Enabling security checking of automotive ECUs with formal CSP models

Heneghan, J, Shaikh, SA, Bryans, J, Cheah, M & Wooderson, P

Author post-print (accepted) deposited by Coventry University's Repository

Original citation & hyperlink:

Heneghan, J, Shaikh, SA, Bryans, J, Cheah, M & Wooderson, P 2019, Enabling security checking of automotive ECUs with formal CSP models. in Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop, DSN-W 2019., 8805994, Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop, DSN-W 2019, Institute of Electrical and Electronics Engineers Inc., pp. 90-97, 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop, DSN-W 2019, Portland, United States, 24/06/19.
https://dx.doi.org/10.1109/DSN-W.2019.00025

DOI 10.1109/DSN-W.2019.00025

Publisher: IEEE

© 2019 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Copyright © and Moral Rights are retained by the author(s) and/ or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This item cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder(s). The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

This document is the author's post-print version, incorporating any revisions agreed during the peer-review process. Some differences between the published version and this version may remain and you are advised to consult the published version if you wish to cite from it.

Enabling Security Checking of Automotive ECUs with Formal CSP Models

John Heneghan*, Siraj Ahmed Shaikh†, Jeremy Bryans†, Madeline Cheah‡ and Paul Wooderson‡
*†Systems Security Group, Institute of Future Transport and Cities (FTC), Coventry University, United Kingdom
‡Horiba MIRA Ltd., Nuneaton, United Kingdom
*Email: henegha3@coventry.ac.uk

***Abstract*—This paper presents an approach, using the process-algebra CSP, that aims to support systematic security testing of ECU components. An example use case regarding Over-The-Air software updates demonstrates the potential of our approach. Initial results confirm application code implemented in a typical automotive development environment can be translated into machine-readable format for the FDR refinement checker to formally verify security functions and identify any existing security flaws. Although still early stage work, the potential contribution towards automatically model-checking ECU components and, by composing several CSP models, larger systems is encouraging.**

***Index Terms*—Software engineering; Automotive; embedded software; concurrent computing; cyber security; system verification; formal verification; model checking**

I. Introduction

Intelligent vehicles promise dramatic changes for future ground transportation, potentially heralding more streamlined journeys, reduced pollution and novel transport services. Such automotive innovation relies on evermore sophisticated software embedded within the now ubiquitous electronic control units (ECUs) in modern automobiles.

However, intelligent vehicles offer would-be cyber attackers a tempting target, which raises security concerns with potential implications for safety, financial loss and privacy. These threaten realisation of potential benefits, and are prompting regulators and OEMs to re-focus efforts to verify automotive security. Yet, testing real-time, distributed vehicle networks is a significant challenge due to heterogeneous and opaque implementations, inaccessible specifications and difficulties associated with long development life cycles. The verification challenge is further complicated by the specialist knowledge needed to both model security and analyse security flaws.

The contribution of this paper is an approach to automatically translate an already implemented ECU application (designed in a typical automotive development environment) into a formal language. Here we use the process algebra Communicating Sequential Processes (CSP) [1] as the accompanying techniques and toolset (see Section IV), are mature enough for formal verification, indicating such an approach is feasible. Our aim is to enable systematic security testing of ECU components, allowing automotive software engineers to continue using familiar modelling and simulation tools, whilst also facilitating automatic verification of security properties.

The rest of this paper starts by presenting the background cybersecurity and software verification challenges facing the automotive industry (Section II). We then introduce our approach and prospective workflow for formally verifying security properties, at component-level, with real ECU applications (Section III). Following this, we provide an overview of techniques and tools underpinning our framework (Section IV). A simple example, based on Over-the-Air (OTA) software update, helps illustrate the overall workflow (Section V). Finally, we consider some limitations of our methods and outline future work (Sections VII and VIII).

II. Background

Automotive software engineering (ASE) is now an integral vehicle engineering activity as motor manufacturers deploy software-enabled features that address more stringent vehicle emission and safety standards, but also differentiate their products in highly competitive markets. Increasingly, embedded software allows vehicles to interact with Intelligent Transport Systems, whilst Advanced Driver Assistance Systems (ADAS) are steadily evolving through automation levels towards the ultimate goal of fully autonomous vehicles. However, this increasingly sophisticated transportation landscape implies more complex vehicle systems that begin to pose problems for all involved in the design, testing and deployment of vehicles, at the same time attracting potential cyber attackers.

A. Automotive Cybersecurity Challenges

Security is not necessarily built into automotive systems, due to cost constraints and a traditional focus on functional safety, which deals with random internal hazards, not deliberate malicious threats. Meanwhile, security testing methodologies in the automotive domain are immature and, seemingly, applied inconsistently [2] [3].

Enterprise IT security practices that could be ported to automotive (embedded) systems are not necessarily appropriate, since vehicle architectures and communication protocols are sufficiently different. Current automotive systems tend to comprise multiple ECUs with computing power that, although increasing, is constrained. More sophisticated ECUs (as used in infotainment units) may use microprocessors with up to 32-bit architectures, but others make do with 16- or even 8-bit processors. Since security processing is typically computationally

heavy, fundamental architectural changes would be required to incorporate security. Likewise, communication protocols, such as CAN bus, traditionally lacked security mechanisms, further limiting the ability to incorporate security functions.

