

D4.2: Report on the Initial Toolset

Revision: v.1.0

Work package	WP4
Task	T4.1, T4.2, T4.3 and T4.4
Due date	30/June/2024
Submission date	30/June/2024
Deliverable lead	Nobuko Yoshida (UOXF)
Version	1.0
Authors	Nobuko Yoshida (UOXF), Ping Hou (UOXF), Alceste Scalas (DTU), António Ravara (NOVA), Carla Ferreira (NOVA), João Costa Seco (NOVA), Sebastian Mödersheim (DTU), Simon Tobias Lund (DTU), Silvia Ghilezan (UNS), Ivan Prokic (UNS), Simona Prokic (UNS)
Reviewers	Roland Kuhn (Actyx), Sotirios Spantideas (NKUA)
Abstract	This document reports D4.2—the initial toolset for communication, data, AI/ML, and security analyses is provided, describing the developed analyses for a subset of the properties outlined in D4.1 across T4.1–T4.4. Additionally, any changes to the identified set of properties in D4.1 are discussed, and other new properties not covered in D4.1 are introduced.
Keywords	tool sets, properties



DISCLAIMER



Funded by
the European Union

Funded by the European Union (TARDIS, 101093006). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

COPYRIGHT NOTICE

© 2023 - 2025 TaRDIS Consortium

Project funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable:	to specify R, DEM, DEC, DATA, DMP, ETHICS, SECURITY, OTHER*	
Dissemination Level		
PU	Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page)	✓
SEN	Sensitive, limited under the conditions of the Grant Agreement	
Classified R-UE/ EU-R	EU RESTRICTED under the Commission Decision No2015/444	
Classified C-UE/ EU-C	EU CONFIDENTIAL under the Commission Decision No2015/444	
Classified S-UE/ EU-S	EU SECRET under the Commission Decision No2015/444	

* R: Document, report (excluding the periodic and final reports)

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

DATA: Data sets, microdata, etc.

DMP: Data management plan

ETHICS: Deliverables related to ethics issues.

SECURITY: Deliverables related to security issues

OTHER: Software, technical diagram, algorithms, models, etc.



EXECUTIVE SUMMARY

The TaRDIS project aims to develop a distributed programming toolbox that simplifies the development of decentralised applications across varied environments. Work Package 4 (WP4) is dedicated to pioneering formal analyses to assess the soundness, security, and reliability of heterogeneous swarms. These analyses are specifically designed for TaRDIS models, ensuring they meet the desired *security*, *data integrity*, *AI coordination*, and *communication* properties, aligning with the TaRDIS use cases and requirements.

In Deliverable D4.1, the properties related to TaRDIS use cases requiring analysis and verification were categorised, and both existing and advanced verification techniques for validating these properties were explored.

In this report, the initial toolset for communication, data, AI/ML, and security analyses is provided, describing the developed analyses for a subset of the properties outlined in D4.1 across T4.1 – T4.4. Additionally, any changes to the identified set of properties in D4.1 are discussed, and other new properties not covered in D4.1 are introduced.

The main contributions of this report are outlined below:

- Descriptions of tools for communication behaviours analysis, data integrity, security verification, and federated learning analysis, along with their developed analyses for the properties described in D4.1.
- Modifications to properties identified in D4.1, tailored specifically to these tools, as well as additional properties suited for analysis by these tools.

The following are some key highlights from the M18 report in terms of academic publications, effectively addressing the challenges associated with formally analysing the soundness, security, and reliability of heterogeneous swarms:

- UOXF
 - Nobuko Yoshida and Ping Hou: [Less is More Revisited: Association with Global Multiparty Session Types](#). To appear in The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II, 2024.
 - Lorenzo Gheri and Nobuko Yoshida: [Hybrid Multiparty Session Types: Compositionality for Protocol Specification through Endpoint Projection](#). Proc. ACM Program. Lang. 7(OOPSLA1): 112-142 (2023).
- DTU
 - Simon Tobias Lund and Sebastian Mödersheim, *Dolev-Yao Information Flow*, submitted, 2024
 - Andreas Hess, Sebastian Mödersheim, Achim Brucker, and Anders Schlichtkrull: [PSPSP: A Tool for Automated Verification of Stateful Protocols in Isabelle/HOL](#), submitted, 2024.
 - Sebastian Mödersheim and Siyu Chen: [Accountable Banking Transactions](#), Open Identity Summit 2024, to appear.
 - Jens Kanstrup Larsen, Roberto Guanciale, Philipp Haller, Alceste Scalas: [P4R-Type: A Verified API for P4 Control Plane Programs](#). Proc. ACM Program. Lang. 7(OOPSLA2): 1935-1963 (2023)
 - Christian Bartolo Burlò, Adrian Francalanza, Alceste Scalas, Emilio Tuosto: [COTS: Connected OpenAPI Test Synthesis for RESTful Applications](#). In: COORDINATION 2024. Lecture Notes in Computer Science, vol 14676. Springer.

- NOVA
 - Marco Giunti, Hervé Paulino, António Ravara: [Anticipation of Method Execution in Mixed Consistency Systems](#). SAC 2023: 1394-1401.
 - Hervé Paulino, Ana Almeida Matos, Jan Cederquist, Marco Giunti, João Matos, António Ravara: [AtomiS: Data-Centric Synchronization Made Practical](#). Proc. ACM Program. Lang. 7 (OOPSLA2): 116-145 (2023).
- UNS
 - Milos Simić, Jovana Dedeić, Milan Stojkov, Ivan Prokić: [A Hierarchical Namespace Approach for Multi-Tenancy in Distributed Clouds](#). IEEE Access 12: 32597-32617 (2024)
 - Simona Prokić: *Probabilistic reasoning in computation and simple type theory*. PhD Thesis, University of Novi Sad (2024).
- ACT
 - Roland Kuhn, Hernán Melgratti, Emilio Tuosto: [Behavioural Types for Local-First Software](#). ECOOP 2023.

TABLE OF CONTENTS

Executive Summary	3
Table of Contents	5
Abbreviations	7
1. INTRODUCTION	8
1.1 Tool Specifications and Modified Properties.....	8
1.2 Results Summary.....	9
1.3 Deliverable Structure.....	10
2. TOOLSET	11
2.1 Communication Behaviours Analysis.....	11
2.1.1 WorkflowEditor & Actyx Middleware (Actyx).....	11
2.1.2 Compositional Verification of Swarm Protocols (DTU and Actyx).....	13
2.1.3 Fair Join Pattern Matching (DTU).....	13
2.1.4 Verified APIs for Software-Defined Networking (DTU).....	14
2.1.5 Model-Based Testing of Swarm Applications (DTU).....	15
2.1.6 Scribble (UOXF).....	17
2.1.7 Java Typestate Checker (NOVA).....	19
2.2 Data Convergence and Integrity.....	23
2.2.1 VeriFx (NOVA).....	23
2.2.2 Ant (NOVA).....	23
2.2.3 AtomiS (NOVA).....	24
2.3 Security Verification.....	25
2.3.1 Channel Information Flow (DTU and NOVA).....	25
2.3.2 PSPSP (DTU).....	33
2.3.3 Cryptographic Interpretations of Choreographies (DTU).....	36
2.4 Orchestration, Verification and Regarding Properties Integrated with WP5 and WP637	
2.4.1 Correct Orchestration of Federated Learning Algorithms (UNS, WP4 for WP5)37	
2.4.2 Correct Hierarchical Namespaces (UNS, WP4 for WP6 T6.3).....	39
3. REVISIONS OF IDENTIFIED PROPERTIES IN D4.1	42
3.1 Properties for Communication Behaviours.....	42
3.1.1 WorkflowEditor & Actyx Middleware.....	42
3.1.2 Compositional Verification of Swarm Protocols.....	43
3.1.3 Fair Join Pattern Matching.....	43
3.1.4 Verified APIs for Software-Defined Networking.....	44
3.1.5 Model-Based Testing of Swarm Applications.....	44
3.1.6 Java Typestate Checker.....	44
3.2 Properties for Data Management and Replication.....	44
3.3 Properties for Security.....	45
4. CONCLUSIONS	46
5. Bibliography	47

LIST OF FIGURES

Figure 1: Sample program using DTU's actor library with join pattern matching.....	14
Figure 2: Sample P4 control program using the P4RType library.....	15
Figure 3: Sample test model for the COTS tool.....	16
Figure 4: Top-down MPST methodology.....	17
Figure 5: Travel agency protocol as a sequence diagram.....	18
Figure 6: Travel agency protocol in Scribble.....	18
Figure 7: EFSM for TravelAgency role A.....	18
Figure 8: Screenshot of the NuScr web interface, showing an adder protocol.....	19
Figure 9: Snippet of JaTyC code.....	20
Figure 10: An e-bank application of Ant.....	24
Figure 11: AtomiS approach.....	24
Figure 12: Generated code of AtomiS.....	25
Figure 13: DCR choreography, modelling a fraction of purchase process.....	26
Figure 14: Security lattice of purchase process.....	27
Figure 15: Security annotated DCR graph of purchase process.....	28
Figure 16: Security annotated DCR graph of purchase process without leaks.....	29
Figure 17: Potential end-state after execution of program.....	32
Figure 18: Example of namespaces resource graph.....	40
Figure 19: Example of application resource graph.....	40
Figure 20: Creation of namespaces.....	41
Figure 21: Example of a DPO based direct graph transformation.....	41

ABBREVIATIONS

ML	Machine Learning
FL	Federated Learning
FLA(s)	Federated Learning Algorithm(s)
PTB-FLA	Python Testbed for Federated Learning Algorithms
ST	Session Types
MPST	Multiparty Session Types
FSM(s)	Finite State Machine(s)
CFSM(s)	Communicating Finite State Machine(s)
EFSM(s)	Endpoint Finite State Machine(s)
API(s)	Application Programming Interface(s)
IDE	Integrated Development Environment
QoS	Quality of Service
SDN	Software-Defined Networking
SUT	System-Under-Test
RDT(s)	Replicated Data Type(s)
CRDT(s)	Conflict-free Replicated Data Type(s)
DCCC	Data Coupling and Control Coupling
IF	Information Flow
IFC	Information Flow Channel
DCR	Dynamic Condition Response
TPM	Trusted Platform Module
CSP	Communicating Sequential Processes Calculus
PAT	Process Analysis Toolkit

1. INTRODUCTION

The main objective of Work Package 4 (WP4) is to develop novel formal analysis tools to ensure that a heterogeneous swarm is *sound*, *secure*, and *reliable*. These tools will be applied to the TaRDIS models to verify that key properties such as *security*, *data integrity*, *AI coordination*, and *communication* are satisfied, with specific properties chosen based on TaRDIS use cases and requirements. WP4 aims to enable the safe use of AI and data primitives developed in WP5 and WP6, and the developed tools will be integrated into the TaRDIS APIs, IDE, and AI optimisation framework.

1.1 TOOL SPECIFICATIONS AND MODIFIED PROPERTIES

This document reports on the M18 delivery (D4.2), highlighting the development of analysis tools for a subset of the properties delivered in D4.1, "Report on the Desirable Properties for Analysis". Specifically, it focuses on tools developed for analysing communication behaviours, ensuring data convergence and integrity requirements, verifying security and privacy, and orchestrating federated learning for heterogeneous swarms. These developments are outlined according to the four specified tasks below.

Communication Behaviours Analysis

- *WorkflowEditor and Actyx Middleware* (by Actyx): utilise Actyx middleware to formally describe and analyse workflows between swarm participants.
- *Compositional Verification of Swarm Protocols* (by DTU and Actyx): verify the well-formedness and deadlock-freedom of a composition of two given swarm protocols, which are individually well-formed and deadlock-free.
- *Fair Join Pattern Matching* (by DTU): specify fair and deterministic join pattern matching for actor-based systems.
- *P4R-Type* (by DTU): a verified P4Runtime API in Scala 3 to perform static checks on programs controlling P4-based software-defined networks.
- *COTS* (by DTU): model-based testing of swarm applications.
- *Scribble* (by UOXF): a description language for application-level protocols among communicating systems.
- *JaTyC* (by NOVA): a Java typestate checker for statically verifying memory-safety, protocol compliance, and protocol completion.

Data Convergence and Integrity

- *VeriFx* (by NOVA): libraries for implementing and verifying Conflict-free Replicated Data Types and operational transformation functions.
- *Ant* (by NOVA): an automated method to identify operations that can safely commute, prioritising those that do not require inter-replica coordination, ensuring data integrity and consistency.
- *AtomiS* (by NOVA): a DCCC approach to mandate specific types for parameters and return values in interface definitions, as well as for fields in class definitions.

Security Verification

- *Channel Information Flow* (by DTU and NOVA)
 - *DCR Choreographies with IFC*: a mechanism to prevent side channels by considering the timing of events, ensuring data confidentiality.
 - Information Flow Channel: a framework for securely extending information flow analysis to systems with communication over an untrusted network.
- *PSPSP* (by DTU): a low-level language for security protocols.
- *Cryptographic Interpretations of Choreographies* (by DTU): a cryptographic choreography language emphasising the practical application of cryptographic operations by agents.

Federated Learning Orchestration

- *Correct Orchestration of Federated Learning Algorithms* (by UNS): formal verification of correctness of centralised and decentralised FL algorithms
- *Correct Hierarchical Namespaces* (by UNS): a hierarchical namespaces model encouraging the organised and efficient distribution of resources.

Additionally, this document includes any modifications to the properties identified in D4.1. The modified and new properties are detailed below for each tool.

- WorkflowEditor and Actyx Middleware: deadlock-freedom, resilience through replication, liveness, eventual consensus, protocol conformance, perfect availability, fault tolerance, and termination of failure.
- Compositional Verification of Swarm Protocols: compositional verification.
- Fair Join Pattern Matching: mailbox communication safety and fair join pattern matching.
- P4R-Type: safety w.r.t. network configurations, deadlock-freedom, and liveness.
- COTS: test correctness and fault detection soundness.
- JaTyC: memory-safety, protocol compliance, and protocol completion.

1.2 RESULTS SUMMARY

Our key contributions are:

- We specify the analysis tools developed for the properties outlined in D3.1, spanning T4.1–T4.4.
- We outline any revisions made to identified properties in D3.1, as well as newly introduced properties, related to each tool.

