Timed B2Scala: Extending B2Scala with Time

Doha Ouardi
Faculty of Computer Science
University of Namur
Namur, Belgium
ORCID: 0009-0007-0213-0748

Jean-Marie Jacquet
Faculty of Computer Science
University of Namur
Namur, Belgium
ORCID: 0000-0001-9531-0519

***Abstract*—Timed B2Scala extends the B2Scala tool by integrating explicit timing into the Bach coordination language and encapsulating topic-based communication into Bach primitives. This extension aims to support the modeling and experimentation of time-sensitive behaviors in distributed systems, particularly in contexts such as cyber-physical systems. While Timed B2Scala provides the means to implement and test scenarios involving temporal constraints and topic-based communication, it does not propose a theoretical model to reason on such systems. Instead, this paper focuses on the tool's implementation and its application to a specific use case inspired by the ROS2 system, demonstrating the capabilities of Timed B2Scala in addressing timing and topic-specific requirements. The presented work highlights the potential of this approach for advancing the exploration and verification of timing-related aspects in distributed systems.**

***Index Terms*—Coordination, Bach, B2Scala, Symbolic Model, Security protocols, ROS2.**

I. Introduction

Distributed systems, particularly those in cyberphysical domains, increasingly rely on precise coordination and real-time communication to meet the demands of modern applications [20]. These systems often require modeling frameworks capable of addressing time-sensitive interactions and dynamic context-aware communication patterns. Errors in timing or message routing can lead to critical failures in applications such as autonomous vehicles, smart grids, and industrial control systems [12].

The Bach coordination language [7]–[10], integrated into Scala through the B2Scala tool, provides a foundation for exploring concurrency and interaction in distributed systems. However, the original Bach language lacks the constructs to explicitly address timing constraints and topic-based communication, which are essential for real-time and context-aware systems. These limitations become significant when attempting to model complex systems like ROS2, which is based on a publish-subscribe communication model that involves timing constraints and topic-specific message exchanges.

This paper introduces Timed B2Scala, an extension of the B2Scala framework that incorporates timing constraints and topic-based communication primitives. Timing constraints enable the representation of time-sensitive interactions, while topic-based communication supports selective message exchange based on defined contexts. Together, these features aim to address the specific requirements of distributed systems operating in real-time environments.

This work was supported by the NADI Research Institute.

The ROS2 communication framework serves as the primary use case for this work. ROS2's publish-subscribe model [5], coupled with its timing and topic-based requirements, is modeled in Bach using the proposed extensions. Furthermore, this paper demonstrates an attempt to represent an attack scenario within this framework. It is important to note that the work presented here focuses on proposing how the model can be implemented and demonstrating its initial application. Testing and validating the implementation in the tool are beyond the scope of this paper but are addressed in D. Ouardi's PhD thesis, which more broadly explores coordination-based process algebra for security protocols.

This paper outlines the proposed extensions to the B2Scala framework and their application to ROS2's communication model and also highlights the potential of these extensions to address the demands of real-time and topic-based systems, providing a foundation for future research and tool refinement in this domain.

II. Background

A. Bach Language

Bach is a coordination language based on the Linda model [3], [4], which uses a shared space for process communication. Its core primitives, ask, nask, get, and tell, offer a synchronization between processes through the availability of information. Table II provides an overview of these primitives.

Primitive	Syntax	Description
`tell`	`tell(t)`	Inserts a tuple `t` into the shared tuple space.
`get`	`get(t)`	Retrieves and removes a tuple matching `t` from the shared tuple space.
`ask`	`ask(t)`	Checks if a tuple matches `t` in the shared tuple space (non-destructive).
`nask`	`nask(t)`	Checks if none of the tuples of the shared tuple space match `t`.

