CANcept: Model-based CAN Traffic Generation and Manipulation

Lino Wertz
Karlsruhe Institute of Technology
Karlsruhe, Germany
wertzlino@gmail.com

Junes Sheikhi
Karlsruhe Institute of Technology
Karlsruhe, Germany
junessheikhi@web.de

Florian Fehrle
Karlsruhe Institute of Technology
Karlsruhe, Germany
florian@fehrle.net

Adrian Rupp
Karlsruhe Institute of Technology
Karlsruhe, Germany
adrian.rupp.2004@icloud.com

Tianhai Liu*
Karlsruhe Institute of Technology
Karlsruhe, Germany
tianhai.liu@kit.edu

Philipp Kern
Karlsruhe Institute of Technology
Karlsruhe, Germany
philipp.kern@kit.edu

Carsten Sinz
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
carsten.sinz@h-ka.de

Bernhard Beckert
Karlsruhe Institute of Technology
Karlsruhe, Germany
beckert@kit.edu

Abstract

Testing timing-related safety properties of Controller Area Network (CAN)-based software requires precise control over transmitted data and communication timing. Existing open-source tools typically encode such scenarios in low-level scripts, making them difficult to maintain and evolve. We present CANcept, an open-source model-based tool for specifying, generating, replaying, and manipulating CAN traffic. A traffic schedule model (TSM) defines message timing and content transformations, and a DBC-based communication model (DCM) defines encoding and decoding between message-level traffic and CAN frames. CANcept combines both models in one execution mechanism for generated and manipulated traffic, including replay of transformed traces. A preliminary evaluation indicates that CANcept realizes the specified scenarios, with low timing deviation and stable execution under elevated traffic rates.

CCS Concepts

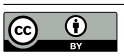
• **Software and its engineering → Software verification and validation**; *Model-driven software engineering*.

Keywords

CAN bus, DBC, model-based testing, test scenario modeling

ACM Reference Format:
Lino Wertz, Junes Sheikhi, Florian Fehrle, Adrian Rupp, Tianhai Liu, Philipp Kern, Carsten Sinz, and Bernhard Beckert. 2026. CANcept: Model-based CAN Traffic Generation and Manipulation. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3837062.3838871

*Corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
MODELS Companion 2026, Málaga, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2903-4/26/10
https://doi.org/10.1145/3837062.3838871

1 Introduction

Controller Area Network (CAN) [3] buses remain widely used in automotive systems to exchange information among Electronic Control Units (ECUs) due to their efficient broadcast mechanism for real-time in-vehicle communication [4]. CAN traffic is transmitted on the bus as raw data link layer frames, which engineers commonly interpret as logical messages containing signals such as engine speed or temperature. These messages are typically defined in Database CAN (DBC) files, which specify how message-level signal values are encoded into and decoded from frames [19]. DBC files, thus, provide a structural basis for interpreting CAN messages and mapping application-level signals to raw frame bytes.

Regression and robustness testing of CAN-based software requires test scenarios controlling both the content and temporal behavior of CAN traffic. Besides defining transmitted signal values, such scenarios must specify characteristics such as message periods, ordering, repetition, or absence. These characteristics are necessary to reproduce expected operating conditions and to exercise timing-dependent faults in interactions between ECUs [11, 12, 14]. For example, an NHTSA recall report [10] describes a software defect where a race condition during initial power-up between an integrated trailer module and a CAN standby-control signal could leave the module powered but unable to communicate with the vehicle. The defect could cause loss of trailer stop lamps, turn signals, and braking functionality, thereby increasing the risk of a crash.

Existing open-source CAN tools, such as can-utils [7] and SavvyCAN [5], support functions such as monitoring and interpreting CAN traffic. However, test scenarios are commonly implemented using low-level scripts in, e.g., Python or Bash. Maintaining and evolving such tests is challenging as the intended traffic schedule is often implicit and interwoven with low-level implementation details. Changes may inadvertently preserve obsolete transmission frequencies, modify unrelated timing behavior, or fail to exercise schedule-dependent faults.

