A Process Algebraic Mutation Framework with Application to a Vehicle Charging Protocol

Benjamin Aziz

School of Computing
University of Portsmouth
Portsmouth PO1 3HE
United Kingdom

Abstract
Modern day renewable and smart energy systems are increasingly playing an important role in our societies. Electric vehicle charging services infrastructures are becoming ever more popular and accessible, although at a pace that we argue is not well understood yet in terms of the reliability and security of this new paradigm. Often solutions are rolled out to the market with little or no analysis of their resilience and even at the standards level, specifications of systems and protocol often omit or skim over the question of reliability. In this paper, we propose a new formal approach for analysing the effects of single failures on process algebraic specifications used often in specifying communication protocols. We apply this new approach to one part of an international standard protocol, the Open Charge Point Protocol, for electric vehicle charging, and demonstrate how the effects of specification faults can impact the safety and security of the protocol.

Keywords: Electric Vehicles, Formal Specifications, Mutation Analysis, Open Charge Point Protocol, Process Algebra

1. Introduction

At the core of most global energy strategies nowadays is the goal of developing smart infrastructures that can harness clean energy ecosystems in order to lower or eliminate the impact of fossil fuel on the environment. For example, the UK government announced recently that it will phase out the sale of diesel and petrol-only vehicles by 2030, with all new vehicles sold
in 2035 being zero transmission at the tailpipe. In order to implement such
green policies, this will require the development of charging services and sup-
port infrastructures for Electric Vehicles (EV), which will aim at providing
information as well as services related to points of charging, traffic events
and other road services that support drivers’ experience throughout their
journey. As a result, standardisation effort has also been on the upbeat in
recent years aiming at regulating EV system communications and interaction
with the EV infrastructure. More specifically, many standards related to the
charging of EVs have emerged proposed [2, 25, 32].

However, like any other critical infrastructure, the resilience of EV charg-
ing infrastructures remains at the forefront of the challenges facing adoption
of the new EV technology [7]. In particular, the reliability of the communi-
cation protocols used in these infrastructures in the face of failures resulting
from bad or ambiguous specifications is an important aspect that contributes
to their resilience, and mutation testing and analysis [33] is one technique
widely adopted by industry through which this reliability can be understood.

In general terms, mutation testing and analysis is concerned with the
concept of introducing small single faults or errors into a system’s design,
specification, implementation or interface and then testing or analysing the
effects caused by those faults or errors. It has also been used as a technique
for evaluating the quality of test suites, as the more faults the tests can
spot, the better the quality of those tests. In some sense, mutating systems
simulate real world scenarios, where behaviour of the system is not well-
defined or well-implemented, and its communicated messages can be subject
to unexpected alterations either by the physical nature of the communication
medium, such as any lossy or harsh network, or by some intended or unin-
tended interference from users or other systems, for example, due to actions
carried out by malicious external intruders.

Yet, another source of such faults, or mutations, is the decisions made
at the specification level, or implementation decisions that we can model
back to the initial specification level. Such “high-level” mutations require
studying and inspection at the level of a system’s specification, since their
effect could be more fundamental than mutations at the more technical levels
of the software development life-cycle. The problem becomes even more
urgent when dealing with specifications of standards, since standards are
implemented in many ways and could have far more reaching impact on
technologies than any one specific system.

Our approach in this paper attempts to make some progress in under-
standing the impact of mutations – in this case, on vehicular charging protocols. First, we attempt to define a mutation framework that works at the level of the formal specification of a system. We achieve this by using a formal language; a timed process algebra called TPi \cite{9}, and then define a general mutation function that builds mutants specified in this language. The second, we apply this formal framework to a small part of a standard for a protocol widely used in the electric vehicle charging industry, called the Open Charge Point Protocol (OCPP) \cite{2}, and statically analyse the communications of the resulting mutations. This achieves to demonstrate the applicability of our approach and at the same time, sheds some light on the effects of mutations on that part of the protocol.

1.1. Contributions of the paper

At a high level, this paper advances the current state of the art literature by defining a combined formal mutation and abstract interpretation framework for a timed message-passing process algebra. The paper, as such, makes two major contributions to research at the intersection of the areas of formal languages, mutation techniques and vehicular communication protocols:

1. Define a formal mutation framework that can be used to introduce mutants of systems and protocols at the specification level, based on a single general mutation function. This framework uses a timed message-passing process algebra, called TPi, as its modelling language, and it is currently defined for three kinds of mutations on this language; namely, mutations of channel names, communicated messages and time duration of input actions

2. Model and statically analyse the effects of the above-mentioned mutations on a critical component of a vehicle charging standard adopted widely in industry, called OCPP. We demonstrate that the mutation framework generates 11 classes of mutants, some with no effects and some with potentially harmful effects

As we will discuss later in Section 2, this approach is both refreshing in terms of its contribution to the area of mutation analysis and practical in terms of its ease of applicability to industrial standards specifications rather than being confined to the space of toy examples.
1.2. Structure of the paper

The structure of the rest of the paper is as follows. Section 2 discusses works in literature related to our approach and analysis. Section 3 gives a general background on the \( TPi \) calculus and the OCPP standard. Section 4 introduces the mutation framework for \( TPi \), where we define the tagging approach and the general mutation function. In Section 5, we apply our mutation function to one example of a widely-used protocol, namely, the Heartbeat component of the OCPP standard. In Section 6, we present a static analysis that can be used to analyse the effects of the mutants generated for the example protocol. Finally, in Section 7, we conclude the paper and discuss avenues for future research.

2. Related Work

Mutation testing is that field of software testing concerned with the measurement of the quality of software tests and test suites through the introduction of artificially generated individual faults known as mutations, therefore, creating mutant variations of the original system. A mutation adequacy score is then used to measure the fitness of tests and test suites against such mutations and hence provide some understanding of the quality of the tests used. The idea was first proposed in 1971 in a student report by Lipton [37] and has gained major momentum in recent years due to its wider applicability and the generality of its underlying principles. Jia and Harman [33] provide an excellent survey of literature on mutation testing and its applicability at various levels of the software life cycle (for a more recent survey, see [44]).

The idea of the mutation testing of protocol specifications itself is not new either. Sidhu and Leung [46] were first to propose in 1988 the application of mutation testing to protocol specifications. Since then, mutation testing has also been applied to finite state machines (e.g. [24]), statecharts (e.g. [50]), Estelle specifications (e.g. [20]) and Petri Nets (e.g. [23]) amongst other formal and semi-formal specification languages. Of particular interest are the works of [18, 19], in which the authors propose a set of common mutations in security protocol specifications that could be considered when performing a formal static analysis. Our work here is inspired by the same approach, although our work focuses strictly on mutations of the system specification, including the transmission of erroneous messages, rather than for example mutations related to the underlying implementation. One limitation in the
works of [18, 19] is that they do not specify a suitable static analysis method to analyse the effects of the mutations.

Another approach is to use input fuzz operators [47], in order to mutate the inputs to a protocol implementation and then test the effects of these inputs on the system’s security. This type of mutation is a more detailed form of message mutation that we consider here, however it does not deal with the other types of mutations like the changing of process actions or operators. One of the earliest works introducing the idea of mutation testing to protocols and their security properties were [34] and [48] using the validation tool AutoFocus. Again, these works are based on the idea of mutating the nature of a message, as they assume that the only source of faults are faults caused by the network, and do not consider errors introduced in the system specification or implementation. Mutation testing and fault injections have also been used as methods to test the robustness of other aspects of systems’ security, such as access control policies, e.g. as in [39], the testing of online attacks using SQL injections, e.g. [10] and specific property-driven testing of security protocols, e.g. [16]. In all of the above works, the difference with our approach is that mutation generation has been used in the classical set up of software testing, rather than in combination with some formal analysis method.

