GRANDSLAM: Linearly Scalable Model Synthesis

Alexander Boll
University of Bern
alexander.boll@unibe.ch

***Abstract*—Commercial cyber-physical system (CPS) development tools are complex and safety-critical software, yet face two major testing challenges. First, the lack of formal specifications necessitates fuzzing for bug discovery. Second, the scarcity of large, open-source models hampers effective scalability testing and empirical research. To address these challenges, we present GRANDSLAM, a Simulink model synthesizer that scales linearly and generates models far larger than prior work. GRANDSLAM enables both fuzzing and scalability testing at unprecedented scales, synthesizing diverse models with over 1M blocks. Our evaluation reveals eleven confirmed bugs, and two confirmed scalability issues in Simulink, demonstrating its effectiveness.**

***Index Terms*—Interfaces, Fuzzing, Cyber-Physical Systems, Simulink, Equivalence Under Substitution, Reproducibility**

I. INTRODUCTION

Commercial cyber-physical system (CPS) development tools, such as MathWorks Simulink [1], are fundamental platforms for designing, simulating, and analyzing CPS models for embedded code implementation. However, these intricate software systems are prone to bugs, posing significant challenges for developers, and risks in safety-critical domains. Detecting these bugs is challenging due to the lack of formal specifications and closed-source tools, so current research relies on fuzzing to uncover them [2]–[5].

Yet, current state-of-the-art CPS fuzzers are limited to small and medium-sized models, while synthesis approaches struggle to generate valid models at scale [3], [6]. These limitations add to broader challenges in tool simulation model research, including reproducibility, replicability [7], and tool development. Large models are particularly critical for testing scalability of tools, and conducting empirical research, both of which rely on models of sufficient size and complexity [8]–[12]. Together, these gaps undermine both the reliability of CPS tools and the rigor of research aimed at advancing them.

To address these challenges, we develop a Simulink model synthesizer that (i) scales more effectively than prior approaches, enabling the synthesis of very large valid models, and (ii) supports the synthesis of models with various constraints, shapes, and sizes. Our synthesizer enables fuzzing and scalability testing, and also provides dataset augmentation for empirical research. We synthesize models larger than any currently available open-source models. While our primary focus in this work is on Simulink as a representative CPS development tool, we design our approach to be generalizable to other modeling and textual languages.

Our synthesis approach leverages our novel concept of *equivalence under substitution*. In a nutshell, our approach guarantees that a valid model remains valid if a model element is substituted. Given a dataset of models, we form equivalence classes, where a model element of a class can safely be substituted with any other element of its class. This approach enables us to implement our linearly scaling Simulink synthesizer GRANDSLAM (**G**iant **R**ecomposition **AND** **S**ynthesis of **LA**rge **M**odels) that synthesizes Simulink models by substituting Subsystems – Simulink's primary abstraction mechanism – that have equivalent interfaces. GRANDSLAM is written in MATLAB and comes with five highly configurable synthesis strategies, generating models of various shapes and sizes. In the evaluation of our approach, we found eleven confirmed bugs, and two confirmed scalability issues in Simulink. We further found missing elements in the Simulink Subsystem interface definition by Jaskolka et al. [13].

Overall, GRANDSLAM enables researchers and developers to systematically test and validate CPS tools at scale, improving both the reliability of the tools and the reproducibility of research. This manuscript is a revised version of material originally presented in the author's PhD thesis [14], which is available under a CC-BY license. We summarize our contributions as follows:

- We develop a novel approach for synthesizing statically valid and/or compilable Simulink models.
- We implement our approach in a prototype GRANDSLAM with five strategies for synthesizing models of various shapes and sizes. To the best of our knowledge, this is the first model synthesizer that scales linearly.
- Our fuzzing revealed eleven confirmed bugs, and two confirmed scalability issues in Simulink.
- We extend the definition of Simulink Subsystem interfaces from prior literature.
- We provide all GRANDSLAM artifacts as open-source and make them available online.[1] Our artifact package includes 4,600 highly diverse synthesized Simulink models, of which 100 models are larger than any previous open-source Simulink model, known to us.

II. SIMULINK BACKGROUND

Simulink [1] is a block-diagram environment integrated with MATLAB and widely used for modeling, simulating, and analyzing dynamic systems across domains such as control engineering, signal processing, and embedded systems design. Its graphical interface enables intuitive representation of complex systems, while its tight integration with MATLAB facilitates

[1]Code available at https://zenodo.org/records/18900319 Data available at https://zenodo.org/records/17296885.

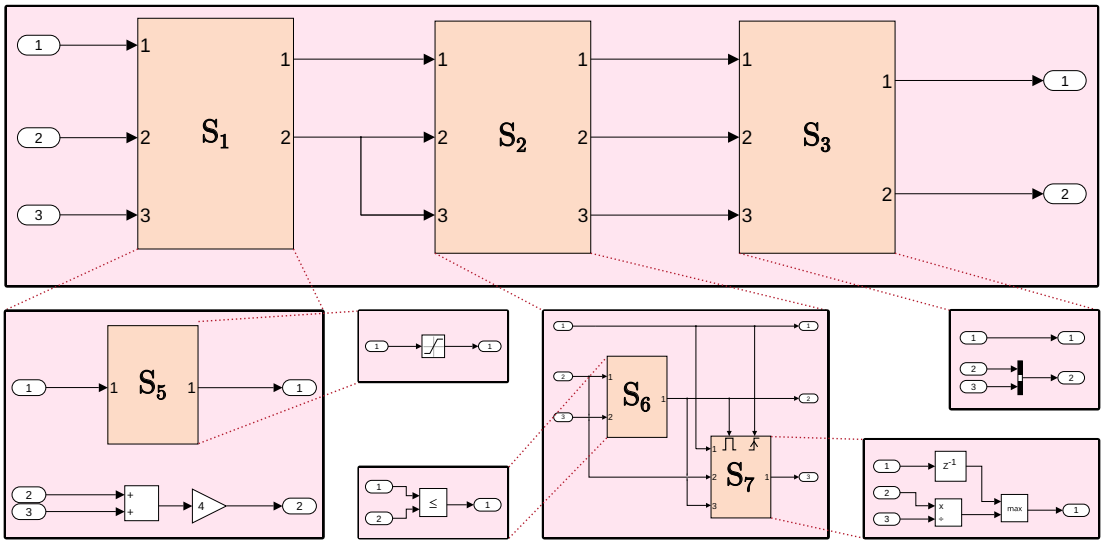


Fig. 1: Complete Subsystem breakdown of an exemplary Simulink model, showing all its views simultaneously.

algorithm development, data analysis, and automated code generation for real-time applications.

Simulink models are block diagrams composed of Blocks and Lines as their core model elements. Blocks are connected by Lines and Lines carry information from Block Outports to Block Inports during model simulation. Lines can hold values of different types (e.g., boolean, double, int16) and dimensionality (e.g., scalars, 1-D vectors, matrices). Meanwhile, Blocks transform their input from Inports to output at their Outports. To handle complex models, developers can use Subsystem Blocks and use them to hierarchize the model. A model's root Subsystem and its content is given at the top of Figure 1. We show all Subsystems' internals at once (normally only one is visible at a time), e.g., the Subsystem S_1's internals are shown on the bottom left of Figure 1. S_1 contains one nested Subsystem: S_5. In addition to normal Inports and Outports, Subsystems may also have a SpecialPort. SpecialPorts in Simulink are: EnablePort, TriggerPort, ActionPort, PMIOPort, or ResetPort. There is an Enable- and a TriggerPort on top of the Subsystem S_7 in Figure 1. A Subsystem's closest analog in textual languages are functions, or methods. While the figure contains several other model elements, only the following ideas are essential for the subsequent understanding: models may have (nested) Subsystems and each Subsystem has a number of typed Ports connecting them with other model elements.