Overall, we consolidate various analysis tools developed for the TaRDIS models to ensure the satisfaction of desirable properties such as security, data integrity, AI coordination, and communication. These tools will be integrated as a toolset into the TaRDIS APIs, IDE, and AI optimisation framework.

1.3 DELIVERABLE STRUCTURE

The report starts by presenting comprehensive specifications for the tools developed to analyse the desirable properties outlined in D4.1 in Section 2. Subsequently, Section 3 specifies any modifications made to these properties and introduces newly identified ones for each tool.

2. TOOLSET

This section provides an in-depth exploration of tools designed to analyse communication behaviours, enforce data convergence and integrity requirements, verify security and privacy, and orchestrate federated learning across heterogeneous swarms.

2.1 COMMUNICATION BEHAVIOURS ANALYSIS

2.1.1 WorkflowEditor & Actyx Middleware (Actyx)

The aim of this work is to formally describe and analyse workflows between swarm participants using the Actyx middleware. This peer-to-peer system lives completely in the realm of edge computing and allows processes distributed across a swarm to reliably send each other information updates in the form of durable event streams. Another important feature of this middleware is that it uses a logical clock mechanism to assign each event a timestamp that captures its causal dependencies with minimal effort and allows a total order to be established between events that does not require any coordination. In summary, Actyx aims for full availability, which implies that it tolerates weak consistency (it achieves *eventual consensus*, a notion known from blockchains like Bitcoin).

While application developers are free to design any interaction or workflow they desire, not all such workflows achieve eventual consensus when the access of participants to some information (i.e. certain event types) is restricted. Eventual consensus is a very useful property to have in a distributed system: its absence means that different swarm participants may have diverging world views without ever reconciling them. Given that it is extremely challenging to program a system with unbounded inconsistency, we deem eventual consensus the least we need to offer in order to be successful—most programmers today routinely use the much stronger notion of strict consensus, which is built into all traditional relational databases. The basic problem we attacked is thus how to retain eventual consensus in a swarm where not every participant is allowed to know everything.

For the full details we refer you to the ECOOP paper [1]. The main idea is to describe a workflow as a state machine, starting out in its initial state and proceeding to new states via transitions that are each labelled by a command name, the participant role that is allowed to invoke the command, and the sequence of event types that are emitted by invoking the command. This representation lends itself well to a diagrammatic visual representation, an aspect that has been used by Actyx with its customers to great success. One important aspect is that each role is assumed to be present within an execution of the workflow with an arbitrary positive number of replicas; in other words, the computational model accommodates the use of redundancy in real-world applications (such as in factories) to achieve operational resilience in the face of hardware and software failures.

Once the workflow is designed from a global view as a swarm protocol, it is projected into locally executable state machines, one for each participating role. These machines differ not only in the commands offered for invocation in each state, they may also lack the input of some of the event types that are present in the global swarm protocol—this filtering mechanism is called the *role's subscription*. Each swarm participant that partakes in the

workflow picks one such role, instantiates the corresponding machine, and uses this as a device to interpret the state of the workflow according to the event logs that are locally available. These logs contain not only the locally emitted events but also the events emitted on other swarm devices and disseminated across the swarm using the Actyx middleware's peer-to-peer protocols. Using the total event order we know that eventually the same log sequence will be available to everyone, filtered locally by each role's subscription.

It is easily visible that not all participants may come to the same overall conclusion—e.g. regarding whether the workflow is still running or has completed—if too many events are filtered out. We identified the following three well-formedness conditions to counter this issue:

1. *causal consistency* requires that a role must subscribe to at least one of the event types emitted by each of the commands it is allowed to invoke, and that it must subscribe to at least one event type emitted by the preceding command in the swarm protocol; these constraints ensure that the local machine awaits its turn and then moves on, preventing infinite command invocation
2. *choice determinacy* requires that a role that remains active later in the protocol must subscribe to the first event type emitted after a choice (i.e. a branch in the diagram), so as to follow along the general progress of the execution
3. *confusion freeness* requires that an event type that immediately follows a choice cannot appear elsewhere in the protocol; otherwise that other appearance may accidentally be confused with making a choice, leading some roles astray in their understanding of the general progress of the execution

While these constraints are sufficient to establish eventual consensus, they are not necessary. In other words, the constraints are stricter than they need to be; the reason is that we have not yet been able to prove the effectiveness of weaker constraints. We are working on such improvements and on the corresponding proofs to give protocol designers more flexibility and in particular to allow more information to be hidden from participants while still keeping their understanding of workflow progress intact.

Another angle that we're following is that workflows in factories often turn out to require one specific machine to perform some of the protocol steps: for example, once the material is on a logistics robot, only that particular robot can deliver the material to its destination. Picking out a single instance of a role and restricting some protocol transitions to that instance will allow designers to more precisely model system behaviour and lead to more faithful implementation of the local agents. On the other hand, adding such constraints makes the system susceptible to the failure of exactly that machine. Hence we are also considering the addition of timeouts and rollbacks whenever such restricted commands are being used. This has also been corroborated by Actyx customers who would like to gain support from an analysis tool to ensure that compensating actions are properly employed in all failure cases—not only in those caused by Actyx's weak consensus model but also those caused by real-world problems.

Current state and next steps: The underlying Actyx middleware is an existing and commercially used product that will receive some updates and new features based on WP6 work that is unrelated to the analytic capabilities of the WorkflowEditor. The well-formedness checking (machine-check) of declared workflows for swarm protocols (machine-runner) is

implemented and commercially used based on the ECOOP paper [1]. These will be enhanced as described in the above two paragraphs within the second half of the TaRDIS project. The visual editor does not yet exist and will be implemented later this year.

2.1.2 Compositional Verification of Swarm Protocols (DTU and Actyx)

DTU is currently working (in collaboration with Actyx) on the compositional verification of swarm protocols — i.e., given two swarm protocols G and G' which are individually well-formed and deadlock-free, determine whether their parallel composition $G|G'$ is also a well-formed and deadlock-free swarm protocol, without analysing the whole combined swarm protocol “from scratch” - and without requiring changes to the actual implementations of the swarm participants projected from G and G' . More specifically, DTU is researching sufficient conditions that, given two correct swarm protocols G and G' , ensure the correctness of their composition $G|G'$; it is also researching the *necessary* conditions for correct composition. This would allow developers to maintain a library of well-formed swarm protocols and participant implementations, that can then be combined without introducing deadlocks or communication errors.

This research work will be implemented as a series verification routines that will be integrated in the swarm design tool currently being implemented by Actyx (see previous section 2.1.1).

2.1.3 Fair Join Pattern Matching (DTU)

Join patterns provide a promising approach to the development of concurrent and distributed applications where different components may need to interact using complex message combinations and conditions. A join pattern (with conditional guard) is reminiscent of a clause in a typical pattern matching construct: it has the form “ J if $\gamma \Rightarrow P$ ” — where J is a message pattern describing a combination of incoming messages and binding zero or more variables, and γ is a guard, i.e., a boolean expression that may use the variables bound in J . A program using join patterns can wait until a desired combination of messages arrives (in any order); when some of the incoming messages are matched by the message pattern J and (their payloads) satisfy the guard γ , the process P is executed.

The theoretical foundation of join patterns was introduced in the join calculus [2], and subsequent research extended the approach in multiple directions. DTU is developing a novel specification of *fair and deterministic join pattern matching for actor-based systems*, i.e., a formal definition of how an actor should perform join pattern matching to select messages out of its mailbox, guaranteeing that older messages are always eventually consumed if they can be matched. DTU has also developed a direct implementation of the fair matching specification, and a novel *stateful, tree-based join pattern matching algorithm* that is proven correct w.r.t. the fair matching specification above [3]. Such algorithms have different performance characteristics, and DTU is evaluating their efficiency and suitability in various settings. Both implementations are research prototypes, and will be further refined and optimised during the rest of the project.

The two implementations above will be contributed (as a ready-to-use library for the Scala 3 programming language) to the TaRDIS toolbox: by construction, any program using the provided join pattern libraries will enjoy the fair and deterministic join pattern matching guarantees. The first intended use of such libraries in the TaRDIS toolbox will be a monitoring

program for the Actyx use case: the program will observe the flow of messages describing ongoing activity on a factory shop floor, and match some combinations of messages to maintain statistics and report quality of service (QoS) issues. The program is outlined in Figure 1 below.

```

1 def monitor() = Actor[Event, Unit] {
2   receive { (self: ActorRef[Event]) => {
3     case ( Fault(_, fid1, _, ts1),
4           Fix(_, fid2, ts2) ) if fid1 == fid2 =>
5       updateMaintenanceStats(ts1, ts2)
6       Continue
7
8     case ( Fault(mid, fid1, description, ts1),
9           Fault(_, fid2, _, ts2),
10          Fix(_, fid3, ts3) ) if fid2 == fid3 && ts2 > ts1 + TEN_MIN =>
11       updateMaintenanceStats(ts2, ts3)
12       log(s"Fault ${fid1} ignored for ${(ts2 - ts1) / ONE_MIN} minutes")
13       self ! DelayedFault(mid, fid1, description, ts1) // For later processing
14       Continue
15
16     case ( DelayedFault(_, fid1, _, ts1),
17           Fix(_, fid2, ts2) ) if fid1 == fid2 =>
18       updateMaintenanceStats(ts1, ts2)
19       Continue
20
21     case Shutdown() => Stop()
22   } }
23 }

```

Figure 1: Sample program using DTU's actor library with join pattern matching.

2.1.4 Verified APIs for Software-Defined Networking (DTU)

Many modern network switches and routers provide Software-Defined Networking (SDN) capabilities, which allow for writing control programs that can define and alter the network's packet processing rules. The de facto standard for programming SDN devices is the P4 language. However, the flexibility and power of P4 (and SDN more generally) gives rise to important risks: errors in SDN control programs can compromise the availability of networks, leaving them in a non-functional state.

For this reason, DTU has developed P4R-Type [4], a novel verified P4Runtime API for Scala 3 that performs static checks for programs that control P4-based software-defined networks. If a control program using P4R-Type can be compiled, then all its operations that may modify the P4 network configuration are guaranteed to be compliant with the network specification (i.e. all updates will respect the packet processing tables, allowed actions, and action parameters).

Figure 2 shows a sample control program written in Scala 3 using P4R-Type. The program connects to two different P4-enabled network routers with different configurations, by instantiating the connections *s1* and *s2*. Then, the program attempts to insert new packet processing rules in both routers. The first *insert* operation is correct (no errors are reported), whereas the second one is wrong: the red highlight on the rule “NoAction” means that the rule is not valid for the selected packet processing table. The error is caught at compile-time and reported via an IDE (in this case, Visual Studio Code).

```
@main def forward_c1() =
  val s1 = config1.connect(0, "127.0.0.1", 50051)
  val s2 = config1.connect(1, "127.0.0.1", 50052)

  insert(s1, TableEntry(
    "Process.ipv4_lpm",
    Some("hdr.ipv4.dstAddr", LPM(bytes(10,0,1,1), 32)),
    "Process.ipv4_forward",
    (("dstAddr", bytes(8,0,0,0,1,17)), ("port", bytes(1))), 1
  ))

  insert(s1, TableEntry(
    "Process.ipv4_lpm",
    Some("hdr.ipv4.dstAddr", LPM(bytes(10,0,1,1), 32)),
    "NoAction", // <-- This action only exists in Process.firewall
    (), 1
  ))
```

Figure 2: Sample P4 control program using the P4RType library.

P4R-Type is currently being investigated as a possible component of the TaRDIS toolbox. A possible application is to regulate swarm membership at the network level, using SDN: the network may drop all packets sent by new swarm members, until they authenticate themselves; the authentication packets, in turn, would be forwarded to a trusted service (written using P4R-Type) that would update the network configuration only after a successful authentication. By using P4R-Type, the authentication service is guaranteed to respect the network configuration.

2.1.5 Model-Based Testing of Swarm Applications (DTU)

The development of a TaRDIS internal application (outlined in Deliverable D3.1) is based on the specification of a global swarm protocol, which is then projected into local workflows, which in turn act as “blueprints” for individual swarm components. This specification-based approach provides a promising starting point for the study of *model-based testing of swarm applications*.

Generally speaking, model-based testing [5] uses a model to automatically generate randomised test cases conforming to the model itself; then, it uses suitable tooling to observe whether the component-under-test behaves as expected by the test model. In the context of TaRDIS, the test model could be based on either a swarm protocol specification, or on a

projected workflow; then, a series of randomised test cases could emit and read events according to the model (e.g. by simulating one or more components in the swarm), checking whether the component-under-test reacts to its inputs by emitting the type of events expected by the model. Model-based testing can provide a useful complement to static verification: if some component of a swarm cannot be statically verified (e.g. because its implementation cannot be inspected), then the behaviour of the component could be checked by performing randomised model-based tests.

DTU has recently studied and developed a novel methodology for the model-based testing of web applications, and a tool (called COTS) based on such a methodology [6]. In this line of work, the system-under-test (SUT) provides a RESTful [7] API, and the test models are *session types* augmented with assertions about the data being received and transmitted by the SUT. For instance, a COTS test model can express that “*if a warehouse item with id X is created, and then deleted, then the id X must not appear in the result of subsequent queries*”.

Figure 3 shows a simple example of the COTS test model (called S_shop) for an e-shop application. The model says that the e-shop protocol expects the client to send an ‘addCust’ request (to add a customer to the e-shop), and then await for a response ‘C201’ (denoting success) carrying the id of the newly-added customer (‘custId’). Then, the client can recursively add information about the credit card or address of the newly-added customer, retrieve the customer information, or delete the customer and terminate. The test model contains assertions like ‘checkCustomer’ (line 9) to ensure the customer data retrieved from the e-shop (c2) matches the data generated at the beginning of the interaction (c1 on line 1). COTS can use this test model to generate random tests that simulate e-shop clients interacting with an e-shop SUT; if the SUT does not respond according to the test model (e.g., if it crashes or sends back invalid customer data), the fault is immediately detected and reported.

```

1 S_shop = !addCust(apiKey: String(genApiKey), c1: Customer(genCustInfo)).
2   ?C201(custId: String).
3   rec X.(
4     +{ !addCard(apiKey, custId, card: Card(getCardInfo)) .
5       ?C201(CardId: String) . X,
6       !addAddr(apiKey, custId, addr: Address(getAddrInfo)) .
7         ?C201(addressId: String) . X,
8       !getCust(apiKey, custId) .
9         ?C200(c2: Customer)<checkCustomer(c1,c2)> . X,
10      !deleteCust(apiKey, custId).?C204() . end
11    }
12  )

```

Figure 3: Sample test model for the COTS tool.