Despite the importance of modern day smart and renewable energy systems, their research as an area of application for formal modelling and verification has remained somehow limited, perhaps due to the complexity of such systems. There are a few exceptions. Patil et al. [45] proposed a model-checking framework based on SMV [40] for the purpose of designing robust smart grid applications that check for systems’ liveliness and perform boundary value checks. Abate [6], on the other hand, discusses the problem of the verification of networks of smart energy systems and suggests several challenges including the transition from verification of these systems to correct-by-design synthesis and the merger of data-driven approaches with classical formal verification. His paper perhaps best highlights the complexity of such systems, as the main challenge.

Another interesting area that has been used as a case study for formal methods is micro-grids, which represent decentralised areas of electricity generation and consumption. In [29], Gentile et al. propose fluid stochastic Petri Nets [30] as a method for modelling the configuration of localised user energy consumption and costs, and present a tool called µGRIMOIRE, that can be used in verifying such configurations. In [49], the authors use linear temporal logic as the underlying formalism to specify the requirements of frequency
regulation in power systems in relation to the integration of electric vehicles to these systems. The symbolic controller approaches adopted in [49] guarantees the settlement of an after-event’s frequency in some specified safe interval, once a failure event is encountered.

Of specific relevance to our paper are works that have targeted the formal modelling and verification of electric vehicle charging protocols, and there have been varieties of such approaches. In particular, we mention [26], in which the authors use the applied $\pi$ calculus [5] and the ProVerif tool [4] to model and verify privacy properties in a privacy-preserving variant of the vehicle-to-grid communication standard, ISO15118 [27], called POPCORN [31]. Their work, which advocates a Dolev-Yao model of the attacker [21], reveals that whilst POPCORN preserves weak secrecy, it does not preserve strong secrecy and strong unlinkability, e.g. e.g. where a user is linked to their use of the system. The authors also suggest some modifications to the protocol to address these issues.

In other relevant works, dos Santos et al. [22] highlighted the importance of formal modelling in understanding the nature and impact of external threats to vehicular communications and automotive systems. Their work focuses on modelling attacks with the use of predicate/transition nets [28], in particular, attacks related to the compromising of systems’ integrity when injected with incorrect data, and whether redundancy countermeasures can help mitigate against such attacks. In [36], the authors combined both the ProVerif [4] and the Tamarin [41] tools to conduct a symbolic analysis of an electric vehicle charging protocol proposed by [38]. Unfortunately, the latter work provides an example of where the application of formal analysis techniques does not rise beyond being a simple exercise with little impact due to the limited usage of the protocols being analysed, even when those protocols are part of an important domain, such as electric vehicle communications. In this paper, we sought from the onset to avoid this problem by focusing on (a subset of) a protocol that is widely used at an industrial level. In all of the works above that use formal analysis methods, no mutation testing or mutation generation techniques have been considered. At best, these analyses are only conducted under the classical conditions and sometimes assuming the presence of a hostile adversary.

So, to summarise, existing literature has mostly focused on either the definition of classical mutation testing techniques, which are defined with the aim of providing some measure of fitness for testing suites, or it has adopted a variety of formal analysis methods without the consideration for
the mutation problem. The current approach we suggest in this paper is unique in that it combines the problem of mutation with that of the formal analysis under one framework, to create what we term a formal mutation analysis rather than the classical approach of mutation testing adopted in literature so far.

3. Background

We provide here some background overview on the formal language and the case study used in our framework.

3.1. Timed π-Calculus

First, we introduce the formal language we use in the framework, called the Timed π-calculus (or TPi for short), a variant of the π-calculus [43] defined in [9] and initially inspired by the calculus of [15]. In TPi, processes, $P, Q, R, \ldots \in \mathcal{P}$, are defined according to the following BNF [14] syntax, using names, $x, y, z, \ldots \in \mathcal{N}$:

$$P, Q ::= \pi(z).P | \text{timer}^t(x(y).P, Q) | !P | (\nu x)P | (P \parallel Q) | (P+Q) | 0$$

The syntax makes use of timed input actions, \text{timer}^t(x(y).P, Q), where $t \in \mathbb{N}$ is a natural number representing a time bound on the input action. Basically, the input action, $x(y).P$, can synchronise with suitable output actions, e.g. $\pi(z).P$, as long as $t > 0$. Otherwise, when $t$ reaches 0 and no matching output actions have become available, the timer will switch to behaving as $Q$. There is an assumption that $t$ is decremented by the environment and that $t$ can be any time unit (e.g. a tick, a second etc.). This can be expressed by means of a clock function, $\partial : \mathcal{P} \rightarrow \mathcal{P}$ that ticks down time [9]:

$$\partial(P) = \begin{cases} 
\text{timer}^t(x(y), Q, R) & \text{if } P = \text{timer}^{t+1}(x(y), Q, R) \\
& \text{and } 0 < t + 1 < \infty \\
\text{timer}^t(x(), Q, R) & \text{if } P = \text{timer}^{t+1}(x(), Q, R) \\
& \text{and } 0 < t + 1 < \infty \\
\partial(Q) | \partial(R) & \text{if } P = Q \parallel R \\
\partial(Q) + \partial(R) & \text{if } P = Q + R \\
(\nu x)\partial(Q) & \text{if } P = (\nu x)Q \\
!\partial(Q) & \text{if } P = !Q \\
P & \text{otherwise}
\end{cases}$$
Sometimes, a communication does not pass any message (i.e. it is a pure synchronisation à la CCS [42]), in which case we write the output action simply as $x().P$ and the input action also as $\text{timer}'(x().P, Q)$.

The other constructs of the syntax include process replication, $!P$, new name creation with scope restriction, $(\nu x)P$, parallel composition, $(P | Q)$, non-deterministic choice $(P + Q)$ and the inactive null process, 0. These are all borrowed from the standard $\pi$-calculus [43]. We sometimes use aliases, in the form of Protocol $\overset{\text{def}}{=} P$, when we want to group some behaviour, $P$, under a specific name (e.g. the name of a protocol or sub-part of a protocol).

We adopt the standard notions of the bound names, $\text{bn}(P)$, and free names, $\text{fn}(P)$, of a process, $P$, and $\alpha$-conversion of bound names [43].

The structural operational semantics of TPi is formalised using a structural congruence relation, $\equiv$, and a labelled transition relation, $\mu \rightarrow$, both shown in Figure 1. The definition of $\equiv$ is standard, except for rules (6)–(9), which deal with expired and infinite timers. We note here that despite the fact that the set of names of a process, $n(P) = \text{bn}(P) \cup \text{fn}(P)$, is syntactically finite, Rule (4) can cause this set to expand to an infinite size. To illustrate this point, we distinguish between the different copies of bound names that could be generated when their processes fall under a replication operator, for example, when $!(\nu y)P$. We achieve this distinction through a number-labelling variant of the $\alpha$-conversion mechanism, leading to successive copies being named $(\nu y_1)P$, $(\nu y_2)P$ etc.

Most of the rules for the $\mu \rightarrow$ relation are straightforward and their explanation can be found elsewhere (e.g. [8, §3.2.2] and [13]). In these, the labels, $\mu \in \{x(y), x(y), x(z), x(l), \tau\rightarrow\}$, express free, bound and no-message outputs, inputs with and without parameters and silent actions. Rules (15) and (16) define message passing, whilst Rule (17) defines no-message synchronisations.