Simulink models adhere to multiple levels of *validity* [3], [15]. In this work, we focus on models being *statically valid* (pass Simulink's loading checks, e.g., Lines connect existing Blocks), *compilable* (e.g., mismatched data types), and *simulable* (e.g., division by zero). Each level presupposes the prior one. As we will show in Section V-A, most compilable models are also simulable.

III. Subsystem Equivalence under Substitution

A. Definition of Subsystem Equivalence

Our main idea in this paper is to synthesize models by substituting Subsystems. A *substitution*, is a model transformation that removes a Subsystem S_o (original) and replaces it with a Subsystem S_s (substitute) exactly in its place, including all prior Line connections. A substitution does not require the two Subsystems to be from two different models, i.e., the most trivial substitution is to substitute a Subsystem with itself ($S_o = S_s$). To ensure meaningful substitutions, we restrict S_s to those that maintain the model's validity after substitution. Specifically, we require the resulting model to retain *static validity* and *compilability*. Unlike textual programming languages, models can provide value even if they are not compilable, as long as they are statically valid [16]. By supporting multiple levels of validity, our approach enables fuzzing and scalability testing for a wider range of Simulink models and tools. As the metamodel of Simulink isn't published, we conjecture substitution validity based on the official Simulink documentation and Jaskolka et al.'s Subsystem interface definition [13]. Our later experiments strengthen and extend their interface definition and our conjectures.

We can ensure a model remains statically valid, if after the substitution S_s is connected exactly at the same Lines as S_o. This gives our first definition of *equivalence under substitution*, or *untyped-equivalence*: S_o and S_s must have the same number of Lines at their Inports, SpecialPorts, and Outports. E.g., the Subsystems S_1 and S_3 in Figure 1 are untyped-equivalent; all other Subsystems of the model have pairwise different numbers of connected Inports and Outports and are not substitutable. With this definition, a substitution becomes easy: we remove a Subsystem, and replace it by connecting the old Lines with the Ports of the new Subsystem.

For the second equivalence level, we aim to ensure compilability. During compilation, Simulink checks the types and

dimensions of Lines: the type and dimensionality from every Outport must conform (in general be identical) to the type that is expected at the other end of the Line (an Inport). This inspires our second definition of *typed-equivalence*: S_o and S_s must be untyped-equivalent, with all connected Lines having identical types and dimensions. Note that this definition is very conservative. During our automatic inspection, we can only observe that a Port *in its current context* supports, e.g., int32. It may also support, e.g., int16, or int8, though.

B. Equivalence Classes and Relaxed Equivalence

All Subsystems that are equivalent to each other fall into (typed or untyped) *equivalence classes*, i.e., every Subsystem is substitutable by all members of its equivalence class. Each class is defined by inputs and outputs, i.e., the 'interface' of the representative of its class.

With our goal of substitution in mind, equivalence classes should not be singletons, i.e., sets with one element, in order for substitution to be possible. To reduce the number of singletons, we define four variations of *relaxation*, that follow the common adapter patterns for harmonizing function calls from classical programming languages. In *order-relaxation* (argument reordering), we allow for *rewiring* of Lines during substitution: the order of the Inports and Outports, and their types or dimensionality may vary, as long as there is a match for each of them. E.g., in Figure 2a for each input and output Port there is an equivalent input and output Port in Figure 2b; inputs 2, and 3 and the outputs need to be rewired. In *type-relaxation* (parameter coercion and return value mapping), we allow for the conversion of data types that can be safely cast, e.g., `int16` to `int32`. Lastly, we define *number-relaxation* (argument truncation and return value suppression): a Subsystem is *number-relaxed* to another if it uses a subset of its inputs and produces a superset of its outputs; *dimensionality-relaxation* can be defined similarly. The concepts of *type-relaxation*, *number-relaxation*, and *dimensionality-relaxation* are shown in Figure 2c.

C. Impact of Substitution

In Figure 3, we depict the impact of substitutions on a model. On the left, we can see the complete Subsystem hierarchy, or *Subsystems tree* of the model from Figure 1. As S_1 and S_3 are equivalent, they can replace each other. The Subsystem tree of S_1 is bigger than the one of S_3: the resulting model on the right grew sligthly after substitution. This observation is key to our GIANT synthesis strategy (cf. Section IV-D), where we repeatedly apply such substitutions.

IV. Implementing our approach in GRANDSLAM

To implement our approach, we first mine a given dataset of models to construct two hash tables: `interface2ids`, which uses a description of a Subsystem's interface as a key to look up its equivalence class, and `id2subinfo`, which is used to look up properties of Subsystems. In the later synthesis step, we use these hash tables extensively. We thus first present the mining and their construction.

A. Encoding and Mining Equivalence Classes

The mining of equivalence classes iterates over every model's Subsystem and each of its Inports, Outports, and SpecialPorts. We encode the hash-key of a Subsystem interface as a string. For untyped equivalence, we quantify their number, such as '`3,2`' for 3 Inports, 2 Outports, and no SpecialPort.

For typed equivalence, we have to differentiate Port type and dimensionality. For each Port, we denote its type and dimensionality as '`<type><dim1>+...+<dimN>`'. The added '+' differentiates a 2x2 matrix from a 22-row vector. For SpecialPorts, we denote their name. Next, an Inport, Outport, and SpecialPort hash is formed by joining their individual Port hashes with a '`:`'. Finally, the overall interface hash is formed by joining the three partial hashes with a '`,`'. Some examples: The hashes of the Subsystems in Figure 2 are '`int3:double2:int1,bool1:int1`' (Fig. 2a), '`int3:int1:double2,int1:bool1`' (Fig. 2b), and '`int3:int1,bool1:single1:int1`' (Fig. 2c).

The values of `interface2ids` are all the *identifiers* of Subsystems that belong to the equivalence class of its interface hash-key. We construct an identifier that describes the absolute position of the Subsystem by combining: the Simulink model file it belongs to, the relative path of the Subsystem within the model, and the Subsystem's name. The identifier of S_6 in Figure 1 may thus be: "`ExampleModel/`S_2`/`S_6`"`. Given an interface of a Subsystem, one can thus quickly lookup equivalent substitute candidates in `interface2ids`.

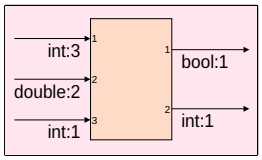
To mitigate filling the equivalence classes with clones, e.g., from libraries, and thus substituting a Subsystem with itself, we deduplicate each equivalence class. We classify Subsystems from the same equivalence class as duplicates, when they have the same number of direct children, number of Subsystem children, and local Subsystem tree depth. These are metrics we compute for our synthesis strategies, and suffice as a quick heuristic for our prototype. However, the heuristic does not guarantee that every Subsystem of an equivalence class is truly unique.

B. Encoding Relaxation

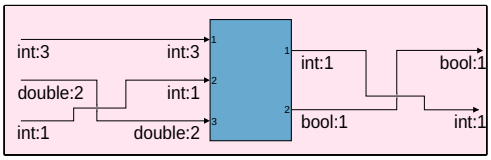
We encode order-relaxation by *normalizing* the interface hashes. This can be achieved, by sorting the Inport, Outport, and SpecialPort hashes alphabetically, before joining them. Both Subsystems in Figures 2a and 2b get the hash '`double2:int1:int3,bool1:int1`' proving they are order-relaxed-equivalent. The other three relaxations could be achieved by adding a Subsystem not only to its direct equivalence class, but also to every other equivalence class, that is more specific. We only followed up to implement order-relaxation in our prototype.