DTU is evaluating whether COTS could be the basis for a similar tool for model-based testing (to be included in the TaRDIS toolkit) that communicates through the TaRDIS toolkit, and uses as test model a swarm protocol or workflow, possibly augmented with assertions about the data being transmitted/received.

2.1.6 Scribble (UOXF)

Scribble is a language to describe application-level protocols among communicating systems. Building on the theory of Multiparty Session Types (MPST) [8], Scribble tackles the challenges of adapting and implementing session types to meet real-world usage requirements. Scribble and Swarm protocols serve different roles in managing distributed systems. Scribble is used to specify and verify communication patterns between software components, ensuring that messages are exchanged correctly between different parts of a system. On the other hand, Swarm protocols focus on managing containerised applications by defining how containers are deployed, networked, and maintained, ensuring the applications run smoothly and efficiently. While Scribble ensures correct communication, Swarm protocols ensure efficient operation and management of containerised applications.

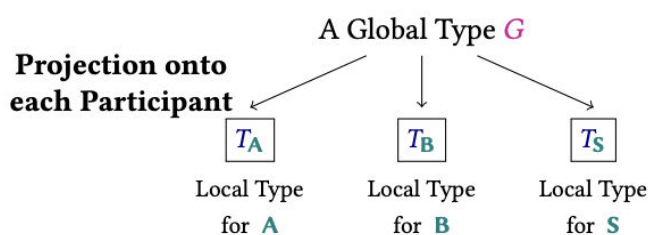


Figure 4: Top-down MPST methodology.

Global Protocol Specification. Following the top-down MPST design methodology, as illustrated in Figure 4, the Scribble framework starts from specifying a global protocol, a description of the full protocol of interaction in a multiparty communication session from a neutral perspective, i.e. all potential and necessary message exchanges between all participants from the start of a session until completion.

Local Protocol Projection. Subsequently, Scribble syntactically *projects* a valid source global protocol to a *local protocol* for each role. Projection essentially extracts the parts of the global protocol in which the target role is directly involved, giving the localised behaviour required of each role in order for a session to execute correctly as a whole. A further validation step is performed on each projection of the source protocol for role-sensitive properties, such as reachability of all relevant protocol states per role. The validation also restricts recursive protocols to tail recursion. A valid global protocol with valid projections for each role is a *well-formed* protocol.

Endpoint FSM. Building on a formal correspondence between syntactic local MPST and communicating FSMs, Scribble can transform the projection of any well-formed protocol for each of its roles to an equivalent *Endpoint FSM* (EFSM). The nodes in an EFSM delineate the state space of the endpoint in the protocol, and the transitions the explicit I/O actions between protocol states. The local type of an endpoint can be then used in the code generation process, to generate APIs that are *correct by construction*.

Consider the following scenario, as shown in Figure 5, where an online travel agency operates a “travelling with a friend” scheme. It starts when a traveller (B) suggests a trip destination to their friend (A), who then queries the travel agency (S) if the trip is available. If

so, the friends discuss among themselves whether to accept or reject the quoted price. If the trip was unavailable, the friends start again with a new destination.

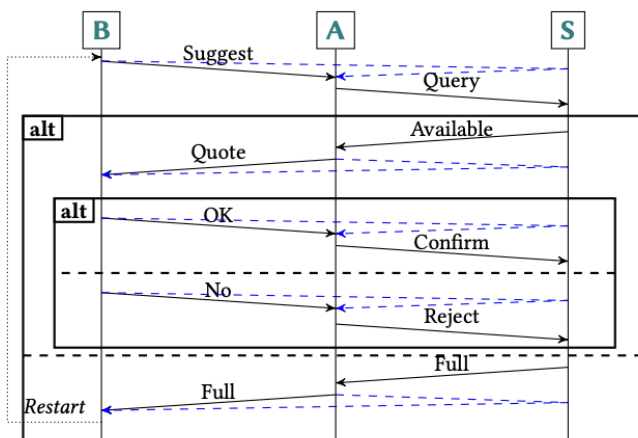


Figure 5: Travel agency protocol as a sequence diagram.

The travel agency protocol is specified in Scribble as shown in Figure 6:

```

1 global protocol TravelAgency(role A, role B, role S)
2 { Suggest(string) from B to A; //friend suggests place
3   Query(string) from A to S;
4   choice at S
5     { Available(number) from S to A;
6       Quote(number) from A to B; //check price with friend
7       choice at B
8         { OK(number) from B to A;
9           Confirm(credentials) from A to S; }
10        or { No() from B to A;
11            Reject() from A to S; } }
12   or { Full() from S to A; Full() from A to B;
13       do TravelAgency(A, B, S); } }
    
```

Figure 6: Travel agency protocol in Scribble.

while the EFSM for role A is demonstrated in Figure 7:

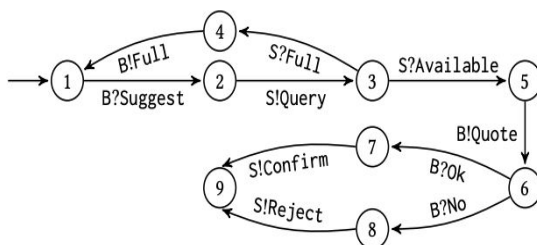
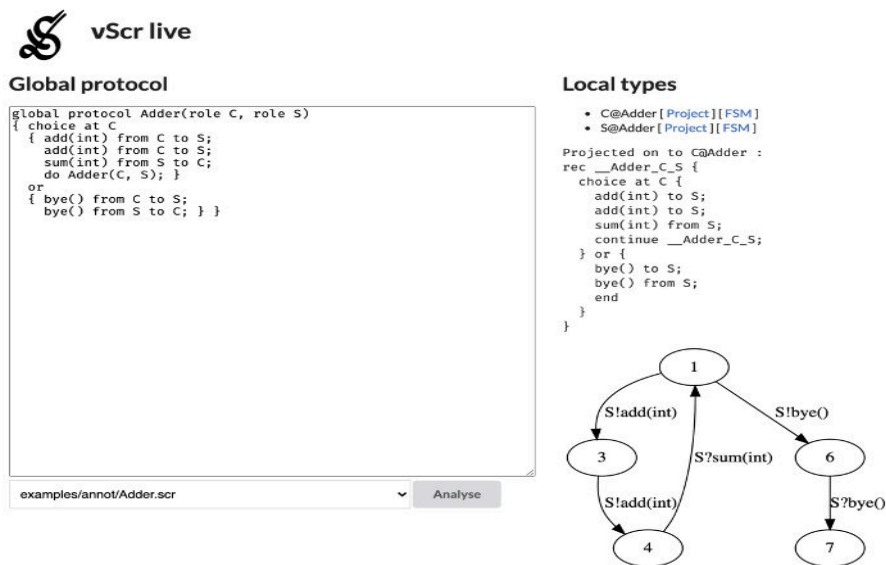


Figure 7: EFSM for TravelAgency role A.

As further work, NuScr is a toolchain for multiparty protocols, designed to handle protocols written in the Scribble language. NuScr implements the core part of the Scribble language, with various extensions to the original MPST. This allows protocols written in the Scribble description language to be accepted by NuScr and then converted into an MPST global type.

From a global type, NuScr can project onto a specified participant to obtain their local type and subsequently generate the corresponding communication finite state machine (CFSM), which can be used for API generation. Additionally, NuScr can generate code for implementing the participant in various programming languages from their local type or CFSM. NuScr can be used either as a standalone command-line application or as an OCaml library for manipulating multiparty protocols. NuScr also features a [web interface](#), allowing users to perform quick prototyping directly in their browsers without the need for installation. See Figure 8 for a screenshot illustrating an Adder Protocol.



vScr live

Global protocol

```
global protocol Adder(role C, role S)
{
  choice at C
  {
    add(int) from C to S;
    add(int) from C to S;
    sum(int) from S to C;
    do Adder(C, S);
  }
  or
  {
    bye() from C to S;
    bye() from S to C;
  }
}
```

Local types

- C@Adder[Project][FSM]
- S@Adder[Project][FSM]

Projected on to C@Adder :

```
rec __Adder_C_S {
  choice at C {
    add(int) to S;
    add(int) to S;
    sum(int) from S;
    continue __Adder_C_S;
  }
  or {
    bye() to S;
    bye() from S;
  }
}
end
}
```

CFSM Diagram:

```

graph TD
  1((1)) -- "S!add(int)" --> 3((3))
  3 -- "S!add(int)" --> 4((4))
  4 -- "S!add(int)" --> 1
  1 -- "S?sum(int)" --> 6((6))
  6 -- "S?bye()" --> 7((7))
  
```

Figure 8: Screenshot of the NuScr web interface, showing an adder protocol.

In the context of TaRDIS, Scribble, NuScr, and their extensions can be used to specify, verify, and validate communication protocols. Here is a specific application: to support the configuration of application components at runtime, which is a primary focus of T6.3 in TaRDIS, T4.4 introduces a model for hierarchical namespaces that promotes the proper organisation and redistribution of resources [9]. All protocols presented for this model are validated using the toolchain [10], an extension of Scribble with explicit connection actions to support protocols with optional and dynamic participants.

2.1.7 Java Typestate Checker (NOVA)

The Java Typestate Checker ([JaTyC](#)) tool statically verifies that, by attaching a protocol declaration to each class in the code, the developer gets:

- memory-safety: no null-pointer exceptions nor memory leaks;
- protocol compliance: client code executes respecting each object correct usage;
- protocol completion: all objects used until the end of their protocols (and released).

When a Java program runs: (i) sequences of method calls follow the object's protocols; (ii) objects' protocols are completed; (iii) subclasses' instances respect the protocol of their superclasses.

With this, JaTyC assists developers in getting their object usage protocols correct, being able to avoid some crashes before the code is executed. This is crucial for application soundness in general and for cybersecurity in particular, since crashes may give unintended access to a machine. For example, in some places, it is a common occurrence that ATMs show the operating system's desktop interface if the bank's application crashes. Another application to cybersecurity is to analyse systems where security clearance levels are needed in order to execute operations: this can be enforced in JaTyC by e.g. requiring that each operation's execution is preceded by the successful verification of the operation's security clearance.

Concretely, JaTyC helps developing sound code, providing means to specify and guarantee correct API behaviour, preventing at the same time critical code vulnerabilities like [CWE-306](#) (Missing Authentication for Critical Function), [CWE-754](#) (Improper Check for Unusual or Exceptional Conditions), or [CWE-841](#) (Improper Enforcement of Behavioral Workflow).

To illustrate, consider the snippet of code in Figure 9, where the class File is supposed to be used as described by the finite-state machine (aka [typestate](#)) below that declares available transitions from a state are external choices (the circles) and internal choices on possible method results (the diamonds):

```
File f = new File(); System.out.println(f.read());
```

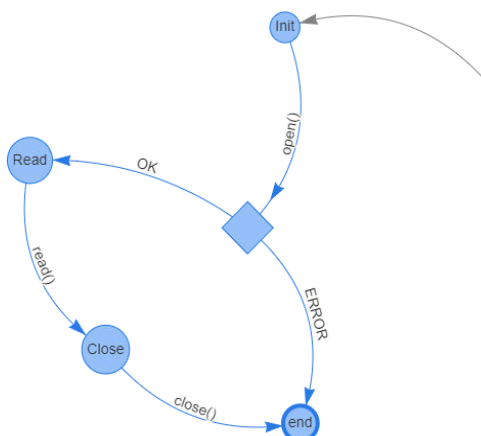


Figure 9: Snippet of JaTyC code.

With JaTyC installed as an expansion of their IDE, the developer that wrote the code above would get either when typing or at compile time the message 'error: Cannot call [read] on State{File, Init}'. The static typechecker detects the error: one must open before read.

If the developer uses the file with the code

```
switch (f.open()) {
  case OK:
    System.out.println(f.read());
  break;
```

```

    case ERROR:
        break;
}

```

the error now is '[f] did not complete its protocol'. In case OK, after reading the file one must close it.

Resource leakage is also detected: the method close below needs to close the file.

```

public class LineReader {
    // error: [this.file] did not complete its protocol
    private @Nullable FileReader file;
    private int curr;
    public Status open(String filename) {
        /* ... */
        file = new FileReader(filename);
        curr = file.read(); /* ... */
    }
    public String read() {
        /* ... */ curr = file.read(); /* ... */
    }
    public boolean eof() { return curr == -1; }
    public void close() {}
}

```

Even in the presence of polymorphic code, the guarantees stay valid: JaTyC supports inheritance (paper to appear in ECOOP'24). Consider a class Bulb with the following typestate

```

typestate Bulb {
    DISCONN = {
        boolean connect(): <true: CONN, false: DISCONN>,
        drop: end
    }

    CONN = {
        void disconnect(): DISCONN,
        void setBrightness(int): CONN
    }
}

```

and a subclass FunnyBulb ruled by

```

typestate FunnyBulb {
    DISCONN = {
        boolean connect(): <true: STD_CONN, false: DISCONN>,

```

```

    drop: end
}
STD_CONN = {
    void disconnect(): DISCONN,
    void setBrightness(int): STD_CONN,
    Mode switchMode(): <RND: RND_CONN, STD: STD_CONN>,
    void setColor(String): STD_CONN
}
RND_CONN = {
    void disconnect(): DISCONN,
    void setBrightness(int): RND_CONN,
    Mode switchMode(): <RND: RND_CONN, STD: STD_CONN>,
    void randomColor(): RND_CONN
}
}
}

```

Up/down casting is supported. The code below is well-typed.

```

public class ClientCode {
    public static void example() {
        FunnyBulb f = new FunnyBulb(); // DISCONN
        while (!f.connect()) {} // STD_CONN
        f.switchMode(); // STD_CONN | RND_CONN
        setBrightness(f);
    }

    private static void setBrightness(@Requires("CONN") Bulb b) {
        if (b instanceof FunnyBulb && ((FunnyBulb) b).switchMode() == Mode.RND) {
            ((FunnyBulb) b).randomColor(); // RND_CONN
        }
        b.setBrightness(10); // CONN
        b.disconnect(); // end
    }
}

```

The tool reached TRL 5 (it is thus ready to be used and will be added to the TaRDIS toolbox) and received the Availability and Functional Badges at [ECOOP'24](#). In the [repository](#) one finds several examples and case studies used to validate the approach. In the context of TaRDIS, the tool can be used to specify, verify, and validate any application developed in Java that is API based.

