AMULET: a Mutation Language Enabling Automatic Enrichment of SysML Models

BASTIEN SULTAN, Mines Saint-Etienne, CEA, Leti, Centre CMP, France
LÉON FRÉNOT, École Normale Supérieure de Lyon, France
LUDOVIC APVRILLE, LTCI, Télécom Paris, Institut Polytechnique de Paris, France
PHILIPPE JAILLON, Mines Saint-Etienne, CEA, Leti, Centre CMP, France
SOPHIE COUDERT, LTCI, Télécom Paris, Institut Polytechnique de Paris, France

SysML models are widely used for designing and analyzing complex systems. Model-based design methods often require successive modifications of the models, whether for incrementally refining the design (e.g. in agile development methods) or for testing different design options. Such modifications, or mutations, are also used in mutation-based testing approaches. However, the definition of mutation operators can be a complex issue and applying them to models is sometimes performed by hand: this is time consuming and error prone. The paper addresses this issue thanks to the introduction of AMULET, the first mutation language for SysML. AMULET encompasses the modifications targeting SysML block and state-machine diagrams, and is supported by a compiler the paper presents. This compiler is integrated in TTool, an open-source SysML toolkit, enabling the full support of design methods including model design, mutation and verification tasks in a unique toolkit. The paper also introduces two case-studies providing concrete examples of AMULET use for modeling vulnerabilities and cyber attacks, and highlighting the benefits of AMULET for SysML mutations.

CCS Concepts: • **Software and its engineering → Compilers**; **Domain specific languages**; **System modeling languages**; *Formal methods*; *Software development techniques*; • **Computing methodologies → Model development and analysis**.

Additional Key Words and Phrases: Formal modeling, Formal verification, Mutations, SysML, Model enrichment

1 INTRODUCTION

SysML models, in particular, have become a prevalent tool in the design and analysis of complex systems. They offer a robust method for formalizing the knowledge of engineers, and can be utilized for various behavioral analyses, including simulations and formal verification, depending on model transformations. Traditionally, these analyses are employed during the design stages or later in a system's life cycle. There, the model acts as a digital twin, and is utilized, for example, to evaluate potential upgrades [12, 24]. However, a model is only a representation of a system's architecture and behavior at a specific point in time. Systems, particularly those that incorporate software components, undergo various changes throughout their life cycle. These changes can be deliberate, such as functional upgrades or security measures, or unexpected, such as cyberattacks or failures. Regardless of the nature of the change, such a system evolution affects the knowledge we have of the system.

Authors' addresses: Bastien Sultan, bastien.sultan@emse.fr, Mines Saint-Etienne, CEA, Leti, Centre CMP, Saint-Étienne, France, F–42023; Léon Frénot, leon.frenot@ens-lyon.fr, École Normale Supérieure de Lyon, 15, parvis René Descartes, Lyon, France, F–69342; Ludovic Apvrille, ludovic.apvrille@telecom-paris.fr, LTCI, Télécom Paris, Institut Polytechnique de Paris, Sophia-Antipolis, France; Philippe Jaillon, philippe.jaillon@emse.fr, Mines Saint-Etienne, CEA, Leti, Centre CMP, Saint-Étienne, France, F–42023; Sophie Coudert, sophie.coudert@telecom-paris.fr, LTCI, Télécom Paris, Institut Polytechnique de Paris, Sophia-Antipolis, France.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1539-9087/2023/9-ART $15.00
https://doi.org/10.1145/3624583

Just Accepted

As a result, models that are no longer up-to-date need to be updated or "mutated" to regain their accuracy by incorporating this specific evolution: this is the objective of a "model mutation".

The need to enrich a system arises in various practical scenarios, and we will focus on three examples to illustrate the motivation and benefits of our contribution. The first scenario is for situations where the model designer is dealing with changes that are out of its will. In some cases, system changes may involve elements that were not designed by the model designer (for example, pre-existing solutions such as software patches for commercial off-the-shelf products or attack vectors). In such cases, the mutations will need to account for these pre-existing elements (cf.Fig. 1 (a)). The second scenario involves situations where the system changes, such as functional upgrades or countermeasures, are actively designed and developed by the model designer. In this case, the mutations will typically capture the specific changes the designer intends to deploy, as illustrated in Fig. 1(b). This scenario is prone to involve several successive mutations in order to adjust the changes after model verification and simulation, especially if the designer applies agile development methods. Lastly, a third scenario is when engineers model the system in an iterative way during the design stage, making continuous incremental improvements to the models. In this case, mutations depict these incremental improvements towards achieving a more accurate model, as illustrated in Fig.1(c). Model enrichment is a crucial task in all these scenarios, but depending on the system's complexity and the modeling language used, it can be a time-consuming and error-prone process for model designers, especially because applying existing model-based design methods can result in a large number of enriched models. For example, in [24] the authors generate up to 47 mutated SysML models to evaluate the impact of four security countermeasures on a rover swarm system.

The paper introduces three contributions that aim at facilitating the enrichment of SysML models:

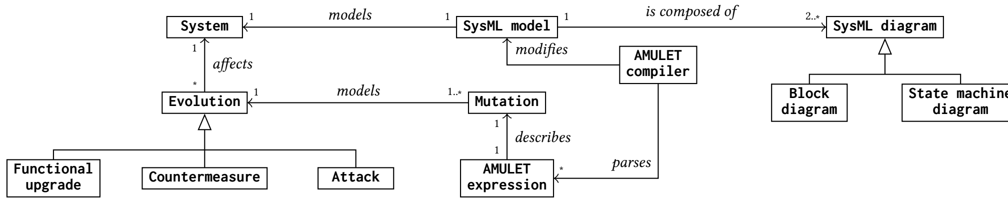
(1) SysML mutation operators are mathematically defined, covering changes both at architectural and behavioral level.
(2) A new language, called AMULET, is introduced. AMULET provides a syntax for describing mutations to be applied to SysML models[1].
(3) A dedicated compiler is introduced. It takes as input a SysML model and a set of mutations, and generates the resulting SysML model. This compiler fully supports AMULET and has been implemented as a new feature of the open-source SysML modeling and verification toolkit TTool[2].

These contributions completes the incremental model-checking algorithms we introduced in [8], that lower the algorithmic complexity of the proofs when verifying a mutated SysML model by reusing the previous proof results. Indeed, by integrating AMULET and incremental model-checking within the framework of model-based design methods the time-consuming tasks of modifying and verifying successive model increments are streamlined.

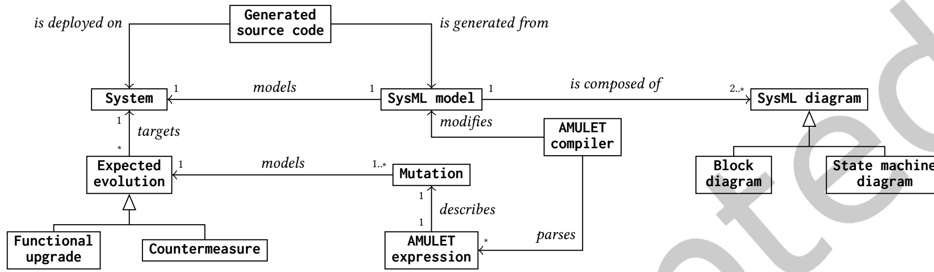
The rest of the paper is organized as follows. Section 2 provides an overview on the related works. Sections 3 and 4 introduces our theoretical and practical contributions: Figure 1 illustrates the conceptual relationships between the key concepts discussed in these sections. Indeed, we always consider a system under design or validation modeled with a SysML model. This model is composed of a block diagram and several state-machine diagrams such as mathematically defined in Subsection 3.1. The system is subject to evolutions that are modeled with a set of mutations: mutations are set functions such as defined in Subsection 3.3. These functions are described with AMULET expressions (defined in Subsection 3.4) that are then parsed by the AMULET compiler (introduced in Section 4) that modifies the SysML model accordingly. Practical application of these concepts are introduced in Section 5 that introduces two case-studies providing results for assessing the relevance of our contributions. Last, Section 6 discusses the strengths and limitations of our contributions and concludes the paper.

[1]AMULET mutations support three SysML diagram types: state-machine, block definition and internal block diagrams.
[2]https://ttool.telecom-paris.fr

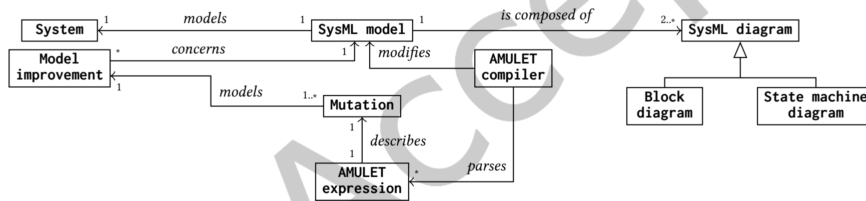
Just Accepted



(a) Generic system evolution case



(b) “Digital twin” system evolution case



(c) Model refinement case

Fig. 1. SysML models, mutations, AMULET and modeled objects: conceptual relations

Just Accepted

2 RELATED WORKS

2.1 Concerning model mutation-based methods

The concept of automata mutation [27] has led to the widespread use of mutation for testing purposes, for evaluating both (formal) models and source code [4]. The traditional mutation testing process involves the following stages: (1) a set of mutation operators is first defined, (2) these operators are systematically applied to a base model (or program) to generate a set of mutated versions, (3) the resulting set of mutated models (or programs) are tested and/or verified in order to assess the capability of the test cases or verification process in eliminating the mutants that don’t comply with the system requirements. In other terms, the aim of a mutation testing process is to determine the implementation errors a set of test cases can detect. For instance, Aichernig et al. [2] introduces such a process for mutation testing relying on UPPAAL timed automata. In a further study [1], they define a set of mutation operators targeting UML state-machine diagrams (SMDs) and introduce a mutation-based test method consisting in applying systematically each mutation operator to each relevant element of the initial SMD (e.g., each operator targeting a transition is applied to each transition, etc.) in order to create

a set of mutant models that is then used to evaluate the detection capabilities of three test suites. UML SMDs also are the target of the method proposed by Mi et al. [18] consisting in the three stages described above. This paper defines a set of mutation operators for UML SMDs that provide a wide coverage of possible SMDs modifications (addition/deletion/modification of states, transitions, guards, actions and events). More recently, Alenazi et al. [4] introduced a SysML mutations-based method aiming at improving automated requirement traceability (traceability analysis returning a link between a LTL property and a subset of states/transitions of a SMD that satisfy it). Based on SysML SMDs mutation operators and a base (correct) SMD, this method generates a wide set of faulty models and checks them against a given property in order to identify the faulty models that still satisfy it. Then, an algorithm is employed to compare the set of mutants that satisfy the specified system requirement against the mutants that do not, in order to minimize the number of "false positives" that are incorrectly identified as satisfying the requirements. Relatedly, mutation of OCL [19] specifications have been investigated in several papers, including [10] that focuses on test data generation through the mutation of OCL specifications. Aichernig et al. [3, 22] used OCL mutations (focusing on pre-conditions and post-conditions) for generating test cases for OCL specifications, and Jin and Lano [15] introduce a set of OCL mutation operators for easying mutation testing. Their mutation operators mainly focus on the "negation of the original specification", i.e., $=$ are replaced with $\neq$ in mutated conditions, $\leq$ with $>$, etc. Our needs are partially addressed by these works since OCL can be used to specify constraints for SysML models. However, OCL semantics are primarily geared towards expressions involving pre-existing model elements whereas we require more extensive SysML model modifications, e.g. block and attribute addition and deletion.

