*standards*

MDPI

*Article*

# Formal Language for Objects' Transactions

**Mo Adda**

School of Computing, University of Portsmouth, Lion Terrace, Portsmouth PO1 3HE, UK; mo.adda@port.ac.uk

**Abstract:** The gap between software design and implementation often results in a lack of clarity and precision. Formal languages, based on mathematical rules, logic, and symbols, are invaluable for specifying and verifying system designs. Various semi-formal and formal languages, such as JSON, XML, predicate logic, and regular expressions, along with formal models like Turing machines, serve specific domains. This paper introduces a new specification formal language, ObTFL (Object Transaction Formal Language), developed for general-purpose distributed systems, such as specifying the interactions between servers and IoT devices and their security protocols. The paper details the syntax and semantics of ObTFL and presents three real case studies—federated learning, blockchain for crypto and bitcoin networks, and the industrial PCB board with machine synchronization—to demonstrate its versatility and effectiveness in formally specifying the interactions and behaviors of distributed systems.

**Keywords:** formal language; activities; interactions; actors; agents; transactions; compartments; junctions; containers

## 1. Introduction

Today's software is accompanied by extensive documentation, yet many systems still fail to meet their requirements due to overlooked detailed specifications, resulting in inadequate designs and implementations. The adoption of formal methods aims to address these issues by employing symbolic techniques rooted in elementary mathematics and logic. These techniques use rules to specify, define, and sometimes prove the correctness of complex systems and their specifications. A formal language, such as the one introduced in this paper, is a crucial component of formal methods. It relies on symbolic notations, syntax, and precise rules to construct formal expressions, statements, and semantics. The use of formal language, while very general to diverse problems, is particularly important in autonomous systems development, notably in security protocols, networks' protocols, cloud systems, and industrial machineries, as it enhances efficiency, quality, and reliability by imposing precision, unambiguity, rigor, and correctness.

Currently, there are several semi-formal languages available in both the market and research domains, tailored to specific system requirements and domains. JSON [1], although widely used, is not considered a completely formal language but rather a lightweight data interchange format over the web. Its simplicity and effectiveness make it suitable for many applications, but it may not be well suited for complex data structures and formal validations. XML [2], on the other hand, offers flexibility and support for validation and transformation but suffers from verbose syntax and complex structure, making documents larger and harder to read. It also presents security concerns such as injections and denial of service attacks.

Predicate logic [3] expresses complete relationships and is often used in formal validations, but it may lack expressive power, efficient reasoning, and consistency in some systems. Regular expressions [4], while powerful in manipulating and validating strings in texts, can be complex, error-prone, and limited in expressiveness. Turing machines [5], while not formal languages themselves, are models of computation used to study the properties of formal languages. However, they can be challenging to work with, have restricted



**Citation:** Adda, M. Formal Language for Objects' Transactions. *Standards* **2024**, *4*, 133–153. https://doi.org/10.3390/standards4030008

Academic Editor: Ahmed Patel

Received: 7 June 2024
Revised: 29 July 2024
Accepted: 9 August 2024
Published: 15 August 2024

problem-solving capabilities, and do not account for real-life constraints in terms of resources and time.

There are many related works in formal languages, static analysis techniques, and formal tools available for modeling and analyzing concurrent and distributed systems. Process algebra [6], for instance, provides a formal mathematical framework for reasoning about the correctness and properties of concurrent systems. Examples of process algebra include CSP (Communicating Sequential Processes) [7], ACP (Algebra of Communicating Processes) [8], CCS (Calculus of Communicating Processes) [9], and ATP (Algebra of Timed Processes) [10]. These languages allow for the composition of larger systems from smaller ones while abstracting away implementation details by exposing behavior at a higher level. However, process algebra may encounter difficulties when modeling certain non-deterministic real-world problems and can be computationally expensive.

Pi-calculus is another formal mathematical model used to represent and analyze concurrent process interactions in distributed systems and networks, including security and privacy analysis [11]. While powerful, $\pi$-calculus can be complex, especially when modeling larger systems with multiple interaction patterns, and it may lack expressiveness in certain scenarios. Spi-calculus, a variant of $\pi$-calculus, extends the model with constructs for modeling cryptographic and security protocols. However, the additional primitives for security and cryptography may limit the expressiveness of the model [12]. Dpi-calculus, an extension of $\pi$-calculus, introduces network primitives to model processes running on different machines [13]. While it offers support for communication primitives and synchronizations, it may incur performance overhead compared to other formalisms due to the need for additional network support.

Numerous software tools are available on the market to verify and support formal languages. FDR (Failure Detection and Recovery) is one such tool used for the formal verification of distributed systems [14]. It supports several formal languages, including CSP [7] and specification languages like B. However, FDR has its limitations, such as false positives, a reduced number of supported languages, and scalability issues when analyzing large systems, leading to state explosion. Other analysis tools for security protocols include Tamarin [15], Scyther [16], ProVerif [17], and AVISPA [18], each with its own approach and differences. These tools take a formal specification of a security protocol, verify its properties, and check for vulnerabilities; Scyther, for example, represents the model using a mathematical language and utilises automated reasoning mechanisms for analysis. However, it requires familiarity with formal methods and languages and is limited to security protocols. Additionally, scalability issues may arise when dealing with larger systems.

This paper has undertaken a review of various formal tools and related works grounded in process algebra, aligning with the formal language introduced in this study [19]. ObTFL presents a straightforward syntax and was crafted with the aim of being accessible to analysts with minimal knowledge of formal methods. It boasts flexibility, and its specifications can be visualized using a modified sequence diagram, which is known to most users. Compared to the other mentioned formal languages, ObTFL offers greater expressiveness or ease of use, with its activity and interaction syntaxes, and shows flexibility in integrating with various programming languages, like Python and C++. Like others, to address the specifications, validations, and requirements of interactive systems, this paper elucidates the principal syntax and semantics, providing three case studies: federated learning, the blockchain for crypto and bitcoin networks, and the IPC-HERMES-9852 standard for machine interactions. It concludes with the comparison of the latter case study with $\pi$-calculus notations.

The concept of ObTFL revolves around interacting objects directly as agents or indirectly through a communication medium as actors [20]. These objects encompass a broad spectrum, including IoT devices, robots, vehicles, drones, sensors, and more. In essence, this formal language serves as a conduit between requirements, specifications, and the design and implementation of interactive distributed systems. The full syntax and semantics of the language, specified in BNF, are outlined in [19].

## 2. Materials and Methods

### *2.1. Concept of the Language: Objects and Actions*

This section outlines the formal template of the language and introduces various elements that contribute to its structure and syntax. The language is grounded in the concept of autonomous objects, specifically as requestors interacting with recipients as receptors to achieve a common goal within a transaction. The purpose is achieved through negotiation, with each entity performing a set of actions and interactions with one another. Requestors and receptors can utilize global communication mediums such as the internet, satellite, data networks, or interfaces based on vision, sound, or touch, as shown in Figure 1. There are three layers for the protocol. Layer 1, the top one, is the application layer where all activities of the objects take place; layer 2 is the identification layer, similar to IP addresses, and is most of the time implicitly defined by the object identifier or indexes; and finally, layer 3 is the communication protocols, which vary from one medium to another, like TCP/UDP, Bluetooth, infrared, etc. Layers 2 and 3 are not required if the objects perform no interactions within their environments; this is like executing local actions. Objects interact or communicate with each other using containers' passing, implemented via local storage primitives, or containers' sharing where an outside storage is instigated with its store and retrieve primitives, supported by the protocols of an interconnection module, like a network card NC. On the other hand, they may communicate by touches, visions, and sounds directly implemented within an interconnection module, IM, acting as an interface. Explicitly, the problem specifications and syntax of ObTFL concentrate on layer 1, activities and containers, while the implementations of layers 2 and 3 are abstracted using labeling for the IP addresses and interaction symbols, respectively.

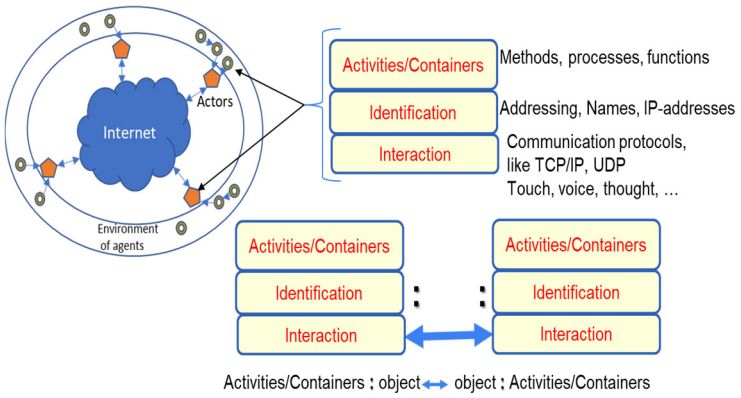


**Figure 1.** The environment of objects, containers, activities, and interactions.

Objects, similar to the concept in object-oriented programming, are entities that perform actions, known as methods or functions in programming languages. Broadly, we classify objects as either agents or actors.

Actors are objects that use communication mediums, such as the internet, satellites, infrared, and similar technologies, to interact with each other. Examples include terminals, servers, switches, routers, IoT devices, robots, vehicles, drones, sensors, and satellites.