Security researchers have already demonstrated practical cyber-attacks on various makes and models of cars, exposing potential safety implications [4]–[7]. These triggered further research examining privacy breaches and potential financial losses where consumer services, such as payment systems, inter-operate with vehicles. Whilst the variety and depth of experimentation have been both eye-opening and informative, formal engineering processes have garnered less attention (see Section II-C).

Maintaining vehicle security is also challenging, in that patches or other fixes for vulnerabilities must be disseminated widely, and in a timely manner. Yet, update mechanisms in vehicles vary greatly, ranging from manual procedures (i.e. bringing the car into a dealership) to self-updating systems, either via peripheral devices like USB sticks, or over-the-air. With vehicles now typically averaging 15 years on the road before decommissioning [8], multiple vehicle configurations must be updated, creating a significant management overhead. Maintenance is further exacerbated by the delay necessary to re-assure that security fixes do not inadvertently cause a safety problem. Automating the verification process would effectively reduce this lag.

B. *ECU Security Validation and Verification*

Automotive network architectures are increasingly complex, supporting distributed concurrent processes that may be implemented as multi-function components or multi-component functions. In-vehicle networks comprise ECU components most likely sourced from multiple suppliers, often without detailed documentation. These factors hamper the ability to integrate and test automotive systems [9].

Generally, concurrent systems are seen as notoriously difficult to verify. For instance, race conditions are difficult to find, whilst any bugs discovered can be tricky to localise, and attempts to better observe a system or component under test may induce a "probe effect" [10]. Concurrency issues may not be reproducible due to timing of events in the wider environment (e.g. other network traffic, competing applications, operating system scheduling, etc.) [11].

In recent years, the automotive industry has made progress towards improving its assurance of automotive safety properties, with compliance to standards such as ISO 26262 now commonplace [12]. Whilst analogous (i.e. safety assurance typically asserts that certain behaviours should or should not happen), methods for assuring security properties have lagged.

Security weaknesses tend to result from subtle, unexpected interactions, as well as lateral or additional functionality that should not exist [6]. These would, therefore, require additional considerations, such as formal verification to expose latent flaws. For example, the Needham-Schroeder authentication protocol, first proposed in 1978 [13], was widely used to secure network communications despite an inherent flaw. The security weakness was only exposed 18 years later through formal analysis using CSP to highlight a successful attack mechanism [14].

C. *Formal Methods in the Automotive Domain*

Despite showing much early promise for designing and verifying software, formal methods seem to have gained little traction within the automotive domain [2]. Broadly, this shortfall may be potentially attributable to accessibility issues and technical challenges.

1) Accessibility: Transforming legacy code-based system development life cycles to a model-based design approach imposes costs for developing skills, designing processes and deploying new tooling. Perceived difficulties arise from business domain experts and software engineers having to master the necessary formal verification techniques and cyber security knowledge, prior to being able to specify expected ECU behaviour and implement the necessary code. Likewise, any new processes must co-exist with existing legacy vehicle programmes, imposing further overheads. In a notoriously cost-driven industry [15], unsurprisingly, other priorities override new ASE approaches.

2) Technical challenges: Implementing the tool support necessary for formal methods suffers from the two-pronged problem of scalability and complexity in real world systems. With a modern vehicle now comprising an average of about 30,000 parts, a significant percentage of which are electronic, the number of interacting components leads to a combinatorial explosion in the number of test cases to consider. Many tools proposed for formal verification are not production-ready in terms of processing power [16] or having suitable support for industrial settings. Moreover, tools are often standalone, and do not readily integrate or inter-operate with existing ASE tool suites. The above is exacerbated by the black box nature of the systems in the vehicle, leading to many unknowns when creating models.

III. Approach

Our model-based approach ultimately aims to automate, at least partially, the security verification process. A key enabling capability for this is programmatically transforming ECU application code into a formal, machine-readable representation for our target model checker.

The concept of operations, outlined in Figure 1, envisages an ECU application created in a typical Integrated Development Environment (IDE). On export from the IDE, possibly with an associated network model, a model extractor application translates the application source code into an ECU component (implementation) model, defined as a CSP process. As such, it can be combined with other CSP models to compose an overall system model. Additional models may be specification models or other ECU implementation models. For instance, specification models may represent intended functional behaviour, define security properties or describe potential threats. Indeed, attacker models (describing attacks from threats) can be modelled as CSP processes [17]. Next, using a refinement