It has to be noted that all these approaches — excepting OCL mutation approaches — focus on the mutations of "atomic" automata, i.e., on one single timed-automata or UML/SysML SMDs. Therefore, the proposed mutation operators don't encompass mutation for more complex models such as networks of timed automata (NTAs) or SysML block diagrams. However, these mutations are of prime importance when it is needed to represent major changes in a modeled system (e.g., the addition of a network switch, or the removal of a communication channel between two components). For these reasons, Sultan et al. [23, 26] proposed a definition for the mutation of a network of UPPAAL timed-automatas in order to model vulnerabilities, attacks and countermeasures targeting cyber-physical systems. Yet atomic mutation operators are not mathematically defined and UPPAAL syntax obviously differs from SysML one. Therefore, we need to introduce new SysML mutations operators targeting block diagrams as well as SMDs[3].

2.2 Concerning mutation automation and mutation languages

The surveyed papers don't systematically explain if the application of model mutations is automated and how they achieve automation. Alenazi et al. [4] rely on a Python script to generate mutants deriving from their base model. Sometimes, it is explicitely stated that models enrichments are performed by hand [24]. In addition, even if mutation operators are defined, none of them proposes a language supporting these operators excepting [23] that introduces a syntax for describing mutations targeting UPPAAL NTAs. However, the compiler for this mutation language has not been fully implemented. In a context of mutation-based testing, Aranega et al. [9] propose an approach for automating the test set improvement stage (i.e., improving test cases in order to enable them to detect more mutants). Their approach is language-independant. The paper introduces metamodels of generic mutation operators in order to analyze mutant models (i.e., identify the mutation operators that have been applied on each mutant model). However, the mutant models generation is out of the scope of this study and the operators metamodels cannot be used off-the-shelf to generate SysML mutated models.

Actually, the major contribution in this field has been introduced by Gómez-Abajo et al. [13, 14] who have designed a language-independent mutation language called Wodel and a framework supporting it. The Wodel

[3]Even if their mathematical definition is different, SMD mutation operators will obviously match with some operators introduced in [1, 4, 18].

Just Accepted

framework enables the users to (1) define a metamodel of the desired modeling or programming language (2) define their own mutation operators with respect to this metamodel, using the Wodel language and (3) generate mutants by applying these operators on a set of entry models (or programs). A major strength of Wodel is that the mutation description language and the framework are fully language-independent, i.e., users can generate mutants of any model or program as long as they provide the relevant syntactic metamodel. Therefore, using Wodel to generate mutants requires prior definition of the target language metamodel and its mutation operators. Thus, users must precisely capture the abtraction levels of the language, and correctly define the mutation operators. For some languages, it may be a difficult task. For instance, in some SysML profiles, modifying a single block in one diagram can trigger changes across diagrams, as demonstrated in Subsection 3.3. Also (see definition ??), the deletion of a signal definition from a block implies the deletion, for each port connection between this block and an other one, of the signal associations involving the deleted signal. This analysis—that may be complex—falls to the user, and AMULET addresses this difficulty thanks to specific operators that already take into account the needed cross-elements propagations. In addition, AMULET is directly integrated in a SysML toolkit where the users can already design and verify the models, therefore no external tool is needed and the complete design/mutation/validation process can be carried out in a unique framework. Yet, AMULET is obviously less polyvalent than Wodel since it targets only one modeling language.

3 SYSML MUTATIONS: THE THEORETICAL WAY

This section introduces our theoretical contributions. Prior to defining our mutation operators and language, we first need to formally define the SysML diagrams they target: Subsection 3.1 focuses on these preliminary definitions[4]. The rest of the section introduces our SysML mutation operators and our language. It firstly provides an easy-to-read summary giving for each operator a simplified notation and a short description of its effect (Subsection 3.2). Then, mathematical definitions of these operators are provided (Subsection 3.3): these definitions describe the full semantics of each operator with respect to the SysML formal semantics given in Subsection 3.1. Finally, the mutation language supporting these operators is introduced (Subsection 3.4).

3.1 SysML models: mathematical definitions

Definition 3.1 (Types, Attributes, Expressions and Signals).

- Types = $\{\mathbb{B}\text{ool}, \mathbb{Z}, \mathbb{N}\}$
- Attr is a set of *attributes*, typed by $type$: Attr $\rightarrow$ Types.
- *Expressions* are usual integer and boolean expressions over attributes. They are typed in the usual way.
- Profiles = $\{(t_1, \cdots, t_n) \mid n \in \mathbb{N} \land \forall 1 \leq i \leq n,\ t_i \in \text{Types}\}$.
- Sign = InSign $\sqcup$ OutSign is[5] a set of *signals*, typed by *profile* : Sign $\rightarrow$ Profiles.
  Signals may be *input* or *output* signals: InSign contains *input* signals and OutSign contains *output* signals.

Definition 3.2 (Basic Sets and associated Abstract Syntax).

- Meth is a set of *methods*, typed by *profile* : Meth $\rightarrow$ Profiles.
- $m(e_1, \ldots, e_n)$ is a *method call*, where m is a method and $e_1, \ldots, e_n$ are expressions respecting the profile of m.
- Port is a set of untyped *ports* enabling the connection of signals between *blocks* over *links*. A *link* is a pair of ports having a communication semantics.

[4]Note that we previously introduced preliminary Definitions 3.1, 3.3, 3.4 and 3.7 in [25], and that Definitions 3.2 and 3.5 are enhanced versions of two definitions provided in the same paper.
[5]"$\sqcup$" denotes the disjoint union.

Just Accepted

- CommSemantics = $\{(\{synchronous\}\times\{broadcast, unicast\}\cup\{asynchronous\}\times\{lossy, unlossy\}\times\{blocking\ write, non-blocking\ write\})\times\{public, private\}\}$ is the set of communication semantics. A link may be *public* or *private*: a *public* link may be eavesdropped by an attacker. It may be *synchronous* or *asynchronous*: when a signal instance is exchanged over a *synchronous* link, both its transmission and reception take place simultaneously. A *synchronous* link l is a *broadcast* link if multiple subblocks[6] of a block linked through l can receive a same signal instance sent over l, or an *unicast* link otherwise. An *asynchronous* link may be *lossy* or *unlossy*: a signal instance sent over a *lossy* link may never be received, while a signal instance sent over an *unlossy* link will eventually be received. An *asynchronous* link may also be *blocking write* or $non-blocking\ write$. Signal instances transmitted over an *asynchronous* link are stored in a FIFO buffer: a *blocking write* link ensures that no new signal instance can be sent over the link if the buffer is full, while in a $non-blocking\ write$ link new signal instances are dropped if the buffer is full.
- We consider four kinds of *actions*:
  - *assignments:* $a := e$ where a is an attribute and e an expression of the same type.
  - *Random assignments:* $a :=?$ where a is an attribute.
  - *Sending signals:* $send_s(e_1, \ldots, e_n)$, where s is an output signal and $e_1, \ldots, e_n$ are attributes respecting the profile of s.
  - *Receiving signals:* $receive_s(e_1, \ldots, e_n)$, where s is an input signal and $e_1, \ldots, e_n$ are attributes respecting the profile of s.

Definition 3.3 (State Machine Diagram).
A *state machine diagram* is a directed (control flow) graph $smd = (s_0, S, T)$, where:

- S is a set of *states*.
- $s_0 \in S$ is the initial state.
- T is a set of *transitions* $t = \langle s_{start}, after, condition, actions, s_{end} \rangle$ where:
  - $after = time \in \mathbb{N}$ constrains the delay before firing t.
  - $s_{start}, s_{end} \in S^2$ are respectively the source and target states of t.
  - $condition$ is a boolean expression that must be true to enable t[7].
  - actions is a sequence of actions/method calls executed when t is fired.

attr(*smd*) (resp. *meth*(*smd*), *sign*(*smd*), *insign*(*smd*), *outsign*(*smd*)) denotes the set of attributes (resp. methods, signals, input signals, output signals) used in *smd*.

A state machine diagram is syntactically correct if all states are reachable from s_0 (by some syntactic path on transitions).

Definition 3.4 (Block Description).
A *block description* is a 6-uple $D = \langle A, M, P, S_i, S_o, smd \rangle$ where $A \subset$ Attr, $M \subset$ Meth, $S_i \subset$ InSign, $S_o \subset$ OutSign, $P \subset$ Port, *smd* is a state machine diagram, and all these sets are finite.

It is syntactically correct if *smd* is syntactically correct, $attr(smd) \subseteq A$ and $meth(smd) \subseteq M$.

Like in [6], we consider here a unique SysML block diagram that merges the block definition diagram (BDD) and the internal block diagram (IBD) defined in the SysML specification [20].

Definition 3.5 (SysML Block Diagram).
A *SysML block diagram* is a 6-uple $\langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ where:

- $\mathcal{B}$ is a finite set of *blocks*.

[6] See definition 3.5 for the definition of a *subblock*.

[7] Note that in our semantics, when $condition$ holds true in two transitions originating from the same state, our system allows non-determinism. This non-deterministic choice enables either of these two transitions to be fired.

Just Accepted

- The function d assigns a description to each block in the set $\mathcal{B}$. For $B \in \mathcal{B}$, we denote $d(B)$ with $\{A_B, M_B, P_B, S_{iB}, S_{oB}, smd_B\}$, $\bigsqcup_{B\in\mathcal{B}} P_B$ with $\mathcal{P}$, $\bigsqcup_{B\in\mathcal{B}} S_{o_B}$ with $\mathcal{S}_o$ and $\bigsqcup_{B\in\mathcal{B}} S_{i_B}$ with $\mathcal{S}_i$.
- $\mathcal{L} \subset \mathcal{P} \times \mathcal{P}$ is a set of *links*. It is an irreflexive and antisymmetric partial injection.
- The function $\sigma : \mathcal{L} \to$ CommSemantics assigns a communication semantics to each link.
- $C \subseteq \mathcal{L} \times \mathcal{S}_o \times \mathcal{S}_i$ is a set of *connections* $\langle\langle p_o, p_i\rangle, s_o, s_i\rangle$ such that p_o and s_o belong to the same block, and p_i and s_i belong to the same block.
- $\mathcal{R} \subset \mathcal{B} \times \mathcal{B}$ is a block containment relation such that
  - its transitive closure $\mathcal{R}^*$ is a noetherian order.
  - its inverse relation $\mathcal{R}^{-1}$ is a function.

  When $\langle B_1, B_2\rangle \in \mathcal{R}$, we say that B_1 "contains"/"is a superblock of" B_2 and that B_2 "is contained by"/"is a subblock of" B_1.

Remark: the block containment relation enables to define a structural hierarchy between blocks and has a a semantic implication on the state-machine diagrams of the contained blocks. Indeed, as stated in definition 3.7, the state-machine diagram of a block can use send/receive actions on signals belonging to the set of signals of its superblock, of the superblock of its superblock, and so on.

Definition 3.6 (Set of all contained blocks of a block B).
Let $\mathcal{M} = \langle\mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R}\rangle$ be a block diagram and $B \in \mathcal{B}$. $sub_B = \{B' \in \mathcal{B} \mid B\mathcal{R}^*B'\}$ denotes the set of all (directly or indirectly) contained blocks of B.

Notice that the constraints on $\mathcal{R}$ ensure that $\{B\} \cup sub_B$ is a finite tree with B as root.

Definition 3.7 (Syntactically Correct SysML Block Diagram).
Let $\mathcal{M} = \langle\mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R}\rangle$ be a SysML block diagram. $\mathcal{M}$ is syntactically correct if and only if:

(1) $\forall\langle\langle p_o, p_i\rangle, s_o, s_i\rangle \in C,\ profile(s_o) = profile(s_i)$.
(2) $\forall B \in \mathcal{B}$, $d(B)$ is syntactically correct and $\forall s_x \in sign(smd_B)$,
(2a) $\exists!\langle\langle p_o, p_i\rangle, s_o, s_i\rangle \in C,\ s_x = s_o \vee s_x = s_i$.
(2b) $\exists B' \in \mathcal{B},\ (B', B) \in \mathcal{R}^* \wedge (s_x \in S_{oB'} \vee s_x \in S_{iB'})$.