Agents are objects that communicate through touch, vision, voice, or indirectly via actors. Examples of agents include humans, intruders, hackers, users, subscribers, vending machines, and software. For instance, a vending machine and a person interact via touch, involving buttons and coins or payment cards. Activities such as selecting an item from the vending machine and receiving it after successful payment are part of the interaction between the person and the machine.

Agents can also communicate remotely with other agents or actors via actors (proxies), as illustrated in Figure 1. For instance, an intruder (agent) may use a terminal (actor) to access a server (actor) and manipulate its software to commit a felony. Agents can use interfaces such as a keyboard or mouse to instruct a terminal to perform activities and interact with a server. For example, a TV remote control, acting as an actor, communicates

with a smart TV using local infrared communication, while the person pressing keys on the remote-control acts as the agent. In this context, agents effectively write scripts for actors to execute. Software or code can also function as agents, instructing actors on how to behave or perform within a scene. This interaction occurs through method calls to objects in a typical software environment. The code (agent) may reside within the actor as a lodger or be permanently embedded. For example, a virus (agent) injected into a mobile phone or server becomes a lodger, directing the actor—mobile or server—on how to behave, similar to the script of a movie.

Special types of objects can behave as agents and/or actors in their environment. Robots, for instance, as intelligent autonomous objects, communicate with other robots (actors) via communication media and interact with other robots (agents) through interfaces such as vision, sound, or private networks. Therefore, they can be both actors and agents simultaneously, depending on their roles within their environment. Sensors, as agents, provide information about their environments to the actors, while events (agents) disrupt the environment or systems, including software or hardware failures, malware attacks, or natural phenomena like earthquakes or volcanoes.

Identifying agents interacting with actors can be challenging, especially from a forensic investigation perspective. This is particularly true when actors like a mobile phone may have multiple users as agents, such as genuine users and potential villains. In the domain of digital forensics, investigations involving IoT devices as actors require identifying not only the device used but also all associated agents, or the users of the IoT device. Overall, the diverse range of objects and interactions underscores the complexity of interactive systems and the importance of understanding their components for effective analysis and management. Therefore, the classification of objects into actors and agents is crucial in the domain of crime investigations and the identification of objects.

### 2.2. Transactions

The generic format of ObTFL consists of containers and transactions, as depicted in Figure 2. Although transactions must be used all the time, not all problems require containers. The syntax and the grammar rules for the formal language, ObTFL, are fully defined with BNF in the technical report [19]. For clarity, Appendix A compiles some of the symbols used in this paper.

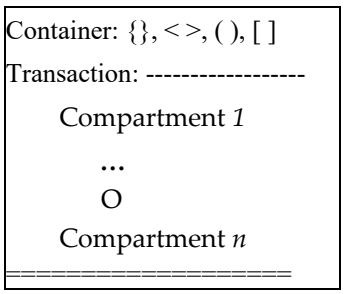


**Figure 2.** The structure of a transition with several compartments of common objectives.

Transactions are represented by one or more compartments which are abstractions of coherent activities with a common goal to achieve a specified objective with zero or more interactions between objects. This grammar indicates a transaction is composed of one or more compartments joined by the compartment operators, O = ⏸, ⊛, ⊗, ⊕. Compartments can run simultaneously, ⏸, consecutively, ⊛, or alternatively, ⊗, or in grouped alternatives, ⊕, with each other. By default, compartments with no operators are executed consecutively.

### 2.3. Compartments

A compartment, as depicted in Figure 3, serves as an abstraction representing a portion of the objectives within a transaction. It comprises one activity and a single object as an

agent, or two objects: a requestor and a receptor as agents and/or actors each with its own autonomous activities, with interactions. Each activity can adopt the form of fractional, linear, junction, or a combination. Fractional activities, shown in Figure 3, consist of numerators and denominators, which delineate blocks of actions. An activity may manifest as a single action, with or without interaction, or a group of actions interconnected by action operators, if they are internal to objects. The internal action operators, $o$, are "*", "$\|$", "$\|^{\circ}$", "$\times$", "+", "*+", "$\|^{+}$", and "$\|^{\circ+}$" and encompass sequential, parallel, rendezvous, selective exclusion, selective inclusion, consecutive, and a combination with selective inclusion "+" with sequential, parallel, and rendezvous operators.

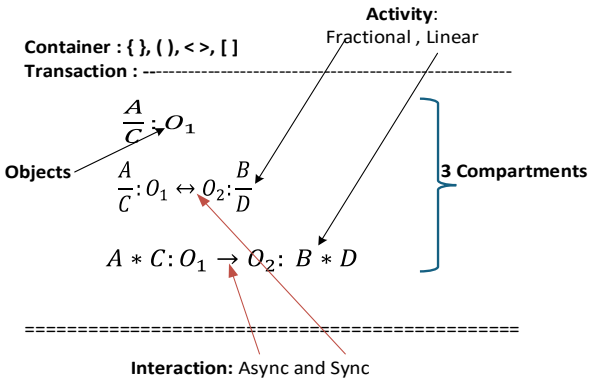


**Figure 3.** The content of a compartment.

Each interaction within a compartment signifies a communication between a requestor object and a receptor object, facilitated by action operators. There are three distinct types of interactions:

1. Lose Interaction (Asynchronous) ($\rightarrow$): In this scenario, the sender does not anticipate a response from the recipient.
2. Strict Interaction (Synchronous) ($\leftrightarrow$): Here, a reply is expected from the recipient. If no response is received, a deadlock situation might occur. Fortunately, this can be mitigated by a special primitive mechanism based on timeout settings, to be seen later.
3. Delayed Interaction ($\rightharpoonup$, $\leftharpoondown$ or $\rightharpoondown$): This interaction involves a delay in receiving a reply, which extends beyond the recipient's immediate scope. A response might eventually be received, but there is a delay.

#### 2.3.1. Independent Compartments

Independent compartments, shown in Figure 3, contain activities and one agent or two objects: requestor and receptor. The statement, Send(virus): hacker $\rightarrow$ server: run (receive(…), is an example of a single or independent statement. Here, the actor hacker sends a virus, and the receptor actor executes what it received, which is in this case the virus. This is a linear activity that consists of a single action Send in the requestor and two actions Receive/Run in the receptor.

#### 2.3.2. Nested Compartments

In the example below, a nested compartment is illustrated. $A_1$ and $A_2$ represent two actions forming a single fractional activity associated with the object $a_1$. Meanwhile, the action $B_1$ and the nested compartment, referenced by the label, $\hat{B}$, are executed in parallel ($\|$), where the actor $b_1$ synchronously interacts with another actor $c_1$, anticipating a reply from the action $C_1$ captured by action $B_3$.

$$\frac{A_1}{A_2} : a_1 \leftrightarrow b_1 : (B_1 \parallel \hat{B})$$

$$B \equiv \frac{B_2}{B_3} : b_1 \leftrightarrow c_1 : C_1$$

Generally, when a compartment includes one or more requestors and/or receptors, such as actor $b_1$ in this example, transitioning to become a requestor to the receptor $c_1$, a nested compartment is formed, which could be written explicitly inside the host compartment or referenced externally. In this instance, the reference label, $B^{\wedge}$, is used to capture the nested compartment, written externally. The internal specification would be written as

$$\frac{A_1}{A_2} : a_1 \leftrightarrow b_1 : (B_1 \parallel \frac{B_2}{B_3} : b_1 \leftrightarrow c_1 : C_1)$$

For complex systems, the implicit notation for nested compartments is preferred.

#### 2.3.3. Grouped Compartments

Compartments can be grouped to eliminate redundancies and create concise structures. There are two possible grouped structures, iterative and miscellaneous.

##### Iterative Compartments

Certain sequences of compartments demonstrate repetitive patterns in terms of actions, activities, and interactions. To mitigate the repetition of identical compartments multiple times, an iterative and indexed object form can be deployed. In this analysis, we assume that indexed activities, such as $A_{jk}$, $B_k$, $C_j$, and $D$ correspond to variations within each compartment of the requestor object, $j$, and the corresponding receptor, $k$. The interaction between requestors and receptors can exhibit various communication patterns, such as one-to-one, one-to-any, many-to-one, and many-to-many interactions, which may incorporate consecutive, simultaneous, and/or selective behaviors. Equation (1) illustrates only synchronous interactions, but asynchronous and delayed interactions can also be employed. The general specification model for interactive compartments can be expressed as shown in Equation (1). There are $n$*$m$ compartments, each comprising a requestor, $S_j$, and a receptor, $R_k$, behaving according to the operators, $O$, defined in Section 2.2. Generally, the activity, $D$, without indexes is common to all objects across all compartments, the activity $C_j$ differs for each requestor, the activity $B_k$ changes with each receptor, and finally the activity $A_{jk}$ varies for each requestor, $j$, and receptor, $k$.

$$\frac{A_{jk}}{C_{jk}} : OS_{j[1 \le j \le n]} \leftrightarrow OR_{k[1 \le k \le m]} : \frac{B_k}{D} \tag{1}$$

##### Miscellaneous Compartments

Ultimately, a combination of operators' activities, objects, and compartment operators, $o$, $O$, and the variations of the interactions facilitate the creation of diverse and organized compartments. It is not always possible to group different compartments if the semantics of the results are ambiguous. In this case, one keeps them as independent compartments. The following example shows how to group two exclusive, $\otimes$, compartments, A/C: $a_1 \leftrightarrow a_2$: B $\otimes$ A/C: $a_1 \leftrightarrow a_3$: E, into $\frac{A}{C} : a_1 \leftrightarrow (a_2 : B \otimes a_3 : E)$.