2.2 DATA CONVERGENCE AND INTEGRITY

Distributed applications (widely common these days) need to replicate data to make it available. The problem is: *how to keep replicated data (eventually) consistent in a scenario where the communication topology is dynamic due to connectivity issues, devices (inherently heterogeneous) come and go at any moment?*

The correctness of an application results from the correctness of its operations. Local views on data should not diverge in an irreconcilable way, so, a certain degree of consistency is necessary. Strong consistency in such a dynamic scenario is unattainable. Depending on the nature of the swarm system, what makes sense is to ask for either eventual or causal consistency.

Eventually, possible conflicts must be solved; how? set specific moments for each replica to do so and coordinate operations only if correctness cannot be guaranteed otherwise. In the context of TaRDIS, we have available two deductive approaches to develop correct applications dealing with replicated data.

In short, conflict resolution mechanisms should guarantee (at least some) of these requirements: (1) safety in sequential execution; (2) (causal and/or eventually) convergence; and (3) the precondition of each operation should be stable under the effect of any other concurrent operation. A typical example is a shared set:: inserts can always happen, as sets do not have repeated elements (although the local view of the set may be outdated), but removals require causal and/or eventual "coordination" (if one cannot remove a non-existing value, as in some contexts, this can block or crash the device).

2.2.1 VeriFx (NOVA)

This tool [11] provides libraries for implementing and verifying Conflict-free Replicated Data Types (CRDTs) and operational transformation functions. These libraries implement the general execution model of those approaches and define their correctness properties. RDTs verified with VeriFx can be transpiled to mainstream languages (currently Scala and JavaScript).

2.2.2 Ant (NOVA)

The tool [12] provides an automatic approach to determine operations that can safely (from the data integrity and consistency point of view) commute, allowing to execute first operations not requiring inter-replica coordination. We develop a language-based static analysis to extract information at compile-time that can be used by the run-time support to decide on call anticipations of operations in replicas without compromising consistency. We illustrate the formal analysis on several paradigmatic examples and briefly present a proof-of-concept implementation in Java.

Consider that:

Locally permissible ops are immediately executed

Strongly consistent ops require coordination among replicated sites

An e-bank application could behave as in Figure 10, using our static analysis and runtime

support.

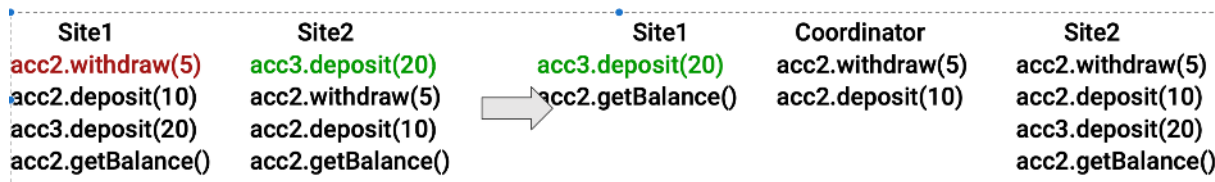


Figure 10: An e-bank application of Ant.

To be usable by third-party developers, the tool needs further work that we will pursue in the coming period. In the context of TaRDIS, a possible adaptation is to consider alternative policies to strong consistency, as the approach can be made parametric.

2.2.3 AtomiS (NOVA)

To ensure data integrity in concurrent applications, we developed AtomiS [13], a new DCCC approach that requires only qualifying types of parameters and return values in interface definitions, and of fields in class definitions. The latter may also be abstracted away in type parameters, rendering class implementations virtually annotation-free. From this high-level specification, a static analysis infers the atomicity constraints that are local to each method, considering valid only the method variants that are consistent with the specification, and performs code generation for all valid variants of each method. The generated code is then the target for automatic injection of concurrency control primitives that are responsible for *ensuring the absence of data-races, atomicity-violations, and deadlocks*.

Basically, the idea is:

- mark resources which need to be accessed in mutual exclusion;
- a type-checking and inference system ensures race freedom.

Figure 11 describes the approach.

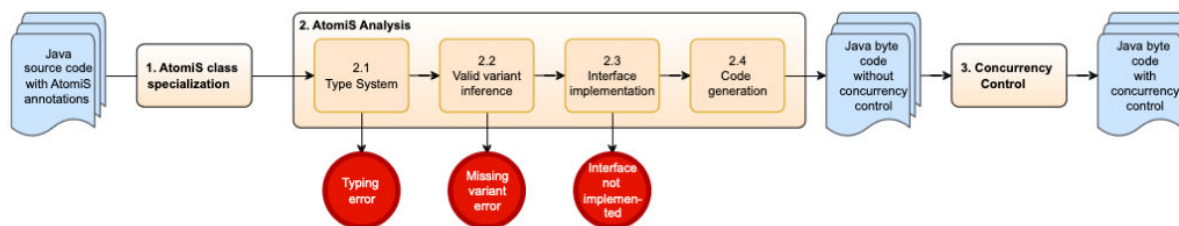


Figure 11: AtomiS approach.

We illustrate how the code looks like in Figure 12, with a simple yet widely used piece of Java code: a concurrent list. The annotations identify the resources to be atomically accessed. The static analysis checks the soundness of the code, produces automatically variants of code and new annotations to guide the generation of concurrency control.


```

1 interface ListAtomic {
2     void add(@Atomic Object element);
3     @Atomic Object get(int pos);
4     boolean containsAll(@All ListAtomic other);
5 }

7 class NodeOfAtomic {
8     @Atomic Object value;
9     NodeOfAtomic next, prev;
10 }

11 class ConcurrentListOfAtomic implements
    ListOfAtomic {
12     @Atomic NodeOfAtomic head, tail;

14     void add(Object element) { ... }
15     ...
16 }

```

Figure 12: Generated code of *Atomis*.

To be usable by third-party developers, the tool needs further work that we will pursue in the coming period. In the context of TaRDIS, the tool can be used whenever developing concurrent applications with Java. Moreover, the can be made parametric on the resource control policy, and thus generalised to other data qualifications.

2.3 SECURITY VERIFICATION

2.3.1 Channel Information Flow (DTU and NOVA)

Information flow control is a language-based technique anchored on the property of noninterference for detecting and preventing confidentiality breaches in target systems through the systematic labelling and tracking of information. There are multiple models for information flow control, with lattice-based models being the most common. These models allow for the hierarchic compartmentalisation of information according to the security lattice, a finite, lower and upper-bounded, partially ordered set of security levels. Information flow policies are detailed in specifications defining the secrecy (security level) of a system's data items and information receptacles (entities holding/receiving information like variables and communication channels). The enforcement of information flow policies boils down to preventing secret information from going to less secret receptacles and is achieved at runtime or compile time through mechanisms such as information flow monitors or type systems. The base model enjoys many improvements, such as value-dependent security levels, which enhance the flexibility of the lattice and allow for finer-grained security policies.

2.3.1.1 DCR Choreographies with IFC

In software development, handling data confidentiality in systems implementing complex business processes is challenging, even more so when considering distributed systems with multiple interacting entities. Mainstream approaches to defining business processes do not adequately address confidentiality properties, leaving room for potential information leaks.

We focus on data and time-aware declarative models for specifying interactions in business processes [14]. We extend dynamic condition response (DCR) graphs, capturing choreographies with data and time, developing a mechanism based on progress-sensitive, time-sensitive noninterference to prevent side channels due to the early or late execution of events, thus ensuring data confidentiality throughout the process.

Consider the DCR choreography depicted in Figure 13, modelling a fraction of a purchase process where the buyer asks multiple sellers for quotes on a product and selects the best

offer (at most one quote) among the (possibly empty) set of available quotes. The buyer must always register the outcome of the process in its ledger: no quotes received, no quotes selected, or the chosen quote. Also, sellers must not know about each other's offers.

Each node represents an event, a stateful entity whose state expresses if it is included/excluded, pending/not pending, has executed, and its value and time steps since its last execution (if previously executed). Event execution may entail the transmission of messages between a sender (on top) and one or more receivers (on bottom). On the graph, solid-line events are included, while dashed-line events are excluded, and events with a "!" on the top left corner are pending events which have to execute for the process to stabilise.

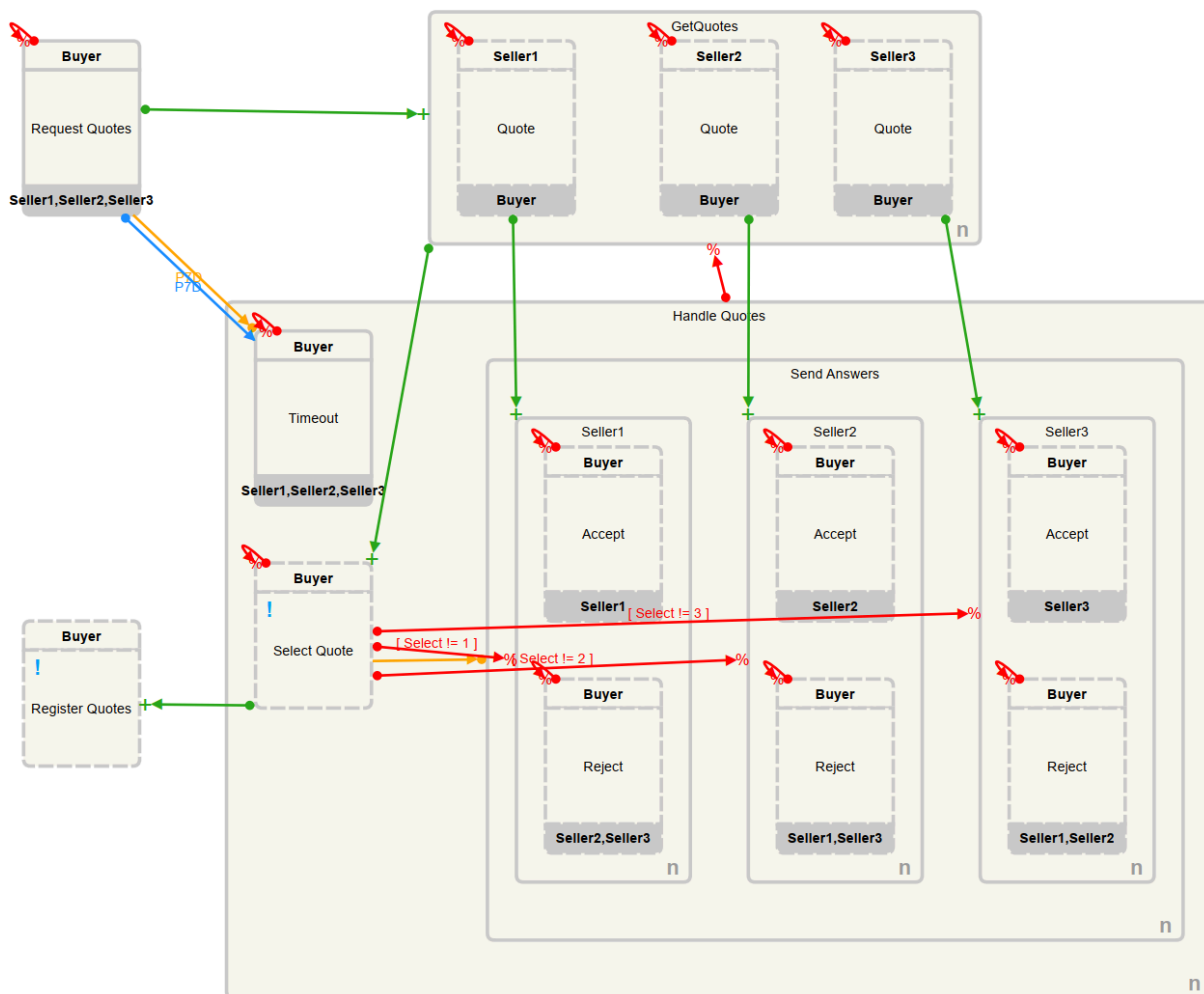
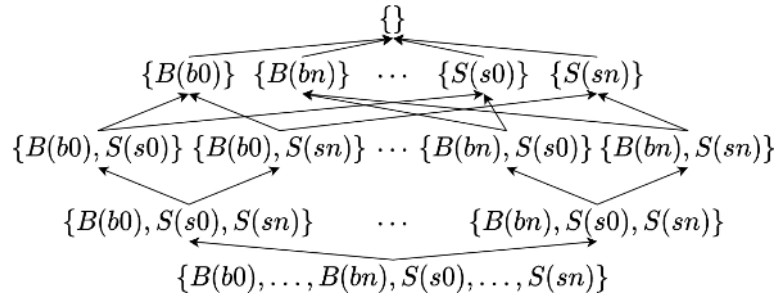


Figure 13: DCR choreography, modelling a fraction of purchase process.

The edges of a DCR graph represent control-flow relations and specify the effect that the execution of an event has on the state and enabledness (possibility of execution) of another. For instance, green (red) arrows define inclusion (exclusion) relations and lead to the inclusion (exclusion) of the event on the "+" ("-%") end of the edge; the yellow arrows denote condition relations and define that the execution of events on the headless end as a requirement for the execution of events on the "⤵●" end of the edge; finally, blue arrows define response relations and make the events on the "⤵" end of the edge pending execution when the events on the "●" end of the edge execute. Condition and response relations can be

associated with time, thus defining a delay and a deadline, respectively. In the example, once the "Request Quotes" event executes, the timed condition relation prevents "Timeout" from executing before seven days have passed, and the response relation forces "Timeout" to execute within a seven-day window; "Timeout" executes in exactly seven days.



We refer the reader to other works [15] for a more comprehensive description of DCR graphs semantics and these relations and others not mentioned in this document.