In the previous definition:

- condition (1) ensures that each pair of signals connected over a link involves signals of the same profile: a signal s_o carrying an integer attribute and two boolean attributes shall be connected to an input signal s_i carrying the same attributes in the same order.
- condition (2a) ensures that each signal instanciated in a state-machine diagram is connected to another signal and that this signal cannot be connected to several signals.
- condition (2b) ensures that each signal used in smd_B belongs to the set of signals of B, or the superblock of B, or the superblock of the superblock of B, etc.

3.2 SysML Mutations: operators summary

Mutation operators are classified according to their scope: the block diagram mutation operators (see Table 1) modify the composition of the block diagrams without affecting the blocks, and the block mutation operators (see Table 2) modify the description of the blocks.

3.3 SysML mutations: full mathematical definitions

For the needs of the definitions given in this section, we introduce the following notations:

- $\mathfrak{M}$ is the set of all SysML block diagrams.

Just Accepted

Table 1. Overview of block-diagram mutation operators

Mutation Operator	Short notation	Description
$addBlock(\mathcal{M}, B)$	$\mathcal{M} \xrightarrow{\mathfrak{B}^{+}(B)} \mathcal{M}'$	Adds block B to block diagram $\mathcal{M}$.
$delBlock(\mathcal{M}, B)$	$\mathcal{M} \xrightarrow{\mathfrak{B}^{-}(B)} \mathcal{M}'$	Deletes block B from block diagram $\mathcal{M}$.
$addLink(\mathcal{M}, (p_1, p_2), cs)$	$\mathcal{M} \xrightarrow{\mathfrak{L}^{+}(p_1,p_2),cs} \mathcal{M}'$	Adds a link between two ports p_1 and p_2 belonging to one or two block(s) of block diagram $\mathcal{M}$. The communication semantics of this new link is given by cs.
$delLink(\mathcal{M}, (p_1, p_2))$	$\mathcal{M} \xrightarrow{\mathfrak{L}^{-}(p_1,p_2)} \mathcal{M}'$	Deletes the link between the ports p_1 and p_2 from block diagram $\mathcal{M}$.
$addConnection(\mathcal{M}, (p_o, p_i), s_o, s_i)$	$\mathcal{M} \xrightarrow{\mathfrak{C}^{+}((p_o,p_i),s_o,s_i)} \mathcal{M}'$	Adds a connection between signals s_o and s_i over the link between the ports p_o and p_i.
$delConnection(\mathcal{M}, (p_o, p_i), s_o, s_i)$	$\mathcal{M} \xrightarrow{\mathfrak{C}^{-}((p_o,p_i),s_o,s_i)} \mathcal{M}'$	Deletes the connection between signals s_o and s_i over the link between the ports p_o and p_i.
$addContainement(\mathcal{M}, (B_1, B_2))$	$\mathcal{M} \xrightarrow{\mathfrak{R}^{+}(B_1,B_2)} \mathcal{M}'$	Adds a containment relation between blocks B_1 and B_2 such that B_2 is a subblock of $B1$.
$delContainement(\mathcal{M}, (B_1, B_2))$	$\mathcal{M} \xrightarrow{\mathfrak{R}^{-}(B_1,B_2)} \mathcal{M}'$	Deletes the containment relation between blocks B_1 and B_2.

Table 2. Overview of block mutation operators

Mutation Operator	Short notation	Description
$addAttribute(\mathcal{M}, B, a)$	$\mathcal{M} \xrightarrow{\mathsf{Attr}^{+}(B,a)} \mathcal{M}'$	Adds attribute a to block B.
$delAttribute(\mathcal{M}, B, a)$	$\mathcal{M} \xrightarrow{\mathsf{Attr}^{-}(B,a)} \mathcal{M}'$	Deletes attribute a from block B.
$addInputSignal(\mathcal{M}, B, s)$	$\mathcal{M} \xrightarrow{\mathsf{InSign}^{+}(B,s)} \mathcal{M}'$	Adds input signal s to block B.
$addOutputSignal(\mathcal{M}, B, s)$	$\mathcal{M} \xrightarrow{\mathsf{OutSign}^{+}(B,s)} \mathcal{M}'$	Adds output signal s to block B.
$delSignal(\mathcal{M}, B, s)$	$\mathcal{M} \xrightarrow{\mathsf{Sign}^{-}(B,s)} \mathcal{M}'$	Deletes signal s from block B.
$addState(\mathcal{M}, B, s)$	$\mathcal{M} \xrightarrow{\mathsf{State}^{+}(B,s)} \mathcal{M}'$	Adds state s to the state-machine diagram of block B.
$delState(\mathcal{M}, B, s)$	$\mathcal{M} \xrightarrow{\mathsf{State}^{-}(B,s)} \mathcal{M}'$	Deletes state s from the state-machine diagram of block B.
$addTrans(\mathcal{M}, B, t)$	$\mathcal{M} \xrightarrow{\mathsf{Trans}^{+}(B,t)} \mathcal{M}'$	Adds transition t to the state-machine diagram of block B.
$delTrans(\mathcal{M}, B, t)$	$\mathcal{M} \xrightarrow{\mathsf{Trans}^{-}(B,t)} \mathcal{M}'$	Deletes transition t from the state-machine diagram of block B.

- $\mathfrak{B}$ is the set of all blocks.
- Attr is the set of all attributes.
- Sign is the set of all signals.
- InSign (resp. OutSign) is the set of all input (resp. output) signals.
- Port is the set of all ports.
- States is the set of all states.
- Trans is the set of all transitions.

Just Accepted

- Given a SysML block diagram $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle \in \mathfrak{M}$, $\forall B \in \mathcal{B}$, B can be used as a shorthand for its description $d(B)$.

3.3.1 Mutations at block-diagram level. We define below all the mutations updating the composition of the sets $\mathcal{B}$, $\mathcal{L}$, C and $\mathcal{R}$, but without modifying the definition of the blocks (attributes, methods, etc.) and their related state-machine diagrams. Eight mutations are formalized in this section:

- Definition 3.8[8] (resp. 3.9) defines the addition (resp. deletion) of a block to (resp. from) a SysML block diagram.
- Definition 3.10 (resp. 3.11) defines the addition (resp. deletion) of a link between two blocks.
- Definition 3.12 (resp. 3.13) defines the addition (resp. deletion) of a signal connection between two blocks.
- Definition 3.14 (resp. 3.15) defines the addition (resp. deletion) of a containment relationship between two blocks.

Definition 3.8 (Block Addition).
A *block addition* is a function

$$\begin{aligned} addBlock: \quad & \mathfrak{M} \times \mathfrak{B} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$: $\mathcal{M}' = \langle \mathcal{B} \cup \{B\}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$.

Definition 3.9 (Block Deletion).
A *block deletion* is a function

$$\begin{aligned} delBlock: \quad & \mathfrak{M} \times \mathfrak{B} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and if we denote by sub_B the set of all sublocks of B and with $P_{sub_B} = \bigsqcup_{B' \in sub_B} P_{B'}$ the set of ports of the blocks of sub_B, $\mathcal{M}' = \langle \mathcal{B}', d, \mathcal{L}', \sigma', C', \mathcal{R}' \rangle$, where:

- $\mathcal{B}' = \mathcal{B} \setminus (\{B\} \cup sub_B)$.
- $\mathcal{L}' = \mathcal{L} \cap \mathcal{P}'^2$, where $\mathcal{P}' = \bigsqcup_{\beta \in \mathcal{B}'} P_\beta$.
- $\begin{aligned} \sigma' \ : \ & \mathcal{L}' \rightarrow \mathsf{CommSemantics} \\ & l \mapsto \sigma(l) \end{aligned}$
- $C' = \{\langle l, s_o, s_i \rangle \in C | l \in \mathcal{L}'\}$.
- $\mathcal{R}' = \mathcal{R} \cap \mathcal{B}'^2$.

Note that $\mathcal{M}' = \mathcal{M}$ if $B \notin \mathcal{B}$ since $\mathcal{B}$ and $\{B\} \cup sub_B$ are obviously two disjoint sets.

Definition 3.10 (Link Addition).
A *link addition* is a function

$$\begin{aligned} addLink: \quad & \mathfrak{M} \times \mathsf{Port}^2 \times \mathsf{CommSemantics} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, (p_o, p_i), cs) \mapsto \mathcal{M}' \end{aligned}$$

[8] Definition 3.8 is an enhanced version of a definition we have previously indroduced in [25].

Just Accepted

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$:

$$\begin{cases} \mathcal{M}' = \langle \mathcal{B}, d, \mathcal{L} \cup \{\langle p_o, p_i \rangle\}, \sigma', C, \mathcal{R} \rangle & \text{if:} \\ & - \langle p_o, p_i \rangle \in \mathcal{P}^2 \\ & - p_o \neq p_i \\ & - \neg\exists \langle p, p' \rangle \in \mathcal{L}, p = p_o \vee p = p_i \vee p' = p_o \vee p' = p_i \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where $\forall l \in \mathcal{L}, \sigma'(l) = \sigma(l) \wedge \sigma'(\langle p_o, p_i \rangle) = cs$.

Definition 3.11 (Link Deletion).
A *link deletion* is a function

$$delLink : \quad \mathfrak{M} \times \mathsf{Port}^2 \to \mathfrak{M}$$
$$(\mathcal{M}, (p_o, p_i)) \mapsto \mathcal{M}'$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$, $\mathcal{M}' = \langle \mathcal{B}, d, \mathcal{L}', \sigma, C', \mathcal{R} \rangle$, where:

- $\mathcal{L}' = \mathcal{L} \setminus \{\langle p_o, p_i \rangle\}$.
- $C' = \{\langle l, s_o, s_i \rangle \in C | l \in \mathcal{L}'\}$.

Note that $\mathcal{M}' = \mathcal{M}$ if $\langle p_o, p_i \rangle \notin \mathcal{L}$.

Definition 3.12 (Connection Addition).
A *connection addition* is a function

$$addConnection : \quad \mathfrak{M} \times \mathsf{Port}^2 \times \mathsf{OutSign} \times \mathsf{InSign} \to \mathfrak{M}$$
$$(\mathcal{M}, (p_o, p_i), s_o, s_i) \mapsto \mathcal{M}'$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$:

$$\begin{cases} \mathcal{M}' = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C \cup \{\langle \langle p_o, p_i \rangle, s_o, s_i \rangle\}, \mathcal{R} \rangle & \text{if:} \\ & - \exists (B_1, B_2) \in \mathcal{B}^2 \text{ s.t. } p_o \in P_{B_1} \wedge s_o \in S_{oB_1} \wedge p_i \in P_{B_2} \wedge s_i \in S_{iB_2} \\ & - \langle p_o, p_i \rangle \in \mathcal{L} \\ & - \neg\exists \langle \langle p, p' \rangle, s, s' \rangle \in C \text{ s.t. } s = s_o \vee s' = s_i \\ \mathcal{M}' = \mathcal{M} & \text{otherwise.} \end{cases}$$

Definition 3.13 (Connection Deletion).
A *connection deletion* is a function

$$delConnection : \quad \mathfrak{M} \times \mathsf{Port}^2 \times \mathsf{OutSign} \times \mathsf{InSign} \to \mathfrak{M}$$
$$(\mathcal{M}, (p_o, p_i), s_o, s_i) \mapsto \mathcal{M}'$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$, $\mathcal{M}' = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C \setminus \{\langle \langle p_o, p_i \rangle, s_o, s_i \rangle\}, \mathcal{R} \rangle$.
Note that $\mathcal{M}' = \mathcal{M}$ if $\langle \langle p_o, p_i \rangle, s_o, s_i \rangle \notin C$.

Definition 3.14 (Containment Addition).