#### 2.3.4. Delayed Compartments

Delayed compartments are based on delayed interactions which consist of a series of compartments connected by the operator, $O$, where the initial compartment entails the requestor soliciting a service, and the final compartment provides the service following intermediate executions of one or more series of compartments, in between. Figure 4 depicts a delayed interaction, where the requestor, $a_1$, receives its response after actor $a_2$ completes its activity, F2. This delay is associated with actor $a_2$ requesting services from actor $a_3$, first, before delivering the response to actor $a_1$. By default, a series of compartments are exe-

cuted consecutively unless an operator that specifies otherwise is utilized. The first and last compartments employ a delayed interaction ($\rightharpoonup$, $\rightharpoondown$).

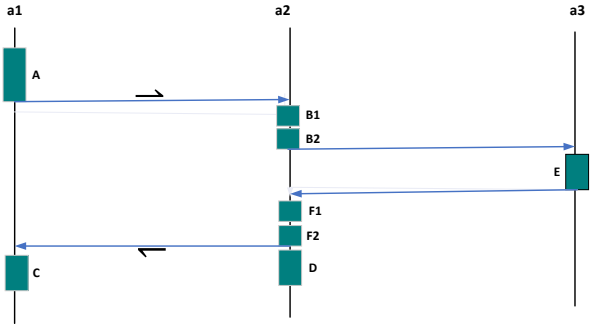


**Figure 4.** A sequence diagram for a delayed compartment.

The formal specification in ObTFL can be written as

$$A : a_1 \rightharpoonup a_2 : B1$$

$$\frac{B2}{F1} : a_2 \leftrightarrow a_3 : E$$

$$\frac{F2}{D} : a_2 \rightharpoondown a_1 : C$$

*2.4. Containers*

Containers encompass a variety of items or elements: values, objects, data, information, parameters, agents, actors, robots, sensors, events, other containers, data structures, databases, and actions. Common container types include arrays, lists, tuples, vectors, sequences, dictionaries, sets, tables, and specialized containers such as queues, stacks, and bags. ObTFL adopts four main container types: sets, dictionaries, tables, and sequences.

A set, denoted by { }, represents an unordered collection of unique items also called elements. These items can comprise both immutable and mutable values, variables, and aliases. If aliases are unique within the virtual space which encompasses all containers and objects in a system specification, they can be directly accessed using their identifiers or, alternatively, via the dot-operator "." to link them to the container they belong to.

A sequence, depicted by ( ), comprises an ordered collection of elements. Depending on the problem context, these elements can be either immutable or mutable, and they may also allow duplicates. Implementations of sequences encompass tuples, strings, vectors, arrays, and lists. Sometimes, sequences are employed as parameters to and from actions in ObTFL, where other containers are grouped into a sequence.

A dictionary, symbolized by <>, alternatively referred to as an associative array, map, or hash-table, comprises key–value pairs. The keys are unique and unordered, mapping to any element or item. Dictionaries are commonly utilized for database-related purposes within the context of ObTFL.

A table contains a set of entries. Each entry has one or many fields represented by columns. A referenced container functions as a pointer or link to the content of another container, much like a file name referencing the content of a file alongside its metadata, or a hyperlink in a browser directing to a server or website. This referenced container is indicated by a hat symbol, "^". Additionally, an orphan container lacks a name or label. For example, (2, 5, 8) denotes an orphan sequence of three elements.

In conversions, such as transforming a set into a sequence, the assignment operator "=" is utilized alongside appropriate symbols. Conversion is not always possible between different types of containers, which leads to some semantics ambiguities. For example, the statement {S} = (2, 3, 4, 4) converts a sequence to a set, resulting in S being equal to {2, 3, 4}, whereas <D> = (2, 3, 4, 4) produces the dictionary, D, as D = <2:3, 4:4> which semantically

has no interesting outcome. Conversely, the assignment operator ":=" inserts an element into a container. The statement {S} := (2, 3, 4, 4) will set the set, S, equal to {(2, 3, 4, 4)}. Elements within a container can be grouped randomly or conditionally. For instance, the notation $n$:$m$ specifies that $m$ elements from a container of $n$ elements are selected randomly, and the set remains unchanged, whereas $n$:$m$[$c$] indicates that the selection is based on a condition, c, to be discussed further in the next sections.

### *2.5. Actions and Activities*

Actions encompass algorithms implemented through various means such as code, pseudocode, lambda calculus, or any other technique outside the realm of ObTFL. Each action is designed to accept a series of parameters or references stored in containers, execute specific operations to accomplish defined objectives, and potentially may yield results stored in containers. The containers returned by actions are adaptable to various types including sets, sequences, dictionaries, and/or tables, with the option to apply rules of promotion and demotion conversions to align with the intended recipient's structure. In a way, the communications between objects and actions are entirely accomplished through containers as parameters. Collections of actions interconnected by operators, $o$, such as sequential (*), parallel ($\|$), rendezvous ($\|^{\circ}$), exclusive selection ($\times$), and/or inclusive selection (+), enclosed within brackets, or distributed in fractional form are termed activities. Organizing activities into numerators and denominators based on their order and purpose enhances formal expression, rendering specifications clearer and more succinct. As a general principle, actions allocated in the denominator are executed by default after the completion of those in the numerator, unless overridden by specified activity operators.

#### 2.5.1. Alternative Actions and Activities

Action can be alternated by the operator (x) when a decision is to be made and a path is to be selected. This can be done through junctions and conditions.

##### Junctions

A junction consists of an initial action followed by brackets containing a sequence of actions separated by the exclusive operator, "$\times$", optionally accompanied by conditions. For instance, an input action marked with (?) requires an examination of its container, and subsequently, based on the outcome, a specific path from the junction is chosen. As an example, consider the statement, Check ($?^{@(t)}\{\ldots\}$) (A$\times$ B $\times$ C), which evaluates the contents, {…}, received at a specific time, $?^{@(t)}$, from the set container { }, and exclusively selects between the actions A, B, and C.

In the formal language syntax of ObTFL, this example can be represented as the junction $?^{@(t)}\{\ldots\}$ ($\circlearrowleft \times \phi \times \triangleright$), where each action associated with the junction is evaluated based on predetermined conditions. The semantics of this syntax depend on the outcome of the chosen path: either repeating the current activity ($\circlearrowleft$), terminating without further progress ($\phi$), or advancing to the next compartment ($\triangleright$).

##### Conditions

The condition involves arithmetic operations performed on indexes, numbers, variables, and/or actions. Moreover, container and fuzzy operators can be utilized to address conditions arising from containers. These arithmetic and container operations are evaluated using relational operators, and subsequently, relational expressions are examined by Boolean operators to determine whether the condition is true, false, or fuzzy. As per the grammar rules of ObTFL, indexes can take various forms. For instance, consider the following examples:

$A_{[Sj=\lightning]}$ represents action $A$ performed on an object $S_j$ in a faulty state, denoted by $\lightning$.
$S_{j[j=j+1]}$ refers to the next object in sequence.
$S_{j[1\leq j\leq n:3]}$ indicates three objects selected randomly from a pool of $n$ objects.
$S_{[s\in Q]}$ addresses an object such that it belongs to a container $Q$.

For example, a condition like $A_{['T]}$ implies that the action path A is always followed in a junction. Lastly, the condition $\{A\}_{[\{A\}\subset\{B\}]}$ introduces the set {A} with the requirement that it must be a subset of another set, {B}.

### *2.6. Primitive Symbolic Actions*

Interactions among objects, such as actors and agents, usually begin with actions. Typically, actions are invoked or selected from libraries using their labels or names. However, in ObTFL, there are predefined actions that perform basic tasks. For simplicity, these actions are identified by symbols in addition to or instead of their names. Some of these symbolic actions are summarized in Appendix A, while others are explained next. Since they are actions, they can be indexed and associated with conditions.

#### 2.6.1. Stop (∅), Null (ε), Repeat (↻), and Progress (▷) Actions

To detect abnormal states, incomplete forms, and errors within a compartment, the stop action, symbolized by ∅, is utilized in various combinations. This action halts the progress of an activity or a compartment before it is completed. An activity may consist of a series of actions enclosed in brackets or a fractional statement. Conversely, the null action, represented by ε, indicates doing nothing. It is commonly employed in junctions when no path is selected.

For instance, the activity $A * \varnothing * B$ specifies that action A is performed, but $B$ will never be reached. However, $A * \varepsilon * B$ denotes that A is executed, followed by $B$; this is equivalent to $A * B$. Additionally, in a compartment with $A/B{:}a_1 \rightarrow a_2{:}\varnothing$, actor $a_2$ receives communication from actor $a_1$ but ignores it. Conversely, $A/B{:}a_1 \rightarrow a{:}\varepsilon$ indicates that actor $a_2$ did not receive the communication, as if the signal is lost or the communication has been hijacked. In both cases, a deadlock occurs if the interaction is synchronous, as a reply from $a_2$ will never be received by $a_1$ to be addressed by action $B$. These actions are used to specify the behaviors of a system under possible faults and errors.