C. Mining Subsystem Properties

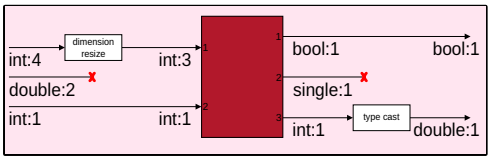
We use the hash table `id2subinfo` to look up various properties of Subsystems. The previously introduced identifiers are the keys of `id2subinfo`. The most valuable properties of a Subsystem stored in `id2subinfo` are the identifiers of all direct child-Subsystems as well as its own interface



(a) Subsystem S_1 from Figure 1.



(b) An adapted order-relaxed-equivalent Subsystem.



(c) An adapted Subsystem under number-, type-, and dimensionality-relaxation.

Fig. 2: For substitutions, adaptations may be required, such as rewiring, dangling lines, type casting, or resizing dimensionality.

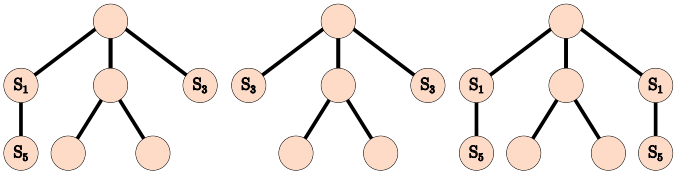


Fig. 3: Substituting within the model from Figure 1 (left): replacing S_1 with S_3 (middle), replacing S_3 with S_1 (right).

which enables the lookup of the Subsystem's equivalence class via `interface2ids`. Depending on synthesis goals, one can also store other properties. We store the Subsystem's own depth within the model, the depth of its local Subsystem tree, and the number of its local Blocks.

D. Synthesizing and Synthesis Strategies

Using the combination of `id2subinfo` and `interface2ids`, we can now elegantly synthesize models using two paradigms:

1. Synthesizing from the ground up: This paradigm starts by synthesizing a new model M using a root Subsystem r from the dataset as the seed. We substitute each of r's Subsystem children, by another Subsystem from its respective equivalence class, if a suitability criterion for the substitution is passed. Depending on relaxation, adaptions may have to be made for a valid substitution (cf. Figures 2b and 2c). This process is then repeated until the Subsystem tree is completed, i.e., all leaf Subsystems are processed. The pseudocode of our approach is given in Listing 1.

We define four suitability criteria (i.e., synthesis strategies) within this paradigm: RANDOM: any Subsystem from the equivalence class passes. ROLE: given a separate 'role model' M' from the dataset, we synthesize M with an isomorphic Subsystem tree to M'. A substitute Subsystem is thus suitable,[2] when it has the same number of children as the role model's Subsystem has at the same position. BUSHY: From a sample from the equivalence class, the Subsystem with the most children is suitable. DEEP: From a sample of Subsystems from the equivalence class, the one with the biggest Subsystem tree depth is suitable.

We propose RANDOM and ROLE to synthesize models that are similar in size to the original models, and BUSHY and DEEP to synthesize large models. We expect BUSHY to create

[2] `isSuitable` in line 14 of Listing 1 needs M' as an additional parameter.

```
1  Input: Subsystem tree rooted in r
2  Output: The synthesized model M
3
4  //Init model M and Stack S of TODO Subsystems
5  M = r;
6  S = id2subinfo(r).childSubsystems;
7  while S ≠ ∅ do
8      s = pop(S);
9      //Lookup equivalent Subsystems
10     E = interface2ids(id2subinfo(s).interface);
11     //Choose k elements e from E as substitute
       candidates; avoids uniform substitutions
12     E' = sample(E, k);
13     for e ∈ E' do
14         if isSuitable(s, e, M, E') then
15             M = substitute(s, e, M);
16             s = e;
17             break;
18         endif
19     endfor
20     //Repeat for substitute's children
21     S = S.append(id2subinfo(s).childSubsystems);
22 endwhile
```

Listing 1: Pseudo-code of synthesis from the ground up.

wide and dense Subsystem trees, and DEEP to create deep Subsystem trees.

Without further moderation, our experiments showed that BUSHY and DEEP tend to grow infinitely. We thus introduce a backtracking mechanism that activates, when a previously selected maximum depth of the model is overshot. The backtracking voids substitutions in shallower levels and tries other candidates instead, until the complete Subsystem tree obeys our constraints. A similar backtracking also activates for ROLE, when no local candidate Subsystems can be found to keep the local Subsystem trees isomorphic.

Note, that while we choose root Subsystems as the seed of our models, any other Subsystem also works. We focus on root Subsystems as seeds because the dataset models are rooted by them by definition. However, if the chosen root Subsystem has no children, the synthesis process terminates immediately. In such cases, another root must be tried, except if M' in ROLE is also just a singular Subsystem model without children.

The big models constructed by BUSHY and DEEP are assembled from hundreds or even thousands of models. Since Simulink's model loading, and Subsystem substitution are computationally expensive – and must be executed hundreds or thousands of times – we also developed an alternative paradigm for large models, based on fewer unique models,

and fewer substitutions:

2. Recomposing existing models: The second paradigm recomposes existing models, embodying our fifth and final strategy: GIANT. Instead of starting the synthesis with a single root Subsystem – for our last strategy GIANT, we use a complete model as its seed. In every step, we select a partial Subsystem tree from anywhere in the model and try to substitute it with an equivalent Subsystem tree (cf. Figure 3), when it is suitable. To quickly grow a model, we found that alternating between two suitability criteria at each depth level improves the synthesis process of GIANT: substitute Subsystem has deeper tree depth, and substitute Subsystem has more direct children. To create large models, we let a model grow with GIANT, until the recomposed model trumps the biggest models from the dataset in their maximum number of model elements, maximum depth, and maximum number of Subsystems all at once. GIANT can be stopped at any point, as after each substitution in GIANT the model's Subsystem tree is complete. Stopping DEEP or BUSHY, however, needs a phase of capping off all unprocessed Subsystem trees with leaves. The other main difference between them is that GIANT recombines existing models, where large parts of the Subsystem trees remain intact. BUSHY and DEEP create a patchwork of Subsystems from various models, because each node of the Subsystem tree is chosen individually.

In our implementation, we decoupled the building of a *blueprint* of the model and the actual *construction* of the model with Simulink API calls. This enables 'completely dry builds', where the Subsystem tree outline is completed but no actual Simulink model is created yet. The construction phase of GIANT copies Subsystem trees (cf. Figure 3), the others synthesize their Simulink models one model element at a time.

V. METHODS

A. Dataset Mining

In order to construct our hash tables `interface2ids` and `id2subinfo`, as described in Section IV, we mine the SLNET data set by Shrestha et al. [17]. SLNET is a diverse and large dataset of 9,095 Simulink models stemming from 2,837 Simulink repositories. The set encompasses various domains, model sizes, modeling goals, and has become a benchmark in the field as it has been used in a number of empirical studies [9], [12], [18]–[22]. Before mining the Subsystem equivalence classes, we filter the SLNET model set for *stable* models: each model should be statically valid, i.e., free of errors while loading, it should also not cause any Simulink bugs, while loading, compiling, simulating, saving and closing. We compile every stable model and simulate it for 10 seconds to assess both compilability and simulability. We do this by performing all the mentioned model operations and restarting the script, whenever Simulink crashes (hard crashes or stops without a catchable exception). This filtering step is vital for our later scripts as they have to handle thousands of models automatically. Especially the synthesis of large models becomes brittle if just a few models in the dataset are unstable. To further 'tame' the models, we remove all Callbacks (cf. our results of **RQ2.3**). After filtering our dataset, it contains 5,912 models that we mine for untyped-equivalence and 761 models for typed-equivalence – both in bold in Table I.

TABLE I: Summary of the SLNET dataset.