Just Accepted

A *containment addition* is a function

$$\begin{aligned} addContainement: \quad & \mathfrak{M} \times \mathfrak{B}^2 \rightarrow \mathfrak{M} \\ & (\mathcal{M}, (B_1, B_2)) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$:

$$\begin{cases} \mathcal{M}' = \langle \mathcal{B}, d, \mathcal{L}, C, \mathcal{R} \cup \{\langle B_1, B_2 \rangle\} \rangle & \text{if } \{B_1, B_2\} \subseteq \mathcal{B} \wedge B_1 \notin sub_{B_2} \wedge \neg \exists B \in \mathcal{B}, \langle B, B_2 \rangle \in \mathcal{R} \\ \mathcal{M}' = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, (\mathcal{R} \setminus \{\langle B, B_2 \rangle\}) \cup \{\langle B_1, B_2 \rangle\} \rangle & \text{if } \{B_1, B_2\} \subseteq \mathcal{B} \wedge B_1 \notin sub_{B_2} \wedge \exists B \in \mathcal{B}, \langle B, B_2 \rangle \in \mathcal{R} \\ \mathcal{M}' = \mathcal{M} & \text{otherwise.} \end{cases}$$

Definition 3.15 (Containment Deletion).
A *containment deletion* is a function

$$\begin{aligned} delContainement: \quad & \mathfrak{M} \times \mathfrak{B}^2 \rightarrow \mathfrak{M} \\ & (\mathcal{M}, (B_1, B_2)) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$, $\mathcal{M}' = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \setminus \{\langle B_1, B_2 \rangle\} \rangle$.
Note that $\mathcal{M}' = \mathcal{M}$ if $\langle B_1, B_2 \rangle \notin \mathcal{R}$.

3.3.2 Mutations at block level. We define below the mutations that change the composition of the elements of a block composing a SysML block diagram and their related state-machine diagrams. Ten mutations are formalized in this section:

- Definition 3.17 (resp. 3.18) defines the addition (resp. deletion) of an attribute to (resp. from) a block.
- Definition 3.19 defines the addition of an input signal to a block.
- Definition 3.20 defines the addition of an output signal to a block.
- Definition 3.21 defines the deletion of a signal from a block.
- Definition 3.22 (resp. 3.23) defines the addition (resp. deletion) of a state to (resp. from) a block's state-machine diagram.
- Definition 3.24 (resp. 3.25) defines the addition (resp. deletion) of a transition to (resp. from) a block's state-machine diagram.

For the needs of these definitions, we first introduce the containment substitution function (Definition 3.16). The aim of this function is to replace a block B with a block B' in a set of subblocks/superblocks pairs $\mathcal{R}$.

Definition 3.16 (Containment Substitution).
Given a set of block pairs $\mathcal{R}$ and a block B, we denote by $\mathcal{R}_B$ the set $\{\langle B_1, B_2 \rangle \in \mathcal{R} | B_1 = B \oplus B_2 = B\}$.
Given the function

$$\begin{aligned} replace: \quad & \mathfrak{B}^2 \times \mathfrak{B} \times \mathfrak{B} \rightarrow \mathfrak{B}^2 \\ & (\langle B_1, B_2 \rangle, B, B') \mapsto \begin{cases} \langle B', B_2 \rangle & \text{if } B_1 = B \\ \langle B_1, B' \rangle & \text{if } B_2 = B \\ \langle B_1, B_2 \rangle & \text{otherwise.} \end{cases}, \end{aligned}$$

a *containment substitution* is a function

$$\begin{aligned} substitute: \quad & (\mathfrak{B}^2)^n \times \mathfrak{B} \times \mathfrak{B} \rightarrow (\mathfrak{B}^2)^n \\ & (\mathcal{R}, B, B') \mapsto (\mathcal{R} \setminus \mathcal{R}_B) \cup \{replace(\langle B_1, B_2 \rangle, B, B') | \langle B_1, B_2 \rangle \in \mathcal{R}_B\} \end{aligned}$$

Just Accepted

Definition 3.17 (Attribute Addition).
An *attribute addition* is a function

$$\begin{aligned} addAttribute: \quad & \mathfrak{M} \times \mathfrak{B} \times \mathsf{Attr} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B, a) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, smd \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A \cup \{a\}, M, P, S_i, S_o, smd \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Definition 3.18 (Attribute Deletion).
An *attribute deletion* is a function

$$\begin{aligned} delAttribute: \quad & \mathfrak{M} \times \mathfrak{B} \times \mathsf{Attr} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B, a) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, smd \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \wedge a \in A_B \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A \setminus \{a\}, M, P, S_i, S_o, smd \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Definition 3.19 (Input Signal Addition).
An *input signal addition* is a function

$$\begin{aligned} addInputSignal: \quad & \mathfrak{M} \times \mathfrak{B} \times \mathsf{InSign} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B, s) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, smd \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\} \cup \{B'\}), d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A, M, P, S_i \cup \{s\}, S_o, smd \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Definition 3.20 (Output Signal Addition).

Just Accepted

An *output signal addition* is a function

$$addOutputSignal: \quad \begin{array}{l} \mathfrak{M} \times \mathfrak{B} \times \text{OutSign} \rightarrow \mathfrak{M} \\ (\mathcal{M}, B, s) \mapsto \mathcal{M}' \end{array}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, smd \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A, M, P, S_i, S_o \cup \{s\}, smd \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Definition 3.21 (Signal Deletion).
A *signal deletion* is a function

$$delSignal: \quad \begin{array}{l} \mathfrak{M} \times \mathfrak{B} \times \text{Sign} \rightarrow \mathfrak{M} \\ (\mathcal{M}, B, s) \mapsto \mathcal{M}' \end{array}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, \langle s_0, S, T \rangle \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C', \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \wedge (s \in S_o \vee s \in S_i) \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $\begin{cases} d(B') = \langle A, M, P, S_i \setminus \{s\}, S_o, smd \rangle & \text{if } s \in S_i \\ d(B') = \langle A, M, P, S_i, S_o \setminus \{s\}, smd \rangle & \text{if } s \in S_o. \end{cases}$
- $\begin{cases} C' = C \setminus \{\langle \langle p_o, p_i \rangle, s_o, s_i \rangle | s_i = s\} & \text{if } s \in S_i \\ C' = C \setminus \{\langle \langle p_o, p_i \rangle, s_o, s_i \rangle | s_o = s\} & \text{if } s \in S_o. \end{cases}$
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Definition 3.22 (State Addition).
A *state addition* is a function

$$addState: \quad \begin{array}{l} \mathfrak{M} \times \mathfrak{B} \times \text{States} \rightarrow \mathfrak{M} \\ (\mathcal{M}, B, s) \mapsto \mathcal{M}' \end{array}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, \langle s_0, S, T \rangle \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A, M, P, S_i, S_o, \langle s_0, S \cup \{s\}, T \rangle \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Just Accepted

Definition 3.23 (State Deletion).
A *state deletion* is a function

$$\begin{aligned} addState: \quad & \mathfrak{M} \times \mathfrak{B} \times \text{States} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B, s) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, \langle s_0, S, T \rangle \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \wedge s \in S \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A, M, P, S_i, S_o, \langle s_0, S \setminus \{s\}, T \setminus \{\langle s_{start}, condition, actions, s_{end} \rangle | s_{start} = s \vee s_{end} = s\} \rangle \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Definition 3.24 (Transition Addition).
A *transition addition* is a function

$$\begin{aligned} addTrans: \quad & \mathfrak{M} \times \mathfrak{B} \times \text{Trans} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B, \langle s_{start}, condition, actions, s_{end} \rangle) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, \langle s_0, S, T \rangle \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \wedge s_{start} \in S \wedge s_{end} \in S \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A, M, P, S_i, S_o, \langle s_0, S, T \cup \{\langle s_{start}, condition, actions, s_{end} \rangle\} \rangle \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

Definition 3.25 (Transition Deletion).
A *transition deletion* is a function

$$\begin{aligned} delTransition: \quad & \mathfrak{M} \times \mathfrak{B} \times \text{Trans} \rightarrow \mathfrak{M} \\ & (\mathcal{M}, B, t) \mapsto \mathcal{M}' \end{aligned}$$

such that, given $\mathcal{M} = \langle \mathcal{B}, d, \mathcal{L}, \sigma, C, \mathcal{R} \rangle$ and $d(B) = \langle A, M, P, S_i, S_o, \langle s_0, S, T \rangle \rangle$:

$$\begin{cases} \mathcal{M}' = \langle (\mathcal{B} \setminus \{B\}) \cup \{B'\}, d, \mathcal{L}, \sigma, C, \mathcal{R}' \rangle & \text{if } B \in \mathcal{B} \wedge t \in T \\ \mathcal{M}' = \mathcal{M} & \text{otherwise,} \end{cases}$$

where:

- $d(B') = \langle A, M, P, S_i, S_o, \langle s_0, S, T \setminus \{t\} \rangle \rangle$.
- $\mathcal{R}' = substitute(\mathcal{R}, B, B')$.
- $B' \notin \mathcal{B} \setminus \{B\}$.

3.4 AMULET: a SysML mutation language

In order to describe and to automate the application of the different mutation operators proposed in Sect. 3.3, we have defined a high-level language called AMULET (**A**ppropriate Sys**ML** m**U**tation **L**anguage **E**nhancing models enrichmen**T**). Although it is formally defined, AMULET has a syntax close to natural language, so that it can be easily learnt by any SysML toolkit user. For instance, the code line `add state s0 in B0` stands for

Just Accepted

$\mathcal{M} \xrightarrow{\text{State}^+(B_0,s_0)} \mathcal{M}'$, and `add block B1` stands for $\mathcal{M} \xrightarrow{\mathfrak{B}^+(B_1)} \mathcal{M}'$. All the operators defined in Section 3.3 are supported by AMULET. Table 3 gives, for each mutation operator at model level, the equivalent syntax in AMULET and one or several examples. In the same way, Table 4 provides the equivalent AMULET syntax and examples for each mutation operator at block level. We had to provide in this table two different syntax rules for the *addTransition* and *delTransition* operators, depending on the actions of the targeted transition. Indeed, in our SysML implementation transitions involving actions on signals (i.e., a send or receive operator in state-machine diagrams) are split into two separate transitions: the first one from the source state to the send/receive operator, and the second one from the send/receive operator to the destination state. As a result, adding a transition from a state s_1 to a state s_2 and containing a send/receive action consists in three consecutive stages: (1) creating the send/receive operator, (2) adding a transition from s_1 to this operator and (3) adding a transition from this operator to s_2. Given several send/receive action operators can instanciate the same signal, we shall provide an ID for this operator at stage (1). In the given example, the ID assigned to the action on signal is `myActionOnSignal`, and it operates on the signal referred to as `inSig`.

Note that AMULET also supports operator composition thanks to further reserved words : for instance, $\mathcal{M} \xrightarrow{\text{Sign}^-(B_0,mySig)} \mathcal{M}' \xrightarrow{\text{OutSign}^+(B_0,mySig)} \mathcal{M}''$ where *mySig* is initially an input signal will be written in AMULET `modify signal mySig in B0 to output`. However, for the sake of brevity, and considering that the syntax outlined in Tables 3 and 4 adequately encompasses all the mutation operators, we have chosen not to furnish a comprehensive syntax of these operator composition commands within this paper.

4 SYSML MUTATIONS: THE PRACTICAL WAY

This section introduces our practical contributions. We firstly provide a brief introduction on the SysML profile we used for the implementation and evaluation of AMULET. Then, we present the implementation of the AMULET compiler and its user interface in the the open-source SysML modeling and verification toolkit TTool.

4.1 SysML and AVATAR: differences

AVATAR is a real-time oriented SysML profile fully compliant with the SysML metamodel [21]. AVATAR provides a formal semantics for SysML block and state-machine diagrams, and we have decided to use this profile for the implementation and evaluation of our mutation language. Additionally, AVATAR includes features that have been added to SysML. It's worth noting that all the mutation operators defined in Section 3.3 are compatible with AVATAR's syntax.