The action, represented by ↻, repeats the activity it belongs to, defined by enclosed brackets, a fractional statement, or the entire block, such as (!Send (msg) * ↻) or !Send(msg)/↻. This effectively repeats the action of sending a message indefinitely. If a condition is associated with the repeat action, ↻, as in $\circlearrowleft_{j[1\leq j\leq n]}$, it indicates repeating the action $n$ times.

To illustrate the usage of the repeat action, let us consider the scenario of a persistent man trying to arrange a date with a lady. He sends her an SMS invitation, !Send {sms}. The lady's phone might be closed or not receiving, ε, or she receives the SMS, ?{…}, and then she decides to ignore it, indicated by ∅, or reply with a yes or no $(!Reply(T) \times (F))$. If she ignores him or her phone is closed, he repeats sending the SMS every 5 min, $\int_{d(5)}^{@(t)}$ for three times. If she refuses, by replying with a false, '$F$', he performs a stop action that terminates the activity and his wish to have a date, ∅.

$$\frac{!Send\ \{sms\}}{(\int_{d(5)}^{@(t)}(\circlearrowleft_{[1\leq j\leq 3]} \times \varnothing))\times ?(\ldots)\Big(\ \rhd_{[T]} \times \varnothing\Big)} : man\ \leftrightarrow woman :?\{\ldots\}(\varepsilon \times (!Reply(T) \times (F)) \times \varnothing)$$

#### 2.6.2. Timer or Delay Action (∫)

The syntax of the timer or delay action is $\int_{d(t_2)}^{@(t_1)}$. The delay process starts at time $t_1$ and lasts for a duration of $t_2$. The delay action is a versatile action that can be used alone to introduce a silent period for a specified duration or in conjunction with another action to defer its execution. In some simulation models, delays, such as service and arrival times, play a significant role in the modeling process.

For example, the statement of an action, $\int_{d(10)}^{@(t)} Delay * A$, or $\int_{d(10)}^{@(t)} A$, delays the operation of an actor for a duration of 10-time units, starting from the current time $t$, before the actor executes the action $A$.

There are several variants to the notations of the timer. The notations $\int_{d(t_2)}^{@}$ , $\int_{d(t_2)}^{@(t)}$ , and $\int_{d(t_2)}^{@(o)}$ indicate a duration of $t2$ starting from the current time. The difference between them lies in the recording of the current time: in the first notation, the current time is not recorded; in the second, it is recorded, in the variable $t$; and in the last, the current time (o) is an absolute value that remains constant across subsequent calls to the timer.

On the other hand, the notations $\int_{d(\infty)}^{@}$, $\int_{d}^{@}$ , $\int_{d(t)}^{@}$ , $\int_{d(0)}^{@}$, and $\int_{d(-t_2)}^{@(o)}$ signify actions occurring at the current time for various durations: indefinitely, an unspecified but finite duration, specified by time $t$, starting immediately, and lasting until the present moment, respectively. The last notation is useful for recording events that occurred in the past.

The following statement of the timed action, schedules activities, by only giving the chance to one of the three sequential activities $A$, $B$, or $C$, to complete after a duration of time, $t_2$, if action D cannot start immediately.

$$\left( \int_{d(t_2)}^{@} (A \times B \times C) \times \int_{d(0)}^{@(t_2)} D \right)$$

With the next statement, if one of the actions $A$, $B$, and $C$ performing concurrently, has not finished within a period, $t_2$, one of the actions, $\circlearrowleft \times \emptyset \times \rhd$, associated with the timer executes. This mechanism can be used for transmission with a timeout, like the TCP/IP protocol. In general, this statement has the same effect as $\int_{d(t_2)}^{@} \times \left|\right| A_{i[1 \leq i \leq n]}$. Some actions of the $n$-parallel actions are interrupted after time $t_2$ if they have not been completed.

$$\int_{d(t_2)}^{@} (\circlearrowleft \times \emptyset \times \rhd) \times (A \,||B\, ||C)$$

2.6.3. Input (?) and Output (!) Actions

All interactions between actors are initiated by two special primitives: (!) and (?). The first action operator (!) injects a collection of items gathered in a container into the communication medium. This is usually associated with action names such as Send, Register, Transmit, Reply, Open, Login, Close, etc. The second action operator (?) extracts any container injected by the corresponding operator (!). The associated action names with the last operator are Receive, Get, Obtain, etc. While these names are not mandatory, they are occasionally used for readability purposes.

These operators solely exchange the payload of a message, while the communication protocol itself is abstracted in the layer of interaction; refer to layer 3 in Figure 1. The underlying communication protocol is concealed, and interchangeably, we replace IP addresses with the names and identifiers of the requestors and receptors.

Both input and output action primitives can be synchronized with buffered or unbuffered communication, depending on the nature of the interaction between two objects. The primitive, Receive (?), regardless of the type of interaction, always implements a blocking receive scheme. In this example, the receptor blocks until a message is sent by the requestor. By default, the requestor and receptors are autonomous and perform concurrently unless they are synchronized to wait for replies from each other. In the example provided, !Send(msg): Requestor $\rightarrow$ Receptor: Action ?(…), the requestor waits for the message collected in the sequence container to arrive and then performs the action on it. The notation, ?(…), signifies receiving something in a container of type sequence ( ). The blocking send notation for the primitive, !, is denoted as, $!^{\circ}$.

2.6.4. Store ($\nabla$) and Retrieve ($\Delta$) Actions

Similarly to the primitives send (!) and receive (?), which are used for passing containers, the primitives store ($\nabla$) and retrieve ($\Delta$), along with the blocking store ($\nabla^{\circ}$) and retrieve ($\Delta^{\circ}$), are employed for container sharing. The blocking retrieve behaves similarly to the receive (?). These seven primitives can be combined within the same model, provid-

ing various perspectives in terms of specification. To illustrate this concept, let us consider a queuing system.

A generator produces customers denoted as C at random intervals, typically every average time interval represented by $t_1$. The customers then join a queue denoted as Q, which is implemented as a sequence and managed by a security agent denoted as S. Once the security agent deems a customer ready to be served, he passes him/her on to the teller denoted as T, as shown in Figure 5. This model finds applications in various real-world scenarios such as transportation, communication networks, and banking systems. In this specification, the generator continuously generates an infinite stream of clients, repeating the action Send as shown in the modified UML sequence diagram of Figure 6. The junctions are depicted by the symbol x. In the formal specification, shown below, once a customer is generated, the actor S places him/her at the back of the queue $\dashv$Q and either terminates or sends a notification to the teller if the teller is not active, identified by the condition of the queue being empty [Q = ( )]. The teller retrieves customers from the front of the queue $\vdash$Q and spends a random time $t_3$ servicing them. It repeats the same process or terminates if the queue is empty.

$$\frac{\int^{@}_{d(Rand(t_1))} *!Send(C)}{\circlearrowleft} : G \to S : \bigtriangledown Put\Big(?(\ldots)^{@},\ \dashv Q\Big)\Big(\hat{A}_{[Q=(\ )]} \times \varnothing\Big)$$

$$\Downarrow\!\Uparrow$$

$$A \equiv \text{ Notify } : \mathrm{S} \to T : \frac{\Delta Get(C, \vdash Q)\Big(\int^{@(t_2)}_{d(Rand(t_3))} Service(C) \times \varnothing_{[Q=(\ )]}\Big)}{\circlearrowleft}.$$

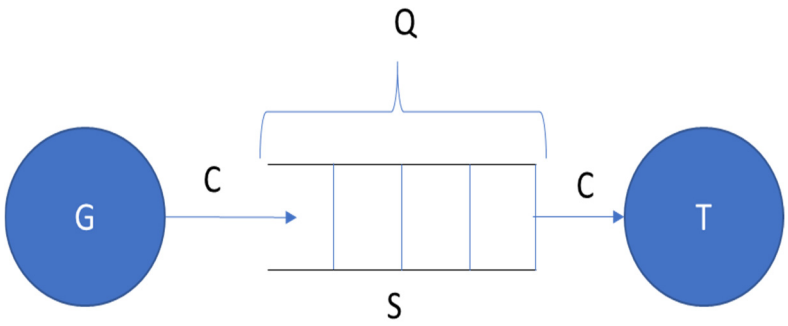


**Figure 5.** A single queuing system.

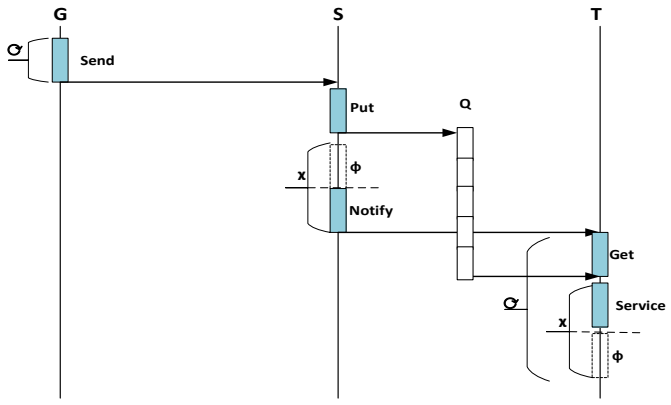


**Figure 6.** Modified sequence diagram for a single-server single-generator queuing system.

### 2.6.5. Fail (⇟) and Recover (↟) Actions

The advancement of compartments can be disrupted by various events. Certain compartments may fail to achieve success and are forced to pause, necessitating corrective measures from external entities or, in cases involving intelligent agents, autonomously executing self-repair protocols. These events may arise internally, due to software or hard-

ware failures, malware attacks, or internal interruptions, or externally, like attacks from malicious actors sending viruses or compromising communication infrastructure, or even natural disasters such as earthquakes, floods, and fires. The impact of events on interactions is symbolized by ($\rightsquigarrow$). Figure 7 depicts the various steps to invoke external or internal repairs, where an external event or internal incident with an action, causing an internal fault like hardware or software, occurs and disrupts the actor that Fails ($\ddagger$) while performing the actions, $A1$. The actor (machine) resumes the execution of action, $A1$, if it is repaired by an external actor (Repairer) or it performs its own self-repair through the execution of the repair action, B.

