Preprints.org

Article

Not peer-reviewed version

Formal Language for Objects' Transactions (ObTFL)

Mo Adda *

Posted Date: 13 May 2024

doi: 10.20944/preprints202405.0756.v1

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

Preprints.org is a free multidiscipline platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Formal Language for Objects' Transactions (ObTFL)

Mo Adda

University of Portsmouth; mo.adda@port.ac.uk

Abstract: The disparity between software design and implementation, especially concerning IoT security and forensics, often lacks clarity and precision. Formal languages, rooted in mathematical rules, logic, and symbols, have proven priceless for expressing specifications 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, cater to specific domains. This paper introduces a novel formal language, named ObTFL (**Obj**ect **T**ransaction **F**ormal **L**anguage), initially developed for IoT forensics and authentication purposes but later extended to encompass general distributed systems. The paper elucidates the syntax and semantics of ObTFL and presents several case studies showcasing its versatility and effectiveness.

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 cyber security, digital forensics and networks' protocols, as they enhance 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 specifications. 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 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 modelling 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

© 2024 by the author(s). Distributed under a Creative Commons CC BY license.

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 modelling 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 modelling 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 modelling 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 in 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 Scyther [16], Tamarin [13], 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 utilizes 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, a format familiar to most users. Like others, to address the specifications, validations, and requirements of interactive systems, particularly those rooted in IoT device security and forensics [20], this paper elucidates the principal syntax and semantics, providing three case studies: federated learning, and the blockchain for crypto and bitcoin networks. It concludes with the comparison with Pi-calculus on the specification of an industrial protocol.

The concept of ObTFL revolves around interacting objects directly as agents or indirectly through a communication medium as actors. 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 syntax, specified in BNF, is also outlined in [19], and the semantics, along with examples, are elaborated upon in the paper.

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. This 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 layers for the protocol. Layer 1, top one, is the application layer where all activities of the objects take place, layer 2 is the identification layer, similar to the IP-addresses, which is most of the time implicit by the object identifier or indexes, and finally

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

the layer 3 is the communication protocols, which vary from one medium to another like the TCP/UDP, Bluetooth, infrared etc. Layers 2 and 3 are not required if the objects perform no interactions within its environment, like executing locale actions. Objects interact or communicate with each other using containers' passing, and 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, sounds directly implemented within the interconnection module, IM, acting as an interface.

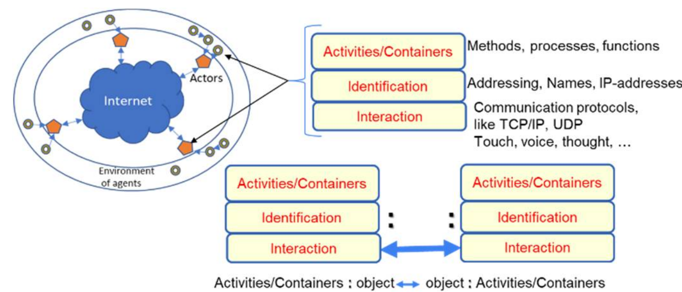


Figure 1. The environment of Objects, Containers, Activities, and Interactions.

Objects as agents or actors encompass a wide range of entities, including software, robots, vehicles, drones, sensors, and events, each playing a distinct role in the interaction. Actors, for example, act by performing activities based on software and communicate with other actors through global communication mediums. Agents, on the other hand, directly perform activities and interact with other agents or actors through hardware or software interfaces. Agents can communicate with remote agents with the support of their attached actors. For instance, an agent acting as an intruder may deploy an actor, such as a terminal, to access another actor, such as a server and manipulate its software, to commit a felony.

Agents utilize interfaces such as a keyboard or mouse to instruct a terminal to perform activities and interact with the server. For instance, a TV remote control, serving as an actor, communicates with a smart TV using local infrared communication, with the person pressing keys on the remote control acting as the agent. In these scenarios, 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, with the actor serving as the carrier of the agents identified by the code's name. The code may reside within the actor as a lodger or be permanently embedded. For instance, a virus injected into a mobile phone or server becomes a lodger, directing the actor on how to behave, like the script of a movie.

Interactions between agents and actors can vary, with physical contact, method calls, or communication through interfaces being common forms. Robots, as intelligent autonomous objects, communicate with actors via communication media and interact with other agents through interfaces such as vision, sound, or private networks. So, they can be both actors and agents at the same time, depending on their roles within their environment. Sensors provide information about their environments, while events disrupt the environment or systems, including software or hardware failures, malware attacks, or natural phenomena like earthquakes or volcanoes. Agents interact with other agents through physical contact as well. For instance, a vending machine and a person interact via push buttons and coins or payment cards. Activities such as selecting an item and receiving it after successful payment are part of the interaction between the person and the machine. Overall, agents play a crucial role in orchestrating interactions between actors, whether through physical contact or digital interfaces, facilitating various actions and transactions within interactive systems

Identifying agents interacting with actors can be challenging, from a forensic investigation point of view, especially when actors may have multiple agents, such as genuine users and potential villains. Investigations involving IoT devices, for example, require identifying not only the device used but also all associated agents. Overall, the diverse range of objects and interactions underscores the complexity of interactive systems and the importance of understanding their components for

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

effective analysis and management. The ObTFL, therefore, is based on containers and transactions with objects as agents or actors, performing activities and interactions.