We present CANcept, an open-source GUI tool for specifying CAN test scenarios and generating the corresponding traffic for timing-aware regression and robustness testing.

Its central abstraction is a *Traffic Schedule Model (TSM)*, which explicitly defines message timing, signal values, and traffic manipulations. CANcept combines this model with a *DBC-based Communication Model (DCM)* and SocketCAN-compatible interfaces [16]. The DCM defines how logical messages and signals are encoded into CAN frames, while the TSM defines when messages occur and how their timing and content evolve. Together, these models separate test-scenario specifications from low-level encoding and transmission details. In a typical workflow, the user loads a DCM file, configures a TSM through the graphical editor, concurrently monitors, records, generates, or manipulates CAN traffic on virtual or physical CAN buses to exercise the software under test. CANcept is model-based because the DCM provides a shared representation throughout the testing workflow, and the TSM captures CAN traffic execution semantics rather than an opaque command sequence.

This paper makes the following **contributions**:

- We present CANcept, an open-source graphical tool for model-based CAN testing.
- We introduce the TSM to specify CAN traffic timing, content, and manipulation using a concrete graphical syntax.
- We provide an execution mechanism that combines TSM-defined traffic behavior with DCM-based frame encoding and decoding.
- We compare CANcept with open-source CAN tools and evaluate its execution and scenario-realization accuracy.

2 Foundations and Models

CANcept operates on a DCM that defines the structure and interpretation of CAN messages, and a TSM that specifies test scenarios with conditional traffic and content transformations.

2.1 Communication Foundations and Model

We first formalize the DCM used by CANcept, based on a subset of DBC-style CAN message definitions [19]. The same model could also be derived from other CAN description formats, such as ARXML [1] and KCD [8]. This DCM provides the structural foundation for the TSM introduced in Section 2.2. Throughout this paper, *message-level* refers to decoded logical messages and signal values, and *frame-level* to their encoded CAN frames and payload bytes.

A DCM consists of a set of message definitions $C = \{m_1, \ldots, m_n\}$, where each message $m_i \in C$ contains a set of signal definitions. Each *signal definition* is defined as the tuple $s = (\text{name}_s, \text{pos}_s, \text{len}_s, \text{ord}_s, \text{sign}_s, \text{factor}_s, \text{offset}_s, \min_s, \max_s, \text{unit}_s, \text{src}_s, \text{dst}_s)$, where the mapping between the message-level value and the frame-level value are defined by factor_s and offset_s, namely

$$v_s^{\text{msg}} = v_s^{\text{frm}} \cdot \text{factor}_s + \text{offset}_s \quad \text{and} \quad v_s^{\text{frm}} = (v_s^{\text{msg}} - \text{offset}_s)/\text{factor}_s.$$

During signal encoding, v_s^{frm} is represented as a signed or unsigned integer according to sign_s, ordered according to ord_s (little-endian or big-endian), and written into the payload byte array starting at bit position pos_s. The field occupies exactly len_s bits. Signal decoding reverses this process with the same interpretation. The components $\min_s$ and $\max_s$ define the admissible range of values. The optional element unit_s specifies the physical unit in which the value is reported. The optional elements src_s and dst_s identify the ECUs that provide and consume the signal, respectively.

A *message definition* is defined as $m = (\text{id}_m, \text{name}_m, \text{len}_m, S_m)$, where the elements determine how a logical message is serialized into a CAN frame. CAN operates at the data-link layer: during message encoding, the message-level values of the signals in S_m are converted and packed into the payload byte array according to their signal definitions. The parameter len_m fixes the payload length in bytes, and payload bits not assigned to any signal remain unused or are initialized according to the encoding policy. The identifier id_m is placed in the CAN frame header and is used for bus arbitration. The communication endpoints for each signal are given by the contained signal definitions through src_s and dst_s.