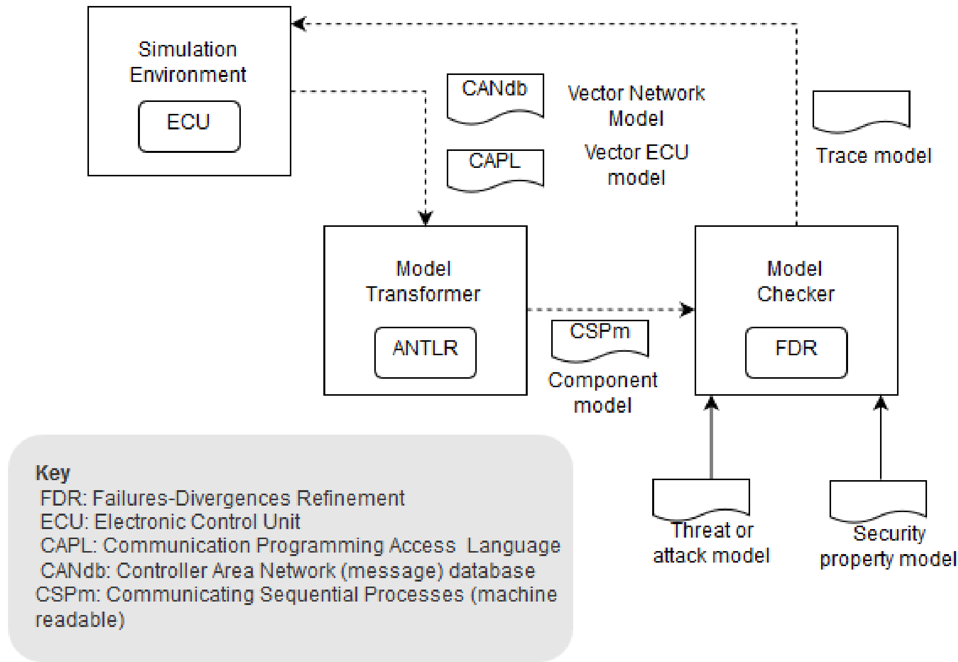

Fig. 1. A workflow and toolchain to enable automated component-level security analysis of automotive ECU components using CSP-based refinement checking with FDR. An innovative model transformation component bridges the automation gap between a CANoe Integated Development Environment (IDE) and the automation-ready FDR model (refinement) checker for CSP models.

checker, the composite system model can be verified to determine if any insecure traces (message sequences) can occur. These counterexamples represent failure traces corresponding to failed functionality, unsatisfied security properties or conditions for a successful attack (security vulnerability). Finally, these can then be fed back to software designers to review and rectify faults.

IV. Tools and Techniques

Implementing our concept requires a selection of tools and techniques that support formal systems modeling, automotive ECU software development, code transformation, and refinement checking for CSP. Additionally, we apply earlier security research techniques developed to model intruder capabilities and to define attacks.

A. Formal System Modelling

As our approach is firmly based on CSP, we provide below a brief overview of the CSP language, its notations and some relevant formal semantics.

1) CSP Overview: CSP emerged in the 1980s in response to the need for a language to describe concurrent systems. Essentially, CSP treats a system as a set of independent processes communicating over channels and, where necessary, synchronising on certain events. Individual process components are sequential programs that interact with other networked components by participating in events. However, since services offered by (or properties of) a network are represented purely in terms of interactions between network components, we avoid having to know too much about their inner workings.

A central tenet of CSP is that systems can be decomposed into subsystems, each interacting with others and the wider environment [18]. This compositional aspect permits successively refining more detailed CSP models as the development lifecycle progresses. Furthermore, external users (both benign and malevolent) who similarly communicate through messages can also be modelled as CSP processes, allowing system models to consider user behaviours.

Importantly, CSP has a sound mathematical basis, thus enabling formal reasoning about system descriptions using algebraic laws, a process easily automated and vital for handling the size of real systems. In fact, CSP now benefits from mature, scalable tools that can automate a wide range of checks, including refinement, deadlock, liveness and termination.

Over three decades, CSP has been applied to a wide range of safety-critical systems, including medical devices, rail signalling systems, and flight control systems [19]. Additionally, security researchers have devised proven methods for verifying various security properties, such as availability (liveness), authentication, confidentiality [20], and anonymity [21].

Overall, CSP is well-suited for modelling and analysing functional and security properties of vehicle networks and other real-time embedded distributed systems.

2) CSP Notation: This section summarises the notation and semantic models for the subset of CSP relevant to our work. A more complete introduction may be found in [22].

Process Definition. Given a set of events Σ, a CSP process P is defined by the following syntax:

$$P ::= Stop \mid e \rightarrow P \mid P_1 \mathbin{\Box} P_2 \mid P_1;\ P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid P_1 \mathbin{|||} P_2$$

where $e \in \Sigma$, and $A \subseteq$ events. In the above definition, we have:

- The process *Stop* is the most basic; it does not engage in any event and represents deadlock.
- The prefix operator $e \to P$ specifies a process that is only willing to engage in the event e, then behaves as P.
- External choice $P_1 \Box P_2$ behaves either as P_1 or as P_2.
- The sequential composition $P_1;\ P_2$ initially behaves as P_1 until P_1 terminates, then continues as P_2.
- The generalised parallel operator $P_1 \underset{A}{\parallel} P_2$ requires P_1 and P_2 to synchronise on events in $A \cup \{\checkmark\}$, ($\checkmark$ is a special event that represents successful termination). All other events execute independently.
- Finally, the interleaving operator $P_1 ||| P_2$ allows both P_1 and P_2 to execute concurrently and independently, except for $\checkmark$.

CSPm Language CSPm is the machine-readable variant of CSP that allows describing concurrent systems in an executable manner. It combines the process algebra of CSP with an expression language based loosely on the functional programming language Haskell [23]. Table I summarises CSPm notation corresponding to the blackboard notation above.