2.2. Transactions

The generic format of ObTFL consists of containers and transactions. Although transactions must be used all the time, not all problems require containers. A transaction contains one or more compartments joined by the compartments' operators, labeled as **O** in Figure 2. The syntax of the formal language, ObTFL, defined with BNF grammar rules is specified in the appendix of the technical report [19]. The grammar of the language specifies that there are zero or more containers followed by a transaction, as depicted in Figure 2.

Transactions are represented by one or more compartments which are abstractions of coherent activities with 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, as summarized in appendix A.

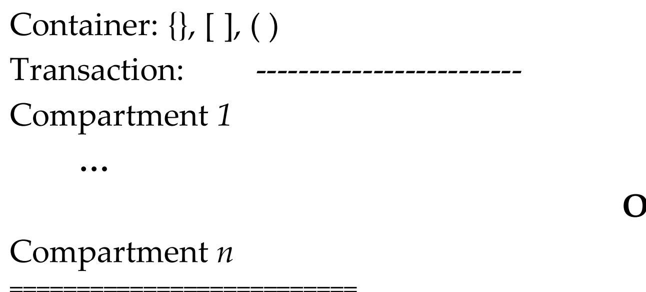


Figure 2. The structure of a transition with several compartments of common objectives.

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 each with its own distributed activities, with interactions between them. Each activity can adopt the form of fractional, linear, junction, or a combination thereof, potentially incorporating interactions. Fractional activities consist of numerators and denominators, which delineate blocks of actions. An activity may manifest as a single action, with or without interaction, or a grouping of actions interconnected by action operators, if they are internal to objects. For distributed activities and objects the compartment operators are used. The internal action operators, ***o***, are "*", "||", "||∘", "×", "+", "*+", "||+", and "||∘+", encompass sequential, parallel, rendezvous, selective exclusion, selective inclusion, consecutively, and a combination with selective inclusion "+"' with sequential, parallel, and rendezvous operators. Comprehensive details are provided in Appendix A.

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

1- Asynchronous Interaction (→): In this scenario, the sender does not anticipate a response from the recipient.

2- Strict Interaction (Synchronous) (↔): Here, a reply is expected from the recipient. If no response is received, it may lead to a deadlock situation. Fortunately, this can be mitigated by a special primitive mechanism based on timeout settings.

3- Delayed Interaction (⇁, ↩): 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's a delay.

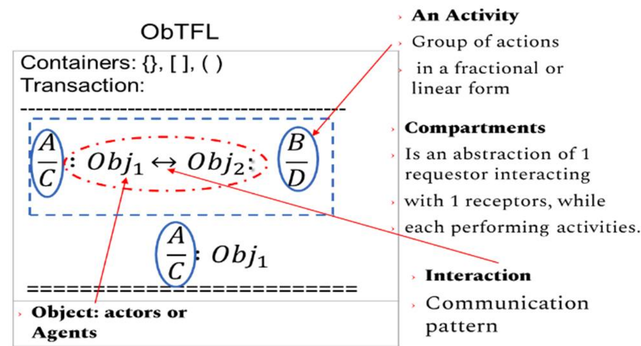


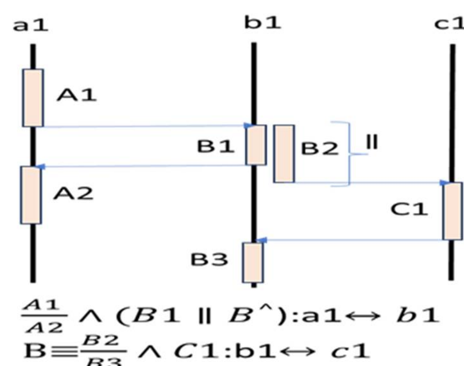
Figure 3. The content of a compartment.

2.3.1. Independent Compartments

Independent compartments, shown in Figure 3, contain activities and one agent or two objects: request and receptor. The statement, *Send(virus): hacker → server: run (receive(…)*, is an example of a single or independent statement. Where the actor hacker sends a virus, and the receptor actor executes what its received.

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 *B1* and the nested compartment, referenced by the label, *B^*, are executed in parallel, where the actor b_1 synchronously interacts with another actor c_1, anticipating a reply captured by the action B_3, as illustrated in the sequence diagram depicted in Figure 4. The notation $B_1||B_2$ in the diagram depicts their concurrent execution.



$$\frac{A1}{A2} \wedge (B1 \parallel B\hat{\ }):a1 \leftrightarrow b1$$
$$B \equiv \frac{B2}{B3} \wedge C1:b1 \leftrightarrow c1$$

Figure 4. An example of a single and referenced compartment with 3 interacting actors.