	SLNET models	**Stable**	**Compilable**	Simulable
Models	9,095	**5,912**	**761**	697
Subsystems	N/A	**313,820**	**1,807**	N/A

B. Research Questions

As we intend to compose models by substituting equivalent Subsystems, we first evaluate:

RQ1: Is our approach of repetitive substitution feasible? Our substitution approach assumes a small set of equivalence classes, ensuring that a significant portion of Subsystems can be replaced by equivalent alternatives. Untyped equivalence might seem to limit the equivalence classes – a maximum of, e.g., 10 Inports and Outports yield just 121 combinations – but SpecialPorts can inflate this number considerably. For typed-equivalence, it may be possible that most Subsystems are fragmented into singletons classes, as inputs could vary by dimension and type. Additionally, not every model is compilable, and thus the number of Subsystems that are available for our analysis is lower from the start. If too many equivalence classes are singletons, then we relax our equivalence requirements (cf. Section III-B).

We further evaluate the feasibility of our proposed strategies (cf. Section IV-D). We run all five strategies with both typed and untyped equivalence, and measure how they perform in regard to success rate and scalability. For RQ1, we answer the following sub-questions:

- **RQ1.1** How are the equivalence classes constituted?
- **RQ1.2** Can we synthesize models with untyped and typed equivalence classes?
- **RQ1.3** How do the blueprint and construction phases scale?

Once we are satisfied with the general feasibility of our approach, we investigate the synthesized models: **RQ2: What are the synthesized models' characteristics?** Our goal in RQ2 is to gauge the synthesized models broadly, before we use them for fuzzing or scalability tests. We measure basic model metrics [9], [16] to answer the following sub-questions:

- **RQ2.1** How large are the synthesized models?
- **RQ2.2** How diverse are the models?
- **RQ2.3** How many models are compilable, and simulable?

Lastly, we evaluate the suitability for our use cases:

RQ3: Are the synthesized models practical for fuzzing and scalability tests? For fuzzing, we analyze all program halts encountered during synthesis and analysis. Most halts stem from implementation bugs in our scripts (which we fuzz 'along the way'), while others reveal Simulink bugs, which we report to MathWorks. Additionally, we identify a few bugs through ad-hoc testing. Related fuzzing approaches are discussed in Section VIII, as they primarily target compiler bugs, whereas our focus is on a broader range of model actions.

TABLE II: Equivalence classes across configurations.

	#classes	singleton class %	mean other Subsystems	mean other models
untyped				
with duplicates	326	20	49,582	1,927
unique	**326**	**42**	**532**	**508**
typed, strict				
with duplicates	202	46	273	249
unique	202	74	40	38
typed, order-relaxed				
with duplicates	199	44	273	249
unique	**199**	**72**	**40**	**39**

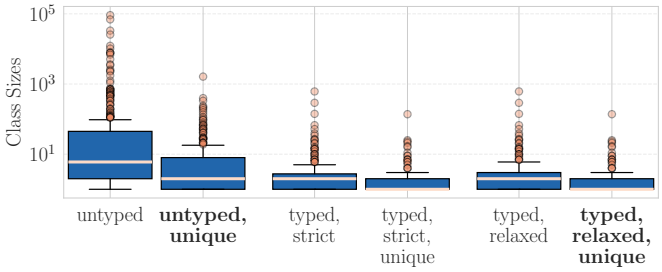
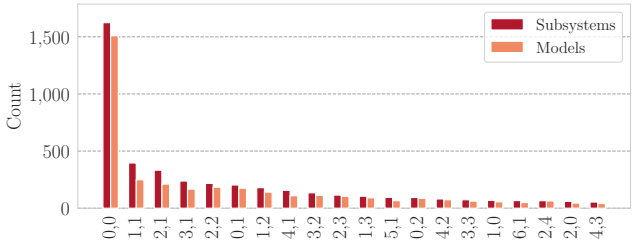
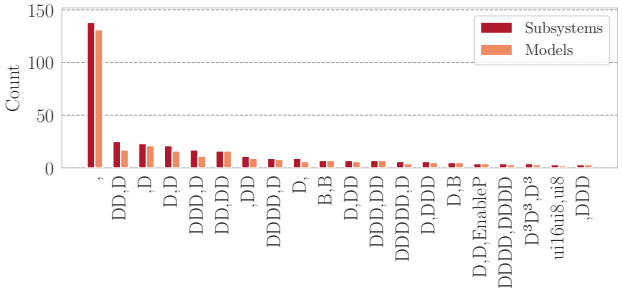


Fig. 4: Box plots of equivalence class sizes.



(a) Interface hashes of untyped, unique equivalence classes.



(b) Interface hashes of typed, relaxed, unique equivalence classes.

Fig. 5: Distribution of the 20 most frequent equivalence classes and their occurrences in Subsystems and Models.

To assess scalability, we reuse the synthesized models from RQ2 and apply various model actions to them. By iteratively computing regression curves while relaxing model size constraints, we determine whether larger models provide deeper insights into scalability behavior. Comparisons with related approaches are deferred to Section VIII, as their synthesized Simulink models are limited in size and thus less informative for scalability analysis.

VI. Results

RQ1: Is our approach of repetitive substitution feasible?

RQ1.1 How are the equivalence classes constituted?

Table II summarizes our findings for the first research question, highlighting the strong impact of deduplication – which removes both intra- and inter-model Subsystem clones – on reducing equivalence class sizes. Since substituting a Subsystem with an identical clone is meaningless, and the two typed configurations differ only marginally (order-relaxing unifies just 3 classes), we focus our analysis on the bolded configurations in Table II and Fig. 4.

42% of the untyped equivalence classes are singletons, which means the equivalence class' representative has a unique interface, and we cannot find a substitute. This number goes up to 72% for the typed equivalence classes. Despite the prevalence of singletons, the data in the final two columns of Table II demonstrates that a randomly selected typed Subsystem retains substantial substitutability – with an average of 40 distinct replacement possibilities from 39 unique models.

Our analysis of the most frequent equivalence classes and their interfaces (cf. Figure 5) reveals that the most common classes have few Inports and Outports, and seldomly feature SpecialPorts. By far the most common Line type is unidimensional `double`, but we also found `boolean`, `uint8`, `uint16`, and three-dimensional `double` in the top 20 interfaces. Furthermore, a class' Subsystems are spread over many models and not just populated by a single model's Subsystems, noticeable by the similar heights of the orange and dark red bars in Figure 5.

RQ1.2 Can we synthesize models with untyped and typed equivalence classes?

We ran each of our five strategies once with the untyped, unique equivalence classes, and once with the typed, relaxed, unique equivalence classes (bold in Table II and Fig. 4). With Random and Role we generated 1,000 models, with the others 100 models, totalling 4,600 models over all runs. Every synthesized model is a statically valid Simulink model and can be opened and edited in Simulink.

While all strategies met their generation quotas, Bushy and Deep exhibited higher failure rates (cf. the first row in Table III). This is due to the high number of singletons in the blueprint phase, which leads to extensive backtracking until a seed is abandoned after three attempts per Subsystem tree node. Giant, however, avoids these bottlenecks by substituting entire Subsystem trees rather than individual Subsystems. Failures in the construction phase were minimal, occurring in only 0.8% of attempts due to implementation bugs.

RQ1.3 How do the blueprint and construction phases scale?

The first two rows of Figure 6 show that the blueprint phase (1,000-10,000 Blocks/s for the large strategies) is much faster than the construction phase (100-1,000 Blocks/s). The linear relationship for untyped Bushy between the times in blueprint and construction phase vs. the number of elements is clearly

TABLE III: Various metrics across our strategies.