$$\left(\begin{array}{c} \left(\begin{array}{c} U1 : Event \rightsquigarrow machine : \ A1 \ * \ \ddagger \\ \otimes \\ machine : A1 * Fault * \ddagger \end{array}\right) \\ \circledast \\ \left(\begin{array}{c} B : repairer \rightsquigarrow Machine : \left(\Updownarrow * A1 \times \varnothing\right) \\ \otimes \\ machine : B\left(\Updownarrow * A1 \times \varnothing\right) \end{array}\right) \\ \otimes \\ \phi \end{array}\right)$$

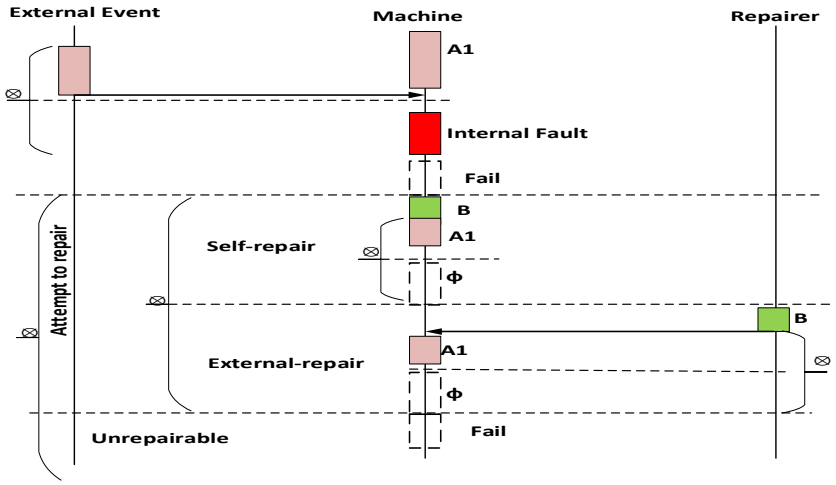


**Figure 7.** UML modified sequence diagram with failures and repairs.

## 3. Case Studies

This section presents three case studies to illustrate the capabilities of the formal language, ObTFL. The case studies cover federated learning [21,22] with data protection, blockchain in cryptocurrency [23–25], and the IPC-HERMES-9852 standard [26]. Case study 1 demonstrates the use of iterative compartments, where multiple autonomous servers interact with a central server. Case study 2 introduces referenced activities and compartments including iterative compartments, along with containers. Lastly, case study 3 showcases the application of exclusive selection between compartments.

### *3.1. Case 1: Federated Learning*

Federated learning is a supervised machine learning approach wherein local servers are trained on decentralized data sources, and the resulting model updates are aggregated on a central server to improve a global model. This method prioritizes privacy by minimizing data exposure during the training process.

Consider a scenario with $n$ hospitals, each with confidential patient records. Each hospital's server, denoted as $Ri$, initiates training by initializing its local model with weights ($wi$) and biases ($bi$) set to zero on its dataset, $Si$. Following training, each hospital server

computes the gradients of its local model with respect to a common loss function. This step, referred to as *UpLocalModel*, sends the evaluated weights and biases to the central server, *C*, as shown in Figure 8.

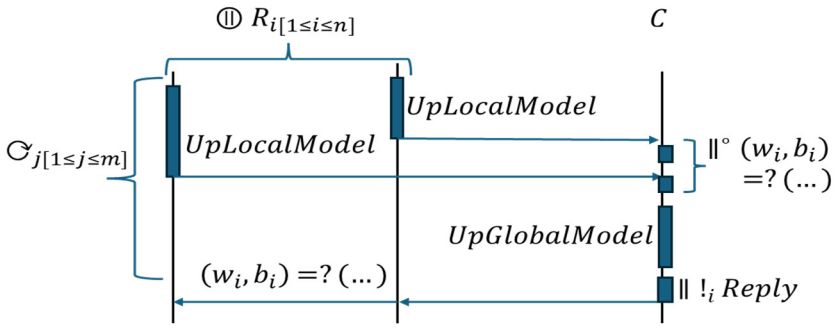


**Figure 8.** UML modified sequence diagram for hospital federated learning.

Upon receiving updates from all hospitals simultaneously, shown by the parallel rendezvous operator, $\Vert^{\circ}$, of the weights and the biases, $(w'_i,\ b'_i)$, the central server gathers them into a set, *S*, of tuples, $(w_i,\ b_i) :=?(\ldots)$, and then performs the action, *UpGlobalModel*, to adjust the global model. This update incorporates the average weights and biases from the hospitals, as well as the previous parameters of the central server, using a speciied learning rate. The central server computes new weights and biases using the formula for the *UpGlobalModel*: *newWeight* = *oldWeight* − *learningRate* ∗ *averageWeight* and *newBias* = *oldBias* − *learningRate* ∗ *averageBias*. The average action takes a set of weights and biases as tuples and produces a tuple with *Weight* and *Bias* to be used by the *UpGlobalModel* action to evaluate the *new weight* and the *new bias*. The expression of the action UpGlobaModel is outside the specification. Once the central server calculates the new parameters for the global model, they are then broadcasted, $\Vert$, to the local servers. The process repeats multiple epochs, $\circlearrowleft_{ij[1\le j\le m]}$, until the global model reaches an acceptable level of performance.

$$\frac{UpLocalModel\ \{S_i\}(w_i,\ b_i)*!Send(w'_i,\ b'_i)}{(w_i,\ b_i):=?(\ldots)*\circlearrowleft_{ij[1\le j\le m]}}$$
$$: \Vert R_{i[1\le i\le n]} \leftrightarrow C :$$
$$S = \{\Vert^{\circ}\ (w_i,\ b_i) :=?(\ldots)\ \} *\ \ \Vert !_i Reply(UpGlobalModel(Average\ \{S\})$$

### *3.2. Case 2: Blockchain for Cryptocurrency*

Blockchain is a distributed digital ledger technology. Nodes, which are computers within the network, exchange transactions that are chained together and stored as permanent, immutable, secure, and transparent records in blocks on each node [23–25]. In the context of cryptocurrency, additional nodes, known as miners, compete to solve a cryptographic puzzle (proof of work) to validate and add transactions to the blockchain. This concept extends beyond cryptocurrencies and has several applications. Blockchain technology has been applied to supply chain management, smart contracts, voting systems, and healthcare.

This case study explores the blockchain in cryptocurrency, and the language ObTFL deploys the containers to model a set of *n* nodes, $N_{i[1\le i\le n]}$, participating in the blockchain network, and a group of *q* of them taking the extra task of acting as miners, $M_{i[1\le i\le n:q]}$ .

$$Nodes = \left\{\ N_{i[1\le i\le n]},\ M_{i[1\le i\le n:q]}\ \right\}$$

A transaction comprises various fields, including a transaction ID, input (representing unspent transaction outputs, or UTXOs, which must cover the specified amount), output (containing sender, recipient, and amount details), signature, and other pertinent information. The sender node signs the transaction data, encompassing sender, recipient, and

amount, among other elements, using their private key (e.g., $K_A^-$), thus generating a digital signature. Each node, $i$, might generate, $m_i$, transactions represented as a dictionary container.

$$Tran_{ij[1\le j\le m_i]} = \ < \text{sender}: \ \ldots, \text{recipient}: \ \ldots, \text{amount}: \ \ldots, \text{sig}: \text{Enc}\left(\text{Hash}\left(\text{sig} \doteq Tran_{ij}\right), K_i^-\right) \text{others}: \ \ldots >$$

A block, depending on its size, has the capacity to accommodate numerous transactions, typically ranging from 2000 to 4000 transactions per block. The block header, as shown in Figure 9, contains essential information, including the hash value of the previous block header, a hash representation of the block's transactions (such as the Merkle tree root hash or hashing of all block transactions), the block's size, and the nonce value, often utilized for mining purposes. It is worth noting that nonce values may or may not be included in all blockchain networks.