Given the example, we aim to detect confidentiality breaches leading to the unintentional disclosure of sensitive information to unauthorised entities. To apply information flow control, we start by defining the security lattice. Considering the requirements of the problem at hand, we opt for a powerset lattice of value-dependent security labels, which allows both for precise security policies and information sharing. The security lattice is depicted in Figure 14, where the labels $S(s_0)$ and $B(b_0)$ define the security compartments seller s_0 and buyer b_0 .

Figure 14: Security lattice of purchase process.

Depicted in Figure 15 is the security annotated DCR graph. The most strict label one can apply to an event is the set containing the security level of each event participant; the greatest lower bound. An event should not be above the security clearance of any of its participants.

We define a safety property over security annotated DCR graphs sufficient to ensure time-sensitive, termination-sensitive noninterference. Simply put, our safety property states that secret events cannot influence public events. Therefore, relations where the left-hand side event has a security level not lower than the security level of the right-hand side event are deemed illegal. Thus, DCR graphs exhibiting this property do not leak information, and given that events are both data and computation, this is a sufficient condition to prevent both explicit and implicit information leaks.

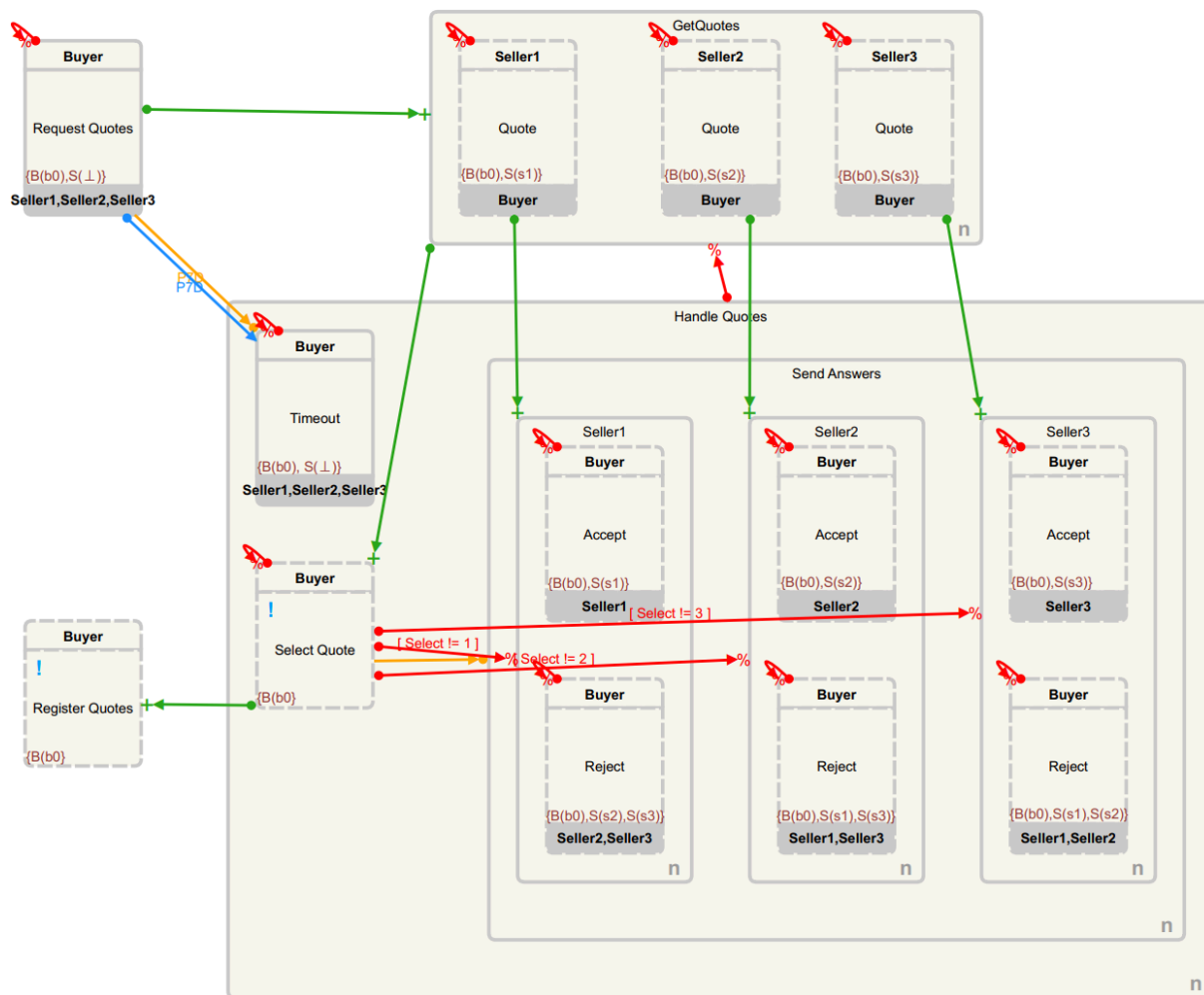


Figure 15: Security annotated DCR graph of purchase process.

A quick glance over the labelled DCR graph above reveals that a few relations between events go against our safety property. For instance, the inclusion relations from "Quote" influence the enabledness of "Reject" events, and "Select Quote" interferes with the less secret events "Reject" and "Accept" through the exclusion relations, allowing for a seller to know who made the accepted quote. By detecting the relations not conforming to our statically verifiable safety property, we can correct the process to one without leaks, such as the one shown in Figure 16.

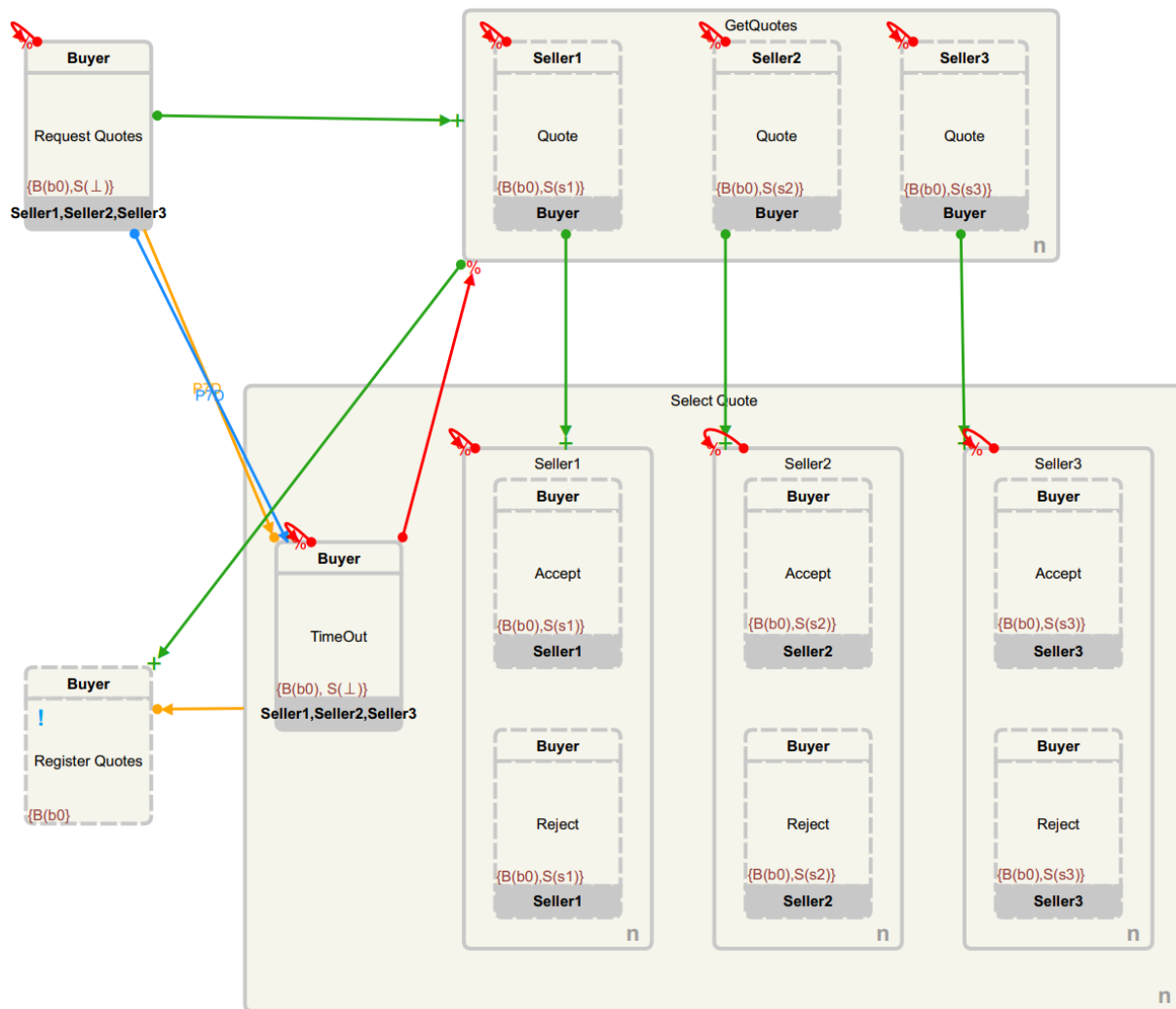


Figure 16: Security annotated DCR graph of purchase process without leaks.

We replace the single selection event followed by acceptance and rejection events with individual pairs of "Accept" and "Reject" for each quote. This graph leaves some requirements unfulfilled. Namely, a seller can accept multiple quotes. This results from the strictness of our progress-sensitive noninterference-ensuring property, which can be overcome through a declassification mechanism or a less strict form of noninterference.

To transpose information flow control from DCR graphs to choreographies, we define a well-formedness property for choreographies that ensures that as long as the underlying graph is safe (does not have any relation whose left-hand side event has a label not lower than the label of the right-hand side event), the choreography and its endpoint projection exhibit time-sensitive, progress-sensitive noninterference. The well-formedness property is quite reasonable, only requiring that the label of an event is lower or equal to the security clearance of any of its participants.

In our analysis of information flow control in DCR choreographies, we base noninterference on the indistinguishability of states during normal system execution. That is to say, without explicit attacks on communications routes or data. Our results only ensure that

choreographies are well formed and the regular users are not exposed to erroneous behaviours embedded in the processes.

To broaden the scope of the analysis, DCR choreographies with IFC, which assumes trusted communication, need to be extended with support for IFC using public channels. We still assume that information leaks due to the analysis of message routing are admissible. Next, we present an approach to use channel information flow as sound support for DCR choreographies over public networks by means of encryption operations that help tunnel data between trusted contexts via insecure media.

2.3.1.2 Information Flow Channel

Traditionally, information flow analyses are most concerned with what can happen in a system that develops in good-faith adherence to specification. For example, one can check that the code executed internally in one of the system's machines does not violate the security policy. This usually ignores a central attack vector: that of an intruder eavesdropping or modifying messages sent over a public network. If the communication primitives used by the programmer are not sufficiently secure or not used in the right way, such an intruder might be able to derive confidential information from patterns in the network traffic or by observing the behaviour of machines when sent unexpected messages by the intruder. We introduce the IFChannel framework for extending an information flow analysis to systems with communication over an untrusted network in a secure way. The main contribution is theoretical: we are proving that if the communication is performed using appropriate cryptography then the information flow analysis provides the very strong privacy guarantee of static equivalence (basically, that the intruder learns nothing about confidential information). This proof then tells us which requirements on the implementation of channels are sufficient for them to be included in the TaRDIS information flow tool.

We first demonstrate some of the problems which may arise when combining an information flow analysis and communication over a public network. Consider the following lines of code:

```
if bid ≥ threshold then
    SEND(PROSUMERCHANNEL, "Auction closed")
end if
```

If *bid* or *threshold* is of any other label than \perp (i.e. public) this program will be illegal since an intruder could infer information about the two variables from the observed network traffic. Note that this is the case even if the prosumerChannel completely hides the content of messages. To fix this, the program could be restructured to the following version which does hide the truth-value of *bid* ≥ *threshold*.

```
if bid ≥ threshold then
    msg ← "Auction closed"
else
    msg ← "Auction still open"
end if
SEND(PROSUMERCHANNEL, msg)
```

Conversely, we might have situations where the traffic pattern itself reveals nothing, while the content of messages does. Consider the following:

```

 $x \leftarrow \text{secret\_value}$ 
if  $x \geq 50$  then
   $a \leftarrow 0$ 
   $b \leftarrow 0$ 
else
   $a \leftarrow 0$ 
   $b \leftarrow 1$ 
end if
 $y \leftarrow \text{ENC}_{\text{key}}(a)$ 
 $z \leftarrow \text{ENC}_{\text{key}}(b)$ 
SEND( $y$ )
SEND( $z$ )

```

Even if the intruder does not have the keys necessary to decrypt the message, they will be able to see whether the secret value is greater than 50 by checking if the two sent messages are the same or not.

Another attack vector is the intruder sending a message out of context to make a machine reveal something confidential. Consider the following two programs running on different machines:

Seller:

```

 $x \leftarrow \text{ENC}_K(\text{seller\_ID}, \text{minimum\_bid})$ 
SEND( $x$ )

```

Announcer:

```

RECEIVE( $x$ )
 $\text{seller\_ID}, \text{minimum\_bid} \leftarrow \text{DEC}_K(x)$ 
SEND( $\text{seller\_ID}$ )

```

The idea is that the first machine is starting an auction and the second machine is responsible for announcing the (public) id of the seller. By chance we might have a third machine running the following program:

Bidder:

```

 $x \leftarrow \text{ENC}_K(\text{secret\_price}, \text{bidder\_ID})$ 
SEND( $x$ )

```

This introduces an attack on the secret price, since an intruder might capture the message from the bidder and send it to the announcer. The announcer will then believe that the secret price is a seller id and announce it to the public.

These examples demonstrate that we need the channels in the information flow analysis to be called only in permitted contexts, introduce randomness, and use formats to avoid the acceptance of out-of-context messages.

We then also want to prove that these requirements are sufficient. Soundness is proven with regards to *static equivalence* [16], which has traditionally been used to show privacy preservation of cryptographic protocols.

In the following, we briefly outline the soundness proof.