	RANDOM	ROLE	BUSHY	DEEP	GIANT
Blueprint failure rates					
untyped	.000	.079	.099	.174	.007
typed	.000	.040	.888	.996	.145
mean pairwise Jaccard similarity (models)					
untyped	.003	.002	.265	.249	.398
typed	.002	.002	.178	.309	.290
mean pairwise Jaccard similarity (Subsystems)					
untyped	0	0	.007	.001	.058
typed	.001	.001	.076	.023	.201
% of SLNet models covered					
untyped	48.3	44.1	79.5	50.2	47.9
typed	87.9	81.2	56.4	61.9	23.5
% of SLNet Subsystems covered					
untyped	4.8	3.7	79.1	15.1	69.6
typed	78.1	60.1	69.9	75.8	52.2
% compilable					
untyped	14.5	13.6	0	0	0
typed	94.1	96.3	19	23	24
% simulable					
untyped	13.9	12.9	0	0	0
typed	89.9	92.9	15	23	15

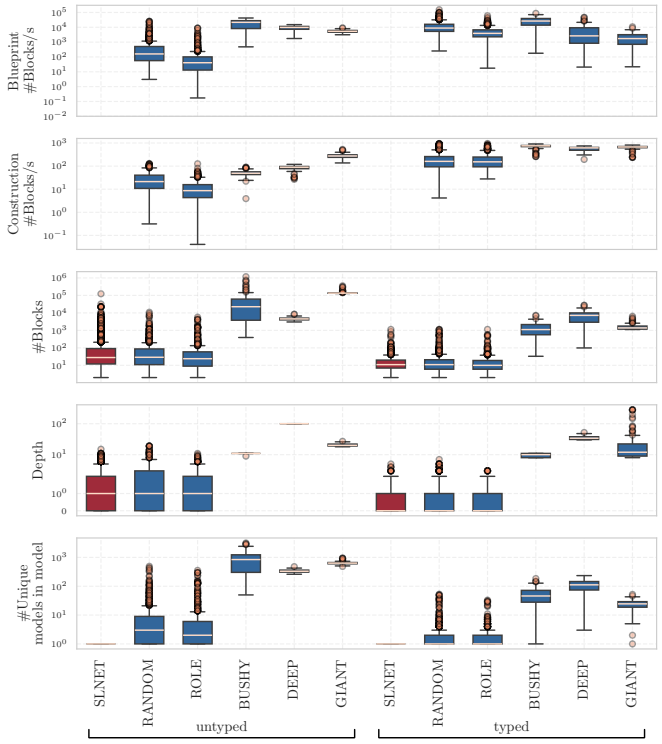


Fig. 6: Box plot comparison of SLNET and our synthesis strategies. All plots are scaled logarithmically, except 'Depth' which is scaled symmetrically logarithmic [23]. All times are measured on an MX Linux laptop with an i9-13980HX CPU, 94GB RAM, and MATLABR2025a.

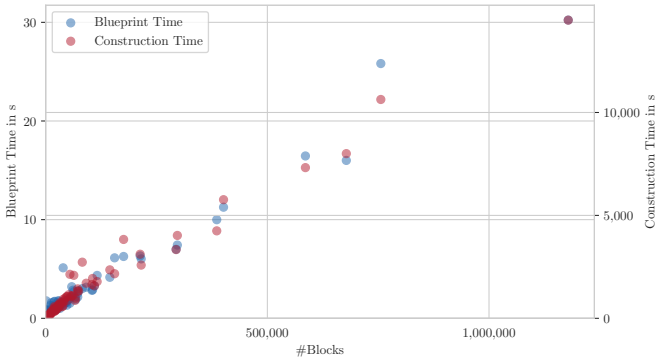


Fig. 7: The linear scaling of untyped BUSHY.

visible in Figure 7. The other methods also scale effectively linear, though with less clarity. We highlight BUSHY because it generated the largest models by a significant margin.

RQ2: What are the synthesized models' characteristics?

RQ2.1 How large are the synthesized models?

The third and fourth rows of Figure 6 show that RANDOM and ROLE generate models with similar sizes to their SLNET datasets. BUSHY, DEEP, and GIANT's models are much larger. The untyped GIANT models all have more than 108,523 Blocks (typed more than 1,122) – the maximum size in SLNET. The largest model of BUSHY comprises more than 1.1 million Blocks. The typed models exhibit a reduction in size by an order of magnitude or greater. For untyped models, our depth constraints (8-12 for BUSHY, 50-99 for DEEP) were nearly always maxed out. We lowered these constraints for typed DEEP (30-50), otherwise the failure rates were impractically high.

RQ2.2 How diverse are the models?

The final row in Figure 6 shows the number of models whose Subsystems were recomposed into each new model. For example, GIANT recomposed Subsystems from approximately 600–1,000 different models per synthesized model. The large strategies lead to very high diversities with more than 100 models for typed and more than 10 models for untyped classes. Judging the mean pairwise Jaccard similarities [24] of models and Subsystems (cf. row three and four in Table III), we find all strategies to be highly dissimilar. Table III further gives the percentages of SLNET models/Subsystems that were covered by a strategy's synthesized models, e.g., the 100 untyped BUSHY models used nearly 80% of the models and Subsystems from the SLNET dataset for their synthesis.

RQ2.3 How many models are compilable, and simulable?

The last two rows of Table III show that the typed models have a much higher chance of being compilable, and the large strategies rarely produce compilable or simulable models. We found the low number of compilable typed models to stem from an incomplete interface definition – specifically, the one we adopted from Jaskolka et al. [13] in our definition of equivalence (Section III).

In our experiments, we found three violations to their definition, where model elements have global dependencies

or side effects: (1) Callbacks: models, Blocks, and Ports may have Callbacks, that are called and executed by various events. These events can be: the model is loaded, closed, it is saved, a Block is renamed, the model simulation starts, etc. These Callbacks are common MATLAB scripts, and they can – in rare cases – rename, or delete other model elements, which may *dynamically change a Subsystem's interface* and thus equivalence class. When synthesizing models with thousands of Subsystems, a single bad Callback can break the whole model. We thus repeated all of our experiments with the previously described 'taming' of the dataset. (2) Asynchronous Blocks: Certain Blocks in Simulink require the entire model, including all other Subsystems, to operate under asynchronous semantics. Thus, each Subsystem's interface needs to be extended with a label 'synchronous' or 'asynchronous'. (3) Locality of type and dimension declaration: Signal types and dimensions can be the result of (i) manual declaration at some point in the model, (ii) implicit generation from a model element that can only generate such type and dimension, (iii) inheriting the type and dimension from either (i) or (ii) from the prior Signal path. Thus, our equivalence definition for each type and Signal needs to be extended with labels (i)-(iii). During the synthesis, one then needs to order Subsystems from Signal sources to sinks (source-sink ordered depth first traversal), to guarantee valid type and dimension propagation in the synthesized model. Cross-dependent type propagations, where Subsystem A needs B, and B needs A for a Signal's type, are not resolvable with this concept though.

For the simulability, we only found divisions by zero, or algebraic loops to be causing halts – both of which our equivalence approach is not intended to catch.

RQ3: Are the synthesized models practical for fuzzing and scalability tests?

In our experiments, we found eleven confirmed bugs (cf. Table IV). Of these, the six hard crashes are critical, as it can lead to data loss. Bug 9 is system critical, as all available system RAM will be hogged. In addition to the bugs listed in Table IV, we found five bugs (two inconsistent outputs during 'taming'; two memory leaks and a failed assertion with synthesized models), which we experienced multiple times but could not consistently reproduce.

For the scalability tests, we computed regression curves for the model actions in Table V. Only 'finding clones in a model' and 'closing a model' showed quadratic scaling. For 'finding clones', the quadratic regression started to be the best fit at around 10,000 elements – for 'closing models' very large models with more than one million elements had to be included. The coefficient of determination R^2 [25] in the last column shows that all regression curves are strong fits. Because of bug 9, we could not perform an automated RAM regression analysis. Our preliminary manual analysis suggests that Simulink allocates roughly 2,900 bytes of RAM per model element, scaling linearly. After we reported the Close Model time scaling and our preliminary RAM analysis to MathWorks, they confirmed both as scalability issues that they plan to fix in future releases.