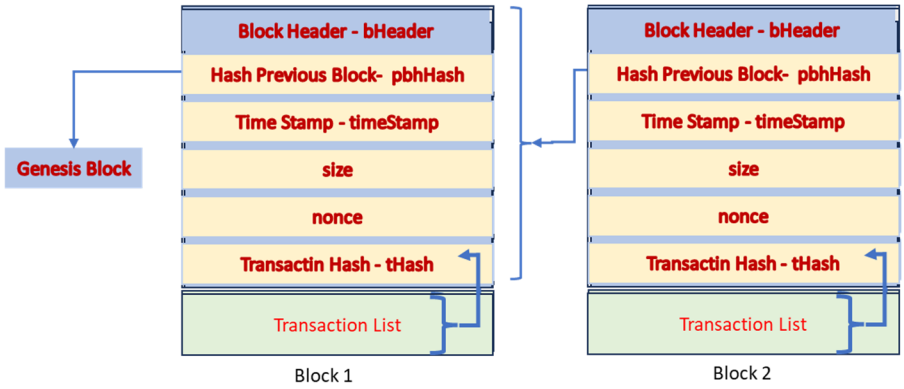


**Figure 9.** Two blocks chained.

Therefore, a block can be modelled as the union of two dictionaries: the overhead and a list of $p_k$ transactions. The total number of blocks is denoted as $b$.

$$Overhead_k = < bHeader: \ldots, \ pbhHash \ : \ldots, \ timeStamp: \ldots, \ size: \ldots, \ once: 0, \ tHash: \ldots >$$

$$Block_{k[1\le k\le b]} = Overhead_k \cup Tran_{kt[1\le t\le p_k]}$$

The ledger in each node, $i$, in the network holds a sequence of valid blocks, $Ledger_i$ = ($Block_k$). We also define two additional parameters the priority as two exclusive sequences, $Pr$ = ($fee$) $\times$ ($size$), and the memory pool as a container sequence used to temporarily hold transactions, $mPool$ = ( ).

The blockchain process in the cryptocurrency network, shown in Figure 10, begins with parallel autonomous, , sender nodes, $N_{i[1\le i\le n]}$, performing three parallel referenced activities continuously, $\frac{(\hat{A_1}\|\hat{A_2}\|\hat{A_3})}{\circlearrowleft}$ and a group of $q$ autonomous miners, $M_{i[1\le i\le n:q]}$ , responding simultaneously with three activities, $\frac{(\hat{B_1}\|\ (\hat{B_2}*\ \hat{B_3})\ )}{\circlearrowleft}$, where the activities $\hat{B_2}$ and $\hat{B_3}$ depend on each other and are performed sequentially. The iterative compartments, shown in Figure 10, can be written as

$$\frac{(\hat{A_1}\ \|\ \hat{A_2}\ \|\ \hat{A_3})}{\circlearrowleft} : \Pi N_{i[1\le i\le n]} \leftrightarrow \ \Pi M_{i[1\le i\le n:q]} \ : \frac{(\hat{B_1}\ \|\ (\hat{B_2}*\hat{B_3}\ ))}{\circlearrowleft}$$

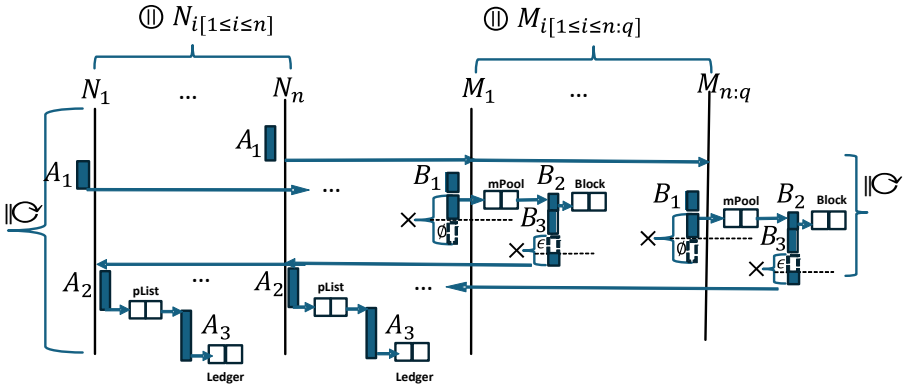


**Figure 10.** UML modified sequence diagram for blockchain.

Randomly, every time $\tau_{ij}$, a node, $i$, prepares and sends a transaction, $j$, during the activity $A_1$.

$$A_1 \equiv \int_{d(rand(\tau_{ij}))}^{@(ti)} * Prepare < Tran_{ij} > *!Send < Tran_{ij[j=j+1]} >$$

Simultaneously, the same node and any other nodes may receive blocks from the miners' nodes and then insert them into the pending list while monitoring the length of the blocks. These actions are grouped in activity $A_2$.

$$A_2 \equiv \nabla^{\circ} insert(?(Block_k),\ pList) * \text{MonitorBlockchainLength}$$

Furthermore, if the pending list has some blocks that are ready, meaning they have reached their highest accumulated proof of work, in the activity $A_3$, the blocks are permanently inserted into the ledger.

$$A_3 \equiv \Delta^{\circ}\text{Extract}(Block_{k[\,C\,]}pList) * \nabla insert(Block_k, Ledger_{i[i=n:1]})$$

$$C \equiv the\ highest\ accumulated\ proof\ of\ work$$

The signed transactions are broadcast to nodes registered within the blockchain network, such as miners, full nodes, and other network participants. Within this context, miners play a vital role in validating and subsequently incorporating the transaction into a block. The three activities performed concurrently by the miners are specified as follows.

The first activity, $B_1$, involves receiving transactions, $? < Tran_{ij} >$, verifying their integrities, after decoding and checking their hashes, and temporarily storing them in the memory pool, *memPool*, in the order of their arrivals, unless a priority order is imposed, as indicated in the formal activity $B_1$ with the priority, $pr$. Due to the finite memory size of the *memPool*, a memory management scheme is usually implemented where unconfirmed transactions are ignored, $\phi$, or removed from the *memPool* to make room for incoming transactions.

$$B_1 \equiv\ ? < Tran_{ij} > * \bowtie \text{Compare}\big(Dec\big(Tran_{ij}.\text{sig}, K_i^+\big),\ \text{Hash}\ (\text{sig} \doteq\ Tran_{ij})\big)$$

$$\big(\phi \times \nabla insert\big(Tran_{ij},\ pr,\ mPool\big)\big)$$

In the activity, $B_2$, transactions are removed from the front, $\vdash$, of the memory pool, and inserted in the consecutive blocks after using the Merk algorithm to obtain the hashes of a list of transactions stored in the block.

$$B_2 \equiv \Big(\nabla^{\circ}(\Delta^{\circ}(tran,\ \vdash mPool),\ Block_{k[k=k+1]}\Big) * Block_k.tHash := Merk < Tran_{kt} >$$

Finally, the activity $B_3$ is devoted to the proof of work consensus. It increments the nonce of the block and performs the laborious hashing process on the block header until the condition of a required number of leading zeros in the proof of work is met, and then the block is sent to be stored in the ledger or ignored, $\epsilon$.

$$B_3 \equiv Inc(Block_k.nonce) * (h) := HashPoW(Overhead_k)\left(\varepsilon \times !_{[h=V]}\text{Send}(Block_k)\right)$$

$$V \equiv \text{Measure the number of leading 0 in PoW consensus}$$

The Merk activity hashes the transactions in pairs until the root which contains the final hash is reached. Alternatively, a different algorithm can be used where the entire list of transactions is hashed together. Miners send blocks upon solving the puzzle in $B_3$; although these blocks are valid, they are placed in a pending list, until nodes independently verify and extend the longest valid chain by adding new blocks to it. The length of the blockchain serves as a proxy for cumulative proof of work.

This example illustrates how the complex blockchain system and its interactions can be specified using the syntax of ObTFL language.

### *3.3. Case 3: IPC-HERMES-9852 Protocol*

The importance of using formal specifications in the manufacturing process is becoming increasingly essential [27]. Several formal languages are now employed to describe production processes in Industry 4.0 manufacturing, including some that feature automatic code generation with domain-specific languages [28]. This approach is based on models similar to UML activity diagrams and Petri nets. Other modelling frameworks specify formal properties at the model level, automatically extracting formal specifications [29]. In this case study, we will use the formal language introduced in this paper to address the specification of the manufacturing process for printed circuit boards (PCBs).

The IPC-HERMES-9852 standard has emerged as a prominent protocol in electronic manufacturing in Industry 4.0. The focus of this case study in particular is on the transportation of the PCBs from one machine to another (M-2-M), as shown in Figure 11. The down machine (DM) and the up machine (UM) are synchronized by flagging their status—ready or not—to each other. This example demonstrates the use of ObTFL alternative activities and alternative compartments.