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, a nested compartment is formed, which could be written explicitly inside the host compartment or referenced externally. In this instance, the references label, *B^*, is used to capture the nested compartment, written externally.

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.

2.3.1.1. Iterative Compartments

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

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 employed, as outlined in the BNF syntax provided in Appendix D of the technical report [19]. In this analysis, we assume that indexed activities, such as A_{jk}, B_k, C_{jk}, 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. Equations 1 and 2 illustrate 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. There are $n*m$ compartments, each comprising a requestor, S_j, and a receptor, k, behaving according to the operators, $\boldsymbol{O}$, defined in section 2.2. Generally, the activity, D, is common to all objects across all compartments, while the activities, A_{jk} and C_{jk} differ for each requestor and receptor, and the activities, B_k, vary for each receptor, only.

$$\frac{A_{jk}}{C_{jk}} : \boldsymbol{O}S_{j[1\le j\le n]} \leftrightarrow \boldsymbol{O}R_{k[1\le k\le m]} : \frac{B_k}{D} \quad (1)$$

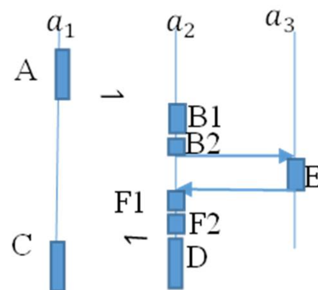
2.3.1.2. Miscelanous Compartments

Ultimately, a combination of operators' activities, objects and compartment operators, $\boldsymbol{o}$ - defined in section 2.3 - and $\boldsymbol{O}$, 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, ⊗ , compartments, A/C : $a_1 \leftrightarrow a_2$:B ⊗ D/F: $a_1 \leftrightarrow a_3$:E .

$$\left(\frac{A}{C} \otimes \frac{D}{F}\right): a_1 \leftrightarrow (a_2: B \otimes\ a_3: E)$$

2.3.4. Delayed Compartments

Delayed compartments are based on delayed interactions which consist of a series of compartments connected by the operator, $\boldsymbol{O}$, 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 5 depicts a delayed, receiving its response after the actor a_2 completes its activity F1. Actor a_2 had to request services from actor a_3 before delivering them to actor a_1. By default, a series of compartments are executed consecutively unless an operator that specifies otherwise is utilized. The first and last compartments employ a delayed interaction.



$$A: a_1 \leftrightharpoons a_2: B1$$
$$\frac{B2}{F1}: a_2 \leftrightarrow a_3: E$$
$$\frac{F2}{D}: a_2 \rightleftharpoons a_1: C$$

Figure 5. A sequence diagram for a delayed compartment.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

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, and databases. Common container types include arrays, lists, tuples, vectors, sequences, dictionaries, sets, and specialized containers such as queues, stacks, and bags. ObTFL adopts three main container types: sets, dictionaries, and sequences.

A set, denoted by { }, represents an unordered collection of unique items also called elements. These items can comprise both immutable values and mutable or immutable 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. In certain cases, items such as actors, agents, robots, and vehicles are treated as immutable.

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. Although unconventional, sequences are sometimes employed as parameters to and from actions in ObTFL notations.

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 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. For example, the statement {s} = (2, 3, 4, 4) converts a sequence to a set, resulting in S being equal to {2, 3, 4}. Conversely, the assignment operator ":=" inserts an element into a container. The statement {s} := (2, 3, 4, 4) will set the set 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, whereas *n:m[c]* indicates that the selection is based on a condition, as illustrated in section 2.5.1.2. appendix A contains a reference to the operators used with containers. Such as union, intersection and alike associated with the sets.

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 yield results stored in containers. The containers returned by actions are adaptable to various types including sets, sequences, dictionaries, or virtual containers, with the option to apply rules of promotion and demotion 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 (‖◦), exclusive selection (×), and/or inclusive selection (+), enclosed within brackets, or distributed in fractional form are termed activities. Organizing activities into numerators and denominators based on their sequence 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.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

2.5.1.1 Junctions

A junction consists of an initial action followed by brackets containing a sequence of actions separated by the exclusive operator, "×", 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 (?@{…}) (A×B×C), which evaluates the contents, {…}, received at a specific time, @, from the set container { }, and exclusively selects between actions A, B, or C.

In the formal language syntax of ObTFL, this example can be represented as the junction, ?@{…} (↻×ф×▻), where each action associated with the junction is evaluated based on intuition or predetermined conditions. The semantics of this syntax depend on the outcome of the chosen path: either repeating the current activity (↻), terminating without further progress (ф), or advancing to the next compartment (▻).

2.5.1.2 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:

$A_{[Sj = ⇞]}$ Action A performed on an object S_j in a faulty state, denoted by ⇞.

$S_{j[j=j+1]}$ 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} ⊂ {B}] introduces a 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 there 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 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*⌀* B specifies that action A is performed, but B will never be reached. However, A* ε* B denotes that A is executed, followed by B; this is equivalent to A*B. Additionally, in a compartment with $A/B : a_1 \rightarrow a_2 :$ ⌀, actor a_2 receives communication from actor a_1 but ignores it. Conversely, $A/B : a_1 \rightarrow a_2 : \varepsilon$ 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.

