Distributed Coordination Runtime Assertions for the Peer Model

Eva Kühn, Sophie Therese Radschek, Nahla Elaraby

Distributed Coordination Runtime Assertions for the Peer Model

eva Kühn[1], Sophie Therese Radschek[1], and Nahla Elaraby[1,2]

[1] TU Wien, Argentinierstr. 8, Vienna, Austria
{eva.kuehn, sophie.radschek, nahla.el-araby}@tuwien.ac.at,
WWW home page: www.complang.tuwien.ac.at/eva
[2] Canadian International College – CIC, Egypt

Abstract. Major challenges in the software development of distributed systems are rooted in the complex nature of coordination. Assertions are a practical programming mechanism to improve the quality of software in general by monitoring it at runtime. Most approaches today limit assertions to statements about local states whereas coordination requires reasoning about distributed states. The Peer Model is an event-based coordination programming model that relies on known foundations like shared tuple spaces, Actor Model, and Petri Nets. We extend it with distributed runtime invariant assertions that are specified and implemented using its own coordination mechanisms. This lifts the concept of runtime assertions to the level of coordination modeling. The concept is demonstrated by means of an example from the railway domain.

Keywords: Coordination model, runtime assertions, distributed systems, tuple space.

1 Introduction

The development of coordination is complex because of the asynchronous nature of distributed systems. Coordination models like the Actor Model [2], Petri Nets [20], Reo [3] and the Peer Model [15] are well suited to model the interactions in concurrent systems. They allow reasoning about coordination at a certain abstraction level. Such a model-driven approach has the advantage to specify the system unambiguously, and to support the verification of system properties. Assertions are a practical programming mechanism to improve the quality of software in general by monitoring it at runtime. Most approaches today limit assertions to statements about local states whereas coordination requires reasoning about distributed states.

The objective of this paper is to introduce event-based, asynchronous, and distributed runtime assertions to a coordination model. This approach shall lift the concept of runtime assertions to the level of coordination modeling. A main problem is to find a good tradeoff between a very costly mechanism and a good quality of verification. A further requirement is to provide a notation that is well integrated into the notation of the coordination model so that developers need

not learn a different language. This way, the assertions can be directly added to the model of the application using the same modeling concepts. Keeping the number of concepts that a developer has to comprehend small contributes to usability [24].

We use the Peer Model (see Section 3) as reference coordination model notation and introduce an assertion mechanism based on its own modeling language. Checking of distributed assertions can in turn be considered a coordination problem. We propose therefore model transformation algorithms that automatically translate the assertion specifications into coordination constructs of the model. Thus they become part of the application model and can be implemented with the same mechanisms. In this way, the extended Peer Model is a contribution towards verification of practical distributed systems.

The paper is structured as follows: Section 2 summarizes related work on distributed assertion mechanisms. Section 3 gives an overview of the Peer Model. In Section 4 we extend the Peer Model by distributed runtime invariant assertions that are specified and implemented using its own coordination mechanisms. In Section 5 the concept is demonstrated by means of an example from the railway domain. Section 6 summarizes the results and Section 7 gives an outlook to future work.

2 Related Work

Proper assertions for verification of coordination models should be declarative, event-driven and distributed. Assertions express correctness properties that must hold in all system states. Coordination model assertions should be model-based and use the same or similar notation of the model to keep the number of concepts small. For concurrent distributed systems, run time assertions [6] provide high confidence that the properties to be verified are fulfilled. However, assertions in this case should be of asynchronous nature so that they neither disturb the execution nor interfere with user code and data by any means. They should be bootstrapped for implementation with native model mechanisms.

[9] introduces a solution for coordinating activities of multiple intelligent agents using the tuple space computational model. The tuple space-based problem solving framework is implemented on an Intel Hypercube iPSC/2 allowing multiple rule-based systems concurrently performing their dedicated interrelated tasks. Their approach to build a rule-based system resembles our approach towards distributed assertions, but on a different level of reasoning.

A language for determining correct behavior in distributed systems is described in [25]. This approach succeeds to verify distributed systems but introduces a different language and notation. Also, waiting states for operators' actions add much overhead to the run time check which is not acceptable in most systems, especially those of safety-critical nature.

Reusable assertions were developed in [13], where the assertions can be used at different abstraction levels. The approach handles the timing but does not provide asynchronous assertions for distributed system modules.

In [14] a lightweight modeling language (Alloy), based on first-order relational logic is used to model and analyze Reo connectors. Reo [3] is a coordination language based on components and connectors. The model presented in this work preserves the original structure of the Reo network, avoiding a complex translation effort. The approach handles basic channel types, compositional construction of more complex connectors, constraints on the environment, and networks exposing context-sensitive behavior. However, in Alloy, no asynchronous assertion mechanism is provided and the properties are defined in terms of first-order predicates, and not in the same model notation.