CAN traffic denotes the ordered occurrence of frames on a CAN interface. Each occurrence consists of a CAN frame and its transmission time. CAN traffic can be observed on a physical CAN bus or a virtual CAN interface, e.g., using SocketCAN [16], and recorded as a CAN trace: $T^x = [(o_1^x, t_1), \ldots, (o_n^x, t_n)]$, $t_i \in \mathbb{R}_{\geq 0}$, where o_i^x denotes the i-th occurrence and t_i its timestamp. The superscript $x \in \{\text{msg}, \text{frm}\}$ distinguishes message-level and frame-level representations. A message-level trace contains message instances with decoded signal values. A frame-level trace contains frames with raw payload bytes. Given a DCM, both representations encode the same signal values and can be converted into one another.

Running Example We use an engine-status message as a running example throughout the paper. The following DBC fragment defines the `EngineStatus` message and its `rpm` and `temp` signals:

```
BO_ 256 EngineStatus: 8 EngineECU
 SG_ rpm  : 0|16@1+ (1,0) [0|65535] "rpm" DashboardECU
 SG_ temp : 16|8@1+ (1,0) [0|255] "degC" DashboardECU
```

The message has CAN identifier 256 = `0x100`, an 8-byte payload, and *EngineECU* as sender. It contains an unsigned 16-bit little-endian engine-speed signal starting at bit 0 and an unsigned 8-bit temperature signal starting at bit 16. Both signals use factor 1 and offset 0, have ranges $[0, 65535]$ and $[0, 255]$, respectively, and *DashboardECU* as receiver. In our DCM, this yields $m_{\text{eng}} = (\texttt{0x100}, 8, \{s_{\text{rpm}}, s_{\text{temp}}\})$, with signals s_{rpm} and s_{temp} represented directly. For the running example below, we assume initial values `rpm`= 3000 and `temp`= 95. A message-level trace where `rpm` increases by 500 RPM and `temp` by 2 °C every 100 ms is $T^{\text{msg}} = [$ (ES(3500, 97), 0 ms), (ES(4000, 99), 100 ms), (ES(4500, 101), 200 ms)], where ES(r, t) abbreviates a message instance with `rpm` = r and `temp` = t. The first generated message-level values are encoded to the frame-level payload `[AC 0D 61 00 00 00 00 00]`.

2.2 Traffic Schedule Model (TSM)

A TSM is a rule-based model for specifying CAN test scenarios in terms of message occurrences, their timing, ordering, and the signal values they carry. It defines message transmission patterns and how signal or payload values evolve across occurrences. These patterns are expressed as conditional rules, each consisting of a condition and an action. Conditions are predicates over expressions involving constants and current or previous values. Actions modify either message occurrences or their signal values. Occurrence actions can delay, drop, reorder, duplicate, or change transmission rates, while content actions can assign, bit-flip, or inject random noise within a specified range to signal and payload values. The TSM applies to both message-level and frame-level traffic because both