TABLE I
CSPM NOTATION

Basic operator	Notation
Prefix	$P1 \to$P2.
Input	?x
Output	!x
Sequential composition	P1;P2
External Choice	P1 [] P2
Internal Choice	P1 [~] P2
Alphabetised parallel	P [A] Q
Interleaving	$P1$ |||P2

Semantic Models. Whilst there are several different semantic models for CSP processes [22], our work focuses only on the simplest, finite trace semantics.

A *trace* is defined as a, possibly empty, sequence of events from Σ that may terminate with $\checkmark$.

Formally, let Σ^* denote the set of all finite sequences of events from Σ, $\langle\rangle$ the empty sequence, and $tr_1 \frown tr_2$ two traces tr_1 and tr_2 concatenated. The set of all traces is then defined as: $\Sigma^{*\checkmark} = \{tr \frown en \mid tr \in \Sigma^* \wedge en \in \{\langle\rangle, \langle\checkmark\rangle\}\}$.

The trace tr_1 is a *prefix* of a trace tr_2, written as $tr_1 \leq tr_2$, iff $\exists\, tr' : tr_1 \frown tr' = tr_2$.

Events in $A \subseteq \Sigma \cup \{\checkmark\}$ may be abstracted away from a trace tr by a hiding operator, written as $tr \setminus A$ and defined as:

$$tr \setminus A = \begin{cases} \langle\rangle & \text{if } tr = \langle\rangle \\ \langle a\rangle \frown (tr' \setminus A) & \text{if } tr = \langle a\rangle \frown tr' \wedge a \notin A \\ tr' \setminus A & \text{if } tr = \langle a\rangle \frown tr' \wedge a \in A. \end{cases}$$

For convenience, when $A = \{a\}$, we shall simply write $tr \setminus a$.

In general, the trace semantics of a process P is a subset $traces(P)$ of $\Sigma^{*\checkmark}$ consisting of all traces that the process may exhibit.

Formally, traces for operators listed above are defined recursively as follows:

- $traces(Stop) = \{\langle\rangle\}$;
- $traces(e \to P) = \{\langle\rangle\} \cup \{\langle e\rangle \frown tr \mid tr \in traces(P)\}$;
- $traces(P_1 \Box P_2) = traces(P_1) \cup traces(P_2)$;
- $traces(P_1;\ P_2) = traces(P_1) \cap \Sigma^*$ $\cup \{tr_1 \frown tr_2 \mid tr_1 \frown \langle\checkmark\rangle \in traces(P_1) \wedge tr_2 \in traces(P_2)\}$;
- $traces(P_1 \underset{A}{\parallel} P_2) = \{tr \in tr_1 \underset{A}{\parallel} tr_2 \mid tr_1 \in traces(P_1) \wedge tr_2 \in traces(P_2)\}$ where $tr_1 \underset{A}{\parallel} tr_2 = tr_2 \underset{A}{\parallel} tr_1$ is defined as follows with $a, a' \in A \cup \{\checkmark\}$ and $b, b' \notin A$:
$\langle\rangle \underset{A}{\parallel} \langle\rangle = \{\langle\rangle\}$;
$\langle a\rangle \frown tr_1 \underset{A}{\parallel} \langle b\rangle \frown tr_2 = \{\langle b\rangle \frown tr \mid tr \in \langle a\rangle \frown tr_1 \underset{A}{\parallel} tr_2\}$;
$\langle a\rangle \frown tr_1 \underset{A}{\parallel} \langle a\rangle \frown tr_2 = \{\langle a\rangle \frown tr \mid tr \in tr_1 \underset{A}{\parallel} tr_2\}$
$\langle a\rangle \frown tr_1 \underset{A}{\parallel} \langle a'\rangle \frown tr_2 = \emptyset$ where $a \neq a'$;
$$\langle b\rangle \frown tr_1 \underset{A}{\parallel} \langle b'\rangle \frown tr_2 = \{\langle b\rangle \frown tr \mid tr \in tr_1 \underset{A}{\parallel} \langle b'\rangle \frown tr_2\} \cup \{\langle b'\rangle \frown tr \mid \in \langle b\rangle \frown tr_1 \underset{A}{\parallel} tr_2\}$$
- $traces(P \setminus A) = \{tr \setminus A \mid tr \in traces(P)\}$;

Trace interleaving is defined as parallel composition with an empty synchronisation set.

- $traces(P_1 ||| P_2) = P_1 \underset{\emptyset}{\parallel} P_2$

Trace Refinement. A usual way to analyse CSP processes is via trace-refinement. A process P is said to *trace-refine* a process Q (written $Q \sqsubseteq_T P$) if $traces(P) \subseteq traces(Q)$. There are other flavors of refinement, but we restrict ourselves to trace refinement below.

B. ECU Software Development

With its significant automotive user base, the CANoe IDE, from Vector Informatik GmbH, serves as a realistic ECU software development tool. CANoe covers the entire ASE lifecycle for coding, testing, and simulating individual ECUs or entire in-vehicle networks, with support for most common automotive network protocols (i.e. LIN, CAN, FlexRay, and MOST). Although our initial work relates to CAN-based networks, as techniques in this paper mature, there is scope to explore other networking protocols with the same environment.