We represent a system state (from the perspective of a single program) as a mapping from program variables to terms and a list of terms representing the messages sent to the network. Figure 17 shows a program and potential end-state after executing the program:

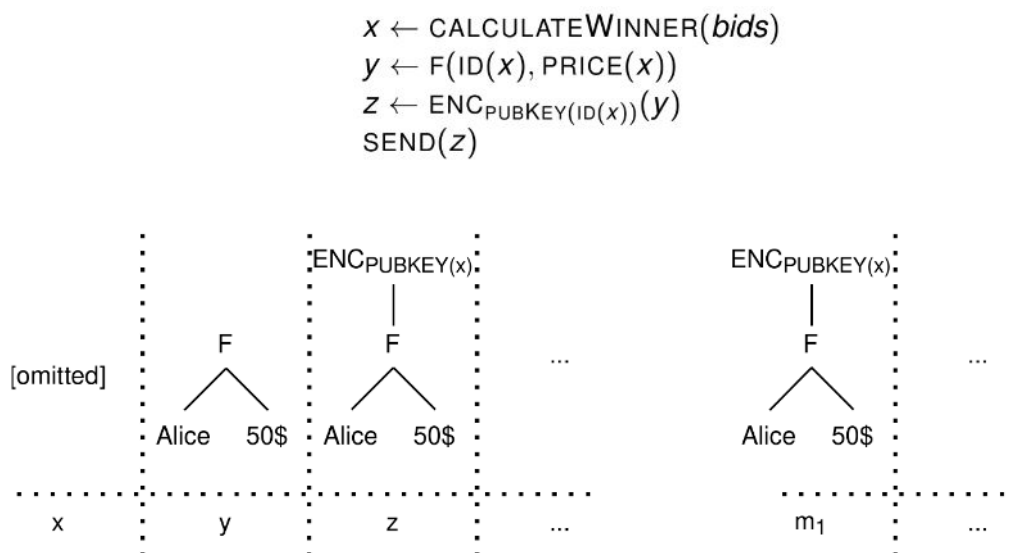


Figure 17: Potential end-state after execution of program.

We then allow the intruder to construct recipes over the information they are allowed to access in such states. A recipe consists of functions the intruder knows and references to variables and network messages. An example is the following (given that *key1* and *key2* are below the intruder’s paygrade):

$$\text{ENCRYPT}_{key2}(\text{DECRYPT}_{key1}(x), m_1 + 1)$$

A recipe, *r*, can be evaluated over a state, written *S(r)*, by replacing the references with the values stored in the state. The evaluation of a recipe only succeeds if there are only references to information that the intruder is permitted to access (we assume the intruder can be a participant in the system with a given security level).

We can then define that two states, S and T , are statically equivalent if for any two recipes, r_1 and r_2 , we have $S(r_1) = S(r_2)$ if and only if $T(r_1) = T(r_2)$.

The soundness proof then consists in showing that for a well-typed program and two permissible starting states (permissible means, among other things, that they are statically equivalent), we have that if the program is executed once from each starting state we will end up in two statically equivalent states. The implication of this is that there is no experiment an intruder can do to figure out which of the two states they are in. In other words, they have not obtained any knowledge about information above their security level.

We have a full formalisation of this result in the proof assistant Isabelle/HOL.

2.3.2 PSPSP (DTU)

The PSPSP tool [23, 24] was developed by DTU before the TaRDIS project. It is based on a protocol security framework we have developed in the proof assistant Isabelle/HOL. At the basic level, protocols are modelled in this framework as a set of traces that consist of honest agents sending and receiving messages on a public network, as well as changing their local state. The network is completely controlled by an intruder, who cannot break the cryptography (Dolev-Yao style intruder model), but who may have their own cryptographic materials like keys and passwords to be able to engage in the protocol under their real name like a normal participant. We can define certain attack events in the protocol and define security as the impossibility to reach such an event.

The PSPSP tool has an input language called Trac which is short for "transactions". Each transaction is an atomic action of an honest agent, consisting of receiving some input messages, performing checks w.r.t. these inputs and the local state of the agent, then changing the state and sending outgoing messages. This is especially designed to model devices like a TPM (Trusted Platform Module) that offers an API to the outside (incoming messages as receiving a command, and outgoing messages as an answer).

In general, Isabelle/HOL is an interactive proof assistant, i.e., a human tries to prove a claim by breaking it down into smaller logical steps that are trivial or can be automatically proved by some heuristics. PSPSP in contrast can automatically verify a given protocol using abstract interpretation techniques. The idea is that agent memory is organised as sets of messages, e.g., a server may maintain a family $\text{valid}(A)$ of sets for every agent A that contain the valid public keys of A . These sets can be changed by the transactions (e.g., new keys can be registered, and old keys discarded). The abstract interpretation abstracts by set membership: all data that belongs to the same sets is not distinguished. If the families of sets are finite, this will lead to a finite fixed point of abstract messages that the intruder can ever know. PSPSP computes this fixed point and then "convinces" Isabelle that everything that can happen in the concrete protocol is covered by the abstract fixed point. If a special symbol "attack" does not occur in the abstract fixed point, then also not in any reachable concrete state, and thus the security of the protocol is proved.

What can potentially go wrong is that the fixed point contains the symbol "attack". Then this either represents an actual way to attack the concrete protocol (and the designer has to fix the security flaws and try again) or it is introduced by the over-approximation inherent in the abstract interpretation. The latter means that the protocol may be fine, but the abstraction is

too coarse to prove it; in this case the modeller needs to review the specification to find maybe a different setup of the sets of messages used by the honest agents that leads to a finer abstraction. Note that any potential bug in the PSPSP tool does not bear the risk of accepting a flawed protocol: in the worst case, such a bug would lead to a failure to "convince" Isabelle.

Utilisation of PSPSP in TaRDIS

The normal TaRDIS programmer should not be involved with cryptography and cryptographic protocols. Rather the TaRDIS API offers functions for the setup and use of channels, even more abstract in the concept of events as basically sending (generating events) and receiving (reacting upon events) messages over appropriate channels. That events or messages are not sent over channels of insufficient security level is part of the information flow analysis (see Channel IF tool). The implementation of the TaRDIS API with respect to cryptography, namely the setup and use of channels, uses cryptographic protocols, i.e., encryptions, digital signatures, challenge-response with random numbers, exchange of keys and certificates. These protocols shall be modelled and analysed with PSPSP. This is part of the implementation of the TaRDIS API, and thus a "project-internal" use of the tool. However, we also plan that in the future, advanced users may want to add custom cryptographic protocols to the TaRDIS API and verify them with PSPSP.

Extensions of PSPSP in TaRDIS

To aid this verification, both TaRDIS-internal and by TaRDIS power-users, we plan to make several extensions. First, we are developing a choreography language for cryptographic protocols to allow for a more high-level specification, as discussed in a section below.

Moreover, currently the tool is a bare-bones extension of Isabelle that does not give a user much support when the verification does not work: basically one can view the abstract fixed point (which is often substantially large) and which of the requirements were violated. We plan to improve this situation by a plugin tool that can analyse the fixed point and give hints as to whether the model indeed exhibits an attack (also on the concrete level) or where the model is possibly too coarse (leading to false positives - attacks on the abstract level that do not exist on the concrete level) and needs to be refined.

Finally, we are working on the question of accountability: that dishonest actors who break the rules or agreements run the risk that they can be identified and held accountable. This is an extension of the concept of non-repudiation, i.e., that an actor cannot deny certain transactions they have performed. Our model of this goes beyond pure non-repudiation. In fact, our model includes a legal framework, i.e., rules that participants are bound to uphold, and a judicial framework, i.e., how an impartial judge (or ombudsperson) can evaluate detected cases of rule violations. This can involve the request that certain information must be kept and can be subpoenaed by the judge; failure to comply with a subpoena is then also a rule violation. This allows for a system where rules are ensured by the fact that dishonest participants are discouraged to break the rules of the contract, because they run a substantial risk of being caught and having to pay a fine.

Besides the immediate practical value of accountability, there is also a more fundamental interest in this kind of systems for TaRDIS: we do not necessarily describe here a protocol as

a sequence or choice of actions of participants, but rather allow them to do everything they are cryptographically able to, and rather have rules that certain messages have legal meaning. For instance a public key may be legally bound to a participant, and signing a message of a particular format using the corresponding private key, is a legal binding statement, e.g. ordering items from another participant. For accountability of such orders, however, the rest of the workflow of the participants is completely irrelevant (i.e., under which conditions a participant chooses to place an order). Leaving open most of the behaviour of the participants does not only lead to more efficient verification of accountability questions, but also makes it independent of changes in the workflows as long as the legal and judicial framework are not affected by the changes.

Compositionality

The initial focus of the PSPSP framework, even before the automated verification itself, is the secure composition of protocols [25]. Given that we have proven two protocols in isolation, can we infer that it is also secure to run them together on the network? The classic results in this area are that this indeed rather straightforward if the protocols do not share any cryptographic material; if they share cryptographic keys such as a public-key infrastructure used by both protocols, then it suffices that the messages of the protocols are made distinguishable, so that an intruder cannot abuse a message from one protocol to achieve something in the other. The composition in the PSPSP framework extends upon these classic results in two significant ways.

First, we do allow a set of shared secrets between the two protocols, and the declassification of such secrets. This allows us to have an overlap between the two protocols, e.g., if we have some general key certificates by a certificate authority, we can use these in both protocols. By default, all these messages in the overlap of the two protocols are secret, but they can be declassified, as it usually makes sense for key certificates.

Second, the participants can maintain a long-term state in terms of one or more sets of messages, e.g., a set of keys for a particular purpose, or on an application level, a set of orders that have been placed and need to be worked off. Crucially, we allow that the protocols share these sets. This is not meant as another way of communication, but rather as an interface between two protocols. For instance, a server may maintain a database of registered public keys of users, and may run several protocols independently on this database.

This stateful aspect allows us also to decompose a larger system into smaller components and verify them each in isolation. For instance, we may model a login protocol that transmits a user password over a channel established by the TLS protocol. This can be done compositionally as follows: clients and servers run the TLS protocol to negotiate a key, and the client enters the negotiated key into a set $\text{clientkey}(A,B)$ and the server into a set $\text{serverkey}(B)$ where A and B are the names of the client and server, respectively. Note that the server here cannot be sure who A is (in the standard setting where users have no key certificates), and hence their set is parameterized only over the name B of the server. The login protocol now can start from these keys: a client A can take (and delete) any key from $\text{clientkey}(A,B)$ and encrypt a message to B containing the password of A . The server can

similarly check that this message decrypts with a key in $serverkey(B)$ and authenticate A in this connection.

The interesting point of composition is now that we can for instance make updates to the TLS protocol (e.g. from version 1.2 to 1.3) and only need to verify that the new version satisfies the requirements of the composition and the interface that the login protocol is expecting, but not the login protocol. Similarly, we could replace the login protocol by a more complex single-sign-on solution with a trusted third party, without re-proving anything about TLS.

Utilisation of Compositionality in TaRDIS

The main use of compositionality is the interfacing between channels and information flow analysis. While the Channel Information Flow tools ensure that the applications will obey the information flow policy, provided that we write messages only into channels that are sufficiently protected by cryptography, the result behind the channel information flow considers rather basic cryptographic implementations. Now compositionality allows us to replace such a basic implementation with a more complex one that gives at least the same security guarantees, without having to repeat the information flow analysis of the application.

The reason for more complex implementations can vary. Typically the basic implementation will assume a simple key infrastructure, e.g., every relevant group of participants have a shared key. Thus cryptographic mechanisms can be simply symmetric encryption or MACing with the respective group key for confidentiality and/or integrity. In contrast, in a heterogeneous landscape we may rely on a key-establishment using a trusted third party or certificates. Similarly other aspects that are independent of security itself (like availability of participants) may require more complex protocols and control flows. It is thus economical to verify the high-level application based on the (unrealistic) assumption of a simple channel, and then replace this channel by a realistic one. Compositionality here allows us to separate the verification tasks of application and channel.

2.3.3 Cryptographic Interpretations of Choreographies (DTU)

This projected tool is relevant for the description, implementation, and verification of the secure channels that TaRDIS offers. For starters, channels should be formalized and verified in PPSPP. PPSPP offers a low-level language for security protocols. It is well-known how to translate Alice-and-Bob notation (aka protocol narrations) to the low-level languages such as PPSPP, which is more user-friendly and has the advantage to show the "whole picture", i.e., how the different roles of the protocol are supposed to interact. However, Alice-and-Bob notation is also a bit limited: we have a linear execution of protocol steps without for instance non-deterministic choices or global mutable state of participants (such as a database or orders, or the key storage of a trusted device).

We are currently working on a cryptographic choreography language that fills that gap. The idea is that choreography languages are indeed providing extensions over Alice-and-Bob languages. For cryptographic protocols - in contrast to standard choreography languages - one must pay attention to how honest agents actually apply cryptographic operations. For instance, when a protocol is based on Diffie-Hellman, we consider exponentiation modulo a prime number p (or elliptic curves over a finite field); let us just write $\text{exp}(g,X)$ for exponentiation modulo p with a secret X and a generator g where p and g are fixed public

values. Alice and Bob each generate a secret X and Y , respectively, and exchange $\text{exp}(g,X)$ and $\text{exp}(g,Y)$, respectively. This exchange needs to be authenticated, i.e., Bob must be sure that $\text{exp}(g,X)$ really comes from Alice, and vice versa. The point of modular exponentiation is that it is computationally hard to get the X from $\text{exp}(g,X)$. After the exchange Alice and Bob have the shared secret key $\text{exp}(\text{exp}(g,X),Y)$. The choreography may thus prescribe that Alice shall encrypt some message with the key $\text{exp}(\text{exp}(g,X),Y)$ and send it to Bob. Note that Alice will not know Bob's secret Y , but have only received some value GY that is supposed to be $\text{exp}(g,Y)$. The key that Alice should generate from this is $\text{exp}(GY,X)$ which, if Bob acted correctly, is $\text{exp}(\text{exp}(g,Y),X)=\text{exp}(\text{exp}(g,X),Y)$.

This example illustrates how tricky it is to understand the protocol execution from just a specification of the messages that are supposed to be exchanged: we need to understand how agents can construct the outgoing messages from their knowledge, and how they are decomposing and checking the incoming messages. For Alice-and-Bob notation this problem has been solved on the semantic level: how to define the execution of each role from an Alice-and-Bob specification. Note that this definition still bears problems that are in general undecidable because they are based on an algebraic theory of operators (e.g. that $\text{exp}(\text{exp}(A,B),C)=\text{exp}(\text{exp}(A,C),B)$ is the very least we need for Diffie-Hellman) and in general even the word problem is undecidable for a given set of algebraic equations. However, for many standard theories that are used in protocol verification we can give an effective procedure.