This 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 $↻_{j[1 \leq j \leq n]}$, it indicates repeating the action n times.

To illustrate the usage of the repeat action, let's 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, ∅, 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 minutes, $\int_{d(5)}^{@(t)}$ for three times. If she refuses, by replaying with a false, 'F', he performs a stop action that terminates the activity, ∅.

$$\frac{!Send\ \{sms\}}{((\int_{d(5)}^{@(t)} \circlearrowright)_{[1\le j\le 3]} \times \emptyset) \times ?(\ldots)(\rhd_{['T']} \times \emptyset)}$$

$$:man \leftrightarrow woman: \emptyset \times ?\{\ldots\}(!Reply('T') \times ('F') \times \emptyset)$$

2.6.2. Timer or Delay Action ($\int$).

The syntax of the timer or delay action is as follows: $\int_{d(t_2)}^{@(t_1)}$. 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 it executes the action *A*.

There are several variants to the notations of the timer. The notations $\int_{d(t2)}^{@}$, $\int_{d(t2)}^{@(t)}$ and $\int_{d(t2)}^{@(o)}$ 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 $\int_{d(-t2)}^{@(o)}$ 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, *t2*, if the action D cannot start immediately.

$$\left(\int_{d(t2)}^{@}(A \times B \times C) \times \int_{d(0)}^{@(t2)} D\right)$$

With this statement, if one of the actions A, B and C performing concurrently, has not finished within a period, t2, one of the actions, $\circlearrowright \times \emptyset \times \rhd$, associated with the timer executes. This mechanism can be used for transmission with a time out, like the TCP/IP protocol. In general, this statement has the same effect as, $\int_{d(t2)}^{@} \times ||A_{i[1\le i\le n]}$. Some actions of the *n*-parallel actions are interrupted after time, t2, if they did not complete.

$$\int_{d(t2)}^{@}(\circlearrowright \times \emptyset \times \rhd) \times (A\,||B\,||C)$$

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. 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 receives(?), regardless of the type of interaction, always implements a blocking receive scheme. In

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

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 → Receptor: Action ?(…),* the requestor waits for the message to arrive and then performs the action on it. The notation, ?(…), signifies receiving something in a container of type sequence.

2.6.4. Store (∇) and Retrieve (Δ) Actions.

Similarly to the primitives send (!) and blocking receive (?), which are used for passing containers, the primitives store (∇) and retrieve (Δ), along with the blocking retrieve (Δ°), are employed for container sharing. The blocking retrieve behaves similarly to the receive (?). These four primitives can be combined within the same model, providing various perspectives in terms of specification. To illustrate this concept, let's consider a queuing system.

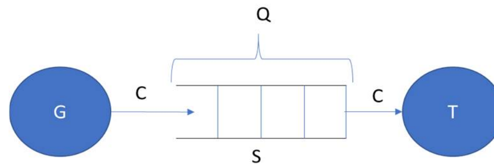


Figure 6. A single queuing system.

A generator acts as an actor or agent within a system, producing customers denoted as C at random intervals, typically every average time interval represented by t1. These 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. 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 7. 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 ⊣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 ⊢Q and spends a random time t3 to service them. It repeats the same process or terminates if the queue is empty.

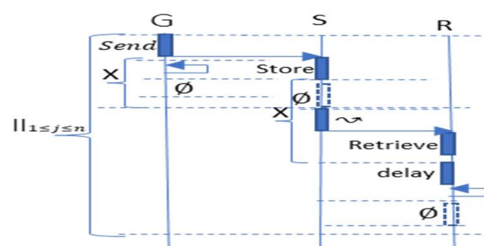


Figure 7. Specification of a single server single generators queuing system.

$$\frac{\int_{d(Rand(t1))}^{@} * ! Send(C)}{\circlearrowright} : G \rightarrow S : \nabla\, Put(?\,(\ldots)^{@}, \dashv Q)\left(A^{\wedge}{}_{[Q=()]} \times \emptyset\right)$$

$$Ⓘ$$

$$A \equiv \text{ Notice} : \mathrm{S} \rightarrow T : \frac{\Delta Get(C, \vdash Q)\left(\int_{d(Rand(t3))}^{@(t2)} Service(C) \times \emptyset_{[Q=()]}\right)}{\circlearrowright}$$

2.6.5. Fail (ǂ) 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

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

entities or, in cases involving intelligent agents, autonomously executing self-repair protocols. These events may arise internally, such as software or hardware 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 (↝). Figure 8 depicts the various steps to invoke external or internal repairs, where an external event with action, U1, or internal incident with action, *U2*, occurs and disrupts the actor that's Fails (ǂ) after performing the actions, *A1*. The actor, *a2*, resume the execution of action, *A2*, if it is repaired by an external actor, *b*, or it perform its own self-repair action, *B*.