1) Programming ECU Software: Developers may simulate network nodes by developing ECU functions in CANoe with typical imperative programming languages used for automotive systems, namely C, C++ and .NET [2]. Alternatively, software designers can use the in-built, albeit proprietary, Communication Programming Access Language (CAPL) from Vector.

CAPL is based on the C language, but adds a superset of pre-defined functions for networking and controlling the IDE [24]. Like C, CAPL programs are compiled using the bundled compiler. However, unlike C, CAPL programs are event-driven, responding to system-generated events like timer expiry or received messages; as such, no main() routine is

needed [25]. A CAPL program comprises four types of code block: optional *includes* and *variables* sections, one or more *event procedures* or (user-defined) *functions*.

2) CANoe Network Database: CANoe may use underlying database(s) to define ECU transmission behaviour and signals exchanged by actuators, sensors and ECUs. These so-called CAN databases are textual files (*.dbc extension) holding all necessary information about message formats, data payloads and relationships of data packets to network components. Despite message semantics for particular automobiles tending to be proprietary, CAN messaging and the CANdb format itself have become a *de facto* standard within the automotive industry. CAPL links seamlessly with any associated CANdb databases to access message formats and signal fields.

C. Code Translation

To extract an implementation model, we apply parsing techniques using the open source ANTLR (ANother Tool for Language Recognition) tool. ANTLR is a parser generator that processes an input grammar file, automatically generating a lexer and a parser [26] in the target language defined at runtime, in our case, we specify Java classes.

Additionally, ANTLR automatically creates an empty program containing skeletal methods, each corresponding to nodes of an Abstract Syntax Tree (AST) representation of the source language. Parser rules defined in the grammar file determine which nodes populate the AST. The outline implementation programm can then be extended to create application(s) for reading, processing and translating text or even binary files.

ANTLR also includes a template engine, called StringTemplate (ST), that aims to separate application logic from display format definitions [27]. By defining a series of templates, into which variable text is inserted, developers have greater control over the order and appearance of textual output, particularly useful feature when translating programming languages with different symbol sets.

D. CSP Refinement Checking

FDR (Failures-Divergences-Refinement) is a refinement checker developed specifically for use with CSP [28]. It uses model-checking techniques and also incorporates visualisation tools to display process transition models and traces. Typical usage of FDR compares two process models, normally a specification (intended behaviour) and an implementation. It is possible to compose quite complex process models using the composition operators described earlier. More detailed information is available regarding tool documentation [29], usage of FDR [22] and latest developments [28].

E. Attack Models

Attack models capture malicious behaviour against a system. With CSP, a common approach is to define an additional intruder process in CSP, based on the Dolev-Yao model. This provides a general, formally-defined model based on a worst-case security threat scenario, defining what the intruder knows and can learn, and capabilities in terms of manipulating messages transmitted over the network. This intruder (attacker) model is then added, in parallel, to existing process models for various network components [30].

Allied to this, attack trees are now gaining acceptance as a graphical way to define step-by-step particular attacks. Recent work confirms that an individual attack tree can be translated into a semantically equivalent CSP process [17]. This equivalence is based on an observation that a series-parallel (SP) graph represents a set of action sequences, each action corresponding to a traverse from a source node to a sink node of the graph. Formally, the set of sequences of actions of an SP graph can be defined recursively as follows:

$(\xrightarrow{a}) = \{\langle a \rangle\}$;
$(G_1 \parallel G_2) = \{s \in s_1 \mid\mid\mid s_2 \mid s_1 \in (G_1) \wedge s_2 \in (G_2)\}$;
$(G_1 \cdot G_2) = \{s_1 \frown s_2 \mid s_1 \in (G_1) \wedge s_2 \in (G_2)\}$.

The function $(\cdot)$ is also generalised to the case of sets of graphs as follows:

$$(\{G_1, \ldots, G_n\}) = \bigcup_{i \in \{1,\ldots,n\}} (G_i)$$

V. Case Study: Vehicle ECU Software Updates

Our case study considers updates to ECU firmware within a road vehicle and draws on United Nations Economic Commission for Europe (UNECE) efforts to harmonise vehicle regulations worldwide, which recently addressed cyber security and Over-the-Air (OTA) issues. The International Telecommunication Union (ITU) supports this work with technical guidance for OEMs, issued as recommendation ITU-T X.1373 [31]. Additionally, new international standard ISO/SAE 21434 "Road vehicles - Cybersecurity engineering", is under development and will specify requirements for updates to vehicles, covering development, delivery and application of updates, whether OTA or by other methods. In the meantime, we use X.1373 to inform our case study.

A. Basic Model of Software Update

X.1373 sets out a basic secure architecture for remote software updates, comprising the following components [31]:

- **Update Server**: An update server may be located at OEM sites, supplier sites or dealer garages, and usually includes a logging database to store status information about software components in each vehicle.
- **Vehicle Mobile Gateway (VMG)**: A conceptual entity acting as an intermediary between the OEM (or supplier) and the vehicle. Its role is to provide communication services to the update server and to manage the update process. Typically, the VMG may be embedded in a vehicle gateway or "Head unit" and uses cellular (mobile phone) network or fixed local wireless network.
- **Target ECU(s)**: One or more ECUs on board the vehicle requiring updated software. Vendors are expected to provide an accessible update module within the core functional services of each ECU.