When applying this concept to choreographies we have, however, the problem of non-deterministic branching. As an example, we have a simple choreography where A can encrypt for B one of two kinds of messages - which one is a non-deterministic choice of A . A cannot however authenticate the message for B and can just use a Mac with a shared key with a trusted server s , and the server should sign the encrypted message for B :

$A \rightarrow s$: $[\text{crypt}(\text{pk}(B), m1)]\text{sk}(A,s)$. $s \rightarrow B$: $\text{sign}(\text{inv}(\text{pk}(s)), \text{crypt}(\text{pk}(B)), m1)$
 $+ [\text{crypt}(\text{pk}(B), m2)]\text{sk}(A,s)$. $s \rightarrow B$: $\text{sign}(\text{inv}(\text{pk}(s)), \text{crypt}(\text{pk}(B)), m2)$

Here $\text{pk}(B)$ is the public key of B , $\text{sk}(A,s)$ is the shared key of A with the server s , $[M]K$ stands for sending the message M along with a Mac of M using key K , and $\text{inv}(\text{pk}(s))$ is the private key of the server. The server, not knowing B 's private key actually cannot see which of the messages A is sending. That is however also not necessary, because the behaviour of B is uniform in this case. Our semantics of cryptographic choreographies understands this and gives for s the following local execution:

$\text{receive}(M)$. $\text{if}(\text{checkMac}(M, \text{sk}(A,s)))$ then $\text{snd}(\text{sign}(\text{inv}(\text{pk}(s))), \text{extract}(M))$ else error

2.4 ORCHESTRATION, VERIFICATION AND REGARDING PROPERTIES INTEGRATED WITH WP5 AND WP6

2.4.1 Correct Orchestration of Federated Learning Algorithms (UNS, WP4 for WP5)

The focus of our work so far has been on the correct orchestration of the federated learning algorithms. In 2023, Python Testbed for Federated Learning Algorithms (PTB-FLA) was

introduced [17]. PTB-FLA was developed with the primary intention to be used as a FL framework for developing federated learning algorithms (FLAs), or more precisely as a runtime environment for FLAs. PTB-FLA supports both centralised and decentralised FLAs. The former is defined as in [18], whereas the latter are generalised such that each process (or node) alternatively takes server and client roles from [18] or more precisely, it switches roles from server to client and back to server. For the full details, we refer to [17].

We have formally verified the correctness of two generic FL algorithms, a centralised and a decentralised one, by proving two properties:

- **Deadlock-freedom (safety property):** an algorithm will never reach a non-terminated state with no further move
- **Successful FLA termination (liveness property):** an algorithm always reaches the terminated state

We have used the Communicating Sequential Processes calculus (CSP) [19] and the Process Analysis Toolkit (PAT) [20] model checker, and proved the correctness of algorithms in two phases. In the first phase, we have constructed by hand CSP models of the generic centralised and decentralised FLAs as faithful representations of the real Python code. These models are constructed in a bottom-up fashion. In the second phase, we formulate desired system properties, namely deadlock freedom (safety property) and successful FLA termination (liveness property) and automatically prove formulated statements in PAT.

There are several directions of work we are following currently. First, we work on the automatic translation of the Python code into CSP, which will ensure that CSP models accurately represent the Python code. Second, in [21] we have used PAT to prove deadlock freedom and liveness property, but we also work on a different approach, where we will use Maude for the verification. Furthermore, during the discussion with members of WP5 we have identified several desirable FL properties:

- **FL Roles of agents:** ensures that clients receive only the data they can process, This contributes to the efficiency of the model resulting in lower energy usage.
- **FL Data privacy:** ensures statically that only the model parameters can be sent by the clients and servers, actual data remains private.
- **FL Message delivery:** have large enough buffers of servers to support receiving messages from all clients, contributes to the liveness.
- **FL Clients equality:** clients should equally contribute to the algorithm, avoid the scenario where a single client sends multiple messages.

We work on the specification and verification of these properties.

For now, we have used CSP models only to specify FL algorithms introduced in [17]. We investigate other FL algorithms with the aim to specify and verify them as well.

Another tool used to model distributed protocols is Multiparty Asynchronous Session Types (MPST), which is a class of behavioural types tailored for describing distributed protocols relying on asynchronous communications. Hu and Yoshida extended MPST in [10] with explicit connection actions to support protocols with optional and dynamic participants. Although these extended MPST enabled modelling and verification of some protocols in cloud-edge continuum [22], we could not use them to model the generic centralised and

decentralised FLAs, because we could not express arbitrary order of message arrivals that take place at an FLA instance. However, we work on the extension of MPST which will enable modelling algorithms introduced in [17].

2.4.2 Correct Hierarchical Namespaces (UNS, WP4 for WP6 T6.3)

One of the main focuses of Task 6.3 in the TaRDIS project is designing a solution for supporting the reconfiguration of application components at runtime. The first step should include designing a system for storing configuration data. Configurations can be versioned so that the changes can be traced over time. Applications live in different namespaces, allowing logical isolation and preventing naming conflicts. As a contribution for this task, a journal paper [9] has recently been published. The paper introduces a model for hierarchical namespaces that promotes the proper organisation and redistribution of resources. The namespaces prevent naming conflicts in the system and preserve logical isolation, thus creating a multi-tenant system. This model relies on the techniques developed in TaRDIS WP4 for the specification and verification. More specifically, WP4 has made the following contributions:

1. the model in the paper relies on remote configuration management and builds upon four protocols, for which we have ensured correctness by employing an extension of asynchronous multiparty session types;
2. resource redistribution has been modelled via record-weighted directed acyclic graphs and accurate resource redistribution is guaranteed through graph transformations.

In the following, we give more detail on these WP4 contributions.

The aforementioned protocols presented for the model are: (i) namespaces mutation protocol, used to rearrange resources, (ii) secure profile mutation protocol, used to create and push new secure computing mode (Seccomp) profiles to selected applications running in namespaces, (iii) context switch protocol, utilised by users for switching between active namespaces and/or distributed clouds, and (iv) upgraded cluster formation protocol, used to create a default namespace, that holds all resources for the newly created element and, accordingly a default security profile for that namespace. Following the top-down approach of asynchronous multiparty session types, all these protocols have been formally modelled using global types. For the complex communication combinations of the protocols, standard works of asynchronous multiparty session types were not suitable for the modelling. Instead, we have used an extended model [10], that allows specifying sending to different participants and the dynamic introduction of participants in the session. From the specification of global types, applying the projection function we obtained the local specifications of all participants, called local types. All our protocols undergo validation using the toolchain presented in [10]. For full details see [9].

The resource redistribution modelled in the paper has been made through graphs and graph transformations. First, to give a faithful model for the hierarchical namespace organisation, we have developed a novel record-weighted directed graphs. There, each node in the graph represents a single namespace and directed edges represent parent-child connections between the namespaces. Weights assigned to nodes represent the resources of the namespace. For example, consider the graph shown in Figure 18. Namespace A has

two child namespaces B and C, that have been given some resources. Here, we consider three different resources that are allocated throughout the namespaces. The two values next to each resource represent the amount of resources available and utilised (by the applications) in the namespace.

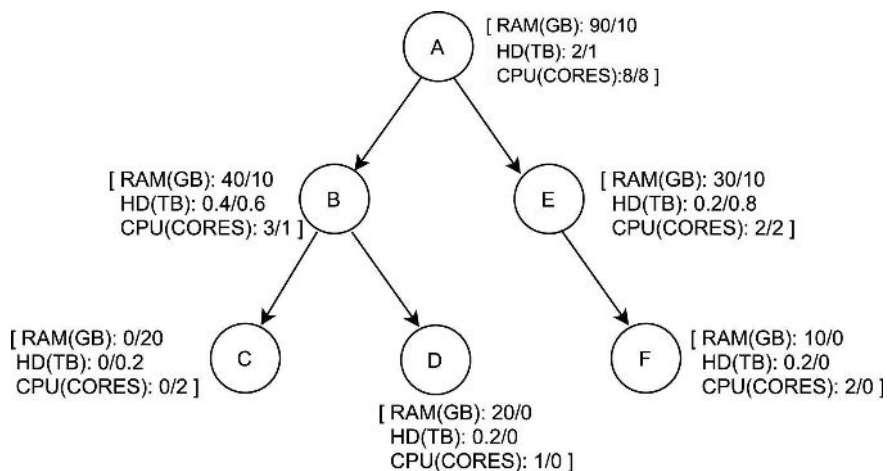


Figure 18: Example of namespaces resource graph.

Applications running in the namespaces are modelled again through graphs. Figure 19 gives an example. The resources mentioned in the applications are the ones that are specified as utilised in the namespace.

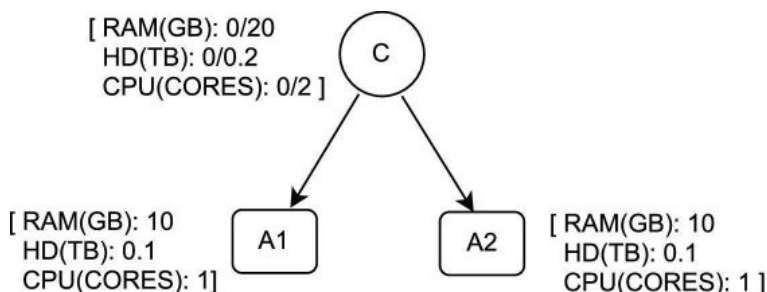


Figure 19: Example of application resource graph.

The namespaces and their resource manipulation that the namespace mutation protocol relies on utilise the (record-weighted) graph representation of the namespaces and are based on the double-pushout (DPO) constructs of graph transformations. The DPO is constructed by defining the set of so-called production rules, which can be applied to an observed graph. We identified a set of four production rules: (i) creation of a child namespace, (ii) deletion of a namespace, (iii) resource allocation between child-parent namespaces, and (iv) transferring from available to utilised (and vice versa) resources in a single namespace. The first rule is given in Figure 20 below. It specifies that for a namespace P, we can create a child namespace C with resources R_c , provided the parent has enough resources since $R_p - R_c$ has to be nonnegative by the definition of graphs.

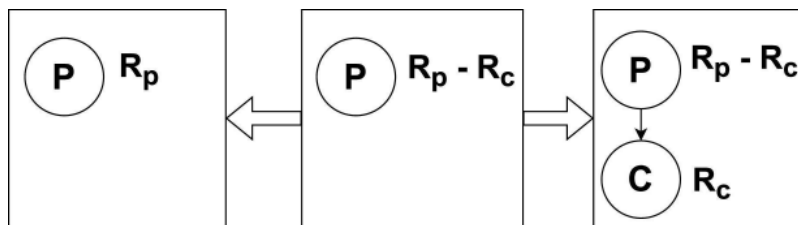


Figure 20: Creation of namespaces.

Using the four production rules and a single node (a default namespace) as a starting graph, we obtain a graph transformation language. An instance of a direct graph transformation, applying the production rule (i), is given in Figure 21 below.

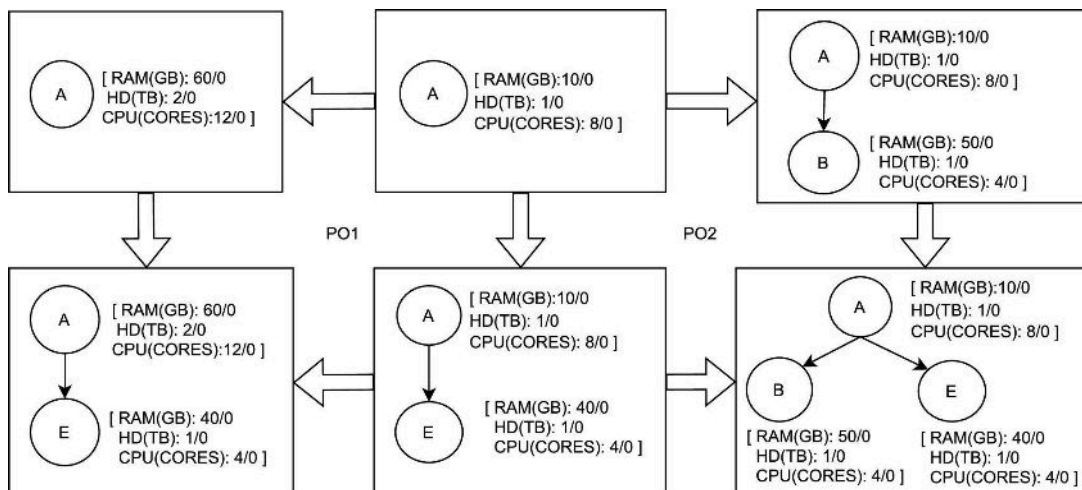


Figure 21: Example of a DPO based direct graph transformation.

Besides giving clear specifications of the system described, these graphs and transformation rules also provide the property that resources used anywhere in the organisation of the namespaces are the ones that are provided. For full details on graphs and graph transformations modelling namespaces and resource manipulations see again [9].

3. REVISIONS OF IDENTIFIED PROPERTIES IN D4.1

This section delves into the modifications applied to properties identified in D4.1, as well as any newly introduced properties. These updated properties are aligned with and can be verified using the tools outlined in Section 2. Tools without revised or new properties compared to D4.1 will not be discussed.

3.1 PROPERTIES FOR COMMUNICATION BEHAVIOURS

3.1.1 WorkflowEditor & Actyx Middleware

Two tools featuring in the planned implementation of the use case of «highly resilient factory shop floor digitalization» were previously neglected in deliverable D4.1, namely the WorkflowEditor and the Actyx middleware (which will be improved as part of WP6 with TaRDIS results). The latter is not directly part of this report, but it is relevant in that it provides some of the underlying event dissemination guarantees required by the theory underlying the WorkflowEditor and its analytic capabilities. In the following we describe the guarantees afforded to end users building software applications using the WorkflowEditor, which include deadlock-freedom, liveness, eventual consensus, protocol conformance, fault tolerance (non-adversarial), and compositional verification. These guarantees are tailored to this tool and differ from the general properties and requirements identified in Deliverable D4.1. Additionally, new properties unique to this tool, such as resilience through replication, perfect availability, and termination of failure, are introduced compared to D4.1.

