite state machine. In: *2017 Annual Reliability and Maintainability Symposium (RAMS)*. IEEE; 2017: 1-7. doi: https://doi.org/10.1109/RAM.2017.7889776
13. Zhu M, Brooks R. Comparison of petri net and finite state machine discrete event control of distributed surveillance network. *IJDSN*. 2009;5:480-501. https://doi.org/10.1080/15501320903048753
14. Ekanayake T, Dewasurendra D, Abeyratne S, Ma L, Yarlagadda P. Model-based fault diagnosis and prognosis of dynamic systems: a review. *Procedia Manuf*. 2019;30:435-442. ISSN 2351-9789. https://doi.org/10.1016/j.promfg.2019.02.060. Digital Manufacturing Transforming Industry Towards Sustainable Growth.
15. Colledanchise M, Ögren P. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Trans Rob*. 2017;33(2):372-389. ISSN 1941-0468. https://doi.org/10.1109/TRO.2016.2633567. Conference Name: IEEE Transactions on Robotics.
16. Bouissou M, Bon J-L. A new formalism that combines advantages of fault-trees and Markov models: Boolean logic driven Markov processes. *Reliab Eng Syst Saf*. 2003;82(2):149-163. ISSN 0951-8320. https://doi.org/10.1016/S0951-8320(03)00143-1
17. Aslansefat K, Kabir S, Gheraibia Y, Papadopoulos Y. Dynamic Fault Tree Analysis: State-of-the-Art in Modelling, Analysis and Tools. *Reliability Management and Engineering*. CRC Press; 2020:73-112. 06 2020. ISBN 9780429268922. doi: https://doi.org/10.1201/9780429268922-4

18. Tundis A, Garro A. On the reliability analysis of systems and SoS: the RAMSAS method and related extensions. *IEEE Syst J*. 2015;9:232-241. doi: https://doi.org/10.1109/JSYST.2014.2321617
19. Colledanchise M, Ögren P. How behavior trees modularize robustness and safety in hybrid systems. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE; 2014: 1482-1488. ISSN: 2153-0866. https://doi.org/10.1109/IROS.2014.6942752. ISSN: 2153-0866.
20. Colledanchise M, Marzinotto A, Dimarogonas DV, Ögren P. The advantages of using behavior trees in mult-robot systems. In: *Proceedings of ISR 2016: 47st International Symposium on Robotics*. VDE Verlag GmbH; 2016:1-8.
21. Klöckner A. Interfacing behavior trees with the world using description logic. In: *AIAA Guidance, Navigation, and Control (GNC) Conference*, Guidance, Navigation, and Control and Co-located Conferences, American Institute of Aeronautics and Astronautics; 2013. https://arc.aiaa.org/doi/10.2514/6.2013-4636
22. Rovida F, Grossmann B, Krüger V. Extended behavior trees for quick definition of flexible robotic tasks. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*; IEEE; 2017: 6793-6800. https://doi.org/10.1109/IROS.2017.8206598. ISSN: 2153-0866.
23. Colledanchise M, Parasuraman R, Ögren P. Learning of behavior trees for autonomous agents. *IEEE Trans Games*. 2019;11(2):183-189. ISSN 2475-1510. https://doi.org/10.1109/TG.2018.2816806. Conference Name: IEEE Transactions on Games.
24. Colledanchise M, Ögren P. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Trans Rob*. 2017;33(2):372-389. https://doi.org/10.1109/TRO.2016.2633567
25. Colledanchise M, Ögren P. Behavior Trees in Robotics and AI: An Introduction. arXiv:1709.00084 [cs] 2018. https://doi.org/10.1201/9780429489105. arXiv: 1709.00084.
26. García S, Pelliccione P, Menghi C, Berger T, Bures T. High-level mission specification for multiple robots. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2019, New York, NY, USA. Association for Computing Machinery; 2019: 127-140. ISBN 978-1-4503-6981-7. https://doi.org/10.1145/3357766.3359535
27. PyTrees. Library Documentation. https://py-trees.readthedocs.io/en/devel/. Last accessed March 2023.
28. BehaviorTree.CPP. Behavior Trees Library in C++. https://github.com/BehaviorTree/BehaviorTree.CPP. Last accessed March 2023.
29. AurynRobotics. Groot - Graphical Editor to create Behavior Trees. Compliant with BehaviorTree.CPP. https://github.com/BehaviorTree/Groot. Last accessed March 2023.

How to cite this article: Christofi N, Baron C, Pucel X, Pantel M, Canu D, Ducamp C. How to early integrate operational diagnosis objectives in model-driven engineering processes: A methodological proposal based on fault and behavior trees. *Systems Engineering*. 2024;27:585–597. https://doi.org/10.1002/sys.21740