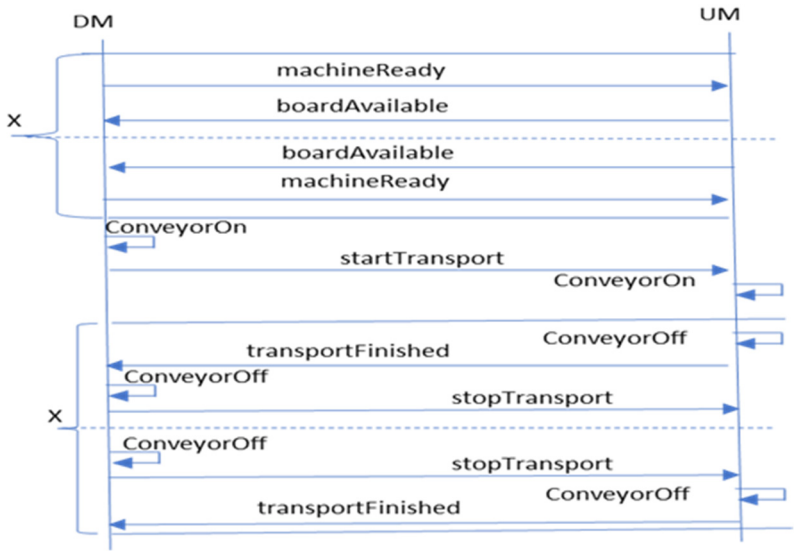


**Figure 11.** The normal operation of Hermes [30] protocol.

The down and up machines are independently dealing with the PCB actions named as a process here as

*Process: DM*
⏸
*Process: UM*

Figure 11 illustrates the scenario where one machine becomes ready and emits a signal to inform the other machine of its readiness to accept more PCBs. The behaviour of these signals is defined using the send (!) and receive (?) primitives of the language. If the machine DM becomes ready first, it sends its readiness signal as !machineReady. If the machine UM is not yet ready, meaning it is still completing the compartment Process: UM as specified earlier, its input action, $?^{@}(\ldots)$, has not been reached yet. Therefore, machine DM will wait until the board available action, !boardAvailable, is performed. The outcome of this action is captured by the first input action, $?^{@}(\ldots)$, of machine DM since the interaction between the two machines is synchronous. Formally, these behaviours can be specified as follows:

$$\frac{!machineReady}{?^{@}(\ldots)*conveyorOn*!StartTransport} : DM \leftrightarrow UM : \frac{?^{@}(\ldots)*\ !boardAvailable}{?^{@}(\ldots)*conveyorOn}$$

$$\otimes$$

$$\frac{!boardAvailable}{?^{@}(\ldots)*\ ?^{@}(\ldots)} : UM \leftrightarrow DM : \frac{?^{@}(\ldots)*\ !machineReady}{conveyorOn*\ !StartTransport}$$

$$\frac{conveyorOff*\ !stopTransport}{?(\ldots)} : DM \leftrightarrow UM :?(\ldots) * ConveyorOff * !transportFinished$$

$$\otimes$$

$$\frac{conveyorOff * !transportFinished}{?^{@}(\ldots)} : UM \leftrightarrow DM : \ ?(\ldots) * ConveyorOff * !stopTransport$$

## 4. Discussion

This section explores the potential and versatility of the formal specification language ObTFL introduced in this paper. As part of the discussion, case study 3—Hermes protocol v1.2—is specified using the π-calculus and compared to the version of case study 3 in ObTFL notation. The complete model of Hermes protocol v1.2 is fully specified in [30] using the π-calculus.

$$UM \stackrel{def}{=} machineReady.\overline{boardAvailable}.UM_{cont} + \overline{boardAvailable}.machineReady.UM_{cont}$$

where

$$UM_{cont} \stackrel{def}{=} startTransport.\tau.(\tau.\overline{transportFinished(\text{complete})}.stopTransport(x).0 + \tau.stopTransport(x').\overline{transportFinished(x')}.0)$$

$$DM \stackrel{def}{=} \overline{machineReady}.boardAvailable.DM_{cont} + boardAvailable.\overline{machineReady}.DM_{cont}$$

where

$$DM_{cont} \stackrel{def}{=} \overline{startTransport}.\tau.(\tau.transportFinished(y').\tau.\overline{stopTransport(y')}.0 + \tau.\overline{stopTransport}(complete).transportFinished(y).0)$$

Upon comparing the formal specification of case study 3 in ObTF and the π-calculus, it is evident that the former offers a more intuitive approach. Reference [30] predicates assume the availability of a PCB for transportation, and the pervasive inclusion of timing sequences (τ) alongside activities can be somewhat perplexing. This representation only indicates silent intervals in the example, which vary, leading to reduced accuracy. Moreover, errors in [30] are specified independently and in more complex formats.

In contrast, the grammar rules of ObTFL make it inherently more expressive, with interactions clearly delineating which actors are involved in specific actions. This also conforms to the sequence diagram exhibiting similar patterns. Its syntax aligns closely with implementation in any programming language. Notably, interactions between UM and DM machines are synchronous, with no assumptions regarding the existence of a PCB. Instead, it simply mandates that two conditions be met: reception of the ready signal, $?^{@}(...)$, and the subsequent wait for the PCB event to become available, irrespective of timing. This waiting process continues indefinitely until a PCB is either available or the triggering event concludes at a specified time (@). Consequently, the timing is more precise, relying on synchronizations between the primitive send (!) and receive (?) operations. Nonetheless, both ObTFL and $\pi$-calculus can specify a wide range of behaviors in distributed systems, albeit with different syntaxes and underlying philosophies.

To demonstrate the versatility of the language, three completely different case studies have been formally specified. The main connection between them is the environment where autonomous objects such as machines, nodes, servers, actors, and/or agents interact with each other after performing actions and activities. The entire world operates as a vast distributed system, where individual objects manage their own affairs while interacting with one another.

The related standards of formal specification languages discussed in the introduction expose the different domains and applications. Like others, ObTFL focuses on distributed systems in general, where parallel and autonomous objects are involved. It also extend to the specifications of security protocols. Its main feature is its closeness to implementation in any programming language such as Python, C++, or Java.

## 5. Conclusions

This paper introduces a formal specification language designed for use in distributed systems, where interactions and parallel activities are central to functionality. The language's grammar rules extend to interactions requiring authentication, confidentiality, and integrity, as demonstrated in case study 2. Its applicability spans various environments, involving interactions and activities within distributed systems and networks.

The language's semantics are based on mathematical models and symbols, ensuring robustness and precision. Its distinguishing feature is a simplified syntax compared to other formal languages, making it accessible to a broader audience interested in specifying and documenting problems within distributed system domains. This accessibility enables stakeholders to articulate and conceptualize complex scenarios before transitioning to implementation using familiar languages such as Python, C++, or Java. Additionally, the modified sequence UML diagram serves as a visualization tool to enhance understanding of the structure, organization, and readability of ObTFL semantics.

The development of this formal language is poised to advance significantly, with upcoming extensions to encompass distributed database systems and their complex interactions through the utilization of table containers. This enhancement will broaden the language's applicability, enabling it to model and specify the behavior of far more complex modern computing environments.

To illustrate the full capabilities and versatility of the language, we have several ambitious case studies in the pipeline. Some of them have already been formally specified with the language. The case studies cover a diverse array of scenarios and challenges, including the following:

WiFi Attacks: Detailed analysis and formal specifications for mitigating WiFi attacks such as wormholes, black holes, and gray holes. These attacks exploit vulnerabilities in wireless networks, and our language aims to provide precise mechanisms for detecting and counteracting them.

IoT Forensics: Comprehensive studies on the forensics of Internet of Things (IoT) devices, including vehicles and drones. As these devices become increasingly integrated into our daily lives, ensuring their security and understanding forensic methodologies will be

paramount. Our language will help specify the interactions and behaviors of these devices in forensic investigations.

Cloud Infrastructure Protection: Addressing the protection of cloud infrastructure that stores critical evidence. This involves developing formal specifications for securing cloud environments against breaches and ensuring the integrity and confidentiality of stored data. The language will include models for safeguarding evidence within cloud platforms.

Security and Privacy in the Dark Web: Exploring the complex issues of security and privacy associated with the dark web. This includes developing formal specifications for monitoring and investigating dark web activities while ensuring user privacy and data protection. The language will provide tools for understanding and mitigating the risks posed by the dark web.

By tackling these varied and intricate case studies, we aim to demonstrate, with possibly additional syntax, the comprehensive spectrum of the language and its practical applications in addressing real-world security and forensic challenges. This future work will not only enhance the language itself but also contribute to the broader field of cybersecurity and digital forensics, offering robust solutions and methodologies for safeguarding digital environments from cybercrimes.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The raw data supporting the conclusions of this article will be made available by the author on request.

**Conflicts of Interest:** The author declares no conflict of interest.

## Appendix A

**Table A1.** The syntax of ObFTL.

| Symbols | The Meanings of Primitive Actions, Operators, and Containers |
|---|---|
| $@$, $d$, $\hat{}$, $\equiv$ | The starting time, $@$, of any actions, and the duration, $d$, of the delay action only, reference ($\hat{}$) to something, ($\equiv$) stands for define as. |
| Status actions applied to objects and conditions | |
| $\ddagger$, $\updownarrow$ | Set/test the status of an object—idle, down, broken, unavailable ($\ddagger$,), or active ($\updownarrow$). |
| Delay actions | |
| $\int^{@}$, $\int_{d}^{@}$, $\int_{d(\ )[\ ]}^{@}$, $\int_{[\ ]}^{@}$ | Delay action with its variants, determined by starting, @, time, duration, d, and/or conditions, [ ]. |
| Decision actions | |
| $\triangleright$, $\trianglerighteq$<br>$\circlearrowright$, $\circlearrowright\!\!\!\!\circlearrowright$<br>$\varnothing$, $\epsilon$ | Move to the next compartment ($\triangleright$), or move to the next transaction ($\trianglerighteq$).<br>Repeat the current activity/compartment ($\circlearrowright$) or transaction ($\circlearrowright\!\!\!\!\circlearrowright$).<br>Stop and no progress ($\varnothing$); it needs to be reactivated. Do nothing ($\epsilon$); does not need to be reactivated. |
| Container passing actions | |
| $!^{@}$, $?^{@}$ | Send (!) and receive (?) actions. The receive action (?) is synchronous by default. |
| Containers sharing actions | |
| $\nabla^{@}$, $\Delta^{@}$,<br>$\nabla^{\circ}$, $\Delta^{\circ}$<br>$:=$, $\doteq$ | Put/Store and Get/Retrieve<br>Blocking Put/Store if container full and Blocking Get/Retrieve if container empty.<br>Insert into ($:=$) or extract from ($\doteq$) a container at/from the back $\dashv$ or the front $\vdash$. |