TABLE I
PRIMITIVES IN BACH COORDINATION LANGUAGE

The Bach coordination language also provides classical compositional operators for building complex instructions in concurrent systems [13], [14], as shown in Table II. The *sequential composition* operator (`P ; Q`) ensures that process `P` is executed fully before process `Q` begins, establishing a strict order of execution. The *parallel composition* operator

(`P || Q`) allows processes `P` and `Q` to execute concurrently, with potential interactions occurring through the shared tuple space. The *non-deterministic choice composition* operator (`P + Q`) enables the system to execute either `P` or `Q` depending on which of the alternatives can make a step.

Composition Operator	Syntax	Description
Sequential Composition	`P; Q`	Executes process `P`, then process `Q` in sequence.
Parallel Composition	`P || Q`	Executes processes `P` and `Q` in parallel.
Choice Composition	`P + Q`	Represents a non-deterministic choice; either `P` or `Q` is executed.

TABLE II
COMPOSITIONAL OPERATORS IN BACH COORDINATION LANGUAGE

B. The R and D Families

In traditional coordination models, time is often implicit or absent, which limits their applicability in systems requiring precise temporal reasoning. The proposed extensions, the **D family** and the **R family** in [15], integrate time into the Bach coordination language, by introducing delays and temporal tuple management [16], [17].

The D Family: This extension incorporates timed delays in process execution. The `delay(d)` primitive forces a process to pause for `d` units of time before proceeding. The semantics of the D family operates in two phases: an initial phase for untimed actions (e.g., `tell`, `get`, `ask`, `nask`) and a time progression phase during which delays decrease. Composition operators such as sequence (`P; Q`), parallel (`P || Q`), and choice (`P + Q`) are adapted to support timed actions, ensuring that temporal constraints seamlessly integrate into process coordination.

Figure 1 illustrates the semantics of this family. We define configurations as pairs of an Agent A and a store σ, denoted as $\langle A, \sigma \rangle$, with E denoting terminated computations. Rule (D1) governs the transition to a delayed version of the process when no immediate actions can be performed. Rule (D2) describes the progression of a zero-delay process, which allows it to move to the next state.

$$(D1) \qquad \frac{A \neq E, A \neq A^-, <A,\sigma> \not\rightarrow}{<A,\sigma> \rightsquigarrow <A^-,\sigma>}$$

$$(D2) \qquad <delay(0),\sigma> \longrightarrow <E,\sigma>$$

Fig. 1. Theoretical Rules (D1) and (D2)

Furthermore, Figure 2 presents how delays (`delay(d)`) and composition operators evolve as time advances. For example:

- `delay(d)⁻ = delay(d-1)`: The delay decreases as time progresses.
- `(B || C)⁻ = B⁻ || C⁻`: Parallel composition adapts as both components evolve over time.

$$\begin{aligned}
tell(t)^- &= tell(t) \\
ask(t)^- &= ask(t) \\
nask(t)^- &= nask(t) \\
get(t)^- &= get(t) \\
delay(0)^- &= delay(0) \\
delay(d)^- &= delay(d-1) \\
(B\ ;\ C)^- &= B^-\ ;\ C \\
(B \parallel C)^- &= B^- \parallel C^- \\
(B\ +\ C)^- &= B^-\ +\ C^-
\end{aligned}$$

Fig. 2. Theoretical Equations for the D Family Semantics

$$(T0) \qquad <tell_0(t),\sigma> \longrightarrow <E,\sigma>$$

$$(Tr) \qquad \frac{d > 0}{<tell_d(t),\sigma> \longrightarrow <E,\sigma \cup \{t_d\}>}$$

$$(Ar) \qquad \frac{d > 0}{<ask_d(t),\sigma \cup \{t_k\}> \longrightarrow <E,\sigma \cup \{t_k\}>}$$

$$(Nr) \qquad \frac{d > 0,\ \nexists k : t_k \in \sigma}{<nask_d(t),\sigma> \longrightarrow <E,\sigma>}$$

$$(Gr) \qquad \frac{d > 0}{<get_d(t),\sigma \cup \{t_k\}> \longrightarrow <E,\sigma>}$$

$$(Wr) \qquad \frac{A \neq E, A \neq A^- \text{ or } \sigma \neq \sigma^-, <A,\sigma> \not\rightarrow}{<A,\sigma> \rightsquigarrow <A^-,\sigma^->}$$