TABLE IV: Summary of confirmed bugs.

Number	Case Number	Description of Bug
Discovered during 'taming'		
1	07130317	Hard crash in `set_param`
2	07131902	Hard crash in `set_param`
3	07144514	Hard crash in `set_param`
4	08020574	Message box flood in `compile`
5	08020724	Irrecoverable exception in `load_system`
Discovered during synthesis experiments		
6	06653446	Loaded model throws exceptions, errors
7	08112233	Irrecoverable exceptions in `compile`
8	08128597	Hard crash in `compile`
9	08147973	RAM usurped in `close_system`
10	08160317	Hard crash in `findClones`
11	08160318	Hard crash in `findClones`

TABLE V: Scalability regression curves $\mathit{Time} = an^2 + bn$ and $\mathit{Space} = bn$, with $n =$ number of model elements.

	a (quadratic)	b (linear)	R^2
Time in microseconds			
Load Model	—	30.947	.98
Find Elements	—	.242	.81
Save Model	—	27.610	.95
Find Clones	.007,978,387	−178.301	.95
Close Model	.000,004,412	7.603	.93
Space in Bytes			
File Size	—	122.811	.98

VII. Discussion

A. Suitability and versatility of our approach

All five GRANDSLAM strategies performed well in the untyped setting. Even with the high number of singleton equivalence classes in the typed setting, all strategies succeeded in synthesizing models, though failure rates increased and compilability rates decreased significantly. We thus suspect that our typed setting approached the upper limit of singleton equivalence classes for which our strategies remain effective. The untyped setting was far more permissive, and it was there that we synthesized the largest statically valid models – by a substantial margin – of which we are aware. The largest statically valid model was generated by untyped BUSHY and has more than 4.3M model elements, 1.1M Blocks, 110k Subsystems, 80k of which are unique and sourced from more than 3k unique models. This is about 400 times larger than synthesized Simulink models in prior work (cf. Section VIII), and much larger than typical industrial models that seldom have more than 20k Blocks [16].

While every model we generated was statically valid, the success rate of creating dynamically valid models was much lower (cf. **RQ2.3**). Still, 66 of the 300 larger typed models we generated are compilable. The largest models that are compilable/simulable were generated by typed GIANT and encompass 1,660/1,419 Blocks, 148/148 Subsystems stemming from 16/12 unique models.

Our five strategies already demonstrate the versatility of our approach in the resulting models and the process, i.e., by dynamically switching the `isSuitable`-function based

on the current synthesis status. Completely different strategies are easily integrated, with our approach: if one, e.g., desires to synthesize models with many Blocks of type t, one can extend the `id2subinfo` in Section IV-C with the count of t. For speed and effectivity of `sample`, one can also sort each equivalence class in `interface2ids` by number of t.

In this work, we tried five strategies in different settings, but left a number of variation points unexplored. (1) While order-relaxation has nearly no impact at all (Section VI), number-, type-, and dimensionality-relaxation may offer a significant reduction of singleton classes for the typed setting. (2) The deduplication of equivalence classes may be sharpened with a better heuristic with less false negatives and false positives. (3) Our strategies always choose root Subsystems as their root Subsystem, but Subsystems from any level could be a model's root – the synthesized models may be smaller, on average though. (4) Furthermore, our strategies have many more tunable parameters, which we chose by intuition and did not systematically explore, e.g., the sample sizes k, minimum/maximum depth of Subsystem trees, etc.

B. Fuzzing and scalability potential

While our approach uncovered eleven bugs, only bug 9 was directly attributable to large model size. Reproducing bugs with large models also proved complex, as it required replaying time-consuming synthesis sequences (cf. Figure 7). In fact, MathWorks declined to fix one reported issue due to its complex reproducibility, highlighting a broader challenge in fuzzing: the value of bugs triggered by nonsensical or overly complex inputs [26]–[28]. This suggests that synthesizing many smaller models may be a more efficient fuzzing strategy.

In contrast, we view large models as excellent subjects for scalability testing, as their performance provides insights into how their underlying dataset – from which they were synthesized – would behave at scale. Our scaling experiments already revealed a surprising asymmetry in core Simulink tool performance: while opening a model scales linearly, closing it exhibits quadratic scaling. MathWorks plans to fix two performance issues in relation to our scalability findings.

Given our success in finding bugs and scalability issues in Simulink, a safety-critical software that is applied in practice since decades, we view GRANDSLAM as promising. We identify two main weaknesses, though. (1) our approach of typed equivalency does not guarantee models to be compilable. Following Chowdhury et al. [29], techniques such as fixing synthesized models depending on their compilation errors could be used to deal with this problem. (2) GRANDSLAM ignores semantics while synthesizing. This often leads to large but nonsensical models. However, MathWorks still confirmed six bugs and two issues that we uncovered using them.

C. Generalizing to other elements and languages

As we have seen, our synthesis approach of substituting equivalent *Subsystems* works very well in Simulink. More general though, any combination of model elements with a definable interface and an equivalence relation is suitable for our substitution approach. This means individual Blocks or groups of connected Blocks could be used to build equivalence classes. We chose Subsystems, as they naturally limit the number of interfaces we have to analyze, and are the natural abstraction unit in Simulink. This means, that our approach can be generalized to other (modeling) languages the easiest, if there is a similar abstraction unit as a Simulink Subsystem, e.g., subnets in Petri Nets [30], EClasses in Eclipse Modeling Framework [31], or methods/functions in textual languages. In fact, our approach may be more suitable in synthesizing valid instances in other languages, when one can construct clean interfaces without Simulink's global effects (cf. **RQ2.3**).

To illustrate the parallel in textual languages, consider the following example: depending on equivalence definition, the Java-methods `int min(int a, int b)` and `int max(int a, int b)` may be typed equivalent – Python's `min(a, b)` and `max(a, b)` may be untyped equivalent.

D. Scaling and Accelerating Our Approach

Our strategies RANDOM, ROLE, BUSHY, and DEEP use the algorithm of Listing 1 in their blueprint phase. It substitutes each Subsystem of the synthesized model once. Thus, in order to scale linearly, the functions `sample`, `isSuitable`, and the backtracking mechanism must produce constant overhead per Subsystem, as in BUSHY, cf. Figure 7. For the models of our experiments, the construction phase scales effectively linearly, but is much slower for RANDOM, ROLE, BUSHY, and DEEP. This is mainly because of Simulink's API for copying Subsystems or Blocks. In our experiment, the largest model's blueprint only took 30 seconds, but its Simulink construction 4 hours. As an additional scaling test, we also synthesized a blueprint-only model encompassing 2.4M Subsystems and 24M Blocks in 15 minutes using BUSHY – its Simulink construction would take about 5 days. While GIANT's blueprint phase is slower, its construction is much faster (cf. Figure 6) as fewer substitutions need to be performed.

We identified two factors to speed up the construction phase: (1) add each Subsystem's interface mapping to its properties in Section IV-C, as currently each substitution computes two interface mappings for the order relaxation: one for the Subsystem before substitution; one for the Subsystem after. (2) The construction (and all other phases) could be parallelized. Currently, our implementation is fully sequential, but the model synthesis steps are completely independent.

E. Threats to Validity

To ensure robust evaluation, we excluded the unstable models from the SLNET dataset (cf. Section V-A) as they caused bugs in Simulink. If we use the whole, 'untamed' dataset, the large strategies BUSHY, DEEP, and GIANT would mostly fail to produce any valid models. Excluding these models thus partially shifted our fuzzing from the synthesis phase to the earlier 'taming' phase (cf. Table IV), while simultaneously ensuring we can synthesize large models.