⟨*schedule*⟩	::=	⟨*generation*⟩ | ⟨*manipulation*⟩
⟨*generation*⟩	::=	`generation {` ⟨*dbcRef*⟩; ⟨*frequency*⟩; ⟨*assign*⟩*; `}`
⟨*manipulation*⟩	::=	`manipulation {` ⟨*dbcRef*⟩$^?$; ⟨*traceRef*⟩; ⟨*rule*⟩*; `}`
⟨*dbcRef*⟩	::=	`dbcFile` ⟨*file*⟩
⟨*traceRef*⟩	::=	`traceFile` ⟨*file*⟩
⟨*frequency*⟩	::=	`every` ⟨*duration*⟩
⟨*rule*⟩	::=	`on` ⟨*msgRef*⟩ `when` ⟨*predicate*⟩ `do` ⟨*action*⟩
⟨*action*⟩	::=	⟨*trafficAction*⟩ | ⟨*contentAction*⟩
⟨*trafficAction*⟩	::=	`delay` ⟨*duration*⟩ | `drop` | `insert` ⟨*valueExpr*⟩
⟨*contentAction*⟩	::=	`set` ⟨*signalRef*⟩ ⟵ ⟨*valueExpr*⟩ | `bitflip` ⟨*valueExpr*⟩ | `perturb` ⟨*signalRef*⟩ ⟨*valueExpr*⟩ ⟨*valueExpr*⟩
⟨*assign*⟩	::=	⟨*signalRef*⟩ ⟵ ⟨*valueExpr*⟩
⟨*valueExpr*⟩	::=	⟨*number*⟩ | ⟨*signalRef*⟩ | ⟨*func*⟩ (⟨*valueExpr*⟩) | ⟨*valueExpr*⟩ ⟨*arithOp*⟩ ⟨*valueExpr*⟩
⟨*duration*⟩	::=	⟨*number*⟩ ⟨*timeUnit*⟩
⟨*timeUnit*⟩	::=	`ns` | `µs` | `ms` | `s`
⟨*predicate*⟩	::=	*atomic predicate over signal values or message identifiers*
⟨*func*⟩	::=	*math functions, e.g., sin, cos, log, abs, and square root*
⟨*signalRef*⟩	::=	*message-name*.*signal-name*
⟨*msgRef*⟩	::=	*message-name*
⟨*file*⟩	::=	*path to a model file*
⟨*number*⟩	::=	*integer* | *decimal*
⟨*arithOp*⟩	::=	+ | - | *

Figure 1: Abstract grammar of the TSM.

share the same temporal structure and can be converted through a DCM. Figure 1 presents the abstract syntax of the TSM. In the running example in Section 2.1, the trace can be generated from this generation schedule:

```
generation {
  dbcFile "vehicle.dbc";
  every 100 ms;
  EngineStatus.rpm <- EngineStatus.rpm + 500;
  EngineStatus.temp <- EngineStatus.temp + 2;
}
```

The resulting message-level trace can then be manipulated by a separate schedule. The following manipulation delays `EngineStatus` occurrences whose engine speed is at least 4,500 RPM by 20 ms and clamps temperature values above 100 °C to 100 °C:

```
manipulation {
  dbcFile "vehicle.dbc";
  traceFile "vehicle.trace";
  on EngineStatus when EngineStatus.rpm >= 4500 do delay 20 ms;
  on EngineStatus when EngineStatus.temp > 100  do set EngineStatus.temp <- 100;
}
```

Assuming the initial values above, the rules are evaluated independently for each matching occurrence. The temperature rule changes the third and subsequent occurrences to 100, and the delay rule shifts them by 20 ms. Consequently, a fragment of the manipulated trace is $T^{\text{msg}} = [(\text{ES}(3500, 97), 0\,\text{ms}), (\text{ES}(4000, 99), 100\,\text{ms}), (\text{ES}(4500, 100), 220\,\text{ms})]$. The first two occurrences is unchanged; the third is delayed from 200 ms to 220 ms and its temperature is clamped from 101 to 100.

The two schedules together describe a controlled engine scenario in which signal values evolve over time and messages satisfying specified state conditions are deliberately delayed or modified. Such scenarios can expose faults that arise only from particular combinations of signal values and communication timing. E.g., this scenario can test whether a receiver correctly handles late messages whose temperature value has been clamped once the engine speed reaches 4,500 RPM.

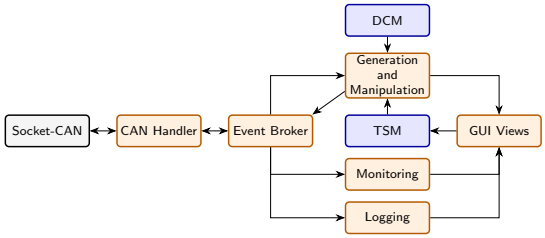


Figure 2: Overview of the CANcept architecture.