### 3.2. The Open Charge Point Protocol

With the rising popularity of Electric Vehicles (EVs), recently industry moved on to define the Open Charge Point Protocol (OCPP) v.2.0.1. [2] standard for establishing a charging infrastructure for EVs. OCPP is maintained by the Open Charge Alliance (OCA) [3], a global consortium of public and private electric vehicle infrastructure leaders promoting open standards.
Rules of the $\equiv$ relation:

1. $(P/\equiv,|,0)$ is a commutative monoid
2. $(\nu x)0 \equiv 0$
3. $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
4. $!P \equiv P \not!P$
5. $(\nu x)(P \mid Q) \equiv (P \mid (\nu x)Q)$ if $x \notin \text{fn}(Q)$
6. $\text{timer}^0(x(z).P,Q) \equiv Q$
7. $\text{timer}^0(x().P,Q) \equiv Q$
8. $\text{timer}^\infty(x(z).P,Q) \equiv x(z).P$
9. $\text{timer}^\infty(x().P,Q) \equiv x().P$

Rules of the $\Rightarrow$ relation:

10. $\pi(y).P \pi(y) \Rightarrow P$
11. $\pi().P \pi() \Rightarrow P$
12. $\text{timer}^{t+1}(x(z).P,Q) \xrightarrow{x/z} P$
13. $\text{timer}^{t+1}(x().P,Q) \xrightarrow{x/()} P$
14. $P \pi(y) \Rightarrow (\nu y)P \pi(y) \Rightarrow Q$ if $x \neq y$
15. $P \pi(y)P',Q \xrightarrow{x(z)} Q' \Rightarrow P \mid Q \xrightarrow{\tau} P' \mid Q'[y/z]$
16. $P \pi(y)P',Q \xrightarrow{x(z)} Q' \Rightarrow P \mid Q \xrightarrow{\tau} (\nu y)(P' \mid Q'[y/z])$
17. $P \pi(),P',Q \xrightarrow{x/()} Q' \Rightarrow P \mid Q \xrightarrow{\tau} P' \mid Q'$
18. $P \xrightarrow{\mu} Q \Rightarrow (\nu x)P \xrightarrow{\mu} (\nu x)Q$ if $x \neq \text{fn}(\mu)$
19. $P \xrightarrow{\mu} P' \Rightarrow P \mid Q \xrightarrow{\mu} P' \mid Q$
20. $P \xrightarrow{\mu} P' \Rightarrow P + Q \xrightarrow{\mu} P'$
21. $P \xrightarrow{\mu} P' \Rightarrow Q + P \xrightarrow{\mu} P'$
22. $P \xrightarrow{\tau} \partial(P)$

Figure 1: The structural operational semantics of $TP_i$ [13].
for electric vehicles and their infrastructure. The architecture underlying the OCPP standard is illustrated in Figure 2.

![Figure 2: Architecture of OCPP](image)

This architecture consists of any number of Charging Stations, each referring to any physical system where an electric vehicle can charge itself. These stations are managed by a single Charging Station Management System (CSMS), which also has the information for authorising users for using the Charging Stations under its management. The Charging Station itself may have multiple units of Electric Vehicle Supply Equipment (EVSE), where each EVSE can have multiple Connectors, though only one Connector can be active at any one moment in time.

OCPP is not a single protocol, but itself consists of a number of sub-protocols, each implementing one of its use cases. In total, there are 119 such use cases. We focus in this paper on one specific critical sub-protocol, namely the Heartbeat Protocol, defined in Use Case G02 [2, p.185]. This sub-protocol will form the basis of our case study presented later in Section 5. Nonetheless, any other protocol that can be specified in TPi (including other OCPP sub-protocols) can also be subject to the same mutation function we define in the next Section.
4. A Mutation Framework for TP\textit{i}

We start the definition of our mutation framework by first defining \textit{tags} that can be used to tag channel and message names as well as time values, i.e. the elements that we will mutate later on. More formally, such tags are defined as the set $\mathcal{T}$ ranged over by elements $\ell, \ell'$ etc. We refer to the set of all \textit{tagged processes} as $\mathcal{P}^T$. A tagged process can be obtained from a normal process through the application of the tagging functions defined in Section 4.1, which will later be used when generating our mutants.

Formally, the syntax of a tagged TP\textit{i} process is defined as follows:

$$P, Q ::= \overline{x}(\bar{y}).P \mid \text{timer}^i(\bar{x}(y).P, Q) \mid !P \mid (\nu x)P \mid (P \mid Q) \mid (P + Q) \mid 0$$

where names, $\bar{x}, \bar{y}$, can either be non-tagged or tagged (again allowing for the case when no names are passed as messages). We call the set of all tagged names, $\mathcal{N}^T$. Similarly, time values, $\bar{t}$, can be either natural numbers and therefore elements of $\mathbb{N}$, or tagged natural numbers, therefore elements of the set $\mathbb{N}^T$. In all what follows, we assume that every tag is unique. This can always be achieved through $\alpha$-renaming of tags. Tagging has no effect on the structural operational semantics of a process.

4.1. Three Tagging Functions

As we mentioned earlier, our focus in this paper is to study the effects of three kinds of mutations: mutations of messages, communication channels and of the duration of input actions. We define three specific tagging functions to assist in generating mutants for each of these three cases:

$$\gamma_{\text{chn}} : \mathcal{P}^T \rightarrow \mathcal{P}^T \quad (1)$$

$$\gamma_{\text{msg}} : \mathcal{P}^T \rightarrow \mathcal{P}^T \quad (2)$$

$$\gamma_{\text{time}} : \mathcal{P}^T \rightarrow \mathcal{P}^T \quad (3)$$

As their names imply, $\gamma_{\text{chn}}$ tags channel names, $\gamma_{\text{msg}}$ tags message names and $\gamma_{\text{time}}$ tags the time duration of an input action. These will simply and generalise the definition of the mutation function, introduced later in Section 4.3. The syntax-directed rules defining these functions are shown in Figures 3, 4 and 5, respectively. The operator,

$$\text{tags} : \mathcal{P}^T \rightarrow \wp(\mathcal{T}) \quad (4)$$

collects the set of tags of a process, as follows:
In Figure 3, we tag the channel names of a process in a syntax-directed manner.

The environment, $\rho \subseteq \mathcal{T}$, holds the set of all the tags that the tagging function has so far “seen”. The rules can be described as follows. Rules ($\gamma_{chn1}$) and ($\gamma_{chn2}$) tag an untagged channel name for the non-empty and empty output actions, respectively, using a tag that has not yet been seen before by the tagging function and does not belong to the set of tags of the process itself. Rules ($\gamma_{chn3}$) and ($\gamma_{chn4}$) pass the tagging function to the body of a non-empty and empty output action, respectively, where the channel of the action is a tagged one, while recording that tag (as well any tag the output message may carry) in $\rho$. In rules ($\gamma_{chn5}$) and ($\gamma_{chn6}$), a non-empty and empty timed input, respectively, which has an untagged channel, is tagged. This is done again in a way that ensures that the chosen tag is not one of the existing ones in $\rho$ or in the timed input itself. In rules ($\gamma_{chn7}$) and ($\gamma_{chn8}$), a non-empty and empty timed input, respectively, using a tagged channel name, will have the tagging function passed to either of the two continuation processes of the action, or none. This will depend on whether we find other channel names that we can tag in either of the two processes, or none. If we did, then the difference between the newly tagged process and the original one will be a non-empty set. In rules ($\gamma_{chn9}$) and ($\gamma_{chn13}$), we use the same idea to tag either side of a parallel composition and a non-deterministic choice, or neither, if neither can be tagged any further.
Figure 3: Definition of the channel tagging function.
In rules \((\gamma_{\text{chn}}10)\) and \((\gamma_{\text{chn}}11)\), we pass the tagging function to the process inside the operator enclosing it. Finally, in rule \((\gamma_{\text{chn}}12)\), the channel tagging function has no effect on a null process.

By contrast, Figure 4 outlines the rules of the message-tagging function.