4.1.1 In block diagrams. In line with the integration of block definition diagrams and internal block diagrams as outlined in Definition 3.5, the AVATAR profile employs a unified block diagram that combines block definition diagram and internal block diagram.

4.1.2 In state-machine diagrams. In AVATAR state-machine diagrams, a variable delay operator has been added to transitions [21], i.e., the $after$ operator takes a time interval as an argument. Therefore, with respect to Definition 3.3 defining SysML state-machine diagrams, AVATAR transitions are of the form $t = \langle s_{start}, after, condition, actions, s_{end} \rangle$, where $after = (time_{min}, time_{max}) \in \mathbb{N}^2$ constrains the delay before firing t.

Just Accepted

Table 3. AMULET syntax for model-level mutation operators. In the **AMULET syntax** column, reserved words are written in bold and optional tokens are written between square brackets. All the IDs are Strings, and fifoSize is an Integer.

Mutation Operator	AMULET syntax	Examples
addBlock (Def. 3.8)	**add block** blockId	`add block Block1`
delBlock (Def. 3.9)	**remove block** blockId	`remove block Block1`
addLink (Def. 3.10)	**add** [**public** | **private**] [**synchronous** [**broadcast**] | **asynchronous** [**blocking**] [**lossy**]] **link** [linkId] [**with maxFIFO** = fifoSize] **between** blockId **and** blockId	`add link between Block1 and Block2` or `add asynchronous lossy link myLink between Block1 and Block2`
delLink (Def. 3.11)	**remove** [**public** | **private**] [**synchronous** [**broadcast**] | **asynchronous** [**blocking**] [**lossy**]] **link** [**with maxFIFO** = fifoSize] **between** blockId **and** blockId **and** [block name] or **remove link** linkId	`remove link between Block1 and Block2` or `remove link myLink`
addConnection (Def. 3.12)	**add connection from** signalId **in** blockId **to** signalName **in** blockId [**in** [**public** | **private**] [**synchronous** [**broadcast**] | **asynchronous** [**blocking**] [**lossy**]] **link** [**with maxFIFO** = fifoSize]] or **add connection from** signalId **to** signalId **in link** linkId	`add connection from sigOut in Block1 to sigIn in Block2` or `add connection from sigOut to sigIn in link myLink`
delConnection (Def. 3.13)	**remove connection from** signalId **in** blockId **to** signalName **in** blockId [**in** [**public** | **private**] [**synchronous** [**broadcast**] | **asynchronous** [**blocking**] [**lossy**]] **link** [**with maxFIFO** = fifoSize]] or **remove connection from** signalId **to** signalId **in link** linkId	`remove connection from sigOut in Block1 to sigIn in Block2` or `remove connection from sigOut to sigIn in link myLink`
addContainment (Def. 3.14)	**attach** blockId **to** blockId	`attach Block2 to Block1`
delContainment (Def. 3.15)	**detach** blockId [**from** blockId]	`detach Block2 from Block1` or `detach Block2`

4.2 Implementation in TTool

In order to facilitate the integration of AMULET, a new compiler was developed and integrated into TTool. This compiler was implemented using Java and its source code is publicly accessible on TTool's Git repository[9] for the compiler package.

Figure 2 illustrates the functional architecture of the compiler, which is composed of three consecutive stages. These stages are activated by four commands that have been added to TTool's command-line interpreter: `a am` (AVATAR Mutation), `a amb` (AVATAR Mutation Batch), `a ap` (AVATAR Print), and `a ad` (AVATAR Draw). The `a am` command, for instance, takes a mutation written in AMULET as an argument, parses it, and applies it to the model that is currently loaded in TTool's memory. For example, if a model $\mathcal{M}$ is loaded in TTool, the mutation $\mathcal{M} \xrightarrow{\mathsf{State}^{+}(B_0,s_0)} \mathcal{M}'$ can be applied to $\mathcal{M}$ using the following command: `a am add state s0 in B0`. The `a amb` command, in turn, takes a file containing several mutations written in AMULET as an argument and applies

[9]https://gitlab.telecom-paris.fr/mbe-tools/TTool.git. Browse in `./src/main/java/avatartranslator/mutation`.

Just Accepted

Table 4. AMULET syntax for block-level operators. In the **AMULET syntax** column, reserved words are written in bold and optional tokens are written between square brackets. All the IDs are String, intVal is an Integer, guard is a boolean expression, actionExpression is a variable assignment or a method call, and actionOnSignalExpression is a signal call.

Mutation Operator	AMULET syntax	Examples

addAttribute (Def. 3.17)	**add attribute** type attributeId [= intVal] **in** blockId
	NB: type = int | bool
		`add attribute bool myBool in Block1`
		or
		`add attribute int x = 42 in Block1`

delAttribute (Def. 3.18)	**remove attribute** attributeId **in** blockId	`remove attribute myBool in Block1`
addInputSignal (Def. 3.19)	**add input signal** signalId **in** blockId	`add input signal inSig in Block2`
addOutputSignal (Def. 3.20)	**add output signal** signalId **in** blockId	`add output signal mySig in Block1`
delSignal (Def. 3.21)	**remove signal** signalId **in** blockId	`remove signal outSig in Block1`
addState (Def. 3.22)	**add state** stateId **in** blockId	`add state myState in Block1`
delState (Def. 3.23)	**remove state** stateId **in** blockId	`remove state myState in Block1`
addTransition (Def. 3.24) (general case)	**add transition** [transitionId] **in** blockId **from** stateId **to** stateId [**with** [guard] | **with after**(intVal, intVal) | **with** "actionExpression" | **with** [guard] **and after**(intVal, intVal) | **with** [guard] **and** "actionExpression" | **with after**(intVal, intVal) **and** "actionExpression" | **with** [guard] **and after**(intVal, intVal) **and** "actionExpression"]	`add transition in Block1 from state1 to state2 with "x=0"`

addTransition (Def. 3.24) (if the set of actions contains an *action on signal*, i.e., a *send* or a *receive* action)	Stage 1. Creation of the action on signal operator
	add action on signal actionOnSignalId **in** blockId **with** actionOnSignalExpression
	Stages 2-3. Creation of the two transitions between the initial state and the action on signal, and between the action on signal and the destination state
	add transition [transitionId] **in** blockId **from** stateOrActionOnSignalId **to** stateOrActionOnSignalId [**with** [guard] | **with after**(intVal, intVal) | **with** "actionExpression" | **with** [guard] **and after**(intVal, intVal) | **with** [guard] **and** "actionExpression" | **with after**(intVal, intVal) **and** "actionExpression" | **with** [guard] **and after**(intVal, intVal) **and** "actionExpression"]
		Stage 1.
		`add action on signal myActionOnSignal in Block1 with inSig(int x)`
		Stage 2.
		`add transition in Block1 from state1 to myActionOnSignal with [x != 0]`
		Stage 3.
		`add transition in Block1 from myActionOnSignal to state2 with "x = x+1" and after(1,2)`

delTransition (Def. 3.25) (general case)	**remove transition in** blockId **from** stateId **to** stateId [**with** [guard] | **with after**(intVal, intVal) | **with** "actionExpression" | **with** [guard] **and after**(intVal, intVal) | **with** [guard] **and** "actionExpression" | **with after**(intVal, intVal) **and** "actionExpression" | **with** [guard] **and after**(intVal, intVal) **and** "actionExpression"]
	or
	remove transition transitionId **in** blockId
		`remove transition in Block1 from state1 to state2 with "x=0"`
		or
		`remove transition myTransition in Block1`

delTransition (Def. 3.25) (if the set of actions contains an *action on signal*, i.e., a *send* or a *receive* action)	**remove action on signal in** blockId **with** actionOnSignalExpression]
	or
	remove action on signal actionOnSignalId **in** blockId
		`remove action on signal in Block1 with outSig`
		or
		`remove action on signal myActionOnSignal in Block1`

them one after another to the model that is currently loaded in TTool's memory. The `a ap` command displays the AVATAR model currently loaded in text format, and the `a ad` command displays it in graphical format.

Just Accepted

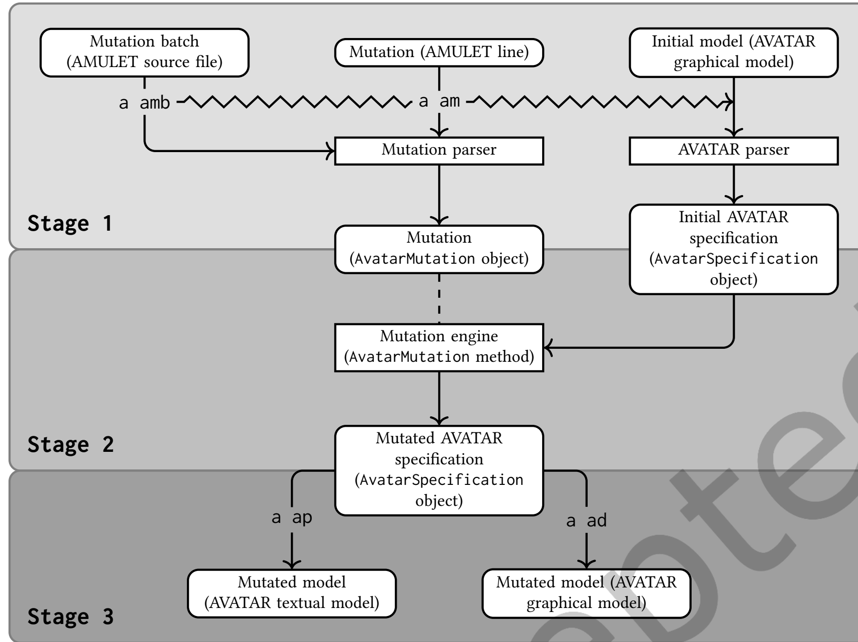


Fig. 2. Functional architecture of the TTool integrated AMULET compiler

Just Accepted

4.2.1 Stage 1: parsing the inputs. Obviously, the compiler takes as input a mutation command, and an AVATAR model. When an `a am` command is provided, the compiler applies the AMULET string after the `a am` command to AVATAR model currently opened in TTool's graphical interface. In the scope of an `a amb` command, each line of the mutation batch is handled one after the other by the compiler. An AMULET string is parsed as follows:

(1) the string is tokenized;
(2) the token list is syntactically analyzed;
(3) an `AvatarMutation` object (the data structure we have designed to representing a mutation after parsing an AMULET command) is created.

The `apply` method of an `AvatarMutation` object (the method that modifies the model according to the parsed mutation) does not directly processes AVATAR graphical models but `AvatarSpecification` objects[10]. Therefore, in parallel with the AMULET string parsing, the AVATAR graphical model is also parsed in order to convert it into an `AvatarSpecification` object. Note that `AvatarSpecification` class and parsing methods were already defined in TTool's source code prior to this work. For instance, its is used as input for other transformation models such as the ones for safety [11] or security verifications [5].

At the end of this first stage, the two inputs have been converted to their relative objects. The compiler can now work on this intermediate formats for applying the mutation.

4.2.2 Stage 2: applying the mutation. The `apply` method of the newly created `AvatarMutation` object is then called with the `AvatarSpecification` object passed as argument. During this stage, the latter is modified according to the mutation (e.g., a new block is added to its list of blocks).

[10]Again, we do not rely on generic data structures for representing AVATAR models, but on tailored `AvatarSpecification` objects.

4.2.3 Stage 3: providing the user with the mutated model. Once the mutated `AvatarSpecification` has been generated, the user can either execute a new `a am` command or display the generated AVATAR model in two formats: textual format (`a ap` command in the command-line interface) or graphical format (`a ad` command).