3 Tool Overview

CANcept operationalizes the DCM and TSM as an interactive CAN testing environment. Figure 2 shows the tool architecture. The GUI provides views for DCM inspection, TSM configuration, live DBC traffic monitoring, trace logging, manual transmission, and replay control. These views exchange commands and updates through an event broker that separates the user interface from the core CAN processing logic. The core layer combines CAN traffic generation and manipulation based on DCM and TSM, as well as monitoring and logging. The CAN handler connects the tool to SocketCAN.

3.1 Traffic Generation and Manipulation

CANcept processes the DCM and TSM through a common model-driven execution mechanism. Users load both models through the GUI, inspect parsed message, signal, and scheduling information, and configure test scenarios at the model level. The tool parses each model into an internal representation and binds them by resolving TSM message and signal names against the DCM. This binding links the two models and could be extended with static analyses that detect inconsistencies between them.

For traffic generation, the execution mechanism evaluates the TSM to create message-level occurrences with concrete timestamps and signal values. Signal values not assigned by the schedule are initialized using defaults derived from the corresponding DCM definitions. Each occurrence is then encoded based on the DCM and transmitted or recorded.

For traffic manipulation, recorded frame-level traffic is first decoded through the DCM into message-level occurrences. The same rule engine then evaluates the TSM over these occurrences. Traffic actions modify the trace structure, e.g., by delaying, dropping, or reordering messages, and content actions modify signal or payload values. The resulting occurrences are encoded through the same DCM before transmission, replay, or logging.

The execution mechanism operates on generic representations of messages, signals, expressions, and actions. Consequently, DCM and TSM instances can be updated by reloading and rebinding without changing the internal model handlers. Introducing new model constructs or action semantics, however, requires extending the corresponding parser or interpreter.

3.2 Monitoring and Logging

CANcept also provides monitoring and logging, which are standard capabilities of open-source CAN tools such as python-can [18] and SavvyCAN [5]. Its GUI displays incoming CAN traffic as time-series views at message and frame levels. When a matching DCM is loaded, frames are decoded into messages and signal values;

otherwise, their identifiers and raw payloads are shown. The current implementation monitors one CAN interface at a time; concurrent multi-interface support is future work.

Incoming traffic can be recorded as reusable traces in CSV. The recorded timestamps, identifiers, payloads, and decoded values can then be inspected offline or loaded back into CANcept. Frame-level traces are decoded using the DCM and can be replayed directly or processed by a TSM. During unmodified replay, CANcept preserves temporal offsets between recorded occurrences, i.e., no schedules are applied. For manipulation, the TSM modifies the decoded occurrences before they are re-encoded through the DCM. Thus, recorded traffic can be monitored, logged, manipulated, and replayed within the same execution workflow.

3.3 Implementation

CANcept is implemented in C++ using Qt 6 [17] and SocketCAN [16]. SocketCAN allows the tool to operate on both physical and virtual CAN interfaces. Virtual interfaces support reproducible demonstrations and benchmarks without requiring vehicle hardware. The current implementation targets Linux-based environments. Core functional modules communicate through the event broker rather than through direct dependencies. This design keeps the GUI responsive, isolates time-critical transmission logic, and allows views, analyses, and schedule actions to be added in a modular way. The abstract grammar in Figure 1 defines the principal model elements independently of a concrete serialization format. TSM instances are created in the graphical editor and imported from or exported to JSON, which also stores workflow metadata. The current implementation supports only atomic predicates for signal equality, signal bounds, and message-identifier equality. Textual syntax, a parser, and richer predicates are future work.