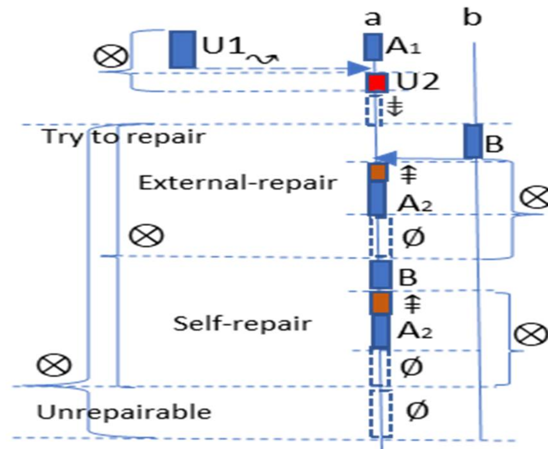


Figure 8. UML modified sequenced diagram with failures and repairs.

$$\begin{pmatrix} U1{:}\,E \rightsquigarrow \text{a: A1} * ǂ \\ \otimes \\ a{:}\,A1 * U2 * ǂ \end{pmatrix}$$

$$\circledast$$

$$\begin{pmatrix} \begin{pmatrix} B{:}\,b \rightsquigarrow \text{a: (} ǂ * \text{A2} \times \emptyset) \\ \otimes \\ a{:}\,B(ǂ * \text{A2} \times \emptyset) \end{pmatrix} \\ \otimes \\ \phi \end{pmatrix}$$

3. Case Studies

This section introduces three case studies to demonstrate the features of the formal language ObTFL. The case studies are about Federated learning with data protection, attacks in ad hoc networks and blockchain in cryptocurrency.

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. It's important to note that the specifics of this

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

process are not defined by ObTFL and can be implemented using various methods, as long as the container parameters are effectively transmitted.

Upon receiving updates from the hospitals simultaneously, the central server performs another 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 specified learning rate. The central server calculates new parameters for the global model, which are then sent back to the local server of each hospital to repeat multiple epochs until the global model reaches an acceptable level of performance. Specifically, the central server computes new weights and biases using the formula for the *UpGlobalModel*: *new-Weight = oldWeight - learningRate * averageWeight and newBias = oldBias - learningRate * averageBias*. The average action takes a set of weights and biases as tuples and produces a type with *old Weight* and *bias* to be used by the *UpGlobalModel* action to evaluate the *new weight* and *bias*. This iterative process is repeated *m* times, demonstrating the use of iterative compartments.

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

3.2. Case 3: Blockchain for Cryptocurrency

In a blockchain protocol, such as that used in cryptocurrency transactions, the process begins with a sender node initiating a transaction. This 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.

For simplicity, let's define the transaction as a dictionary with specific keys. The signed transaction is then broadcasted 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. These miners uphold copies of the blockchain ledger, composed of interconnected blocks. Each block, depending on its size, has the capacity to accommodate numerous transactions, typically ranging from 2000 to 4000 transactions per block. The block header 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's worth noting that nonce values may or may not be included in all blockchain networks. The memory pool, known as MemPool, serves as a temporary repository for transactions that have been received but not yet validated and included in a block. During the mining process, new transactions continue to arrive and are queued in the MemPool. Transactions are typically stored in the memory pool in the order of their receipt. Some blockchain systems employ prioritization schemes to sequence transactions based on factors like fees and size. Due to the finite memory size of the MemPool, a memory management scheme is implemented, whereby unconfirmed transactions may be removed from the MemPool to make room for incoming transactions.

$Tran_{ij[1\le j\le m_i]}$ = [sender: …, recipient: …, amount: …, sig: Enc (Hash (sig $\dotminus$ $Tran_{ij}$), K_i^-), others: …]

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

Pr = (fee) × (size)

$$Block_{j[1\le j\le q]} = Overhead_j \cup Tran_{ji[1\le i\le m_i:p_j]}$$

$$Overhead_j = [bHeader{:}\ \ldots, pbhHash : \ldots, timeStamp{:}\ \ldots, size{:}\ \ldots, once{:}\ 0, tHash{:}\ \ldots\ ]$$

$$MemPool_{k[1\le k\le n:p]} = (\ )$$

$$Ledger_{i[1\le i\le n]} = (Block_{j[1\le j\le q]})$$

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

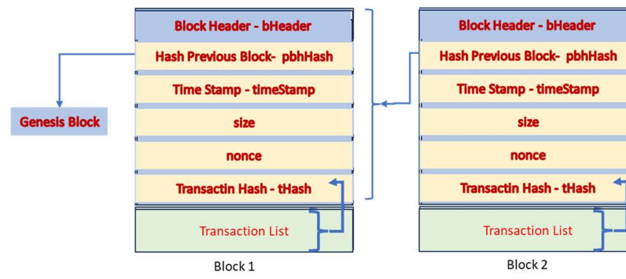


Figure 9. two blocks chained.