In these rules, Rule \((\gamma_{\text{msg}}1)\) tags an untagged message name in a non-empty output action, using a tag that has not yet been seen by the tagging function and does not belong to the set of tags of the process itself. Rule \((\gamma_{\text{msg}}2)\) passes the tagging function to the body of a non-empty output action with a tagged message, while recording that tag (as well any tag the channel name may carry) in \(\rho\). In rule \((\gamma_{\text{msg}}3)\), the same is done for the case of an empty output action, since such action would have no message to tag. In rules \((\gamma_{\text{msg}}4)\) and \((\gamma_{\text{msg}}5)\), non-empty and empty timed input actions naturally have no output messages, therefore, we pass the tagging function to either of the two continuation processes, or none, depending on whether we find other channel names that we can tag in either of these two continuation processes (or none). If we did, then the difference between the newly tagged process and the original one will be a non-empty set. In rules \((\gamma_{\text{msg}}6)\) and \((\gamma_{\text{msg}}10)\), we use the same idea to tag either side of a parallel composition and a non-deterministic choice, or none if neither have message names that can be tagged. In rules \((\gamma_{\text{msg}}7)\) and \((\gamma_{\text{msg}}8)\), we pass the message tagging function to the process inside the operator enclosing it. Finally, in rule \((\gamma_{\text{msg}}9)\), the message-tagging function has no effect on a null process.

Finally, Figure 5 shows the rules of the input time-tagging function.

The rules can be described as follows. Rules \((\gamma_{\text{time}}1)\) and \((\gamma_{\text{time}}2)\) do not tag anything in non-empty and empty output actions, respectively, and simply pass the tagging function to the body of the output action, while recording any tags the action may carry in \(\rho\). Rules \((\gamma_{\text{time}}3)\) and \((\gamma_{\text{time}}4)\) tag an untagged time value of a non-empty or empty input action, respectively, using a tag that has not been recorded in \(\rho\) so far and does not belong to the tags of the input process itself. In rules \((\gamma_{\text{time}}5)\) and \((\gamma_{\text{time}}6)\), we pass the tagging function to either of the two continuation processes or neither, for a non-empty or empty input action, respectively, where the time value has already been tagged. In rules \((\gamma_{\text{time}}7)\) and \((\gamma_{\text{time}}11)\), we use the same idea to tag either side of a parallel composition and a non-deterministic choice, or none if neither have time values that can be tagged. In rules \((\gamma_{\text{time}}8)\) and \((\gamma_{\text{time}}9)\), we pass the time tagging function to the process inside the operator enclosing it. Finally, in rule \((\gamma_{\text{time}}7)\), the time tagging function has no effect.
Figure 4: Definition of the message tagging function.
\begin{align}
(\gamma_{\text{time}}1) & \quad \gamma_{\text{time}}(\overline{x}(y).P)\rho = \overline{x}(y).\gamma_{\text{time}}(P)(\rho \cup \text{tags}(\overline{x}(y).0)) \\
(\gamma_{\text{time}}2) & \quad \gamma_{\text{time}}(\overline{x}().P)\rho = \overline{x}().\gamma_{\text{time}}(P)(\rho \cup \text{tags}(\overline{x}().0)) \\
(\gamma_{\text{time}}3) & \quad \gamma_{\text{time}}(\overline{\text{timer}}(\overline{x}(y).P, Q))\rho = \overline{\text{timer}}(\overline{x}(y).P, Q) \\
(\gamma_{\text{time}}4) & \quad \gamma_{\text{time}}(\overline{\text{timer}}(\overline{x}().P, Q))\rho = \overline{\text{timer}}(\overline{x}().P, Q) \\
(\gamma_{\text{time}}5) & \quad \gamma_{\text{time}}(\overline{\text{timer}}(\overline{x}(y).P, Q))\rho = \\
& \quad \left\{ \begin{array}{ll}
\overline{\text{timer}}(\overline{x}(y).\overline{x}(\gamma_{\text{time}}(P)\rho_\ell), Q) & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) \neq \{\} \\
\overline{\text{timer}}(\overline{x}(y).\gamma_{\text{time}}(P)\rho_\ell, \gamma_{\text{time}}(Q)\rho_\ell) & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{\} \land \\
\overline{\text{timer}}(\overline{x}(y).P, Q) & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{} \land \\
\end{array} \right.
\end{align}

\begin{align}
(\gamma_{\text{time}}6) & \quad \gamma_{\text{time}}(\overline{\text{timer}}(\overline{x}().P, Q))\rho = \\
& \quad \left\{ \begin{array}{ll}
\overline{\text{timer}}(\overline{x}().\gamma_{\text{time}}(P)\rho_\ell, Q) & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) \neq \{\} \\
\overline{\text{timer}}(\overline{x}().\gamma_{\text{time}}(P)\rho_\ell, \gamma_{\text{time}}(Q)\rho_\ell) & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{\} \land \\
\overline{\text{timer}}(\overline{x}().P, Q) & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{} \land \\
\end{array} \right.
\end{align}

\begin{align}
(\gamma_{\text{time}}7) & \quad \gamma_{\text{time}}(P | Q)\rho = \\
& \quad \left\{ \begin{array}{ll}
\gamma_{\text{time}}(P)\rho_\ell | Q & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) \neq \{\} \\
\gamma_{\text{time}}(P)\rho_\ell | \gamma_{\text{time}}(Q)\rho_\ell & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{\} \land \\
P | Q & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{} \land \\
\end{array} \right.
\end{align}

\begin{align}
(\gamma_{\text{time}}8) & \quad \gamma_{\text{time}}(\overline{\{P\}})\rho = \gamma_{\text{time}}(P)\rho \\
(\gamma_{\text{time}}9) & \quad \gamma_{\text{time}}(\overline{(a)P})\rho = (a)\gamma_{\text{time}}(P)\rho \\
(\gamma_{\text{time}}10) & \quad \gamma_{\text{time}}(0)\rho = 0 \\
(\gamma_{\text{time}}11) & \quad \gamma_{\text{time}}(P + Q)\rho = \\
& \quad \left\{ \begin{array}{ll}
\gamma_{\text{time}}(P)\rho_\ell + \gamma_{\text{time}}(Q)\rho_\ell & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) \neq \{\} \\
P + \gamma_{\text{time}}(Q)\rho_\ell & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{\} \land \\
P + Q & \text{if tags}(\gamma_{\text{time}}(P)\rho_\ell) \setminus \text{tags}(P) = \{} \land \\
\end{array} \right.
\end{align}

Figure 5: Definition of the time tagging function.
on a null process.

We can prove the termination of the tagging functions, as follows.

**Theorem 1** (Termination of the Tagging Functions).
The aforementioned $\gamma_{chn}$, $\gamma_{msg}$ and $\gamma_{time}$ tagging functions terminate.

**Proof.** This is possible to prove by showing that the size-change principle applies [35]. First, we define a size function on the number of terms that may need to be tagged:

$$
size(R) = \begin{cases} 
1 + size(P) & \text{if } R = \overline{x}(y).P \\
1 + size(P) + size(Q) & \text{if } R = \text{timer}^i(\overline{x}(y).P, Q) \\
size(P) & \text{if } R = !P \lor R = (\nu x)P \\
size(P) + size(Q) & \text{if } R = P \mid Q \lor R = P + Q \\
0 & \text{if } R = 0 
\end{cases}
$$

We can express any of the rules for $\gamma_{chn}$, $\gamma_{msg}$ and $\gamma_{time}$, in one of the following two generic forms:

$$
\gamma([R]\rho) = C[\gamma([P]\rho')] \\
\gamma([R]\rho) = P
$$

where $C[.]$ is the context of a process, defined itself as being a process with a hole. For the first form, it is possible to show by structural induction on the rules of $\gamma_{chn}$, $\gamma_{msg}$ and $\gamma_{time}$, that it is always the case that:

$$
size(P) < size(R)
$$

This proves that the size-change principle applies here. For the second form, the tagging function naturally comes to a termination point.