Fig. 2. Scope of Software Update Case Study Demonstration System.

1) Scope: As our initial aim was to demonstrate the feasibility of the approach, the demonstration scope includes only the VMG and ECU components, as shown in Figure 2. This restricts the associated message types to those listed in Table II; future work will expand the scope to include the update server and other message types defined in [31], namely, *diagnose, update_check, update, and update_report* (see Section VIII-A).

TABLE II
MESSAGE TYPES AND MESSAGE USED [31]

Type	Id	From	To	Description
Diagnose	reqSw	VMG	ECU	Request diagnose software status
	rptSW	ECU	VMG	Result of software diagnosis
Update	reqApp	VMG	ECU	Request apply update module
	rptUpd	VMG	ECU	Result of applying update module

2) Requirements and Assumptions: Consider the high-level requirements, derived from [31], listed in Table III).

TABLE III
SECURE UPDATE SYSTEM REQUIREMENTS

ID	Requirement Text
R01	At start of update process, the VMG shall send a software inventory request message to all ECUs.
R02	On receipt of software inventory request, the ECU shall send a software list response message.
R03	On receipt of apply update message from the VMG, the ECU shall check the package contents and apply the update.
R04	On completion of update module installation, the ECU shall send software update result message to the VMG.
R05	It is assumed the system uses shared keys (see below).

To accommodate a variety of vehicle makes, model types, and model years, X.1373 acknowledges that different cryptographic capabilities for securing messages between components may be installed [31]. It offers use of either digital signatures, where asymmetric cryptography is available, or Message Authentication Code (MAC), where symmetric cryptography based on shared keys is applicable. Initially, to simplify our demonstration by avoiding the need to include a Certification Authority, we assume use of shared keys; further work will expand this to include asymmetric cryptography.

B. ECU Specification Model

In our case, the ECU specification model needs to addresses security properties. Strictly speaking, the FDR model checker is actually a refinement checker, so we must capture security properties as abstract CSP models, then use FDR to check that these security property models are refined by a CSP model extracted from CAPL code.

Suppose that we wish to check the integrity of a transmission from the VMG to the ECU (as per requirement R02 in Table III). Considering integrity in the context of OTA software updates means avoiding receipt of unauthorised software update messages, in other words the message exchange must progress, as specified, in the correct sequence.

A simple implementation of R02 would be a security process SP_{02} that ensures that every time a software inventory request (denoted by *reqSw*) is received, a software list response message (denoted by *rptSw*) is returned.

If we define the channels transmitting messages as *channel send, rec: msgs*, then we can define SP_{02} as

```
SP02 = rec?reqSw -> send!rptSw -> SP02
```

We then would expect that process SP_{02} is refined by the system composed of VMG and ECU processes, ($SYSTEM = VMG \parallel ECU$), which we could check within FDR.

More sophisticated models, to be developed, would allow other messages to be received on a different channel (other) provided a *rptSw* message was sent as soon as a request is received. Similarly, our definition of datatype *msgs* can be extended with additional fields needed in message formats to implement correct security protocols, (e.g. nonces, source and destination identifiers). A detailed description of CSP models used to analyse security protocols in available at [30].

VI. RESULTS

The focus of work to date has been the core element, translating ECU application code into a machine language representation (CSPm) for FDR. In preparation, a simulated CANbus network was implemented in CANoe, with components (per Figure 2) programmed to exchange simple messages as defined in our requirements. Also, a CAPL-specific, ANTLR grammar was created to parse CAPL constructs specific to CANbus messages, namely *on message* event procedures and *output* statements. Our parse rules also recognise message declarations, which are then output as CSPm channel type declarations. We also developed several ST templates to correctly place chunks of source code, like message identifiers and event procedure names, into the target code.

By first running ANTLR with the CAPL grammar, we successfully generated Java classes for both a CAPL lexer and a parser, plus an outline listener application, which was modified with overridden methods that write CSPm statements, as required. The final model extractor application combines these classes to process source files through successive lexing, parsing, template generating stages before finally writing a target file. Essentially, this creates a pipeline processor to automate extracting a CSP model from CANoe for FDR to process. An example output is shown Figure 3.

```
Console ☒  Problems  Error Log  Debug Shell
<terminated> CAPL Translator (ST) [Java Application] /Library/Java/.
{- ****************************

   CSPm Module generated on: dd-mmm-yy
   Source filename: test.can

   ****************************
-}

channel : DataOutMsg, InMsg

ECU =  InMsg?x DataOutMsg!x -> ECU
```

Fig. 3. Example of ECU Implementation Model (CSPm script) automatically generated from the application code for simulated CAN bus network in a CANoe environment.

VII. Discussion

For acceptance by OEMs and their component vendors, our security verification methods must function properly, but also scale for real-world systems, be accessible for business and engineering users, and be compatible with existing processes.

A. Advantages

Our CSP-centric approach builds on a mature and effective formal language for modeling the type of distributed, embedded system used in modern vehicles. It is proven for verifying properties of wide ranging safety-critical systems, whilst CSP-based analysis is known to successfully verify, or identify flaws with, diverse security properties.