4.3 About the syntactic correctness of the mutated model

It is important to note that the mutated model may not be syntactically correct. Most of the operators defined in Sect. 3.3 ensure by definition the syntactic correctness (as per definition 3.7) of the resulting block diagram. However, depending on the input block diagram, the following operators may lead to a syntactically incorrect block diagram:

- *delBlock*, if a signal of the deleted block is associated with a signal used in another block's state-machine diagram.
- *delLink*, if a pair of signals used in a block's state-machine diagram is connected over the deleted link.
- *delConnection*, if one of the signals involved in the deleted connection is used in a block's state-machine diagram.
- *delContainment*, if one of the signal of the superblock (or of the superblock's superblock, and so on) is used in the detached subblock's state-machine diagram.
- *delAttribute*, if the attribute is used in the block's state-machine diagram.
- *delInputSignal*, *delOutputSignal*, if the signal is used in the block's state-machine diagram.

Indeed, AMULET was initially designed to assist TTool users in applying identical mutations to multiple models, mirroring the process observed in model-driven design methods like W-Sec [24] or SysML-Sec [7]. As a result, AMULET encompasses all modifications allowed by the TTool's editor, even those that may result in syntactically incorrect block diagrams. While applying these methods, it falls to the user to design a mutation that leads to a syntactically correct block diagram. In this context, the AMULET operators represent successive steps undertaken to achieve a syntactically correct block diagram. For instance, if a *delBlock* operator is used, transitions using signals associated with signals of the deleted block may be modified, or the signals used in those transitions may be connected to signals of another block. These further modifications can be achieved by using other mutation operators. Thus, in this case, since we support operators that can lead to syntactically incorrect diagrams, users can apply these mutation operators in any order they prefer.

Furthermore, the support for mutations should not prevent the use of AMULET in design methods involving automatic mutation generation and application, such as mutation-based model testing. Indeed, since TTool already incorporates a SysML syntax-checker, it is trivial to automatically identify and eliminate the syntactically incorrect mutants generated from the application of mutation operations. This helps ensuring that only the syntactically correct mutations are retained for subsequent stages of the process (e.g., testing).

5 CASE-STUDY: AUTOMATIC ENRICHMENT OF A MODEL WITH AN ARP SPOOFING ATTACK SCENARIO

This section illustrates the AMULET language and its compiler with two case-studies. The first case-study is basic since it only intends to give a better idea on how our approach works and illustrate its relevance in the context of various system evolution scenarios. This first case-study is illustrated with examples of AVATAR diagrams and AMULET source code. The second case study is clearly more ambitious since it is based on a real industrial system we had to secure. Due to the complexity of the models, diagrams and AVATAR source code are not provided here. However, we provide metrics highlighting the benefits of AMULET in terms of reliability and modeling time.

5.1 First case-study: a simple network

5.1.1 Model and tested evolution scenarios.

Just Accepted

This case-study illustrates a practical and visual example of typical AVATAR model mutations, and how AMULET helps in applying them. The system is built upon two computers communicating through a router. Each component of the system is modeled with a distinct block (see Figure 3). In order to provide concrete examples of the three system evolution cases mentioned in Sect. 1, we have applied mutations on the intial model in order to model the following evolutions:

Scenario 1 (generic system evolution case and model refinement). We assume that a *man-in-the-middle* attack could be performed in the system: here, the goal of the mutations is to better understand how such an attack could impact our system. The man-in-the-middle attack is captured as a new block modeling the attacker and with a slight modification to the state-machine diagrams of the existing blocks (see Figure 5). Then, we assume that we want to evaluate the impact according to different functional scenarios. One functional scenario differs from another one in the content of the message sent from a computer to the other computer (PC1, PC2). Here, the mutations focus on the modification of the value of the attribute `valueToSend` in the block `PC1`.
Scenario 2 (model refinement). Model-driven engineering often involves incremental refinement of the models: in this scenario, we add a additional boolean parameter to the messages exchanged between `PC1` and `PC2`. The mutations consist in adding a new attributes in the three blocks and slightly modifying some signals and the three state-machine diagrams (see Figure 4). This refinement targets the ten models generated in Scenario 1.
Scenario 3 ("digital twin" evolution case). Here we assume that the model designer wants to improve the system design in order to mitigate the man-in-the-middle attack. The countermeasure consists in adding a message authentication code (MAC) to the messages exchanged from `PC1` to `PC2`. The mutations involve adding new attributes in the three blocks, modifying the signals and the state-machine diagrams. Here again, we assume that this evolution targets the ten models generated in Scenario 1.

5.1.2 Refining the model: mutations and AMULET script.

Prior to discuss the metrics comparing the AMULET and the "by-hand" approach for deploying these mutations, we provide here an AMULET script used for generating mutated models in the context of Scenario 2. Applying this script on the diagrams shown in Fig. 3 generates the diagrams shown in Fig. 4. This model generation involves 17 successive mutations provided below. However, thanks to the AMULET syntax providing specific tokens for operator composition, the script only comprises 11 lines.

$$\mathcal{M}_1 \xrightarrow{\mathsf{Attr}^+(PC1, booleanToSend{=}true)} \mathcal{M}_2 \xrightarrow{\mathsf{Attr}^+(PC2, receivedBoolean)} \mathcal{M}_3 \xrightarrow{\mathsf{Attr}^+(Router, b)} \mathcal{M}_4$$

$$\mathcal{M}_4 \xrightarrow{\mathsf{Sign}^-(PC1, PC1_toRouter(int\ valueToSend))} \mathcal{M}_5$$

$$\mathcal{M}_5 \xrightarrow{\mathsf{OutSign}^+(PC1, PC1_toRouter(int\ valueToSend, bool\ booleanToSend))} \mathcal{M}_6$$

$$\mathcal{M}_6 \xrightarrow{\mathsf{Sign}^-(PC2, PC2_fromRouter(int\ receivedValue))} \mathcal{M}_7$$

$$\mathcal{M}_7 \xrightarrow{\mathsf{InSign}^+(PC2, PC2_fromRouter(int\ receivedValue, bool\ receivedBoolean))} \mathcal{M}_8$$

$$\mathcal{M}_8 \xrightarrow{\mathsf{Sign}^-(Router, Router_fromPC1(int\ n))} \mathcal{M}_9 \xrightarrow{\mathsf{InSign}^+(Router, Router_fromPC1(int\ n, bool\ b))} \mathcal{M}_{10}$$

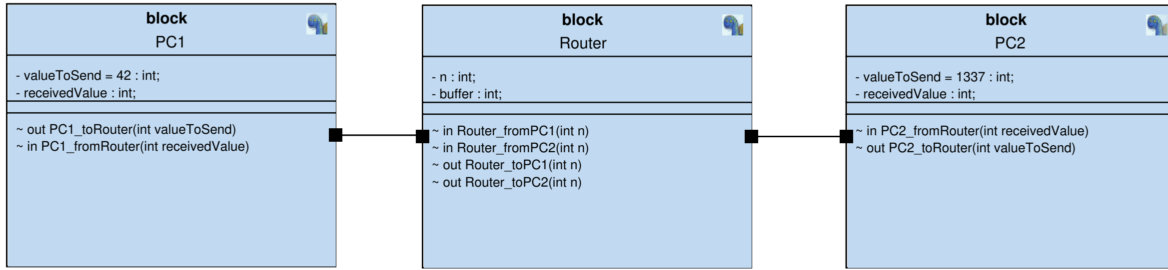
$$\mathcal{M}_{10} \xrightarrow{\mathsf{Sign}^-(Router, Router_toPC2(int\ n))} \mathcal{M}_{11} \xrightarrow{\mathsf{OutSign}^+(Router, Router_toPC2(int\ n, bool\ b))} \mathcal{M}_{12}$$

$$\mathcal{M}_{12} \xrightarrow{\mathsf{Trans}^-(PC1, \langle idle, PC1_toRouter(int\ valueToSend), idle\rangle)} \mathcal{M}_{13}$$

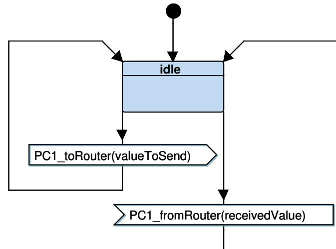
$$\mathcal{M}_{13} \xrightarrow{\mathsf{Trans}^+(PC1, \langle idle, PC1_toRouter(int\ valueToSend, bool\ booleanToSend), idle\rangle)} \mathcal{M}_{14}$$

$$\mathcal{M}_{14} \xrightarrow{\mathsf{Trans}^-(PC2, \langle idle, PC2_fromRouter(int\ receivedValue), idle\rangle)} \mathcal{M}_{15}$$