Fig. 3. Theoretical Rules for R family

The R Family: This extension introduces lifetimes for tuples in the shared tuple space. Each tuple is associated with a duration `d`, specifying its validity. The R family defines new primitives for interacting with these timed tuples:

- $\texttt{tell}_d$`(t)`: Inserts a tuple `t` with a lifetime `d`.
- $\texttt{get}_d$`(t)`: Retrieves and removes a tuple `t` if it exists within its lifetime `d`.
- $\texttt{ask}_d$`(t)`: Checks if a tuple `t` exists within its lifetime `d` (non-destructive).
- $\texttt{nask}_d$`(t)`: Ensures that a tuple `t` does not exist within its lifetime `d`.

The six operational rules of the R family, as shown in Figure 3, govern the behavior of tuples with respect to their lifetime. These rules are described as follows:

- `(T0)`: A tuple with a lifetime of zero is immediately discarded upon insertion.
- `(Tr)`: A tuple with a lifetime `d > 0` is inserted into the tuple space with its remaining duration `d`.
- `(Ar)`: A tuple with a lifetime `d > 0` is retrieved from the tuple space if it exists within its valid lifetime. If the tuple is still valid, it is returned without being removed.
- `(Nr)`: A tuple with a lifetime `d > 0` is not retrieved if

$$
\begin{aligned}
tell_d(t)^- &= tell_d(t) \\
ask_d(t)^- &= ask_{max\{0,d-1\}}(t) \\
nask_d(t)^- &= nask_{max\{0,d-1\}}(t) \\
get_d(t)^- &= get_{max\{0,d-1\}}(t) \\
E^- &= E \\
(B\ ;\ C)^- &= B^-\ ;\ C \\
(B\ ||\ C)^- &= B^-\ ||\ C^- \\
(B\ +\ C)^- &= B^-\ +\ C^-
\end{aligned}
$$

$$\sigma^- = \{t_{d-1} : t_d \in \sigma, d > 1\}$$

Fig. 4. Theoretical Equations for the R Family Semantics

it is not present in the tuple space or has expired. If no valid tuple is found, the operation has no effect.

- `(Gr)`: A tuple with a lifetime `d > 0` is retrieved and removed from the tuple space if it exists within its lifetime. The tuple is returned and the tuple space is updated accordingly.
- `(Wr)`: When a process is attempting to transition from one state to another, and either the tuple or the tuple space has changed (i.e., a tuple has expired or been removed), this rule adjusts the state of the system by ensuring that the tuple space is updated accordingly.

These rules ensure that tuples automatically expire when their lifetime elapses, maintaining temporal consistency in the tuple space. Additionally, Figure 4 explains how the semantics of the R family handle tuple operations as time advances. For instance:

- $\texttt{tell}_d\texttt{(t)}^- = \texttt{tell}_d\texttt{(t)}$: The operation remains unaffected by time progression.
- $\texttt{ask}_d\texttt{(t)}^- = \texttt{ask}_{\max(0,d-1)}\texttt{(t)}$: The remaining lifetime of the tuple decreases as time progresses.

C. B2Scala Tool

The B2Scala tool [21], [22] provides an implementation of the Bach coordination language as an internal domain-specific language (DSL) embedded within Scala [19]. It facilitates the modeling and experimentation of coordination mechanisms in distributed systems by leveraging Scala's type system and functional programming features. The tool implements the core primitives of Bach, including operations for tuple-based communication and synchronization in a shared tuple space. In B2Scala the Needham-Schroeder protocol was analyzed, revealing a man-in-the-middle attack first identified by G. Lowe [19].