**Lemma 1** (Completeness).
Every relevant non-tagged term is tagged.

**Proof.** The proof of this lemma falls from the proof of Theorem 1 and by structural induction on the rules of the tagging functions, $\gamma_{chn}$, $\gamma_{msg}$ and $\gamma_{time}$, which show that no non-tagged terms are missed.
4.2. Application of the Tagging Functions

The application of the tagging functions depends on what combination of mutations we are seeking to generate and this will depend further on the scenario the mutation analysis is being used for. However, the most general case is that we would like to tag every channel name, message and time value. In order to do that, we first define the following function:

\[ \omega = \lambda f. \lambda \gamma. \lambda p. \lambda \rho. \begin{cases} \text{if } \gamma([p]) \rho = p \text{ then } p \text{ else } f \ (\gamma([p]) \rho) \ tags(\gamma([p]) \rho) \end{cases} \]  

(5)

then for a specific tagging function, \( \gamma \), we compute its fixed point using Curry’s fixed-point combinator [17, p.178]:

\[ Y \omega \gamma P \ tags(P) = \omega (Y \omega) \gamma P \ tags(P) \]  

(6)

If we name the above fixed point for the \( \gamma \) tagging function, \( \text{fix}_\gamma(P) \), then the full application of our three tagging functions to a process, \( P \), yielding our final process, \( P_{\text{final}} \), can be expressed as follows:

\[ P_{\text{final}} = \text{fix}_\text{time}(\text{fix}_\text{msg}(\text{fix}_\text{chn}(P))) \]

4.3. Definition of the General Mutation Function

We now define a general mutation function that takes a tagged process and a specific tag, and produces a mutant, which is another (possibly tagged) process:

\[ \mu : P^T \times T \times ((\mathcal{N} \cup \mathbb{N}) \rightarrow (\mathcal{N} \cup \mathbb{N})) \rightarrow P^T \]  

(7)

In these rules, the function \( f : (\mathcal{N} \cup \mathbb{N}) \rightarrow (\mathcal{N} \cup \mathbb{N}) \) is a polymorphic name- and time-changing function, defined as follows:

\[ f(r) : \begin{cases} g(r) & \text{if } r \in \mathcal{N} \\ h(r) & \text{if } r \in \mathbb{N} \end{cases} \]  

(8)

where \( g : \mathcal{N} \rightarrow \mathcal{N} \) is a name-changing function and \( h : \mathbb{N} \rightarrow \mathbb{N} \) is a time-changing function. The actual definitions of \( g \) and \( h \) depend on the specific scenario. For example, \( g \) could be defined as the most obvious function that selects randomly some name in \( \mathcal{N} \) not equal to the name it receives:

\[ g(x) = y, \text{ where } y \in \mathcal{N} \text{ and } y \neq x \]

The case of \( h \) is more interesting. This is because time, in the context of the timed input action, may have different effects. Take for example,
\begin{align*}
\mu([\bar{\epsilon}(\bar{\mu})].P)]\ell f &= \begin{cases} 
\bar{\epsilon}(\bar{\mu}).P & \text{if } \bar{\epsilon} = \bar{x}, \text{ where } z = f(x) \\
\bar{\epsilon}(\bar{\mu}).P & \text{if } \bar{\epsilon} = \bar{y}, \text{ where } z = f(y) \\
\bar{\epsilon}(\bar{\mu}).P & \text{otherwise}
\end{cases} \\
\mu([\bar{\epsilon}].P)]\ell f &= \begin{cases} 
\bar{\epsilon}.P & \text{if } \bar{\epsilon} = \bar{x}, \text{ where } z = f(x) \\
\bar{\epsilon}.P & \text{otherwise}
\end{cases} \\
\mu([\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q)]\ell f &= \begin{cases} 
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{if } \text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) = f(t), \text{ where } t' = f(t) \\
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{if } \bar{\epsilon} = \bar{x}, \text{ where } z = f(x) \\
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{if } P \neq \mu([P])\ell f \\
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{otherwise}
\end{cases} \\
\mu([\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q)]\ell f &= \begin{cases} 
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{if } \text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) = f(t), \text{ where } t' = f(t) \\
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{if } \bar{\epsilon} = \bar{x}, \text{ where } z = f(x) \\
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{if } P \neq \mu([P])\ell f \\
\text{timer}^{\ell f}(\bar{\epsilon}(\bar{\mu})].P, Q) & \text{otherwise}
\end{cases} \\
\mu(P | Q)]\ell f &= \begin{cases} 
\mu([P])\ell f \mid Q & \text{if } P \neq \mu([P])\ell f \\
P \mid \mu([Q])\ell f & \text{otherwise}
\end{cases} \\
\mu([P])\ell f &= \mu([P])\ell f \\
\mu([u_{\epsilon}]P)]\ell f &= (u_{\epsilon})\mu([P])\ell f \\
\mu([0)]\ell f &= 0 \\
\mu([P + Q)]\ell f &= \begin{cases} 
\mu([P])\ell f + Q & \text{if } P \neq \mu([P])\ell f \\
P + \mu([Q])\ell f & \text{otherwise}
\end{cases}
\end{align*}

Figure 6: Definition of the general mutation function.
\( \text{timer}^\infty(x(y).P,Q) \) means that the input action will always be available (and hence \( Q \) is practically non-reachable). On the other hand, \( \text{timer}^0(x(y).P,Q) \) means that the input action itself is bypassed and \( Q \) is the only practically available process. Therefore, the choice of how we mutate \( t \) is important. Here, we are interested in choices of \( h \) that cover three important points on the time line: the initial point \((t = 0)\), a random mid point \((0 < t < \infty)\) and the point of infinity \((t = \infty)\), since these three points affect the behaviour of input actions at the abstract level. Therefore, we consider the following three definitions of \( h \) (called \( h_{\rightarrow 0} \), \( h_{\rightarrow \infty} \) and \( h_{\rightarrow t} \)):

\[
\begin{align*}
 h_{\rightarrow 0}(t) &= 0 \\
 h_{\rightarrow \infty}(t) &= \infty \\
 h_{\rightarrow t}(t) &= \begin{cases} 
 t' & \text{if } t = 0 \lor t = \infty, \text{ where } 0 < t' < \infty \\
 t & \text{otherwise}
\end{cases}
\end{align*}
\]

\( h_{\rightarrow 0} \) changes every time value to the initial point in time. \( h_{\rightarrow \infty} \), on the other hand, changes every time value to the infinite point in time. And finally, \( h_{\rightarrow t} \), changes both the two extremes, to some middle point in time, or if the point is already somewhere in the middle, it returns the value unchanged. There are other definitions that \( h \) can be given, e.g. defining \( h \) as a piece-wise function. That will depend on the context of the system being analysed. However, in our case, we don’t think that our static analysis later benefits from such other definitions.

The set of all mutants that can be derived from a process, \( P \), is:

\[
\{ m \mid (m = \mu([P]f) \land (\ell \in \text{tags}(P)) \land (f \in F)) \}
\]

where \( F \) is the space of all name- and time-changing functions, \( f \), specifically adopted in the case of the mutation scenario. To ensure that the mutation of a process terminates, \(|F|\) must be finite.

5. Example: The OCPP Heartbeat Protocol

Use Case G02 [2, p.185] defines a heartbeat protocol between the CSMS and a Charging Station in the OCPP standard. Like any other heartbeat protocol, the OCPP heartbeat protocol has the purpose of ensuring that the Charging Station can inform the CSMS, every now and then, of the fact that it is still alive. The Charging Station does that by sending regular heartbeat
The informal description of the scenario, quoted from [2, p.185], is as follows:

1. “If there is no activity for a certain time, the Charging Station sends HeartbeatRequest for ensuring that the CSMS knows that a Charging Station is still alive.”

2. “Upon receipt of HeartbeatRequest, the CSMS responds with HeartbeatResponse. The response message contains the current time of the CSMS, which the Charging Station MAY use to synchronize its internal clock.”

The use case also states 7 requirements on the protocol implementation, which can be referred to in [2, p.185–186]. In order for us to study the mutation effects on the OCPP Heartbeat protocol, we give first a formal specification of the behaviour of a single Charging Station and its CSMS in TPi. This specification is shown in Figure 8.