Moreover, powerful, complementary tools now exist in the guise of the FDR refinement checker. Recent FDR development work has introduced support for large-scale verification using either grid-based or cloud-based computing. This now opens the door for automating component-level security checks at the scale needed for the sophisticated ECUs now seen in vehicles. Yet the gap between the typical automotive software programming languages and the machine-readable language of the tools has been an obstacle.

To help address this, our translation approach is entirely model-driven, with models for ECU source code (the CAPL grammar), an intermediate model (the parser-generated AST), and a target language model (CSPm templates). By working with models, we hope the method is more accessible to business domain experts and software engineers; this hides the underlying formalism allowing conversations at the right level using familiar terms.

A model-based approach should also be compatible with existing processes, where organisations already have adopted such model-based design and testing practices. The capability to work at different abstraction levels enables this method to be used directly in the design stage, helping to identify potential vulnerabilities earlier.

Using a grammar, plus output templates, creates an opportunity for re-purposing our techniques to translate ECU source code written in other programming languages, or even producing output code that is consumable by alternate process algebra tools.

B. Limitations

Despite the clear advantages, some limitations are also evident. Firstly, for simplicity, we have confined our initial use of CSP to its basic, untimed version. Whilst this appears to limit the utility of our approach for analysing time-triggered tasks in ECUs, two proven approaches can potentially address this situation; either using the Timed CSP variant [32], which provides a continuous time model, or by simply extending the alphabet of our models to include a specific *tock* event [19]. The latter, more practical approach would be appropriate for extending our work to modeling time-dependent features.

Secondly, we are constrained by the need to access related design and implementation artifacts (such as requirement specification, source code) to help develop specification and implementation models. These may be unavailable where OEMs and ECU vendors seek to protect their intellectual property. Here, we assume that external, independent testing organisations (or internal testing teams) would have access to necessary specifications or high-level designs, and source code.

At this stage, our implementation is simply a proof-of-concept, with an incomplete CAPL grammar. Further work is needed to extend the model extractor application and evaluate it, using real code, to confirm correctly handling of real world ECU implementations.

From a software development lifecycle perspective, our focus thus far is on the implementation phase, whereby ECU code has already been produced. It would be useful to also consider models in work products from earlier phases, such as requirements specifications.

In terms of development tools, we have only worked with Vector tools, using CAPL and CANdb files. Many real-world ECUs instead are developed using C or C++, while some automoitve application domains rely on MATLAB programming. Although our model-based approach seems adaptable to other programming language sources, further evaluation with other programming languages is worthwhile.

We also have concerns about potential limitations in fully capturing the semantics of imperative program, like CAPL, and faithfully embedding that into the functional programming paradigm used by CSPm. CSPm is limited as a functional language, deliberately so, as its primary purpose was describing concurrent processes rather than full implementations [18]. Faithfully reproducing CAPL-based (or other) applications may be further complicated by diverse programming styles and coding standards used in different development teams. However, we note industry efforts to promote common coding standards, such as MISRA and AUTOSAR, may mitigate this.

VIII. Conclusions and Future Work

Our paper establishes a cornerstone on which to build further work. The concept demonstrated can be generalised to accommodate other input languages or formats and different translations (e.g. other process algebras). CSP is suited to different abstraction levels, so feasibly may be applied earlier in the ASE lifecycle. This benefits industry since there is no

need to produce real-world prototypes (thereby reducing cost), but also allows for quicker engineering lifecycle phases.

Our approach also allows existing development processes to remain in place, since we provide a parallel formalised modeling structure (based on CSPm); this structure should also be flexible enough to accommodate future work practices. Automotive business analysts and software engineers can also be shielded from mastering formal methods, therefore helping bridge the gap between application domain experts and those within the security or mathematics domains.

Finally, our work, which utilises a common automotive industry development environment, coupled with mature, public domain tools, may offer a cost-effective, pragmatic way to improve software security verification for real ECU components.

A. Future Work

Next steps would follow two routes. Firstly, improve our model extraction capability, for example, by extending the CAPL grammar and the translator to parse functions and data structures into corresponding CSPm elements. Additionally, identifying and writing CSP parallel operation constructs, including external choice and sequential composition, would allow building composite ECU models. Possibly, a second parser and model generator is warranted to handle CAN database files, extracting message formats as CSPm declarations for data types, name types, and data ranges.

Secondly, extending the breadth of our formal models. These could include more automotive subsystem models, whilst also looking to develop CSP models for other security property types and compose them with realistic attack(er) models. These could then be used to evaluate representative real-world components. Finally, exploring CSP model extraction at higher levels (e.g. from requirements) may enable automatic generation of ECU specification models from work products developed in early phases of an ASE lifecycle.

REFERENCES

[1] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 26, no. 1, pp. 100–106, Jan 1983.

[2] H. Altinger, F. Wotawa, and M. Schurius, "Testing methods used in the automotive industry: results from a survey," in *Proc. of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing (JAMAICA)*. California, USA: ACM Press, 2014, pp. 1–6.

[3] A. Haghighatkhah, A. Banijamali, O. P. Pakanen, M. Oivo, and P. Kuvaja, "Automotive software engineering: A systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 25–55, 2017.