In the next section, we present the extension of B2Scala to Timed B2Scala, which incorporates the time management features introduced by the R and D families of the Bach coordination language. Due to space limitations, the detailed implementation of these features will not be covered in the following sections; however, the reader will find the elements necessary to understand our work.

III. Timed B2Scala

In Timed B2Scala, agents are designed to incorporate time-based behaviors, enabling the modeling of temporal dynamics in protocol execution. Each agent is associated with a time duration, which by default is set to `infiniteDuration`, meaning that the agent persists indefinitely unless explicitly defined otherwise. For instance, the `tell` operation creates a `TelltAgent` with a specified duration. This agent holds both an `SI_Term` and a duration. While the `TelltAgent`'s duration is unaffected by the `increase_time` function, other agents such as `GettAgent`, `AsktAgent`, and `NasktAgent` evolve over time, reflecting time-sensitive behaviors.

Agents in Timed B2Scala are defined using constructs of the following form:

```
val P = Agent { (tellt(1,f(1, 2))
              * delay(4)
              || (tellt(a) + tellt(b)) }
```

The `Agent` object encapsulates a Bach agent within Scala, mapping a (internal) `BSC_Agent` into a so-called `CalledAgent` via a function that returns an agent (a thunk). This design enables lazy evaluation, which is particularly useful for recursively defined agents. The following Scala declaration is therefore key in the implementation.

```
object Agent {
  def apply(agent: BSC_Agent) =
    CalledAgent(() => agent) }
```

The `BSC_Agent` trait refers to our tool's implementation of these agents following Bach semantics. It provides methods for agent composition, supporting different composition operators. Since the semicolon (`;`) is a reserved symbol in Scala, sequential composition is represented by the $\star$ operator. Additionally, the operators $\star$, `||`, and `+` are implemented using Scala's postfix method facility. For example, the construct `tell(t) + tell(u)` invokes the `+` method on `tell(t)` with `tell(u)` as its argument. These operators utilize *call-by-name* for agents, delivering structures via thunks. This composition enables agents to interact concurrently or sequentially, with their execution inherently influenced by timing constraints.

The store in Timed B2Scala is time-aware, designed to track not only the presence of terms but also their timestamps and durations. The structure of the store is as follows:

```
class BSC_Store {
var theStore = Map[SI_Term,
    List[(TTimeStamp, TDuration)]]() }
```

This design ensures that time-sensitive operations can be performed efficiently, enabling or disabling actions based on the current time context.

The execution model in Timed B2Scala introduces both time progression and dynamic agent selection. As a basic ingredient, it executes a transition step through a so-called `run_one` function. However, instead of processing a single agent in a

loop, the model maintains a list of agents and randomly selects one during each iteration. Once the `run_one` function executes the selected agent, it is removed from the list. Following this, both the belief base and the remaining agents have their time values updated through the `increase_time()` function. The variables `it_current_agent` and `it_sigma` are used to track these updates, ensuring that execution accurately reflects time evolution while allowing flexible agent interactions. The execution loop continues until a termination condition is met, where the `failure` in the execution loop acts as a stop condition. It is updated inside the run_one function: if an agent cannot be processed (due to an invalid state or unmet conditions), run_one returns false, setting failure to true. This prevents further execution, terminating the loop. The failure state is checked at each iteration, ensuring execution stops immediately upon detecting a failure. The execution loop is as follows:

```
while (current_list_agent.nonEmpty
   && !failure
   && (it_current_agent != current_agent
   || it_sigma)) {
    run_one(ag_choice, formula) match {
       . . .  }
    if (current_agent != EmptyCxtAgent) {
       . . .
       it_sigma = BB.increase_time() } }
```

Timed B2Scala is applied to analyze time-sensitive protocols and potential attacks, such as a man-in-the-middle attack in a ROS2-based system. It is important to note that this paper focuses on testing scenarios within the ROS2 protocol, rather than verifying the correctness properties of the protocol.