At the top level, the protocol process, Heartbeat, is defined in terms of two sub-protocol processes, Charging Station and CSMS, composed in parallel with each other. In addition to these, there are two signals (in the form of output actions), also composed in parallel with the Charging Station and the CSMS processes. The first of these signals, ChargingStation(zerotime), bootstraps the Charging Station process, while at the same time passing it the initial time, which we refer to as zerotime. It is worth pointing out here that these time messages utilised in the protocol, are a different concept from our time values, used to define the behaviour of a timed input action. The second signal, CSMSc(), bootstraps the CSMS process. For clarity, we refer to these two signals from now onward as the top-level signals. Note that both
the ChargingStation and CSMSc channel names are restricted at this level, in order to ensure that no interference will occur across multiple heartbeat protocol instances running among multiple Charging Stations and a CSMS instances.

The Charging Station process itself is a replicated process, which allows for any number of times for its behaviour to be run. This behaviour is timed input action that waits forever to receive a time value. After receiving its initial time value from the top-level signal, waits to receive an external event from the CSMS process, not related to the Heartbeat protocol. This second input is timed with the value of the HeartbeatInterval variable, stated in the use case, and it is performed over a hypothetical channel we call event. If such activity event is received, the Charging Station will then start a new copy of itself, passing itself the current time message it had initially received. If, however, no such event is received, the actual heartbeat sequence of messages kicks in. This sequence of messages starts by sending a heartbeat request, and then waiting for $t$ amount of time to receive the heartbeat response from the CSMS process. If such response is received, it will carry with it the current time message, which is then used to restart a new copy of the Charging Station process. If, however, it is not received, the existing time message received from the above top-level signal is used to start the new copy of Charging Station. The specification of the Heartbeat protocol in [2, p.185] does not specify what the value of $t$ should be. In other words, the

\[
\text{Heartbeat} \overset{\text{def}}{=} (\nu \text{ChargingStation})(\nu \text{CSMSc})(\text{Charging Station} \mid \text{ChargingStation(zeroTime).0} \mid \text{CSMS} \mid \text{CSMSc}().0)
\]

\[
\text{Charging Station} \overset{\text{def}}{=} ! \text{timer}^\infty(\text{ChargingStation}(time).\
\text{timer}^{\text{HeartbeatInterval}}(\text{event}.\text{ChargingStation}(time).0),\
\text{HeartbeatRequest}().\text{timer}^t(\text{HeartbeatResponse}(xt).\text{ChargingStation}(xt).0,\text{ChargingStation}(time).0),0)
\]

\[
\text{CSMS} \overset{\text{def}}{=} ! \text{timer}^\infty(\text{CSMSc}.\text{timer}^\infty(\text{HeartbeatRequest}().\text{HeartbeatResponse}(\text{currentTime}.\text{CSMSc}().0,0),0)
\]

Figure 8: Formal Specification of the OCPP Heartbeat Protocol
document does not define the time limit that the Charging Station should wait for, between sending a heartbeat request and receiving the corresponding heartbeat response from CSMS.

The CSMS replicated process, on the other hand, has simpler behaviour. It will wait infinitely to be initiated by the top-level signal, then it will wait again infinitely for a heartbeat request signal to arrive from Charging Station. When such a signal is received, it will send the currentTime message back to the Charging Station, and restart a new copy of itself.

In all of the above definitions, whenever an alternative to an expired timed input is not specified, it is assumed to be the inactive process, 0. This is because the use case itself does not state such alternative behaviour, and so, we do not care about it for the purpose of our analysis. We next present the resulting mutants that were generated based on our three tagging functions. In total, there are 28 mutants.

5.1. Charging Station Mutants

The first set of mutants were generated for the Charging Station process. There are 16 such mutants, resulting from alterations to channel names (mutants m1–m7), output messages (mutants m8–m10) and input time values (mutants m11–m16). The definitions of all of these mutants are shown in Figure 9.

5.2. CSMS Mutants

The second group of mutants are the CSMS mutants. There are 9 such mutants, resulting from alterations to channel names (mutants m17–m20), output message names (mutant m21) and input time values (mutants m22–m25). These mutants are shown in Figure 10.

5.3. Protocol-Level Mutants

The final group of mutations were generated at the protocol level itself. There are 3 mutants at this level, covering changes to channel names (mutants (m26–m27)) and output messages (mutant m28). These mutants are shown in Figure 11. There are no time value mutations as there are no input actions at the protocol’s level.
Figure 9: Charging Station Mutants

\[
\begin{align*}
\text{Charging Station}_1 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} (\text{ChargingStation}'(t), \\
& \text{timer} \text{HeartbeatInterval} \text{(event)} \text{. ChargingStation}(t). \text{0}, \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_2 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_3 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_4 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_5 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_6 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_7 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_8 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0}) \\
\text{Charging Station}_9 & \overset{\text{def}}{=} \text{def}\ ! \text{timer}^{\infty} \text{HeartbeatRequest}(\text{event}' \text{. ChargingStation}(t). \text{0}), \text{ChargingStation}(\text{event}' \text{. ChargingStation}(t). \text{0})
\end{align*}
\]
<table>
<thead>
<tr>
<th>Charging Station</th>
<th>$m_{10}$</th>
<th>$m_{11}$</th>
<th>$m_{12}$</th>
<th>$m_{13}$</th>
<th>$m_{14}$</th>
<th>$m_{15}$</th>
<th>$m_{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Charging Station}<em>{m</em>{10}}$</td>
<td>$! \text{timer}^\infty(\text{ChargingStation}(\text{time})).$</td>
<td>$\text{timer}_{\text{HeartbeatInterval}}(\text{event}).\text{ChargingStation}(\text{time}).0,$</td>
<td>$\text{HeartbeatRequest}().\text{timer}^\infty(\text{HeartbeatResponse}(\text{xt}).\text{ChargingStation}(\text{xt}).0,\text{ChargingStation}(\text{time}).0),0)$</td>
<td>$! \text{timer}^4(\text{ChargingStation}(\text{time})).$</td>
<td>$\text{timer}_{\text{HeartbeatInterval}}(\text{event}).\text{ChargingStation}(\text{time}).0,$</td>
<td>$\text{HeartbeatRequest}().\text{timer}^4(\text{HeartbeatResponse}(\text{xt}).\text{ChargingStation}(\text{xt}).0,\text{ChargingStation}(\text{time}).0),0)$</td>
<td>$! \text{timer}^6(\text{ChargingStation}(\text{time})).$</td>
</tr>
</tbody>
</table>

Figure 9: Charging Station Mutants (cont.)
$CSMS_{m17} \overset{def}{=} \text{! timer}^\infty(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(0,0), 0)$

$CSMS_{m18} \overset{def}{=} \text{! timer}^\infty(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(0,0), 0)$

$CSMS_{m19} \overset{def}{=} \text{! timer}^\infty(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(), 0, 0), 0)$

$CSMS_{m20} \overset{def}{=} \text{! timer}^\infty(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(0,0), 0)$

$CSMS_{m21} \overset{def}{=} \text{! timer}^\infty(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime}')(\text{CSMSc}(0,0), 0)$

$CSMS_{m22} \overset{def}{=} \text{! timer}'(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(0,0), 0)$

$CSMS_{m23} \overset{def}{=} \text{! timer}'(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(0,0), 0)$

$CSMS_{m24} \overset{def}{=} \text{! timer}'(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(0,0), 0)$

$CSMS_{m25} \overset{def}{=} \text{! timer}'(\text{CSMSc}().\text{timer}^\infty(\text{HeartbeatRequest}()))$