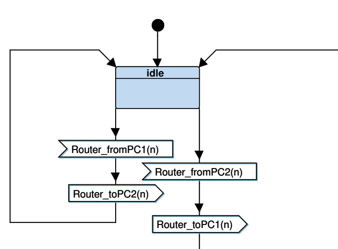
Just Accepted



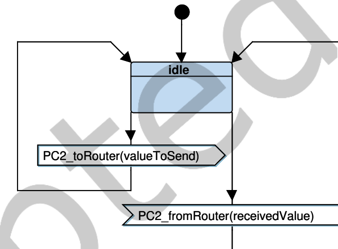
(a) Block diagram



(b) PC1's SMD

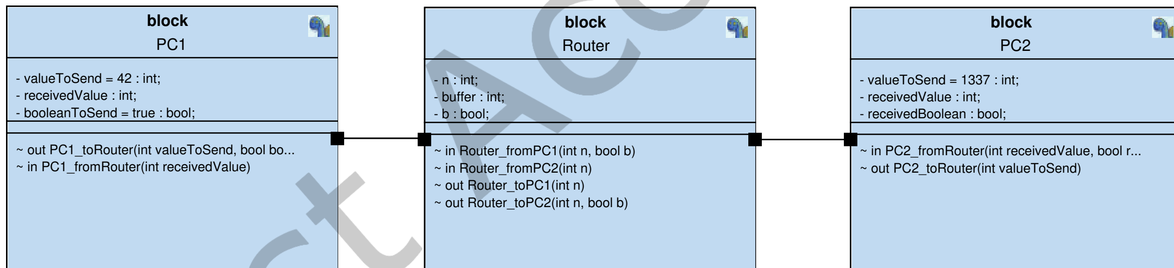


(c) Router's SMD

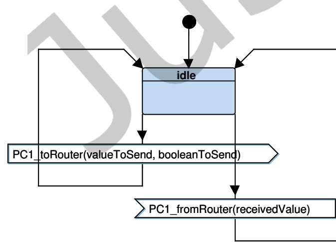


(d) PC2's SMD

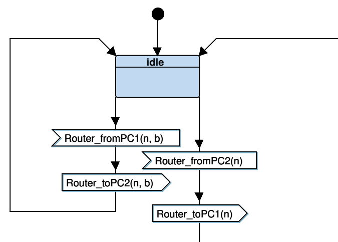
Fig. 3. Initial AVATAR model



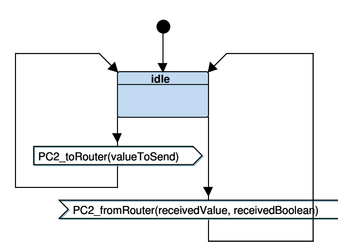
(a) Mutated block diagram



(b) Mutated PC1's SMD



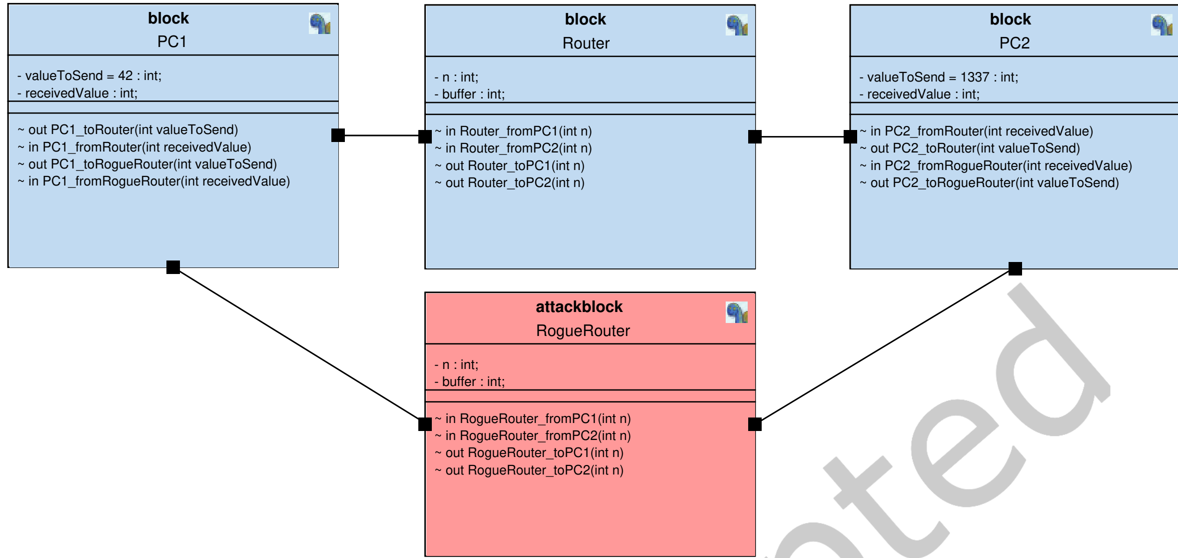
(c) Mutated Router's SMD



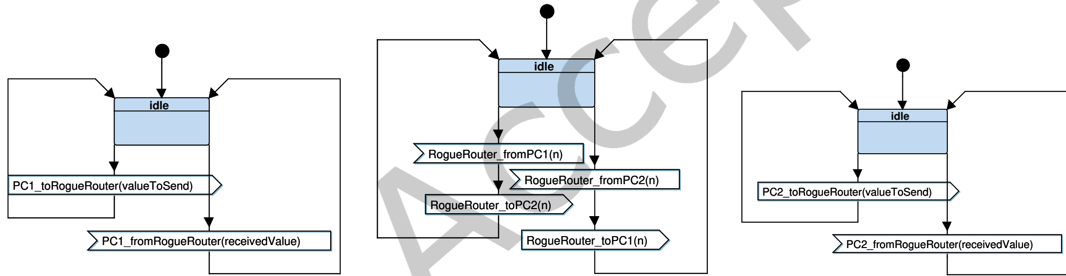
(d) Mutated PC2's SMD

Fig. 4. AVATAR model after refinement

Just Accepted



(a) Block diagram



(b) PC1's SMD (c) Malicious router's SMD (d) PC2's SMD

Fig. 5. AVATAR model after attack scenario integration

Just Accepted

$$\mathcal{M}_{15} \xrightarrow{\text{Trans}^{+}(PC2,\langle idle,PC2_fromRouter(int\ receivedValue,bool\ receivedBoolean),idle\rangle)} \mathcal{M}_{16}$$

$$\mathcal{M}_{16} \xrightarrow{\text{Trans}^{-}(Router,\langle idle,Router_fromPC1(int\ n)|Router_toPC2(int\ n),idle\rangle)} \mathcal{M}_{17}$$

$$\mathcal{M}_{17} \xrightarrow{\text{Trans}^{+}(Router,\langle idle,Router_fromPC1(int\ n,bool\ b)|Router_toPC2(int\ n,bool\ b),idle\rangle)} \mathcal{M}_{18}$$

```
1  add attribute bool booleanToSend = true in PC1
2  add attribute bool receivedBoolean in PC2
3  add attribute bool b in Router
4  modify signal PC1_toRouter in PC1 to PC1_toRouter(int valueToSend, bool booleanToSend)
5  modify signal PC2_fromRouter in PC2 to PC2_fromRouter(int receivedValue, bool receivedBoolean)
6  modify signal Router_fromPC1 in Router to Router_fromPC1(int n, bool b)
7  modify signal Router_toPC2 in Router to Router_toPC2(int n, bool b)
8  modify action on signal with PC1_toRouter(valueToSend) in PC1 to PC1_toRouter(valueToSend,booleanToSend)
9  modify action on signal with PC2_fromRouter(receivedValue) in PC2 to PC2_fromRouter(receivedValue,receivedBoolean)
10 modify action on signal with Router_fromPC1(n) in Router to Router_fromPC1(n,b)
```

Table 5. Comparison metrics for the simple network case-study

Scenario 1 – Generic system evolution case: integration of functional scenarios and attack model		
	With AMULET	**By hand (optimized – naive approach)**
Number of manual operations	40 AMULET lines written	39 – 176 modifications in TTool
Time needed for applying the mutations	7 min 29	4 min 21 – 11 min 52
Scenario 2 – Model refinement case: adding a parameter in the signals exchanged from PC1 to PC2		
	With AMULET	**By hand (optimized – naive approach)**
Number of manual operations	18 AMULET lines written	43 – 145 modifications in TTool
Time needed for applying the mutations	5 min 14	5 min 27 – 19 min 15
Scenario 3 – "Digital twin" system evolution case: designing a cryptographic countermeasure		
	With AMULET	**By hand (optimized – naive approach)**
Number of manual operations	33 AMULET lines written	52 – 180 modifications in TTool
Time needed for applying the mutations	6 min 47	6 min 57 – 26 min 45

```
11 modify action on signal with Router_toPC2(n) in Router to Router_toPC2(n,b)
```

Note that this script applies to the models of the system without the attack: a second script, that differs from this one in 7 lines, has been written for refining the mutated models integrating the attack.

5.1.3 Results.

For each evolution scenario, we have carried out two mutation approaches on the base models in order to highlight the benefits of AMULET: a "by-hand" approach where we have modified the models using TTool's graphical interface, and an automated approach where we have written and executed AMULET scripts using the AMULET compiler. We have quantified the number of manual operations (performing a model modification using TTool GUI or writing an AMULET script line) and the time it took to perform these operations for both approaches. For the manual approach, the time measurement only includes the application of mutations using the GUI of TTool. For the automated approach, the time measurement includes both the writing of the script (including the time to identify and correct typing errors) and the execution of the script. In other words, both measurements of time exclude the duration spent on conceptualizing the mutations. As explained above, this case-study has mainly an illustrative aim. Nevertheless, Table 5 provides some key metrics for discussing the benefits of AMULET even on this simple example. Note that in this case-study, two "by-hand" mutation approaches actually exist. Indeed, given the models of the five different functional scenarios we want to study only differs in an attribute initial value, the optimized approach consists in (1) mutating a given block diagram and (2) using a TTool feature enabling block diagram cloning for duplicating four times this block diagram and then modifying in the four new models the relevant attribute. The naive approach consists in generating firstly the five models and then integrating each of them with the modeled system evolution. While this approach is suboptimal in this context, it has been included to simulate real-world scenarios where engineers are required to apply the same mutation across a large set of models. For example, in [24] each mutation modeling a countermeasure was manually applied on five models.

In Scenario 1, with the naive approach up to 176 modifications were needed in TTool while 40 lines of AMULET were needed in the AMULET approach (4.4 times less needed operations for the model designer), and the relative execution times of the mutations were 11 min 52 vs 7 min 29 (37% gain in needed time). However, in comparison with AMULET, the optimized approach leads to a shorter modeling time and the same number of operations. In Scenario 2, AMULET has a benefit whatever the "by-hand" mutation approach is chosen, with a reduction in the number of manual operations ranging from 58% to 88% and a gain in needed time ranging from 4% to 73%. In the

Just Accepted

same way, in Scenario 3 AMULET enables for a reduction in the number of manual operations ranging from 37% to 82%, and a time gain from 2% to 75%.

In Scenarios 2 and 3, the main benefit of AMULET lies in the reduction in the number of needed operations for the model designer, thus lowering the risk of errors. Obviously, another benefit is the reusability of the scripts that could apply to similar models with possibly only few modifications.

Finally, it is worth noting that the reduction in the number of manual operations does not directly correspond to a proportional reduction in the required time. Indeed, some elementary mutations are more efficiently executed using the GUI. For example, if we want to modify a parameter sent by a signal in a state-machine diagram it is faster to double-click on the relevant send operator and directly update the parameter name, as opposed to writing and executing the command `modify action on signal mySig(oldParameter) in myBlock to mySig(newParameter)`. However, AMULET provides a time-saving advantage for these elementary mutations when their repeated application across a large set of models is required, since the command is written only once and can be applied to each model without the need to repeatedly locate and open the relevant panel and the relevant operator in the GUI.

5.2 Second case-study: an automated packaging chain

5.2.1 System and evolution scenario.

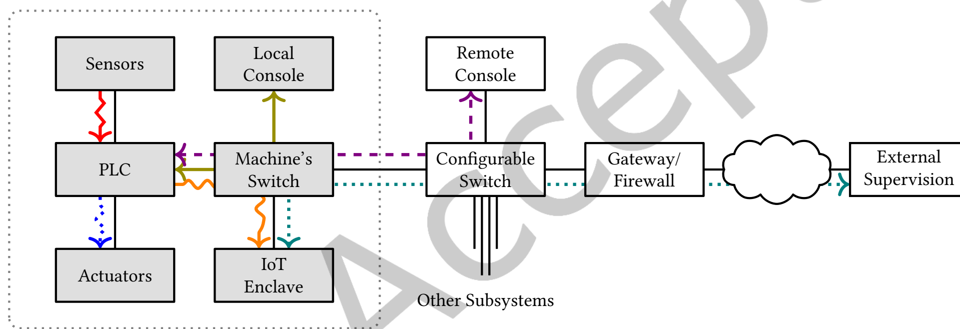


Fig. 6. Network architecture of the relevant factory's subpart (adapted from [25])

This second case-study deals with a packaging chain which is part of IT'm Factory, a research and training platform hosted in École des Mines de Saint-Étienne[11]. The packaging chain relies on four distinct machines that:

- provides the chain with pots to fill (warehouse)
- fills the pots with granules (filling machine)
- closes and moves the pots from the filling machine's conveyor belt to the packer's conveyor belt (collaborative arm)
- packs the pots in crates (packer).

The evaluation focuses on the filling machine. This machine is composed of six kinds of components: a PLC[12] and a set of sensors of actuators, a local control panel, an IoT enclave enabling for an external supervision, and a network switch connecting together the PLC, the control panel, the IoT enclave and the factory's network (see Figure 6). Similarly to the simple network case-study, we assume that the system has a vulnerability in its ARP protocol such as in [17] and we integrate through model mutations an attacker model that (1) connects to the

[11] https://itm-factory.fr/index.php/objectif_et_visite_360/
[12] Programmable Logic Controller.

Just Accepted

Table 6. Comparison metrics for the packaging chain case-study

	With AMULET	**By hand**
Number of manual operations	77 AMULET lines written	370 modifications in TTool
Time needed for applying the mutations	19 min 27	38 min 9

factory's configurable switch, (2) exploits this vulnerability, and finally (3) performs a man-in-the-middle attack by intercepting and modifying the control messages sent by the remote console to the PLC.

5.2.2 *Models and mutations.*

Five initial AVATAR models were designed to model the packaging chain with and without four distinct countermeasures aiming at mitigating the man-in-the-middle attack:

1. a model of the system without any security countermeasure. This model has 9 blocks with a total of 91 attributes. State-machine diagrams have in total 33 states and 72 transitions.
2. a model of the system with a plausibility check countermeasure verifying if the commands received by the PLC are legitimate (9 blocks and state-machine diagrams totalizing 35 states, 75 transitions and 91 attributes)
3. a model of the system with a cryptographic countermeasure (9 blocks and state-machine diagrams totalizing 34 states, 75 transitions and 97 attributes)
4. a model of the system with a countermeasure based on the use of a static ARP table on the devices targeted by the attack (9 blocks and state-machine diagrams totalizing 33 states, 72 transitions and 91 attributes)
5. a model of the system with a workaround consisting in disconnecting the configurable switch from the system (9 blocks and state-machine diagrams totalizing 33 states, 72 transitions and 91 attributes).