IV. Case Study: The ROS2 Attack Scenario

ROS2 is a framework designed for the development of distributed systems, particularly within the context of robotics. It utilizes a publish-subscribe communication model, where the nodes exchange messages via topics. ROS2 includes real-time communication and a modular design, making it suitable for systems that require precise timing and context-aware behaviors. In the Bach framework, ROS2's communication patterns are modeled using a tuple space, where topics are represented as parameters. Publishers insert tuples into the space, while subscribers retrieve them based on topic and timing constraints.

Figure 5 illustrates the communication flow in the ROS2 attack scenario, where information is exchanged between two nodes (v1 and v2) refering to vehicle1 and vehicle2 via the shared space. The figure also highlights the role of an intruder who intercepts and updates data within the shared space. This scenario demonstrates how time-sensitive interactions within the tuple space can be exploited, making it an ideal use case for examining vulnerabilities in time-aware distributed systems.

The attack scenario closely resembles the Dolev-Yao model [11] of security analysis, in which the attacker is assumed to have the ability to intercept, modify, or forge messages in transit but lacks access to private keys or internal system states. In this case, the intruder's manipulation of the tuple space aligns with the Dolev-Yao model's assumption of an attacker capable of altering communications, potentially exposing vulnerabilities in real-time systems such as ROS2.

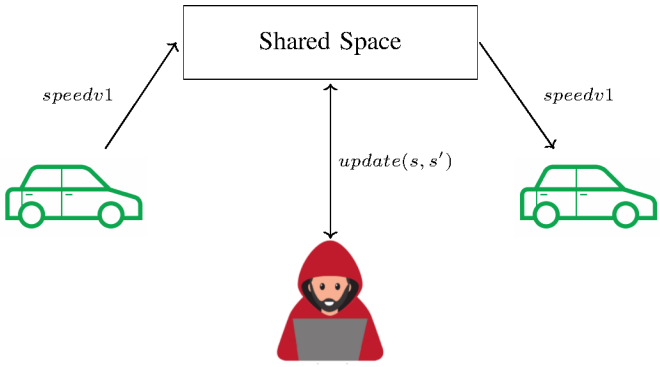

Fig. 5. The ROS2 Attack Scenario

A. Analyzing ROS2

1) In Bach: Publishers and subscribers are represented as processes interacting through a shared tuple space. Tuples are tagged with topics, enabling selective message exchange. Timing constraints are applied to simulate real-time behavior. We consider the nodes `v1` and `v2` as agents representing "vehicle1" and "vehicle2". The publisher, `v1`, sends the speed limit `speedv1` via the shared space with the topic using the primitive $\texttt{tell}_1\texttt{(t)}$, where 1 represents the duration d. The subscriber, `v2`, retrieves this information from the shared space using the primitive $\texttt{get}_1\texttt{(t)}$. The `Intruder` behaves differently by first retrieving the original message ($\texttt{get}_1\texttt{()}$), modifying the speed value (`update()`), publishing the modified message ($\texttt{tell}_1\texttt{()}$). This introduces a manipulation of the information in the shared space.

```
proc
    v1 = tell_1(msg(speedv1, topic)) ;
         delay(1).

    v2 = get_1(msg(speedv1, topic));
         delay(1).

     i = get_1(msg(speedv1, topic));
         tell_1(update(speedv1, s_prime);
         tell_1(msg(s_prime, topic);
         delay(1).
```

2) In Timed B2Scala: The publisher (vehicle v_1) sends the speed limit s to a shared space using the topic t with the primitive `tellt()`, and the duration is set to 1 time unit. The subscriber (vehicle v_2) retrieves this information from the shared space using the primitive `gett()`.