4 Evaluation

We evaluate two aspects of CANcept: execution-level behavior for standard CAN tasks and realization of declarative TSM-based scenarios. As open-source baselines, we use python-can [18] and SavvyCAN [5] where comparable functionality is available. The evaluation comprises two benchmark groups. The *execution-level benchmarks* cover four standard CAN tasks: (i) transmission (periodic sending without injected faults), (ii) bit corruption (deterministic payload corruption during transmission), (iii) raw replay (replay of recorded frame-level traffic), and (iv) DBC-based replay (replay of message-level traffic decoded and re-encoded through the DCM). The *scenario-realization benchmarks* cover four test scenario realization tasks: (i) frame loss (probabilistic dropping), (ii) timing violation (threshold-triggered delay), (iii) frame duplication (one-to-one duplication), and (iv) ordering constraint (enforced relative order between messages). For both benchmarks, we report: (i) frame loss (percentage of scheduled frames not observed), (ii) average jitter (mean absolute timing deviation), (iii) peak jitter (maximum timing deviation), and (iv) throughput (observed frames per second).

Although the experiments run on Linux with a real-time kernel, CANcept does not use a dedicated real-time API. The reported jitter therefore indicates application-level timing stability rather than a bound on hard real-time precision. In both evaluation results, ‘n/a‘ denotes metrics that are not applicable to the tested tool for the corresponding scenario.

Table 1: Performance results for standard functions.

Scenario	Tool	Frame loss (%)	Jitter (μs)	Peak jitter (μs)	Fault fidelity (%)
Transmission	python-can	0.01	219	3018	100.00
Transmission	SavvyCAN	10.59	431	1665	n/a
Transmission	CANcept	0.00	2	50	100.00
Bit corruption	python-can	0.01	424	3019	100.00
Bit corruption	SavvyCAN	10.52	421	1870	n/a
Bit corruption	CANcept	0.00	2	34	100.00
Replay raw	python-can	0.00	29	3722	n/a
Replay raw	CANcept	0.00	5	841	n/a
Replay DBC	python-can	0.00	15	2681	n/a
Replay DBC	CANcept	0.00	6	1334	n/a

4.1 Execution-Level Comparison

The evaluation compares standard CAN execution tasks against available baseline tools. The transmission and bit-corruption benchmarks transmit 60,000 frames at 1.000 FPS, while the raw and DBC-based replay benchmarks process recorded traffic for 60 seconds at 10.000 FPS. We additionally report *fault fidelity*, i.e., the percentage of observed fault-injected frames whose payload matches the configured deterministic bit corruption.

Table 1 summarizes the results. Fault fidelty is not applicable to SavvyCAN because it lacks deterministic bit corruption. The results indicate that CANcept exhibits substantially more stable timing behavior than the evaluated open-source baselines. A possible explanation is that CANcept uses a dedicated execution pipeline for both traffic generation and replay, whereas other tools rely on more general-purpose scripting mechanisms.

In the transmission and bit-corruption scenarios, CANcept has 2 μs jitter with zero frame loss and 100% fault fidelity. For replay, CANcept again achieves lower average and peak jitter than python-can in both raw and DBC-based replay while preserving frame contents and zero frame loss. SavvyCAN was not included in the replay comparison because it became unresponsive on the virtual CAN setup. This indicates a more stable common execution basis for timing-aware generation, fault injection, and replay.

4.2 Scenario Realization Evaluation

This evaluation investigates how accurately CANcept realizes test scenarios. Four benchmarks cover frame loss, timing violation, frame duplication, and ordering constraint. We additionally report *delay error* (absolute deviation from the configured delay), *frame amplification* (ratio of generated frames to original frames), and *ordering violations* (number of occurrences violating the configured order), which directly characterize the intended manipulations.

Table 2 shows that neither tool violates the configured order or loses frames in the time-independent scenarios. In the time-dependent scenarios, CANcept achieves consistently lower jitter than python-can while meeting the configured delay with slightly smaller error (15 µs versus 19 µs). We configured the scenarios in CANcept via the GUI in about *one minute*; equivalent python-can scripts required about *one engineer-hour* and roughly 150 lines of code. This suggests that CANcept reduces manual effort while maintaining accurate scenario realization.

Table 2: Accuracy of scenario realizations.