[4] A. Greenberg, "Hackers Remotely Kill a Jeep on the HighwayWith Me in It," 2015. [Online]. Available: http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[5] T. Hoppe, S. Kiltz, and J. Dittmann, "Security threats to automotive CAN networks - Practical examples and selected short-term countermeasures," *Reliability Eng. & System Safety*, vol. 96, no. 1, pp. 11–25, Jan 2011.

[6] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces." in *Proceedings of 20th USENIX Security Symposium*. San Francisco, CA: USENIX Association, Aug 2011, pp. 77–92.

[7] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Snachám, and S. Savage, "Experimental security analysis of a modern automobile," in *Proceedings - IEEE Symposium on Security and Privacy*, 2010, pp. 447–462.

[8] A. Bento, K. Roth, and Y. Zuo, "Vehicle lifetime and scrappage behavior: Trends in the US used car market," *Energy Journal*, vol. 39, no. 1, pp. 159–183, 2018.

[9] D. S. Fowler, J. W. Bryans, S. A. Shaikh, and P. Wooderson, "Fuzz testing for automotive cyber-security," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, Luxembourg, Jun 2018, pp. 239–246.

[10] W. Schutz, "Fundamental issues in testing distributed real-time systems," *Real-Time Systems*, vol. 7, no. 2, pp. 129–157, sep 1994. [Online]. Available: http://link.springer.com/10.1007/BF01088802

[11] H. Thane and H. Hansson, "Towards systematic testing of distributed real-time systems," in *Proc. of 20th IEEE Real-Time Systems Symposium*. IEEE Comput. Soc, 2003, pp. 360–369.

[12] ISO, "ISO26262-1:2011 Road vehicles - Functional Safety - Part 1: Vocabulary," 2011.

[13] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[14] G. Lowe, "An attack on the Needham-Schroeder public-key authentication protocol," *Information Processing Letters*, vol. 56, no. 3, pp. 131–133, Nov 1995.

[15] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *FoSE 2007: Future of Software Engineering*. IEEE, may 2007, pp. 55–71.

[16] M. Cheah, S. A. Shaikh, J. W. Bryans, and H. N. Nguyen, "Combining third party components securely in automotive systems," in *Proc. 10th Int. Conf. for Information Security Theory and Practice*, Crete, Greece, Sep 2016, pp. 262–269.

[17] M. Cheah, H. N. Nguyen, J. Bryans, and S. A. Shaikh, "Formalising Systematic Security Evaluations using Attack Trees for Automotive Applications," in *11th WISTP International Conference on Information Security Theory and Practice*, 2017.

[18] C. A. R. Hoare, *Communication Sequential Processes (electronic version)*. International Association for Energy Economics, 2015. [Online]. Available: http://www.usingcsp.com/cspbook.pdf

[19] A. W. Roscoe, *Communicating Sequential Processes. The First 25 Years*, A. E. Abdallah, C. B. Jones, and J. W. Sanders, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3525.

[20] S. Schneider, *Concurrent and Real-time Systems - The CSP Approach*, 1st ed. Chichester: John Wiley & Sons Ltd., 2000.

[21] M. Moran, J. Heather, and S. Schneider, "Verifying anonymity in voting systems using CSP," *Formal Aspects of Computing*, vol. 26, no. 1, pp. 63–98, Jan 2014.

[22] A. W. Roscoe, *Understanding concurrent systems*. Springer, 2010.

[23] B. Scattergood and P. Armstrong, "CSPm: A Reference Manual," Oxford University, Oxford, Tech. Rep., Jan 2011. [Online]. Available: http://www.cs.ox.ac.uk/ucs/cspm.pdf

[24] Vector Informatik GmbH, "CANoe: ECU and Network Testing on Highest Level." [Online]. Available: https://www.vector.com/gb/en-gb/products/products-a-z/software/canoe/

[25] M. Lobmeyer and R. Marktl, "Tips and tricks for the use of CAPL (part 2)," *CAN Newsletter*, pp. 10–12, 2014. [Online]. Available: www.vector.com

[26] T. J. Parr and R. W. Quong, "ANTLR: A predicatedLL(k) parser generator," *Software: Practice and Experience*, 1995.

[27] T. J. Parr, "Enforcing strict model-view separation in template engines," in *Proc. of the 13th Int. Conf. on World Wide Web*, 2004, pp. 224–233.

[28] P. Armstrong, M. Goldsmith, G. Lowe, J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell, "Recent Developments in FDR," in *Proc. of 2012 International Conference on Computer Aided Verification*, vol. 7358 LNCS. Springer, Berlin, Heidelberg, 2012, pp. 699–704.

[29] U. of Oxford, "FDR Manual Release 4.2.3," University of Oxford, Tech. Rep., 2017. [Online]. Available: https://www.cs.ox.ac.uk/projects/fdr/downloads/fdr-manual.pdf

[30] P. Ryan and S. Schneider, *Modeling and Analysis of Security Protocols: The CSP approach*. Addison-Wesley, 2001.

[31] International Telecommunications Union (ITU), "X.1373 : Secure software update capability for intelligent transportation system communication devices," ITU, Tech. Rep., 2017.

[32] G. Reed and A. Roscoe, "A timed model for communicating sequential processes," *Theoretical Computer Science*, vol. 58, no. 1-3, pp. 249–261, Jun 1988.