Two series of mutations will be applied to this set of model. Firstly, mutations will integrate each model with the attacker model. These mutations consist, for each model, in adding a new block to their block diagram, creating new links between this block and two existing blocks, and slightly modifying the signals and state-machine diagrams of the existing blocks. After these mutations are applied, we have ten models: five initial models, and their mutated counterpart. Secondly, as we want to evaluate the impact of the attack according to two different functional scenarios (a scenario where the PLC is supervised from the remote console, and a scenario where the PLC is supervised from the local console only), we will apply a new series of mutations to the ten models. These mutations consist in modifying two transitions in a state-machine diagram.

In order to apply these mutations we have written five AMULET scripts: four scripts for integrating the five initial models with the attacker model (75 AMULET lines written), and a script (2 AMULET lines written) for integrating the five initial models and the five mutated models with the functional scenarios.

5.2.3 *Results.*

For this case-study, we also carried out two mutation approaches ("by-hand" with TTool GUI and with AMULET) on the initial models. In comparison with the results provided for the simple network case-study, Table 6 illustrates the benefits of AMULET on a "real-world" example: the script-and-compilation approach spares the user from performing 293 additional manual operations that are repetitive and error-prone (77 AMULET lines written vs 370 modifications in TTool, a 79% decrease in needed operations for the model designer). Here, the time needed for applying the mutations on five models is also divided by two (38 min 9 by hand vs 19 min 27 with AMULET). We believe here again that the most interesting benefit of AMULET is the reduction in the number of manual operations (nearly 300 on these models) that should logically lead to a reduction of the modeling errors (and, in addition, in the time spent on correcting them).

Just Accepted

6 DISCUSSION AND CONCLUSIONS

Model mutations are widely used in model-based design and test methods. Yet, applying mutations can be time consuming and error prone when done manually, and may require external tools when automated. The contributions introduced in the paper address this issue for SysML models. Indeed, AMULET is the first language targeting SysML mutations and relying on a set of predefined SysML mutation operators. The compilation of AMULET source code is supported by TTool, an open-source SysML modeling and verification toolkit: engineers can now use a unique toolkit for design, automatically enrich and validate SysML models. AMULET and its compiler have been evaluated against two case-studies including a model of a real industrial system. These evaluations highlight the strength and the limitations of the language and its implementation.

6.1 Strengths

Applying mutations with AMULET helps in improving the model modification process in TTool with respect to three key software quality criteria [16]:

User error protection (subcriterion of **Usability**): AMULET compiler provides an exception handling feature based on the formal definitions of mutation operators. This feature prevents incorrect atomic mutations (i.e., adding a transition between two states that does not exist in the state-machine diagram). Thus, the AMULET-based approach ensures that the model modification operations prevented by TTool's GUI are also impossible to apply from a script. Moreover, as noticed in almost every tested scenario (see Sect. 5) the AMULET-based mutation approach substantially reduces the number of needed manual operations. We believe that the conjunction of these two features leads to a decrease in the number of potential faulty modifications.

Operability (subcriterion of **Usability**): AMULET users only need to use a text editor, learn a syntax close to natural language and four simple commands. This avoids them from performing numerous GUI interactions in several panels — which can be tedious when modifying large models involving numerous blocks. Even if TTool mainly relies on a graphical language for designing SysML models, we believe that a textual language for describing mutations is the most efficient approach. Actually, the graphical support of mutations already exists (it is the model editor) and our case-studies show that it is less efficient than AMULET.

Performance efficiency: as highlighted by the case-studies results, when applying mutations on large models or on numerous similar models AMLUET enables for a more efficient modification process by reducing (sometimes substantially) the needed time.

6.2 Limitations and future works

Yet AMULET syntax and implementation can significantly be improved with respect to three aims. Firstly, AMULET currently does not provide syntactic shortcuts for applying the same mutation in several diagrams: for instance, adding a given state in every state-machine diagram of a model requires writing one AMULET line per state-machine diagram instead writing, for instance, a single `add state s0 in all blocks`. Therefore adding tokens enabling for multiple diagrams modification as in Wodel [13] is an interesting improvement perspective. Secondly, another implementation improvement we have identified is the integration of AMULET in TTool's GUI: currently, the compiler necessitates the use of an external text editor and the command-line interface. Therefore adding to TTool a panel for writing AMULET commands, including autocomplete features, and dedicated buttons for the `am/amb`, `ap` and `ad` commands should improve the compiler's operability. Integrating a pre-compilation feature that alerts the user if a mutation he has written will lead to a syntactically incorrect model is another approach to improving the compiler. Finally, AMULET current implementation exclusively targets one of the two SysML profiles supported by TTool: since several methods such as SysML-Sec [7] or W-Sec [24] rely on the both

Just Accepted

profiles and involve model mutations, it should be relevant to extend AMULET in order to cover the other SysML profile.

In addition, a limitation of our case-studies is that AMULET scripts and mutations in TTool GUI have been written and performed by a single expert. Thus conducting additional testing of AMULET with diverse case studies involving mutliple engineers of varying technical backgrounds and expertise levels is likely to provide (1) more reliable comparison metrics and (2) help in identifying additional strengths and limitations. Furthermore, conducting evaluations of AMULET on more ambitious case studies would yield more insightful feedback regarding the productivity gains within the context of model-driven engineering processes.

Finally, we believe that AMULET paves the way to several research directions relying on its capabilities. Firstly, it could provide a TTool feature for displaying the models dynamically so that the users can easily navigate between the successive versions of their models. This could easily be done by saving a history of the mutations successively applied along the model lifecycle, and replay them when needed. Secondly, a machine-learning algorithm integrated in TTool could be used to suggest modeling improvements (such as security countermeasures or algorithm optimizations) on the basis of the previously used modeling patterns: the output of this algorithm would be a list of AMULET commands to apply to the model under design. Finally, like in mutation-based testing approaches AMULET could be used to exhaustively explore the modeling alternatives for a given system, and automatically select the most efficient ones with respect to the results of safety, security and performance assessments embedded in TTool.

REFERENCES

[1] Bernhard K Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korošec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. 2014. Model-based mutation testing of an industrial measurement device. In *International Conference on Tests and Proofs*. Springer, 1–19.

[2] Bernhard K Aichernig, Klaus Hörmaier, and Florian Lorber. 2014. Debugging with timed automata mutations. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 49–64.

[3] Bernhard K Aichernig and Percy Antonio Pari Salas. 2005. Test case generation by OCL mutation and constraint solving. In *Fifth International Conference on Quality Software (QSIC'05)*. IEEE, 64–71.

[4] Mounifah Alenazi, Nan Niu, and Juha Savolainen. 2020. A novel approach to tracing safety requirements and state-based design models. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 848–860.

[5] Rabéa Ameur-Boulifa, Florian Lugou, and Ludovic Apvrille. 2019. SysML Model Transformation for Safety and Security Analysis. In *Security and Safety Interplay of Intelligent Software Systems*, Brahim Hamid, Barbara Gallina, Asaf Shabtai, Yuval Elovici, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Cham, 35–49.

[6] Ludovic Apvrille, Pierre de Saqui-Sannes, Hoana Hotescu, and Alessandro Tempia-Calvino. 2022. SysML Models Verification Relying on Dependency Graphs. In *MODELSWARD*. 174–181.

[7] Ludovic Apvrille and Yves Roudier. 2013. SysML-Sec: A SysML Environment for the Design and Development of Secure Embedded Systems. In *APCOSEC 2013*. Yokohama, Japan. https://hal.telecom-paris.fr/hal-02288385

[8] Ludovic Apvrille, Bastien Sultan, Oana Andreea Hotescu, Pierre de Saqui-Sannes, and Sophie Coudert. 2023. Mutation of Formally Verified SysML Models. In *Proceedings of the 11th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*.

[9] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. 2015. Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 653–683.

[10] Luciano C Ascari and Silvia R Vergilio. 2010. Mutation testing based on ocl specifications and aspect oriented programming. In *2010 XXIX International Conference of the Chilean Computer Science Society*. IEEE, 43–50.

[11] Pierre de Saqui-Sannes, Ludovic Apvrille, and Rob Vingerhoeds. 2021. Checking SysML Models Against Safety and Security Properties. *Journal of Aerospace Information Systems* (Nov. 2021), 1 – 13. https://doi.org/10.2514/1.i010950

[12] David P Gluch and Charles B Weinstock. 1998. *Model-Based Verification: A Technology for Dependable System Upgrade*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[13] Pablo Gómez-Abajo, Esther Guerra, and Juan de Lara. 2016. Wodel: a domain-specific language for model mutation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 1968–1973.

[14] Pablo Gómez-Abajo, Esther Guerra, Juan de Lara, and Mercedes G Merayo. 2021. Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling* 20, 3 (2021), 767–793.

[15] Kunxiang Jin and Kevin Lano. 2021. Mutation Operators for Object Constraint Language Specification.. In *STAF Workshops*. 128–134.

Just Accepted

[16] Attila Kovács and Kristóf Szabados. 2014. Test software quality issues and connections to international standards. *Acta Univ. Sapientiae, Informatica* 5 (05 2014), 77–102. https://doi.org/10.2478/ausi-2014-0006
[17] Aditya P. Mathur and Nils Ole Tippenhauer. 2016. SWaT: a water treatment testbed for research and training on ICS security. In *2016 International Workshop on Cyber-physical Systems for Smart Water Networks (CySWater)*. 31–36. https://doi.org/10.1109/CySWater.2016.7469060
[18] Lei Mi and Kerong Ben. 2011. A method of software specification mutation testing based on uml state diagram for consistency checking. *Procedia Engineering* 15 (2011), 110–114.
[19] OMG. 2014. *Object Constraint Language – Version 2.4*. Technical Report.
[20] OMG. 2017. *OMG Systems Modeling Language – Version 1.5*. Technical Report.
[21] Gabriel Pedroza, Ludovic Apvrille, and Daniel Knorreck. 2011. AVATAR: A SysML environment for the formal verification of safety and security properties. In *2011 11th Annual International Conference on New Technologies of Distributed Systems*. IEEE, 1–10.
[22] Percy Antonio Pari Salas and Bernhard K Aichernig. 2005. Automatic Test Case Generation for OCL: a Mutation Approach. *UNU-IIST Report* 321 (2005).
[23] Bastien Sultan. 2020. *Maîtrise des correctifs de sécurité pour les systèmes navals*. Ph. D. Dissertation. Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire.
[24] Bastien Sultan, Ludovic Apvrille, and Philippe Jaillon. 2022. Safety, Security and Performance Assessment of Security Countermeasures with SysML-Sec. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*. INSTICC, SciTePress, 48–60. https://doi.org/10.5220/0010832300003119
[25] Bastien Sultan, Ludovic Apvrille, Philippe Jaillon, and Sophie Coudert. 2023. W-Sec: A Model-Based Formal Method for Assessing the Impacts of Security Countermeasures. In *Model-Driven Engineering and Software Development*, Luís Ferreira Pires, Slimane Hammoudi, and Edwin Seidewitz (Eds.). Springer Nature Switzerland, Cham, 203–229.
[26] Bastien Sultan, Fabien Dagnat, and Caroline Fontaine. 2018. A Methodology to Assess Vulnerabilities and Countermeasures Impact on the Missions of a Naval System. In *Computer Security*, Sokratis K. Katsikas, Frédéric Cuppens, Nora Cuppens, Costas Lambrinoudakis, Christos Kalloniatis, John Mylopoulos, Annie Antón, and Stefanos Gritzalis (Eds.). Springer International Publishing, Cham, 63–76.
[27] John von Neumann, Arthur W Burks, et al. 1966. *Theory of self-reproducing automata*. Vol. 1102024. University of Illinois press Urbana.

Just Accepted