**Table A1.** *Cont.*

| Symbols | The Meanings of Primitive Actions, Operators, and Containers |
|---|---|
| Container handling actions | |
| $\Rightarrow$, $\not\Rightarrow$, $\bowtie$, $\varnothing$ | Search ($\Rightarrow$), remove ($\not\Rightarrow$), compare items ($\bowtie$) from a container, clear the whole ($\varnothing C$) container, $C$. |
| Action operators | |
| $*$, $*^{+}$, $\|$, $\|^{+}$, $\|^{\circ+}$, $\times$, $+$, $\|^{\circ}$ | Sequential, sequential from a group of "ored"-activities, parallel, rendezvous, parallel group of "ored"-activities, parallel "or" with rendezvous, select only one, select a group (or). The $\times$ operator is prioritized. When more than one action is ready to be performed, the priority starts from the left-hand side. |
| Compartment operators | |
| $\circledast$, $\otimes$, $\oplus$, , $^{+}$, $^{\circ\circ+}$<br>**:** | Consecutive, selective, group selective, simultaneous, parallel groups, rendezvous, group rendezvous.<br>Separator between actions and objects *action*:*object* and *object*:*action* |
| Container declarations | |
| [ ] < > { } ( )<br>= [ ], < >, { }, ( )<br>S := {}, [ ], ( ), < ><br>= [•], {•}, (•), <•><br>[… ]{… } (… ) <…> | Table, Dictionary, Set, Sequence containers.<br>Used with conditions [], to check if a container is empty<br>Clearing a container with (:=), similar to the action $\Phi$.<br>Used with conditions to check if a container is full.<br>Container has something. It can be used with the receive primitive action (?) to receive something. |
| Interaction symbols | |
| $\leftrightarrow$, $\rightarrow$, ($\rightharpoonup$, $\leftharpoondown$ or $\rightharpoondown$), $\rightsquigarrow$ | Synchronous ($\leftrightarrow$), Asynchronous ($\rightarrow$), Delayed ($\rightharpoonup$, $\leftharpoondown$, $\rightharpoondown$), Alter ($\rightsquigarrow$) the state of an object. |
| Symbols, operators, values, aliases used inside the [Condition] | |
| $\subset$, $\subseteq$, $\notin$, $\in$, $\exists$,$\forall$, $\therefore$ , $\eta(C)$ | Container operators, inclusion, there exist, for all, such that, cardinality of a container |
| 'T', 'F' | Boolean values—true and false |
| &, \|, ~, $\times$ | Boolean operators, "and", "or", negation, exclusion |
| "string" | A word, a sentence, a set of characters |
| numbers, indexes, variables | Integer, decimals, or container values |
| $+$, $*$, $-$, %, /, $\div$ | Arithmetic operators |
| $<>\leq$ $\geq=\neq\simeq$ | Relational operators |
| *n*:*m*:*p*:..<br>*n*:*m*:*p*:..[*condition*] | Group random extractions: *p* items, selected from *m* items from *n* items of a container, etc.<br>Group selective extractions with conditions |

## References


1. Serrano, D.; Iglesias, C.A. JSONbis: A Proposal to Extend JSON with Type Information. In Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020), Virtual Event, 8–10 June 2020; pp. 290–297.
2. Jacobs, S. *Beginning XML with DOM and Ajax: From Novice to Professional*; Apress: New York, NY, USA, 2020.
3. Kaye, R. *The Mathematical Theory of Predicate Logic*; Cambridge University Press: Cambridge, UK, 2020.
4. Stubblebine, T. *Regular Expressions Pocket Reference*; O'Reilly Media: Sebastopol, CA, USA, 2021.
5. Cooper, S.B.; Soskova, M. *Turing Machines and Computational Theory*; Springer: Berlin/Heidelberg, Germany, 2020.
6. Bernardo, M.; Nicola, R.D.; Loreti, M. *Process Algebra and Probabilistic Models: Performance and Dependability Analysis*; Springer: Berlin/Heidelberg, Germany, 2020.
7. Whitney, J.; Gifford, C.; Pantoja, M. Distributed execution of communicating sequential process-style concurrency: Golang case study. *J. Supercomput.* **2018**, *75*, 1396–1409. [CrossRef]
8. Vrancken, L.M. The algebra of communicating processes with empty process. *Theor. Comput. Sci.* **1997**, *177*, 287–328. [CrossRef]
9. Friedman, A. *Communicating with Process Calculus*; A Major Qualifying Project Submitted to the Faculty of Worcester Polytechnic Institute: Worcester, MA, USA, 2023.

10. Nicollin, X.; Sifakis, J. An overview and synthesis on timed process algebras. In *Computer Aided Verification*; Springer: Berlin/Heidelberg, Germany, 1992; pp. 376–398.
11. Umer, M.; Ali, A. *Automated Analysis of the Security of the IoT Using Pi-Calculus*; Springer: Berlin/Heidelberg, Germany, 2021.
12. Abadi, M.; Gordon, A.D. *A Calculus for Cryptographic Protocols: The Spi Calculus*; Elsevier: Amsterdam, The Netherlands, 1999; Volume 148, pp. 1–70.
13. Hennessy, M. *A Distributed Pi-Calculus*; Cambridge Press: Cambridge, UK, 2007; ISBN 9780521873307.
14. Liao, Y.; Yeaser, A.; Yang, B.; Tung, J.; Hashemi, E. Unsupervised fault detection and recovery for intelligent robotic rollators. *Robot. Auton. Syst.* **2021**, *146*, 103876. [CrossRef]
15. Al Fikri, M.; Ramli, K.; Sudiana, D. Formal Verification of the Authentication and Voice Communication Protocol Security on Device X Using Scyther Tool. In *IOP Conference Series: Materials Science and Engineering, Proceedings of the 5th International Conference on Information Technology and Digital Applications (ICITDA 2020), Yogyakarta, Indonesia, 13–14 November 2020*; IOP Publishing Ltd.: Bristol, UK, 2021; Volume 1077.
16. Cortier, V.; Delaune, S.; Dreier, J.; Klein, E. Automatic generation of sources lemmas in Tamarin: Towards automatic proofs of security protocols. In *Computer Security–ESORICS 2020*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 3–22.
17. Blanchet, B.; Cheval, V.; Cortier, V. ProVerif with Lemmas, Induction, Fast Subsumption, and Much More. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022.
18. Yogesh, P.R.; Devane Satish, D. Formal Verification of Secure Evidence Collection Protocol using BAN Logic and AVISPA. *Procedia Comput. Sci.* **2020**, *167*, 1334–1344. [CrossRef]
19. Adda, M. *ObTFL Formal Language for Spider Network*; Internal Technical Report; University of Portsmouth: Portsmouth, UK, 2023.
20. Adda, M. A Formal Language for Actors' Interactions. In Proceedings of the ITT 2023 Information Technology Trends, Dubai, United Arab Emirates, 24–25 May 2023.
21. Crowson, M.G.; Moukheiber, D.; Arévalo, A.R.; Lam, B.D.; Mantena, S.; Rana, A.; Goss, D.; Bates, D.W.; Celi, L.A. A systematic review of federated learning applications for biomedical data. *PLoS Digit. Health* **2022**, *1*, e0000033. [CrossRef] [PubMed]
22. Ogundokun, R.O.; Misra, S.; Maskeliunas, R.; Damasevicius, R. A Review on Federated Learning and Machine Learning Approaches: Categorization, Application Areas, and Blockchain Technology. *Information* **2022**, *13*, 263. [CrossRef]
23. Gad, A.G.; Mosa, D.T.; Abualigah, L.; Abohany, A.A. Emerging Trends in Blockchain Technology and Applications: A Review and Outlook. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 6719–6742. [CrossRef]
24. Unata, D.; Hammoudehb, M.; Kiraz, M.S. Policy specification and verification for blockchain and smart contracts in 5G networks. *ICT Express* **2020**, *6*, 43–47.
25. Macrinici, D.; Cartofeanu, C.; Gao, S. Smart contract applications within blockchain technology: A systematic mapping study. *Telemat. Inform.* **2018**, *35*, 2337–2354. [CrossRef]
26. *IPC-HERMES-9852*; The Global Standard for Machine-to-Machine Communication in SMT Assembly (v1.2). Technical Report. IPC: Bannockburn, IL, USA, 2019.
27. Tolmach, P.; Li, Y.; Lin, S.-W.; Liu, Y.; Li, Z. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* **2021**, *54*, 1–38. [CrossRef]
28. Vještica, M.; Dimitrieski, V.; Pisarić, M.; Kordić, S.; Ristić, S.; Luković, I. Towards a Formal Specification of Production Processes Suitable for Automatic Execution. *Open Comput. Sci.* **2021**, *11*, 161–179. [CrossRef]
29. Jnanamurthy, H.K.; Henskens, F.; Paul, D.; Wallis, M. Formal specification at model-level of model-driven engineering using modelling techniques. *Int. J. Comput. Appl. Technol.* **2021**, *67*, 340–350. [CrossRef]
30. Aziz, B. *Formal Analysis by Abstract Interpretation, Case Studies in Modern Protocols*; Springer: Berlin/Heidelberg, Germany, 2022.