$HeartbeatResponse(\text{currentTime})(\text{CSMSc}(0,0), 0)$

Figure 10: CSMS Mutants
6. Analysing the Mutants

We recall that in [8, 11], we defined a non-standard name-substitution semantics for the π-calculus, which when abstracted using an approximation function, was capable of yielding an abstract environment:

\[ \phi : \mathcal{N}^\sharp \rightarrow \phi(\mathcal{N}^\sharp) \in D^\sharp_\bot \]

where \( \mathcal{N}^\sharp \) represents the set of abstract names. Unlike \( \mathcal{N} \), \( \mathcal{N}^\sharp \) restricts the number of copies of bound names that can be created under replication up to some finite number. As a result, the set of names of a process, \( n(P) \subset \mathcal{N}^\sharp \), will always remain finite, as well as the power-set \( \phi(n(P)) \). The resulting semantic domain, \( D^\sharp_\bot \), guarantees termination for an abstract interpretation computed over it (with applications e.g. such as in [12]). The bottom element of \( D^\sharp_\bot \), \( \bot = \phi_0 \), is the empty environment where \( \forall x \in \mathcal{N}^\sharp : \phi_0(x) = \{ \} \). \( \mathcal{N}^\sharp \) and consequently \( D^\sharp_\bot \) can be constructed using some approximation function that keeps the size of \( \mathcal{N}^\sharp \) finite by limiting the number of copies of input parameter names and freshly created names. The static analysis of a process, \( P \), can then be obtained through the application of a suitable abstract interpretation function, \( \mathcal{A}([P]) \phi_0 \in D^\sharp_\bot \). The rules of such an abstract interpretation function were defined in [9]. When analysing the specification of a process, we use the resulting \( \phi \) environment to formalise the definitions of the properties we are interested in.
As a simple example, consider the following process:

\[ P \overset{\text{def}}{=} (! \text{timer}^\omega(x(y),0,0)) \mid (! (\nu z)\tau\langle z \rangle,0) \]

where we could apply an approximation that limits the number of copies of the input parameter \( y \) and the freshly generated message \( z \), to a maximum of two such copies. Then we would obtain the following abstract environment, \( A(P) \phi_0 = \phi_{\text{example}} \), where:

\[ \phi_{\text{example}}[y_1 \mapsto \{z_1, z_2\}, y_2 \mapsto \{z_1, z_2\}] \]

We keep sub-environments resulting from choices of process executions explicitly separate for clarity. For example, consider the following process:

\[ Q \overset{\text{def}}{=} \text{timer}^\omega(x(y),0,0) \mid (\tau\langle z \rangle,0 + \tau\langle u \rangle,0) \]

When analysed, \( A(Q) \phi_0 = \phi_{\text{example}2} \), this would result in the following two sub-environments:

\[ \phi_{\text{example}2} = \phi'_{\text{example}2}[y_1 \mapsto \{z\}] \cup \phi''_{\text{example}2}[y_1 \mapsto \{u\}] \]

which we prefer to keep separately, instead of writing:

\[ \phi_{\text{example}2}[y_1 \mapsto \{z, u\}] \]

for reasons related to clarity of presentation of results.

### 6.1. Analysis of the Heartbeat Protocol

To understand later the impact mutations will have on the protocol, we start here by analysing the Heartbeat protocol itself without any mutations applied. Therefore, we perform the analysis, \( A([\text{Heartbeat}]\phi_0 = \phi_{\text{normal}} \), where \( \phi_{\text{normal}} = \phi' \cup \phi'' \), and where:

\[
\phi'[time_1 \mapsto \text{zeroTime}, time_2 \mapsto \text{time_1}, time_2 \mapsto \text{time_2}]
\]

\[
\phi''[time_1 \mapsto \text{zeroTime}, \text{xt}_1 \mapsto \text{currentTime}, time_2 \mapsto xt_1, xt_2 \mapsto \text{currentTime},
\text{time}_2 \mapsto xt_2]
\]

We next apply the analysis to the protocol mutants, comparing to the above normal case.
<table>
<thead>
<tr>
<th>Mutant</th>
<th>Analysis Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heartbeat&lt;sub&gt;m1&lt;/sub&gt;</td>
<td>φ&lt;sub&gt;0&lt;/sub&gt;</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m2&lt;/sub&gt;</td>
<td>φ[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m3&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m4&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m5&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;2&lt;/sub&gt;] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → currentTime]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m6&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;2&lt;/sub&gt;] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, xt&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;1&lt;/sub&gt;,</td>
</tr>
<tr>
<td></td>
<td>time&lt;sub&gt;2&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m7&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;2&lt;/sub&gt;] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;1&lt;/sub&gt;, xt&lt;sub&gt;2&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m8&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time′] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;1&lt;/sub&gt;,</td>
</tr>
<tr>
<td></td>
<td>time&lt;sub&gt;2&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m9&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;2&lt;/sub&gt;] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;1&lt;/sub&gt;,</td>
</tr>
<tr>
<td></td>
<td>time&lt;sub&gt;2&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m10&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;2&lt;/sub&gt;] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;1&lt;/sub&gt;, xt&lt;sub&gt;2&lt;/sub&gt; → currentTime]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m11&lt;/sub&gt;</td>
<td>φ = φ′ ∪ φ″</td>
</tr>
<tr>
<td></td>
<td>where, φ′[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;1&lt;/sub&gt;, time&lt;sub&gt;2&lt;/sub&gt; → time&lt;sub&gt;2&lt;/sub&gt;] and</td>
</tr>
<tr>
<td></td>
<td>φ″[time&lt;sub&gt;1&lt;/sub&gt; → zeroTime, xt&lt;sub&gt;1&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;1&lt;/sub&gt;,</td>
</tr>
<tr>
<td></td>
<td>time&lt;sub&gt;2&lt;/sub&gt; → currentTime, time&lt;sub&gt;2&lt;/sub&gt; → xt&lt;sub&gt;2&lt;/sub&gt;]</td>
</tr>
<tr>
<td>Heartbeat&lt;sub&gt;m12&lt;/sub&gt;</td>
<td>φ&lt;sub&gt;0&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Table 1: Results of the Static Analysis of the Heartbeat Protocol
<table>
<thead>
<tr>
<th>Mutant</th>
<th>Analysis Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Heartbeat_{m13}$</td>
<td>$\phi'[time_1 \mapsto zeroTime, xt_1 \mapsto currentTime, time_2 \mapsto xt_1, \newline \quad \quad xt_2 \mapsto currentTime, time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m14}$</td>
<td>$\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m15}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime, time_2 \mapsto xt_1, \newline \quad \quad xt_2 \mapsto currentTime, time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m16}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, xt_1 \mapsto currentTime, \newline \quad \quad time_2 \mapsto time_1, \newline \quad \quad xt_2 \mapsto currentTime, time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m17}$</td>
<td>$\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m18}$</td>
<td>$\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m19}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime', time_2 \mapsto xt_1, \newline \quad \quad xt_2 \mapsto currentTime', time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m20}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime', \newline \quad \quad time_2 \mapsto time_1, \newline \quad \quad xt_2 \mapsto currentTime', \newline \quad \quad time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m21}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime, time_2 \mapsto xt_1, \newline \quad \quad xt_2 \mapsto currentTime, time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m22}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime, \newline \quad \quad time_2 \mapsto time_1, \newline \quad \quad xt_2 \mapsto currentTime, \newline \quad \quad time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m23}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime, \newline \quad \quad time_2 \mapsto time_1, \newline \quad \quad xt_2 \mapsto currentTime, \newline \quad \quad time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m24}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime, \newline \quad \quad time_2 \mapsto time_1, \newline \quad \quad xt_2 \mapsto currentTime, \newline \quad \quad time_2 \mapsto xt_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m25}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime, time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime, \newline \quad \quad xt_1 \mapsto currentTime, \newline \quad \quad time_2 \mapsto time_1, \newline \quad \quad xt_2 \mapsto currentTime, \newline \quad \quad time_2 \mapsto time_2]$</td>
</tr>
<tr>
<td>$Heartbeat_{m26}$</td>
<td>$\phi = \phi' \cup \phi''$ where, $\phi'[time_1 \mapsto zeroTime', time_2 \mapsto time_1, time_2 \mapsto time_2]$ and $\phi''[time_1 \mapsto zeroTime', \newline \quad \quad xt_1 \mapsto currentTime, \newline \quad \quad time_2 \mapsto time_1, \newline \quad \quad xt_2 \mapsto currentTime, \newline \quad \quad time_2 \mapsto xt_2]$</td>
</tr>
</tbody>
</table>

Table 1: Results of the Static Analysis of the Heartbeat Protocol (cont.)
6.2. Analysis of the Heartbeat Protocol Mutants

We analysed each case of the Heartbeat protocol mutants (m1–m28) under an approximation that places a bound on the number of copies of input parameters and freshly generated names. We consider a bound of two such copies only. Results of the analysis are shown in Table 1.