Deadlock-freedom. Assuming at least one live replica per role in the swarm protocol, the workflow will eventually make progress—more precisely progress will only depend on information transport from the previously active role to some replica playing a role that can act next.

Resilience through replication. The system will remain deadlock free as long as it is properly maintained. This implies that a role may be played by multiple replicas so that the real system can employ redundancy and thus ensure that the precondition of deadlock freedom is always met.

Liveness. All live replicas of roles that may act according to their current knowledge of the workflow's progress can perform an allowed action without further restrictions. In particular, the ability to act does not depend on the availability of any other replica or the ability to communicate.

Eventual consensus. The system guarantees that once all events up to a certain (logical) timestamp have been disseminated to all live nodes, these nodes will agree on the path of execution of the workflow up to the point indicated by this prefix of the event log. This agreement is reflected in the workflow state being presented to the application, and it is achieved without any further coordination (as is implied by the liveness property above and the availability demanded below). This is not strict consensus because the moment in time after which this agreement holds is not known to any of the replicas, and replicas will typically present invalid intermediate workflow states to the application while event dissemination is still ongoing.

Protocol conformance. The sequence of actions recorded as events in the federated event log adheres to the workflow as designed using the WorkflowEditor. Note that the availability and liveness requirements imply that some events may be emitted and later ignored due to conflict resolution. The goal of this requirement is that the execution settled upon by eventual consensus can intuitively be understood by considering only the global workflow design, specified under the assumption of instantaneous and reliable event dissemination.

Perfect availability. The application can obtain the local view on the workflow state at any time without depending on the communication with other replicas. Whenever that state permits the local role to act, such action can be taken also without depending on the communication with other replicas, and taking the action will update the local view on the workflow state.

Fault tolerance (non-adversarial). Failure of replicas (crash stop) or transient communication outages do not affect the other guarantees. Note that this does not cover malicious behaviour like falsifying events in the log. Fault tolerance relies heavily on resilience through replication.

Termination or failure. Assuming that no workflow branch is taken for an unbounded number of times (in case of cyclic workflows) and that there always is at least one live replica for each role, any well-formed workflow will eventually reach either a designed terminal state or a failure state.

Compositional verification (in collaboration with DTU). Workflows can be composed from smaller workflows in a black box fashion: the understanding and analysis of the composed system does not require an understanding of the internal structure of the included workflows, it only requires them to be well-formed.

3.1.2 Compositional Verification of Swarm Protocols

The focus of compositional verification in swarm protocols lies in identifying sufficient conditions to ensure that when two correct swarm protocols, G and G' , which are well-formed and deadlock-free, are composed ($G|G'$), the resulting composition remains correct. Additionally, the exploration of the necessary conditions for such compositions aims to enable developers to maintain a library comprising well-formed swarm protocols and participant implementations such that these protocols and implementations can be combined without introducing deadlocks or communication errors.

3.1.3 Fair Join Pattern Matching

The join pattern matching library adheres to all communication behaviour properties outlined in D4.1, while also inherently providing the following new properties.

Mailbox communication safety. Messages and mailboxes are strongly-typed, and all message exchanges are type-safe.

Fair join pattern matching. Any message in a mailbox that can be potentially consumed by a join pattern will be eventually consumed.

3.1.4 Verified APIs for Software-Defined Networking

The verified APIs for Software-Defined Networking in P4 modify the safety, deadlock-freedom, and liveness properties, diverging from those described in D4.1.

Safety w.r.t. network configurations. Control programs attempting invalid updates w.r.t. the P4 tables of a P4-defined network do not type-check.

Deadlock-freedom, liveness. A typed program that attempts a network update will always succeed (i.e., enjoys progress).

3.1.5 Model-Based Testing of Swarm Applications

The model-based testing tool COTS introduces the following new properties compared to D4.1. (Currently, the tool utilises a test model based on session types, with potential future extensions to incorporate a test model based on swarm protocols.)

Test correctness. Each autogenerated test run represents a valid execution that conforms to the test model.

Fault detection soundness. Each failing test corresponds to a case where the system-under-test violates the test model.

3.1.6 Java Typestate Checker

The Java typestate checker adheres to all communication behaviour properties identified in D4.1, while also inherently providing the following new properties.

Memory-safety. programs that statically type-check respect the typestate protocols of all objects and thus no method call will ever raise a null-pointer exception. Moreover, given that programs are free of races because the access to objects is linearly controlled, there will be no memory leaks (the linear discipline implies that objects are fully used - according to their protocol - and disposed).

Protocol compliance. Client code executes respecting each object correct usage, which means that no method is called when the protocol does not allow it.

Protocol completion. All objects are used until the end of their protocols (and released).

3.2 PROPERTIES FOR DATA MANAGEMENT AND REPLICATION

The properties for data management and replication identified in D4.1 remain unchanged; the assertions made therein retain their validity. The tools developed at NOVA are *Language-based Data Consistency Approaches*, eventually avoiding data conflicts (consistency) and ensuring safe concurrent updates (convergence).

We are looking for opportunities to work with TaRDIS partners, using idealised core versions of the use cases. Concretely, we foresee the following collaborations:

Actyx - develop a setting for “elevating” eventual consensus to consensus, without sacrificing availability.

EDP - statically ensure policy compliance for all (client) scenarios.

Telefonica - statically ensure safe concurrent data updates.

3.3 PROPERTIES FOR SECURITY

With respect to D4.1, there are only minor changes to the security properties we want to verify. Recall that we are going to employ two basic approaches: Firstly, the verification of communication protocols that use cryptographic means to protect communication from leaking information, tampering with information and unauthorised access. Secondly, we will use information flow control techniques applied to event-based languages to analyse systems for illegal flows that are introduced by programming mistakes. This aims to prevent classified information from being “leaked” into public places and to prevent untrusted information from “leaking” into a trusted information base.

Transmission Security Properties

This is one of the core verification tasks: verifying security properties (confidentiality and integrity) of given transmission protocols with the PSPSP tool. This will be an internal use of the tool for verifying security of the TaRDIS API.

Information Flow Properties

This is the main aim of the verification tool for DCR graphs, relying on secure channels. We may need to make one revision here in practice: The transmission over channels is in general observable by an attacker, who cannot open encrypted messages, but who can see that messages are being sent and link messages that belong together, possibly even link them to particular entities. This in general breaks the strong guarantees of non-interference, unless one starts with anonymization techniques like onion routing (which is in most cases not desirable as an additional layer). Thus, we have to make some concessions in terms of so-called implicit information flow: the attacker may learn some information about conditions being true or on the relation between messages. We are currently investigating how to limit the exposure and how to allow for clear feedback for developers to allow them to make conscious decisions about what information is fine to release and what needs further protection.

Privacy-type Properties

In general, the information flow would ensure privacy properties, but due to the compromises we have to make in information flow when transmissions are observable, the privacy properties are similarly affected, at least with respect to an attacker who can do long-term surveillance of the entire communication medium. However, we will at every point aim for the maximally achievable privacy.

4. CONCLUSIONS

The primary goal of the TaRDIS development environment is to assist developers in constructing correct systems by automatically analysing interactions between various components within a distributed system. This approach ensures that applications are inherently designed for correctness, taking into account both application invariants and the specifics of the execution environment. By integrating these elements, TaRDIS promotes the development of robust and reliable distributed systems.

To tackle these challenges, this document introduces an initial toolset consisting of tools tailored for application to the TaRDIS models. These tools ensure the fulfilment of desirable properties that align with the specific TaRDIS use cases and requirements outlined in Deliverable D4.1. Additionally, to facilitate calibration, this document outlines any adjustments to the properties detailed in D4.1, as well as any new properties that can be addressed by the tools specified herein. This ensures a more comprehensive application of the toolset within the TaRDIS models.

As future work for the subsequent deliverable of this work package, the team will explore the integration of the developed analyses and tools into the APIs and IDE developed as part of work package 3 (WP3).

5. BIBLIOGRAPHY

- [1] Roland Kuhn, Hernán Melgratti, and Emilio Tuosto. (2023). Behavioural Types for Local-First Software. *ECOOP 2023*. <https://doi.org/10.4230/LIPIcs.ECOOP.2023.15>
- [2] Cédric Fournet and Georges Gonthier. (1996). The reflexive CHAM and the join-calculus. *In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 372–385. <https://doi.org/10.1145/237721.237805>
- [3] Philipp Haller, Ayman Hussein, Hernan Melgratti, Alceste Scalas, Antoine Sébert, and Emilio Tuosto. A New Take on Join Patterns. *NWPT 2023 (34th Nordic Workshop on Programming Theory)*. <https://conf.researchr.org/details/nwpt-2023/nwpt-2023-papers/16/A-New-Take-on-Join-Patterns>
- [4] Jens Kanstrup Larsen, Roberto Guanciale, Philipp Haller, and Alceste Scalas. 2023. P4R-Type: A Verified API for P4 Control Plane Programs. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 290 (October 2023), 29 pages. <https://doi.org/10.1145/3622866>
- [5] Utting, M., Pretschner, A. and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22: 297-312. <https://doi.org/10.1002/stvr.456>
- [6] Christian Bartolo Burlò, Adrian Francalanza, Emilio Tuosto, and Alceste Scalas. COTS: Connected OpenAPI Test Synthesis for RESTful Applications. *COORDINATION 2024*. https://doi.org/10.1007/978-3-031-62697-5_5
- [7] Fielding, R.T. and Taylor, R.N. (2000) Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D. thesis, University of California, Irvine*.
- [8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. (2016). Multiparty Asynchronous Session Types. *J. ACM* 63(1): 9:1-9:67 (2016). <https://dl.acm.org/doi/10.1145/2827695>
- [9] Milos Simic, Jovana Dedeic, Milan Stojkov, and Ivan Prokic. (2024). A Hierarchical Namespace Approach for Multi-Tenancy in Distributed Clouds. *IEEE Access* 12: 32597-32617 (2024). <https://doi.org/10.1109/ACCESS.2024.3369031>
- [10] Raymond Hu and Nobuko Yoshida. (2017). Explicit connection actions in multiparty session types. *In FASE 2017, Proceedings. Lecture Notes in Computer Science, vol. 10202, pp. 116-133. Springer (2017)*. https://doi.org/10.1007/978-3-662-54494-5_7
- [11] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. (2023). VeriFx: Correct Replicated Data Types for the Masses. *ECOOP 2023*. <https://doi.org/10.4230/LIPIcs.ECOOP.2023.9>
- [12] Marco Giunti, Hervé Paulino, and António Ravara. (2023). Anticipation of Method Execution in Mixed Consistency Systems. *ACM SAC 2023*. <https://doi.org/10.1145/3555776.3577725>
- [13] Hervé Paulino, Ana Almeida Matos, Jan Cederquist, Marco Giunti, João Matos, and António Ravara. (2023). AtomIS: Data-Centric Synchronization Made Practical. *OOPSLA 2023*. <https://doi.org/10.1145/3622801>
- [14] Eduardo Geraldo, João Costa Seco, and Thomas Hildebrandt. (2024). Declarative Choreographies with Data, Time and Information Flow Security. *Under submission*.
- [15] Thomas T. Hildebrandt, Hugo-Andrés López-Acosta, and Tijs Slaats. (2023). Declarative choreographies with time and data. *Proc. 21st Int. Conf. of Business Process Management*, 490:73–89. https://doi.org/10.1007/978-3-031-41623-1_5
- [16] Stéphanie Delaune and Lucca Hirschi. (2017). A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *J. Log. Algebraic Methods Program.* 87: 127-144 (2017). <https://doi.org/10.1016/j.jlamp.2016.10.005>
- [17] M. Popovic, M. Popovic, I. Kastelan, M. Djukic, and S. Ghilezan. (2023). A Simple Python Testbed for Federated Learning Algorithms. *Zooming Innovation in Consumer Technologies Conference (ZINC), Novi Sad, Serbia, 2023, pp. 148-153*, <https://doi.org/10.1109/ZINC58345.2023.10173859>
- [18] McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B.A. (2017). Communication-efficient learning of deep networks from decentralized data. *AISTATS 2017. Proceedings of Machine Learning Research, vol. 54, pp. 1273-1282. PMLR (2017)*. <http://proceedings.mlr.press/v54/mcmahan17a.html>
- [19] Hoare, C.A.R. (1985). *Communicating Sequential Processes*. Prentice Hall (1985).
- [20] Sun, J., Liu, Y., and Dong, J.S. (2009). PAT: Towards flexible verification under fairness. *CAV 2009. Lecture Notes in Computer Science, vol. 5643, pp. 709-714*. https://doi.org/10.1007/978-3-642-02658-4_59

- [21] Ivan Prokic, Silvia Ghilezan, Simona Kasterovic, Miroslav Popovic, Marko Popovic, Ivan Kastelan. (2023). Correct Orchestration of Federated Learning Generic Algorithms: Formalisation and Verification in CSP. *ECBS 2023*: 274-288. https://doi.org/10.1007/978-3-031-49252-5_25
- [22] Simic, M., Prokic, I., Dedeic, J., Sladic, and G., Milosavljevic, B. (2021). Towards edge computing as a service: Dynamic formation of the micro data-centers. *IEEE Access* 9, 114468-114484 (2021). <https://doi.org/10.1109/ACCESS.2021.3104475>
- [23] Andreas V. Hess, Sebastian Mödersheim, Achim D. Brucker, and Anders Schlichtkrull. (2021). Performing Security Proofs of Stateful Protocols. *CSF 2021*. <https://doi.org/10.1109/CSF51468.2021.00006>
- [24] Andreas Hess, Sebastian Mödersheim, Achim Brucker, and Anders Schlichtkrull. (2024). PSPSP: A Tool for Automated Verification of Stateful Protocols in Isabelle/HOL. *Submitted, 2024*. [Manuscript](#)
- [25] Andreas Hess, Sebastian Mödersheim, and Achim Brucker. (2023). Stateful Protocol Composition in Isabelle/HOL. *ACM Transactions on Privacy and Security*, 2023. <https://doi.org/10.1145/3577020>