$$\frac{(\hat{A_1} \parallel \hat{A_2} \parallel \hat{A_3})}{\circlearrowright} : ⦶\, N_{i[1\leq i\leq n]} \leftrightarrow ⦶\, M_{i[1\leq i\leq n:q]} : \frac{(\hat{B_1} \parallel \hat{B_2} \parallel \hat{B_3})}{\circlearrowright}$$

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

$$A_2 \equiv \nabla insert(?\,(\ldots), PendingList) \; * \text{MonitorBlockchainLength}$$

$$A_3 \equiv \mathbf{\Delta}^{\circ}\text{Extract}(Block_{j[\,C^{\wedge}\,]}, PendingList) * \phi\text{PendingList} * \; \nabla insert(Block_j, Ledger_{i[i=n:1]})$$

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

$$B_1 \equiv ?\left[Tran_{ij}\right] * \bowtie \text{Compare}(Dec(Tran_{ij}.\text{sig}, K_i^{+}), \text{Hash}\ (\text{sig} \dotdiv \ Tran_{ij}))$$
$$(\phi \times \ \nabla insert(Tran_{ij}, pr, MemPool))$$

$$B_2 \equiv \left(\nabla(\Delta^{\circ}(tran, \vdash MemPool), Block_{j[j=j+1\ \&\ 1\leq j\leq q]}\right) * Block_j.tHash{:}= Merk\ \left\{Tran_{ji[1\leq i\leq m_i:p_j]}\right\}$$

$$B_3 \equiv Inc(Block_j.once) * (h) \coloneqq Hash(Overhead_j)(\varepsilon \times !_{[h=V^{\wedge}]}\ \text{Send}(Block_j)\,)$$

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

The Merk activity hashes the transactions in pairs until the root is reaches which contains the final hash. It is also possible to use a different algorithm where the list of all transactions is hashed all together. Miners send blocks when they finish solving the puzzle, B_3, these blocks although valid they are put in a pending list, A_2, until nodes independently verify and extend the longest valid chain by adding new blocks to it. The length of the blockchain is a proxy for cumulative proof of work.

4. Discussion

This section explores the potential and versatility of the formal specification language introduced in this paper, comparing it to the **π**-calculus for specifying the Hermes protocol v1.2. The IPC-HERMES-9852 standard has emerged as a prominent protocol in the electronic manufacturing industry 4.0. To provide a comparative analysis with previous work [21], the focus is on the transportation of Printed Circuit Boards (PCBs) from one machine to another (M-2-M). The revised sequence diagram, depicted in Figure 10, illustrates the operational dynamics of this protocol, reflecting the alterations discussed.

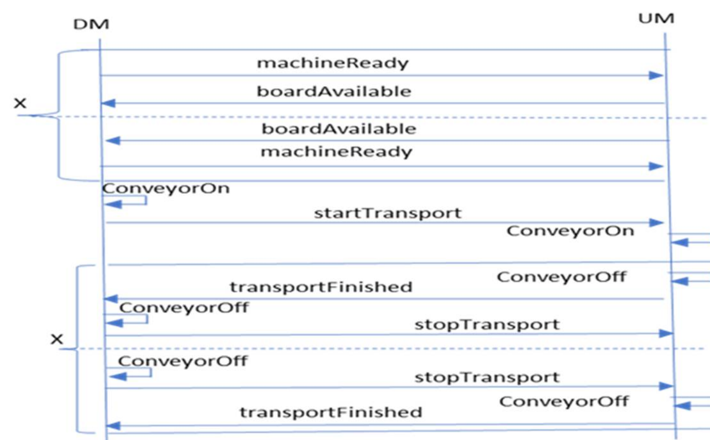


Figure 10. The normal operation of Hermes [21] protocol.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

The model of Hermes protocol v1.2 is specified in [22] using the **π**-calculus, as

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

where

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

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

where

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

The equivalent specification of ObTFL syntax introduced in this paper can be written as,

$$\frac{!\,machineReady}{?^{@}\,(\dots)\ast conveyorOn \ast !\,StartTransport} : DM \leftrightarrow UM : \frac{?^{@}\,(\dots)\ast\ !\,boardAvailable}{?^{@}\,(\dots)\ast conveyorOn}$$

$$\otimes$$

$$\frac{!\,boardAvailable}{?^{@}\,(\dots)\ast\ ?^{@}\,(\dots)} : UM \leftrightarrow DM : \frac{?^{@}\,(\dots)\ast\ !\,machineReady}{conveyorOn \ast\ !\,StartTransport}$$

$$\frac{conveyorOff \ast\ !\,stopTransport}{?\,(\dots)} : DM \leftrightarrow UM : ?\,(\dots)\ast\ ConveyorOff \ast\ !\,transportFinished$$

$$\otimes$$

$$\frac{conveyorOff \ast\ !\,transportFinished}{?^{@}\,(\dots)} : UM$$

Upon comparing the two formal specifications, it is evident that the **π**-calculus notations offer a more intuitive approach. Reference [22] 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 solely indicates silent intervals in the example, which vary, leading to reduced accuracy. Moreover, errors in [22] are specified independently and in more complex formats.