Each case represents one change, in the Charging Station process, the CSMS process or in the protocol process itself. We have analysed each run of a mutant, Heartbeat$_{ni}$ (for $i = 1 \ldots 28$), in parallel with two events emitted from the CSMS:

\[ A([Heartbeat_{ni} \mid \overline{event}().0 \mid \overline{event}().0])\phi_0 \]

where these events (referred to as CSMS activity in the informal description of the scenario in [2, p.185]) allow the Charging Station to follow the normal case where no heartbeat is required. Our choice of two events only coincides with the approximation bound on the number of copies mentioned above, as more events would have generated indistinguishable copies. We also limit our choice of the name- and time-changing function space to the following:

\[ F = \{(\lambda g,h.f) \ g \rightarrow h_0, (\lambda g,h.f) \ g \rightarrow \infty, (\lambda g,h.f) \ g \rightarrow t\} \]

The results of the analysis reveal the following classes of indistinguishable mutants, shown in Table 2. Note that our analysis is purely testing the effects of mutations in an otherwise non-malicious set-up. In other words, we do not consider the presence of any adversaries. Such an adversarial environment would require a different, more detailed, analysis.

We now discuss each of these mutation classes, at an informal level.

**Normal Class.** This case represents all the mutants that have analysis results similar to results obtained in the normal (un-mutated) case, and therefore, they are indistinguishable from the normal specification. In other words, $\phi = \phi_{normal}$. These mutants are harmless, in the context of our analysis, as they do not affect the behaviour of the system.

**Inactive Class.** This class represents all the cases where the mutation caused complete stalling of the protocol. As a result, $\phi = \phi_0$. These mutants cause “deadly” failures, and hence, they are easy to spot.
No-heartbeat Class. This class represents all the mutants that exhibit only behaviour corresponding to responses to the CSMS external events, over `event()`, but that are never able to enter the heartbeat stage if no such external events are received. These mutants represent partial behaviour according to what the standard protocol requires.

Different Update Time. This mutant represents the case where the updating of the time of the Charging Station, at the end of the heartbeat stage, is done using a different value than the current time at the CSMS. This mutant is an example of harmful mutations, since a wrong current time value can have further effects on the overall OCPP protocol. For example, it can influence the time validity of the “ChargingProfile” datatype used by the CSMS to influence, through the Charging Station, the power or current drawn by an EV (see Use Case K01 [2, p.233].)

Different Initial Time. This mutant represents the case where the Charging Station is initialised with a time value different from the normal initial time, when it bootstraps (see Use Case BO1 [2, p.43].). Again, if the Charging station is used by an EV prior to its first heartbeat communication with the CSMS, this could cause negative impact on the charging profile, as stated for the mutant above.

<table>
<thead>
<tr>
<th>Class</th>
<th>Indistinguishability Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Case</td>
<td>{Heartbeat\textsubscript{m7}, Heartbeat\textsubscript{m9}, Heartbeat\textsubscript{m11}, Heartbeat\textsubscript{m16}, Heartbeat\textsubscript{m22}, Heartbeat\textsubscript{m24}}</td>
</tr>
<tr>
<td>Inactive</td>
<td>{Heartbeat\textsubscript{m1}, Heartbeat\textsubscript{m12}, Heartbeat\textsubscript{m26}}</td>
</tr>
<tr>
<td>No-heartbeat</td>
<td>{Heartbeat\textsubscript{m4}, Heartbeat\textsubscript{m14}, Heartbeat\textsubscript{m17}, Heartbeat\textsubscript{m18}, Heartbeat\textsubscript{m19}, Heartbeat\textsubscript{m23}, Heartbeat\textsubscript{m25}, Heartbeat\textsubscript{m27}}</td>
</tr>
<tr>
<td>Different Update Time</td>
<td>{Heartbeat\textsubscript{m21}}</td>
</tr>
<tr>
<td>Different Initial Time</td>
<td>{Heartbeat\textsubscript{m28}}</td>
</tr>
<tr>
<td>Class 1</td>
<td>{Heartbeat\textsubscript{m5}, Heartbeat\textsubscript{m15}}</td>
</tr>
<tr>
<td>Class 2</td>
<td>{Heartbeat\textsubscript{m2}, Heartbeat\textsubscript{m3}, Heartbeat\textsubscript{m13}}</td>
</tr>
<tr>
<td>Class 3</td>
<td>{Heartbeat\textsubscript{m6}}</td>
</tr>
<tr>
<td>Class 4</td>
<td>{Heartbeat\textsubscript{m8}}</td>
</tr>
<tr>
<td>Class 5</td>
<td>{Heartbeat\textsubscript{m10}}</td>
</tr>
<tr>
<td>Class 6</td>
<td>{Heartbeat\textsubscript{m20}}</td>
</tr>
</tbody>
</table>
Classes 1–6. These classes of mutants represent various effects that have no high-level interesting impact on the system, other than changing its operational behaviour. Therefore, we do not discuss them any further.

It is worth noting here that the above eleven classes we identified are specific to the type of analysis we discussed at the beginning of this section, i.e. the name-substitution analysis. Our method is more general in that any other type of analysis can equally be applied leading to a (possibly) different classification of mutants with different effect and impact on the protocol’s behaviour and its properties. One of the interesting future ideas would be to apply data-driven (quantitative) analysis methods to understand if changes in system behaviour lead to changes measurable changes to the environment within which the system is operating. For example, how would changes in numbers of messages or changes in time ranges affect performance.

7. Conclusion

We defined in this paper a formal mutation framework that can be used to introduce mutants of systems and protocols specified using a variation of the $\pi$-calculus. The framework uses a single general mutation function, which we used to perform mutations on messages, names of communication channels and duration of input actions. We applied these mutations to a case study of an electric vehicle charging protocol, called OCPP. Results of the analysis revealed eleven classes of mutants, some were indistinguishable from the normal un-mutated case, some had no impact on the system and some had interesting implications for the safety and security of the protocol.

Future research will focus on combining the mutation function $\mu$ with other types of analysis, $A$, which captures other behaviour different from the name substitutions, and hence, we can observe other effect of mutations created by $\mu$. Another interesting area would be to extend the definition of the mutation function itself to include other mutations of process specifications. More specifically, these could include changes to the structure of the process itself, for example, in the manner which sub-processes are combined whether in parallel or in a non-deterministic choice. There is also, of course, the prospect of applying the current mutation and analysis functions to a different process algebra, or indeed, a different formal language altogether, if one wanted to cover some other class of systems. Finally, although this paper did not consider any models of adversaries since the main concern was to demonstrate only the effects of mutations on systems, however, an immediate
line of research more concerned with the security and safety of systems would be to consider adding the adversary to the analysis of the OCPP protocol (or for that matter, any other protocol under similar study). Such an adversary (e.g. Dolev-Yao’s most powerful adversary [21]) can easily be modelled as a process running in parallel with the specific mutant being analysed.

References

https://www.openchargealliance.org/protocols/ocpp-201/.

https://www.openchargealliance.org/protocols/ocpp-201/.

https://www.openchargealliance.org/.

http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.