Our scalability results of Table V showed good fits ($R^2 > 0.6$) for four regression curves when applied to the 'tamed'

dataset. However, 'taming' removed model elements (e.g., Callbacks) that impact time/space complexity. We expect the original SLNET models to: (i) Run slower (due to Callback computations), (ii) Impede prediction based solely on element count (again, due to Callbacks), and (iii) Yield larger files (from restored elements) or smaller files (if library Blocks dominate). Thus, our regression may only reflect scaled-up behavior of the 'tamed' dataset – not the original models.

We treated some Subsystems with complicated or 'clunky' [32], [33] interfaces (Subsystems with Bus Ports, physical Ports, and Stateflow Charts) as common Blocks that are not substitutable. Including them would result in more equivalence classes, potentially as singletons.

VIII. Related Work

A. Fuzzing in CPS and other Domains

Approaches for creating fuzzing candidates can be roughly divided into two categories: generative, and transformative [34]. In generative approaches, rules, or LLMs are leveraged to generate candidates from scratch, while the latter ones mutate existing seeds into candidates. A number of fuzzing techniques – mostly promising approaches from textual languages – have already been employed in the CPS domain [3], [4], [15], [29], [35]. Chowdhury et al. [29] developed CyFuzz (generates models up to size of $\sim$35 Blocks), a generative approach, which was inspired by CSmith [36] to fuzz Simulink's code generator. Shrestha et al. [15] then employ the long short term memory [37] approaches DeepSmith, and Deepfuzz [38], [39] (40 Blocks; 8 connected), and later improve it with a trained version of the LLM GPT-2 [40] to generate their models [4] (30 Blocks; 12 connected). With the rise of LLMs in modern textual language fuzzers [41], [42], we anticipate their adoption in the CPS domain in the future.

Prior transformative approaches in the CPS domain rely on equivalence modulo inputs (EMI) [43], which mutates unexecuted code in compilable models and detects generated code changes. Note that in their context, *equivalence* refers to preserving semantics after mutation, whereas our *equivalence* ensures that two Subsystems can be substituted without invalidating the model. Using EMI, Chowdhury et al. [2], [3] developed SLforge, and later SLEMI ($\sim$3,000 Blocks). Both, Guo et al.'s COMBAT [35], and Li et al.'s RECORD [5], combine generation and transformation to first seed with SLforge, and then mutate with EMI ($\sim$3,000 Blocks).

The most comparable approaches to our work were not yet used in the CPS domain. Both Holler et al. and Han et al. [27], [44] mine building blocks from a program collection and then recombine them. Han et al.'s fuzzer, CodeAlchemist, recombines building blocks by putting fitting ones in sequence, the blocks can then grow in size to be recombined again (8 JavaScript statements). While CodeAlchemist does check block types, it can only put them in sequence. Instead, we put building blocks into 'suitable holes' of building blocks: in effect, we leave large local sub-ASTs intact. Already in 2011, Holler et al. proposed LangFuzz, which first mines AST-fragments to then replace a fragment in a program with another of the same type (no size given). In our work, we also replace a type of AST-fragments – Subsystems – but guarantee validity by only substituting equivalents.

Wang et al. [45] developed another related idea to construct more natural fuzzing candidates. Their fuzzer, SkyFire, learns the patterns from a code dataset, which are recreated in a probabilistic context-sensitive grammar (< 200 elements).

B. Model Synthesis

Models are also synthesized for scalability or stress tests. Semeráth et al. [6] synthesize valid and diverse Yakindu statecharts and EMF models by iteratively extending partial models until acceptance criteria are met. This is similar to our ground-up synthesis, though their approach is less scalable (300-1,000 model elements per hour). Nassar et al. [46] generated large EMF models, leveraging given metamodels for rule-based model transformations, which they iteratively apply. While their approach does scale super-linearly, it still can generate up to 500k model elements. He et al. [47]–[49] generate various model types in quadratic time, relying on constraint solvers and metamodels. In contrast, our approach does scale linearly and works without a known metamodel.

C. Clone Detection

There are various approaches to detect clones in models. Clone detectors may also cluster Subsystems into *clone classes* [50], [51]. These are only loosely related to our *equivalence classes*: while Subsystems of the same clone class are similar in structure, substituting two Subsystems of a class often invalidates the model. However, our equivalence classes guarantee substitutions work. Furthermore, most clone detectors ignore dynamic aspects of typing or dimensions, which is central to our concept of typed equivalence.

IX. Conclusion

Based on our concept of equivalence, we introduced GRANDSLAM with five synthesis strategies – each generating models with Subsystem trees of different shape or size. All models are statically valid, with many also being compilable and simulable. Our approach uncovered eleven bugs and two scalability issues in Simulink, a safety-critical and mature tool that is used in industry for decades. GRANDSLAM's linear scaling enables the generation of very large Simulink models, making it ideal for scalability tests.

In future work, we will extend GRANDSLAM to support more fine-grained substitutions. As discussed, our approach can be generalized to any group of Blocks. It could also be combined with another fine-grained mutator such as the modeling assistant SimGESTION from Adhikari et al. [51], which can mutate models one element at a time, or an LLM model completion tool like RAMC [52]. GRANDSLAM could first generate the model's rough outline (i.e., the Subsystem tree) and then delegate the finer details to another approach.

References

[1] H. Klee and R. Allen, *Simulation of Dynamic Systems with MATLAB and Simulink*, 3rd ed. CRC Press, Taylor & Francis Group, 2018. [Online]. Available: https://doi.org/10.1201/9781315154176

[2] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "Slemi: equivalence modulo input (emi) based mutation of cps models for finding compiler bugs in simulink," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 335–346. [Online]. Available: https://doi.org/10.1145/3377811.3380381

[3] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, "Automatically finding bugs in a commercial cyber-physical system development tool chain with slforge," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 981–992. [Online]. Available: https://doi.org/10.1145/3180155.3180231

[4] S. L. Shrestha and C. Csallner, "SLGPT: Using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 260–265. [Online]. Available: https://doi.org/10.1145/3463274.3463806

[5] X. Li, S. Guo, H. Cheng, and H. Jiang, "Simulink compiler testing via configuration diversification with reinforcement learning," *IEEE Transactions on Reliability*, vol. 73, no. 2, pp. 1060–1074, 2024. [Online]. Available: https://doi.org/10.1109/TR.2023.3317643

[6] O. Semeráth, A. A. Babikian, B. Chen, C. Li, K. Marussy, G. Szárnyas, and D. Varró, "Automated generation of consistent, diverse and structurally realistic graph models," *Software and Systems Modeling*, vol. 20, no. 5, pp. 1713–1734, 2021. [Online]. Available: https://doi.org/10.1007/s10270-021-00884-z

[7] Y. Huang, "Reproducibility and replicability of simulation models," in *2025 Annual Modeling and Simulation Conference (ANNSIM)*, 2025, pp. 1–10.

[8] A. Boll, N. Vieregg, and T. Kehrer, "Replicability of experimental tool evaluations in model-based software and systems engineering with matlab/simulink," *Innovations in Systems and Software Engineering*, pp. 1–16, 2022. [Online]. Available: https://doi.org/10.1007/s11334-022-00442-w

[9] S. L. Shrestha, A. Boll, T. Kehrer, and C. Csallner, "Scoutsl: An open-source simulink search engine," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 12 2023, pp. 70–74. [Online]. Available: https://doi.org/10.1109/MODELS-C59198.2023.00022

[10] Z. Jiang, X. Wu, Z. Dong, and M. Mu, "Optimal test case generation for simulink models using slicing," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2017, pp. 363–369. [Online]. Available: https://doi.org/10.1109/QRS-C.2017.67

[11] A. Hussain, H. A. Sher, A. F. Murtaza, and K. Al-Haddad, "Improved restricted control set model predictive control (ircs-mpc) based maximum power point tracking of photovoltaic module," *IEEE Access*, vol. 7, pp. 149 422–149 432, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2946747

[12] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Replicability study: Corpora for understanding simulink models & projects," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2023, New Orleans, LA, USA, October 26-27, 2023*. IEEE, 2023, pp. 1–12. [Online]. Available: https://doi.org/10.1109/ESEM56168.2023.10304867

[13] M. Jaskolka, V. Pantelic, A. Wassyng, and M. Lawford, "Supporting modularity in simulink models," 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2007.10120

[14] A. Boll, "Bridging the data desert: Mitigating challenges of model accessibility in simulink research," Dissertation, University of Bern, Bern, Switzerland, 2 2026. [Online]. Available: https://doi.org/10.48549/7066

[15] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Deepfuzzsl: Generating models with deep learning to find bugs in the simulink toolchain," in *2nd workshop on testing for deep learning and deep learning for testing (DeepTest)*, 2020.