Scenario	Tool	Loss (%)	Avg. jit. (μs)	Peak jit. (μs)	Throughput (fps)	Delay err. (μs)	Amp. (×)	Order vio.
Frame Loss	python-can	10.18	14	1020	898.17	n/a	n/a	n/a
Frame Loss	CANcept	9.91	1	39	900.95	n/a	n/a	n/a
Timing Violation	python-can	0.01	2465	6559	999.92	19	n/a	n/a
Timing Violation	CANcept	0.01	2465	5013	999.92	15	n/a	n/a
Frame Duplication	python-can	0.00	5	505	18518.51	n/a	2.00	n/a
Frame Duplication	CANcept	0.00	1	136	18518.52	n/a	2.00	n/a
Ordering Constraint	python-can	0.00	6	1315	10000.01	n/a	n/a	0
Ordering Constraint	CANcept	0.00	1	198	9999.84	n/a	n/a	0

Table 3: Comparison with existing CAN tools.

Tool	GUI	DBC	Send/Replay	Timing support	Schedule model	Open source
can-utils [7]	no	no	yes	trace-based	no	yes
python-can [18]/cantools [9]	no	yes	script	periodic/script	no	yes
SavvyCAN [5]	yes	yes	yes	trace-based	no	yes
CANoe [20]	yes	yes	yes	yes	proprietary	no
CANcept	yes	yes	yes	yes	yes	yes

5 Related Work

Table 3 compares tools for testing CAN-based software systems. Can-utils [7] provides lightweight frame-level sending, recording, generation, and replay. Python-can [18] and cantools [9] provide scriptable CAN access and DBC-based encoding and decoding. SavvyCAN [5] supports capture, visualization, replay, reverse engineering, and fuzzing. Kayak [15] offers monitoring with XML-based bus definitions. BUSMASTER [2] supports simulation and testing, and Scapy automotive extensions [13] enable programmable protocol manipulation. These tools are useful building blocks, but timing and message mutations must generally be encoded in scripts or traces rather than in a reusable scenario model.

Research on CAN fault injection and timing-based intrusion detection highlights the importance of temporal behavior. Rim'en and Christmansson [12] consider omitted, delayed, and modified CAN frames, while timing-based approaches [11, 14] analyze message timing to detect anomalous traffic. These works motivate explicit modeling of temporal behavior. However, they target specific fault-injection or anomaly-detection techniques rather than a reusable model for specifying and generating CAN test scenarios.

Commercial environments such as CANoe [20] provide comprehensive mechanisms for simulation and testing, including programmable timing behavior through simulation nodes and test modules, but lack a comparable open, declarative specification language for jointly describing traffic generation, signal evolution, temporal dependencies, and manipulations. CANcept provides such a language and its underlying model as part of an open-source toolchain.

Unlike tools that mainly use DBC files for encoding and decoding, CANcept derives a DCM from a DBC file and uses it as a shared semantic artifact throughout monitoring, replay, logging, generation, and manipulation. The TSM complements the DCM by capturing temporal and dynamic aspects not represented in structural communication descriptions, such as timing relationships, signal evolution, and dynamic triggers.

6 Conclusion

We presented CANcept, an open-source tool for model-based CAN traffic generation, replay, monitoring, and manipulation. Its core idea is to combine a DCM with a TSM in one execution mechanism for generated and manipulated traffic. This lets users inspect, configure, and execute CAN test scenarios at the model level rather than through ad hoc scripts and raw-frame processing. Recorded traces can be decoded via the DCM, transformed via the TSM, and re-encoded for replay or export, while generated traffic follows the same path. Our evaluation indicates that CANcept realizes the specified loss, delay, duplication, and ordering behaviors with low timing deviation and stable execution under elevated traffic rates.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) – SFB 1608 – 501798263.

Data Availability Statement

All prototype and evaluation artifacts are publicly available at [6], the source code is publicly available on GitHub https://github.com/CANcept/CANcept.