We define tokens for the topic (t), vehicles (v_1 and v_2), the speed (s), and the modified speed (s'). Additionally, we declare two case classes:

- `msg()`, which represents a message containing the speed and topic.
- `update()`, which models an update where the speed s is modified to s'.

The system consists of three agents:

1) v_1 (Publisher): Sends the speed value s to the topic t.
2) v_2 (Subscriber): Retrieves the speed value s from the topic t.
3) i (Intruder): Retrieves s, updates it to s', and republishes the modified speed.

Note that the `Agent` construct refers to the Scala construct introduced in Section C above.

```
val t = Token("topicv1")
val v1 = Token("vehicle1")
val v2 = Token("vehicle2")
val s = Token("speedv1")
val s_prime = Token("s_prime")

case class msg(s: SI_Term, t: SI_Term)
                      extends SI_Term
case class update(s: SI_Term,
     s_prime: SI_Term) extends SI_Term

val v1 = Agent {
      tellt(1, msg(s, t)) *
      delay(1) *
      v1 }
val v2 = Agent {
      gett(1, msg(s, t)) *
      delay(1) *
      v2 }
val i = Agent {
      gett(1, msg(s, t)) *
      tellt(1, update(s, s_prime)) *
      tellt(1, msg(s_prime, t)) *
      delay(1) *
      i }
```

In B2Scala, logical formulas are used to constrain system execution and verify security requirements. For the ROS2 attack scenario, security properties can be modeled by using formulas to ensure data integrity. For example, a formula can check that the second vehicle (v2) always receives the correct speed value sent by the first vehicle (v1), and detect violations if the value is altered by an intruder. While B2Scala doesn't directly verify formal properties like safety and liveness, its logical formulas help identify vulnerabilities and security breaches in time-sensitive systems like ROS2 by constraining execution and highlighting deviations from expected behavior. It is important to note that the work on the ROS2 scenario is still a work in progress, with ongoing efforts to refine and extend the tool's capabilities.

V. Conclusion

The Timed B2Scala tool represents a important step in extending the B2Scala framework to address time-sensitive and topic-based requirements in distributed systems. While this work demonstrates the feasibility of incorporating these extensions and showcases their application in modeling and experimentation scenarios, such as the ROS2 attack scenario, it stops short of delivering a formalized model or comprehensive verification capabilities.

Future research aims at elevating the maturity of Timed B2Scala by enhancing its theoretical foundations and practical implementations to align with the capabilities of established tools like ProVerif [2], Tamarin [18], Ivy [6], and Uppaal [1]. This evolution will involve rigorous formalization of the proposed extensions, integration of automated reasoning techniques, and validation against a broader range of complex use cases. Achieving these goals will position Timed B2Scala as a robust tool for analyzing and verifying real-time, context-aware distributed systems in both research and applied domains. This is the direction we aim to take in our PhD thesis, focusing on refining these tools and frameworks to offer stronger guarantees in the modeling, analysis, and verification of time-sensitive distributed systems.

References

[1] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: Uppaalâa tool suite for automatic verification of real-time systems. In: Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems IIIâ¯: Verification and Control: Verification and Control. p. 232â243. Springer-Verlag, Berlin, Heidelberg (1996)

[2] Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001. pp. 82–96 (2001)

[3] Brogi, A., Jacquet, J.M.: On the Expressiveness of Linda-like Concurrent Languages. Electronical Notes in Theoretical Computer Science **16**(2), 61–82 (1998)

[4] Brogi, A., Jacquet, J.M.: On the Expressiveness of Coordination via Shared Dataspaces. Science of Computer Programming **46**(1-2), 71–98 (2003)

[5] Choton, J.C., Gupta, L., Prabhakar, P.: Formal modeling and verification of publisher-subscriber paradigm in ros 2 (2025), https://arxiv.org/abs/2412.16186