[16] A. Boll, F. Brokhausen, T. Amorim, T. Kehrer, and A. Vogelsang, "Characteristics, potentials, and limitations of open-source simulink projects for empirical research," *Software and Systems Modeling*, vol. 20, no. 6, pp. 2111–2130, 4 2021. [Online]. Available: https://doi.org/10.1007/s10270-021-00883-0

[17] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "SLNET: A redistributable corpus of 3rd-party simulink models," in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 1–5. [Online]. Available: https://doi.org/10.1145/3524842.3528001

[18] Z. Yu, Y. Yang, Z. Su, R. Wang, Y. Tao, and Y. Jiang, "Knight: Optimizing code generation for simulink models with loop reshaping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 2, pp. 444–457, 2025. [Online]. Available: https://doi.org/10.1109/TCAD.2024.3438691

[19] A. Boll, T. Kehrer, and M. Goedicke, "Smoke: Simulink model obfuscator keeping structure," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS Companion '24. New York, NY, USA: Association for Computing Machinery, 10 2024, p. 41–45. [Online]. Available: https://doi.org/10.1145/3652620.3687788

[20] A. Boll, P. Rani, A. Schultheiß, and T. Kehrer, "Beyond code: Is there a difference between comments in visual and textual languages?" *Journal of Systems and Software*, vol. 215, p. 112087, 9 2024.

[21] S. L. Shrestha, A. Boll, S. A. Chowdhury, T. Kehrer, and C. Csallner, "Evosl: A large open-source corpus of changes in simulink models & projects," in *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 12 2023, pp. 273–284. [Online]. Available: https://doi.org/10.1109/MODELS58315.2023.00024

[22] J. Zhang, D. Ghobari, M. Sabetzadeh, and S. Nejati, "Simulink mutation testing using codebert," in *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2025, pp. 24–28. [Online]. Available: https://doi.org/10.1109/AST66626.2025.00009

[23] J. B. W. Webber, "A bi-symmetric log transformation for wide-range data," *Measurement Science and Technology*, vol. 24, no. 2, p. 027001, 2012. [Online]. Available: https://doi.org/10.1088/0957-0233/24/2/027001

[24] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 4th ed. Elsevier, 2022.

[25] N. R. Draper and H. Smith, *Applied Regression Analysis*, 3rd ed., ser. Wiley Series in Probability and Statistics. New York: Wiley-Interscience, 1998.

[26] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 2011, pp. 80–89. [Online]. Available: https://doi.org/10.1109/ICST.2011.53

[27] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[28] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2372785

[29] S. A. Chowdhury, T. T. Johnson, and C. Csallner, "Cyfuzz: A differential testing framework for cyber-physical systems development environments," in *Cyber Physical Systems. Design, Modeling, and Evaluation*, C. Berger, M. R. Mousavi, and R. Wisniewski, Eds. Cham: Springer International Publishing, 2017, pp. 46–60. [Online]. Available: https://doi.org/10.1007/978-3-319-51738-4_4

[30] M. Markiewicz and L. Gniewek, "Review of hierarchy in petri nets," *International Journal of Electronics and Telecommunication*, vol. 71, no. 3, pp. 1–9, 2025. [Online]. Available: https://doi.org/10.24425/ijet.2025.155452

[31] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *Eclipse Modeling Framework, Second Edition*. Addison-Wesley, 2009. [Online]. Available: https://sisis.rz.htw-berlin.de/inh2009/12368099.pdf

[32] T. Amorim, A. Boll, F. Bachman, T. Kehrer, A. Vogelsang, and H. Pohlheim, "Simulink bus usage in practice: an empirical study," *Journal of Object Technology*, vol. 22, no. 2, pp. 2:1–14, 7 2023, the 19th European Conference on Modelling Foundations and Applications (ECMFA 2023). [Online]. Available: https://doi.org/10.5381/jot.2023.22.2.a12

[33] M. Jaskolka, V. Pantelic, A. Wassyng, M. Lawford, and R. Paige, "Repository mining for changes in simulink models," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2021, pp. 46–57. [Online]. Available: https://doi.org/10.1109/MODELS50736.2021.00014

[34] H. Ma, "A survey of modern compiler fuzzing," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.06884

[35] S. Guo, H. Jiang, Z. Xu, X. Li, Z. Ren, Z. Zhou, and R. Chen, "Detecting simulink compiler bugs via controllable zombie blocks mutation," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1061–1072. [Online]. Available: https://doi.org/10.1145/3540250.3549159

[36] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[37] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[38] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 95–105. [Online]. Available: https://doi.org/10.1145/3213846.3213848

[39] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 1044–1051, Jul. 2019. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33011044

[40] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, Tech. Rep., 2019. [Online]. Available: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[41] L. Huang, P. Zhao, L. Ma, and H. Chen, "On the challenges of fuzzing techniques via large language models," in *2025 IEEE International Conference on Software Services Engineering (SSE)*, 2025, pp. 162–171. [Online]. Available: https://doi.org/10.1109/SSE67621.2025.00028

[42] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[43] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[44] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[45] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 579–594. [Online]. Available: https://doi.org/10.1109/SP.2017.23

[46] N. Nassar, J. Kosiol, T. Kehrer, and G. Taentzer, "Generating large emf models efficiently: A rule-based, configurable approach," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2020, pp. 224–244. [Online]. Available: https://doi.org/10.1007/978-3-030-45234-6_11

[47] X. He, W. Li, T. Zhang, and Y. Liu, "Towards parallel model generation for random performance testing of model-oriented operations," in *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2016, pp. 57–64. [Online]. Available: https://doi.org/10.1109/TASE.2016.26

[48] X. He, T. Zhang, C.-J. Hu, Z. Ma, and W. Shao, "An mde performance testing framework based on random model generation," *Journal of Systems and Software*, vol. 121, pp. 247–264, 2016. [Online]. Available: https://doi.org/10.1016/j.jss.2016.04.044

[49] X. He, T. Zhang, M. Pan, Z. Ma, and C.-J. Hu, "Template-based model generation," *Software & Systems Modeling*, vol. 18, no. 3, pp. 2051–2092, 2019. [Online]. Available: https://doi.org/10.1007/s10270-017-0634-5

[50] J. R. Cordy, "Submodel pattern extraction for simulink models," in *Proceedings of the 17th International Software Product Line Conference*, ser. SPLC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 7–10. [Online]. Available: https://doi.org/10.1145/2491627.2492153

[51] B. Adhikari, E. J. Rapos, and M. Stephan, "Simima: a virtual simulink intelligent modeling assistant: Simulink intelligent modeling assistance through machine learning and model clones," *Software and Systems Modeling*, vol. 23, no. 1, pp. 29–56, 2024. [Online]. Available: https://doi.org/10.1007/s10270-023-01093-6

[52] C. Tinnes, A. Welter, and S. Apel, "Software model evolution with large language models: Experiments on simulated, public, and industrial datasets," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 950–962. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00112