References

[1] AUTOSAR. 2017. *AUTOSAR XML Serialization Rules.* Specification AUTOSAR_TPS_ARXMLSerializationRules. AUTOSAR. Release 4.3.1.
[2] BUSMASTER Team. 2026. BUSMASTER: Open Source Software tool to simulate, analyze and test data bus systems. Accessed: 2026-07-15. https://rbei-etas.github.io/busmaster/
[3] Robert Bosch GmbH. 1991. CAN Specification Version 2.0. Accessed: 2026-07-15. http://esd.cs.ucr.edu/webres/can20.pdf
[4] International Organization for Standardization. 2024. ISO 11898-1:2024: Road vehicles – Controller area network (CAN) – Part 1: Data link layer. Accessed: 2026-07-15. https://www.iso.org/standard/86384.html
[5] Collin Kidder. 2025. SavvyCAN – A CAN Bus Reverse Engineering and Capture Tool. Accessed: 2026-07-15. https://github.com/collin80/SavvyCAN
[6] Lino Wertz, Florian Fehrle, Junes Sheikhi, Adrian Rupp and Nele Spatzier. 2026. CANcept. doi:10.5281/zenodo.21385303
[7] Linux-can community. 2026. can-utils. Accessed: 2026-07-15. https://github.com/linux-can/can-utils
[8] Jan-Niklas Meier and Jens Krueger. 2020. KCD: Kayak CAN Definition Format. Accessed: 2026-07-15. https://github.com/julietkilo/kcd
[9] Erik Moqvist et al. 2026. cantools – CAN Bus Tools for Python. Accessed: 2026-07-15. https://github.com/cantools/cantools
[10] National Highway Traffic Safety Administration. 2026. Part 573 Safety Recall Report 26V104. Accessed: 2026-07-15. https://static.nhtsa.gov/odi/rcl/2026/RCLRPT-26V104-3946.pdf
[11] Habeeb Olufowobi, Clinton Young, Joseph Zambreno, and Gedare Bloom. 2020. SAIDuCANT: Specification-Based Automotive Intrusion Detection Using Controller Area Network Timing. *IEEE Transactions on Vehicular Technology* 69, 2 (2020), 1484–1494.
[12] Marcus Rimén and Jörgen Christmansson. 1999. Testing CAN-based Safety-Critical Systems using Fault Injection. In *Proceedings of the International CAN Conference.*
[13] Scapy Team. 2018. Scapy Automotive-specific Documentation. Accessed: 2026-07-15. https://scapy.readthedocs.io/en/stable/layers/automotive.html
[14] Hyun Min Song, Ha Rang Kim, and Huy Kang Kim. 2016. Intrusion Detection System Based on the Analysis of Time Intervals of CAN Messages for In-Vehicle Network. In *30th International Conference on Information Networking.* 63–68.
[15] Kayak Team. [n. d.]. Kayak: CAN bus diagnosis and monitoring tool. Accessed: 2026-07-15. https://github.com/dschanoeh/Kayak
[16] The Linux Kernel Organization. 2008. SocketCAN - Controller Area Network. Accessed: 2026-07-15. https://docs.kernel.org/networking/can.html
[17] The Qt Company. 2020. Qt 6 Documentation. Accessed: 2026-07-15. https://doc.qt.io/qt-6/
[18] Brian Thorne et al. 2025. python-can – CAN Bus Support for Python. Accessed: 2026-07-15. https://github.com/hardbyte/python-can
[19] Vector Informatik GmbH. 2026. CANdb++ Manual. Accessed: 2026-07-15. https://cdn.vector.com/cms/content/products/candb/Docs/CANdb_Manual_EN.pdf
[20] Vector Informatik GmbH. 2026. CANoe: Development, test and analysis of individual ECUs and entire ECU networks. Accessed: 2026-07-15. https://www.vector.com/int/en/products/products-a-z/software/canoe/