[6] Crochet, C., Rousseaux, T., Piraux, M., Sambon, J.F., Legay, A.: Verifying quic implementations using ivy. In: Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC. p. 35â41. EPIQ '21, Association for Computing Machinery, New York, NY, USA (2021)

[7] Darquennes, D., Jacquet, J.M., Linden, I.: On Density in Coordiantion Languages. In: Canal, C., Villari, M. (eds.) CCIS 393, Advances in Service-Oriented and Cloud Computing, ESOCC 2013, Proceedings of Foclasa Workshop. pp. 189–203. Springer, Malaga, Spain (2013)

[8] Darquennes, D., Jacquet, J.M., Linden, I.: On the Introduction of Density in Tuple-Space Coordination Languages. In: Science of Computer Programming. Springer (2013)

[9] Darquennes, D., Jacquet, J.M., Linden, I.: On Distributed Density in Tuple-based Coordination Languages. In: Cámara, J., Proença, J. (eds.) Proceedings 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems. EPTCS, vol. 175, pp. 36–53. Springer, Rome, Italy (2015)

[10] Darquennes, D., Jacquet, J.M., Linden, I.: On Multiplicities in Tuple-Based Coordination Languages: The Bach Family of Languages and Its Expressiveness Study. In: Serugendo, G.D.M., Loreti, M. (eds.) Proceedings of the 20th International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 10852, pp. 81–109. Springer (2018)

[11] Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory **29**(2), 198–208 (1983)

[12] Eidson, J.C., Lee, E.A., Matic, S., Seshia, S.A., Zou, J.: Time-centric models for designing embedded cyber-physical systems. University of California, Berkeley, Technical Memorandum. UCB/EECS-2009-135 (2009)
[13] Jacquet, J.M., Barkallah, M.: Scan: A Simple Coordination Workbench. In: Nielson, H.R., Tuosto, E. (eds.) Proceedings of the 21st International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 11533, pp. 75–91. Springer (2019)
[14] Jacquet, J.M., Barkallah, M.: Anemone: A workbench for the Multi-Bach Coordination Language. Science of Computer Programming **202**, 102579 (2021)
[15] Jacquet, J.M., Bosschere, K.D., Brogi, A.: On Timed Coordination Languages. In: Porto, A., Roman, G.C. (eds.) Proc. 4th International Conference on Coordination Languages and Models. Lecture Notes in Computer Science, vol. 1906, pp. 81–98. Springer (2000)
[16] Linden, I., Jacquet, J.M.: On the Expressiveness of Absolute-Time Coordination Languages. In: Nicola, R.D., Ferrari, G., Meredith, G. (eds.) Proc. 6th International Conference on Coordination Models and Languages. Lecture Notes in Computer Science, vol. 2949, pp. 232–247. Springer (2004)
[17] Linden, I., Jacquet, J.M., Bosschere, K.D., Brogi, A.: On the Expressiveness of Relative-Timed Coordination Models. Electronical Notes in Theoretical Computer Science **97**, 125–153 (2004)
[18] Meier, S., Schmidt, B., Cremers, C., Basin, D.: The tamarin prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 696–701. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
[19] Ouardi, D., Barkallah, M., Jacquet, J.M.: The b2scala tool: Integrating bach in scala with security in mind. Electronic Proceedings in Theoretical Computer Science **414**, 58â76 (Dec 2024)
[20] Shrivastava, A., Derler, P., Baboud, Y.S.L., Stanton, K., Khayatian, M., Andrade, H.A., Weiss, M., Eidson, J., Chandhoke, S.: Time in cyber-physical systems. In: Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. CODES '16, Association for Computing Machinery, New York, NY, USA (2016)
[21] UNamurCSFaculty: B2scala (2024), `https://github.com/UNamurCSFaculty/ B2Scala`, software
[22] UNamurCSFaculty: B2scala (version 1.0) (2024), `https://github.com/ UNamurCSFaculty/ B2Scala/releases/ tag/ 1.0`, software