In contrast, the grammar rules for ObTFL make it inherently more expressive, with interactions clearly delineating which actors are involved in specific actions. 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, contingent upon synchronizations between the primitive send (!) and receive (?) operations.

5. Conclusions

This paper introduces a novel formal specification language tailored for use in distributed systems, where interactions and parallel activities drive the core functionality. Additionally, it extends to interacting with IoT devices, addressing needs such as authentication, confidentiality, integrity, and forensic investigations. Crucially, its applicability transcends IoT environments, encompassing scenarios involving interactions and various activities within disturbed systems and networks.

The language's semantic foundation relies on mathematical models and symbols, ensuring robustness and precision. What distinguishes this language is its simplified syntax compared to other

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

formal languages, making it accessible to a wider audience interested in specifying and documenting problems within distributed system domains. This accessibility is vital as it 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 comprehend the structure, organization, and readability of the ObTFL semantics.

Appendix A

Symbols	Primitive action operators, by symbol, label or symbol and label
@, d	Time to specify the time an action starts, ***o*** means 0, a number, or nothing means any time, if omitted. A reference to the a starting time that is maintained. it means the starting time is not important. When used actions, If d is 0 or omitted, it means immediately. If it is the duration is infinity, ∞, it means wait forever, in all other cases a time slot is supplied to wait for that duration. They cannot be used directly with containers only through the actions. It both, @ and d, are omitted, the action start now.
	Status actions apply to objects
⇞ ⇟	Declare the status of an object to be, ⇞, idle, down, broken, unavailable, ⇟ active.
	Delay action
$\int^{@}, \int^{@}_{d}, \int^{@}_{d()[]}, \int^{@}_{[]}$	Delay action is used to make an elapsed time. Action delay starting from time now, o, duration for a duration, d, which is 0 or ∞, or any identifier holding time. If d is omitted it means the duration is 0 by default, do not wait. Delay action and can be used to create a delay or with the operator × to create the time limit for an action to complete or stop. Or can be delayed until a condition occurs. The timer can also be interrupted by another object with the use of signal interaction.
	Container passing actions
!@ , ?@	Send/Output a container or Receive/Input the content of a container or its reference ^ . The action can record its starting time, if @(time) is specified, otherwise it is undefined. They can be synchronous and buffered or unbuffered. This behaviour is also supported by the interaction notation. The receive by default is a blocking receive, this action wait forever until an event or a container has been received.
	Container Sharing actions
∇@, Δ@ , Δ°	Put/Store and Get/Retrieve and blocking Get/Retrieve. The application can also use additional actions support the store and retrieve synchronously. They can be buffered and unbuffered. Container passing and sharing can be missed. If the item cannot be inserted or removed nothing happens, unless it is captured by a junction.
:= , ∸	Default of := in the set is insert anywhere, and in sequence is insert at the back, unless specifies with ⊣, ⊢,.
	Decision action operators
▹, ⊵	Next compartment or activity, next iterative compartment or activity, next transaction.
⥀, ⥁	Repeat the current activity, fractional statement, or compartment or an iterative activity, compartment, or a transaction, an activity could be just a fractional compartment.
∅, ∅	stop current action, iteration, stop transaction.
ϵ	Action that does nothing – sometimes used in junctions, ϵ/↻ does nothing then call the repeat action which keeps looping forever, ∅/↻ terminates .
	Operator action for container
⇒,⇏, ⋈, ⏀	Search, remove, compare items from a container, clear the whole container.
^	Pointer or links to a container waiting to be filled by an action, the reference to an action, or a compartment.
	Actions' operators

,₊, ‖, ‖⁺, ‖°⁺, ×, +, ‖°	Sequential, sequential from a group of "ored"-activities, parallel, rendezvous, parallel group of "ored"-activities, parallel "or" with rendezvous, selective only one, selective a group (or). The x operator is prioritized more than one actions are ready to perform, the priority starts from the left hand side, that is the action at the left is selected and so on.
	Compartment Operators
⊛,⊗,⊕, ⦶,⦶⁺,⦶° ⦶°⁺,⊛°	Consecutive, selective, group selective, simultaneous, rendezvous operators used with compartments.
	Container declarations
[ ] { } ( ) = [ ], { }, ( ) S := {}, [ ], ( ) = [●], {●}, (●) […] {… } (…)	Dictionary container – Set container – Sequence container. Container status, specifying empty. Clearing a container with (:=) The container is full. Container with something, used with an input action.
	Container arithmetic operators
⊣, ⊢,	Back/middle/front of a container, used for insertion, when the container can be a queue or a stack.
⫛	Stripping removes the membership of the items, and they need to be allocated into another container if they are not containers.
	Fuzzy set operators
∧ ∧ ⟶ ⊕ ⌝	And, Or, Imply, aggregation and negation operators on members of fuzzy sets
	Container additional operator actions.
∩,∪,∸ …	All set theory symbols.
	Interaction symbols
↔ ,→, ↼,⇁ ↜	Synchronous, Asynchronous, Delayed interactions and signal for interrupting an object
↛,↮ ↛ n,n ↮ n	Asynchronous and Synchronous, can be used mainly with shared containers databases … The indexes n, can represent the relationship 1 to many or many to many ect… For anthology and distributed relational database
Symbols	These symbols are used inside the [Condition]
⊂, ⊆, ∉, ∈, ∃, ∀, ∴ , $\eta(C)$	Container operators, inclusion, there exist, such that or where, for all ect … cardinality of a container
'T', 'F'	Boolean values
&, ǀ , ~, x	Boolean operators, "and", "or" and negation
"string"	A word, sentence a set of characters
Numbers, indexes, and identifiers	Integer, decimals, or container values
+, *, -, %, /, ÷	Arithmetic operators
$1 \le index \le n$ $< \ > \le \ \ge \ = \neq \simeq$	Relational operators
n:m:p:.. *n:m:p:..[condition]*	Group random extractions: p items, selected from m items from n items of a container, or object in the virtual space. Group selective extractions: with a nested condition

References

1. D. 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), June 8-10, **2020**, pp. 290-297.
2. Jacobs, S. *Beginning XML with DOM and Ajax: from Novice to Professional*. Published A press, **2020**.
3. Kaye, R. *The Mathematical Theory of Predicate Logic*. Published by Cambridge University Press, **2020**.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 13 May 2024 doi:10.20944/preprints202405.0756.v1

4. Stubblebine, T. *Regular Expressions Pocket Reference.* Published by O'Reilly Media, **2021**.
5. S. B. Cooper, and M. Soskova, "Turing machines and Computational Theory," Published by Spring, **2020**.
6. Bernardo, M.; R. D. Nicola, R.D.; M. Loreti, M. *Process Algebra and Probabilistic Models: Performance and Dependability Analysis*. Published by Springer, **2020.**
7. Whitney, J.; Gifford. C; Pantoja, M. *Distributed execution of communicating sequential process-style concurrency: Golang case study.* The Journal of Supercomputing, Springer, **2019**.
8. Vrancken, L.M. *The algebra of communicating processes with empty processes*. Elsevier, Theoretical Computer Science, 15 May **1997**, Volume 177, Issue 2, pp. 287-328.
9. Friedman, A. *Communicating with Process Calculus.* A Major Qualifying Project submitted to the Faculty of Worcester Polytechnic Institute, August 24, **2022** – May 3, **2023**.
10. Nicollin, X.; Sifakis, J. *An overview and synthesis on timed process algebras. Timed Specification and Verification.* Conference paper, Computer Aided Verification, Springer. First Online: 01 January **2005**, pp 376–398,
11. Umer, M.; Ali, A. *Automated Analysis of the Security of the IoT using Pi-calculus.* Publish by Springer, **2021**.
12. Awais, M.; Yasin, M.; Umar, A. I. *Formal Verification of a Secure Message Protocol using the Applied Spi-Calculus*. In the Journal of Computer Science and Technology, Springer, **2021**.
13. Vasconcelos, V. T.; Kok, J. N. *A Theory of Distributed Program Composition*. Information and Computation, **2003**.
14. Y Liao, Y.; Yeaser, A.; Yang, B.; Tung, J.; Hashemi, E. *Unsupervised fault detection and recovery for intelligent robotic rollators*. Elsevier, Robotics and Autonomous Systems, December **2021**, Volume 146.
15. Al Fikri, M.; Ramli, K.; Sudiana, D. *Formal Verification of the Authentication and Voice Communication Protocol Security on Device X Using Scyther Tool.* IOP Conference Series: Materials Science and Engineering, The 5th International Conference on Information Technology and Digital Applications (ICITDA 2020) 13th-14th November **2020**, Yogyakarta, Indonesia, Volume 1077.
16. Cortier, V.; Delaune, S.; Dreier, J.; Klein, E. *Automatic generation of sources lemmas in Tamarin: Towards automatic proofs of security protocols. Computer Security* – ESORICS **2020**, Springer, pp. 3–22.
17. Blanchet. B.; Cheval, V.; Cortier, V. *ProVerif with Lemmas, Induction, Fast Subsumption, and Much More.* **2022** IEEE Symposium on Security and Privacy (SP).
18. Yogesh. P. R.; Devane Satish, D. *Formal Verification of Secure Evidence Collection Protocol using BAN Logic and AVISPA.* Elsevier, **2020**, Volume 167, pp. 1334-1344.
19. Adda, M. *ObTFL formal language for Spider Network*. Internal Technical Report, University of Portsmouth, **2023**.
20. Scheidt, N.; Adda, M. *Internet of Things:* Threats, Landscape, and Countermeasures. CRC Press Inc, **2021**, pp. 137-166.
21. Initiative T.H.S., *IPC-HERMES-9852: The global standard for machine-to-machine communication in SMT assenbly(v1.2*), Technocal report, IPC 2019.
22. Aziz, B. *Formal Analysis by Abstract Interpretation, case studies in modern protocols*, Publish by Springer, **2022**

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.