[4] investigates runtime verification where the properties are expressed in linear time temporal logic (LTL) or timed linear time temporal logic (TLTL). Runtime verification is identified in comparison to model checking and testing. A three-valued semantics (with truth values true, false, inconclusive) is introduced as an expressive interpretation indicating whether a partial observation of a running system meets an LTL or TLTL property. For LTL, a minimal size deterministic monitor is generated identifying a continuously monitored trace as either satisfying or falsifying a property as early as possible, similarly for TLTL. The approach is successful and provides foundation for real-time monitoring of system properties. The approach does not provide asynchronous assertions and uses a different notation from the model.

Mining of the simulation data as an effective solution for generation of assertions is presented in [5]. The research provides an efficient way to develop a highly trusted set of assertions and solves the incompleteness problem of Assertion Based Verification (ABV). The assertions are not based on the model notation and need effort of the developer to integrate them into the runtime. Distributed asynchronous mechanisms for assertions are not provided.

The work presented in [19] investigates the realization of infrastructures to maintain and access data and use it in assertions. Assertions can be managed across distributed system to support sophisticated assertions for intercommunications in distributed systems. A tool-chain is provided for programming assertions on interaction history written in regular expressions that incorporate inter-process and inter-thread behavior amongst multiple components in a distributed system. The mechanism is promising but not model-based.

A novel approach described in [26] uses machine learning for automatic assertion generation. The approach releases the burden of manually specifying the assertions, which is a time-consuming and error-prone process. Similarly the authors of [21] also deal with the problem of insufficiently written assertions and propose an automatic approach for assertion generation based on active learning. The approach targets complex Java programs, which cannot be symbolically executed. Test cases are used as a base for assertion generation and active learning to iteratively improve the generated assertions is applied. Both approaches are runtime-based and automatic, but they are not applicable for distributed systems where coordination needs to be defined and asserted asynchronously to validate the system functionality.

[8] introduces runtime verification as a complementary validation technique for component-based systems written in the BIP (Behavior, Interaction and Priority) framework. The proposed solution dynamically builds a minimal abstraction of the current runtime state of the system so as to lower the overhead. Monitors are directly generated as BIP components where the C code generator of BIP generate actual monitored C programs and remain compatible with previously proposed runtime verification frameworks. However, an adaptation layer is needed to adapt monitors generated by other existing tools.

A general approach to monitor the specifications of decentralized systems is presented in [7]. The specification is considered as a set of automata associated with monitors attached to different components of the decentralized system. A general monitoring algorithm is developed to monitor the specifications. Execution History Encoding (EHE) data structure is introduced as a middle ground between rewriting and automata evaluation. Rewriting is restricted to Boolean expressions, parameters are determined and their respective effect on the size of expressions and the upper bounds are fixed. A similar decentralized algorithm for runtime verification of distributed programs is introduced in [18]. The technique works on asynchronous distributed systems, where a global clock is not assumed. The specification language used is full LTL so that temporal properties can be monitored. The algorithm can also determine the verification verdict once a total order of events in the system under inspection can be constructed. The presented algorithm addresses shortcomings in other algorithms as it does not assume a global clock, is able to verify temporal properties and is sound and complete. However, the main concern is the large number of monitoring messages introducing an increased communication and memory overhead.

[12] proposes a session-based verification framework for concurrent and distributed ABS models. Applications are categorized with respect to the sessions in which they participate. Their behaviors are partitioned based on sessions, which include the usage of future. The presented framework extends the protocol types through adding terms suitable for capturing the notion of futures, accordingly the communication between different ABS endpoints can be verified by the corresponding session-based composition verification framework. Timing is handled but not in an asynchronous way.

In summary, none of the listed approaches provides assertions that are specified at the model level, are runtime-based, asynchronous and distributed, and the implementation of which can be bootstrapped with native model mechanisms. The motivation was therefore to use the Peer Model (see Section 3) as reference coordination model notation and to provide a full set of assertions that can be translated to the same model notation, whereby all mentioned requirements for assertions are fulfilled. This way, the intent is to extend the Peer Model to become capable to trustfully monitor distributed systems.

3 The Peer Model in a Nutshell

The Peer Model [15] is a coordination model that relies on known foundations like shared tuple spaces [10, 11, 17], Actor Model [2], and Petri Nets [20]. It clearly separates coordination logic from business logic and is well suited to model reusable coordination solutions in form of patterns. It provides a refinement-based modeling approach where you start with the highest abstraction layer of your application model, and stepwise improve the model[3]. The main concepts are briefly explained in the following.

Peer. A peer relates to an actor in the Actor Model [2]. It is an autonomous worker with in- and outgoing mailboxes, termed peer input container (PIC) and peer output (POC) container.

Container. A container is a sub-space in the tuple space that stores entries that are written, read, or taken (i.e., read and removed) in local transactions.

Entry. Entries are the units of information passed between peers. An entry has system- and application-defined properties, which are name/value pairs. Examples for system-defined properties are e.g.: `ttl` (time-to-life; if it expires, the entry becomes an exception entry that wraps the original entry), and `fid` (flow id). All entries with the same `fid` belong to the same flow, i.e., they logically belong together. An exception entry is a special system-defined entry. If it is caused by the expiration of an entry `ttl` it inherits the `fid` of the entry causing the exception, and "wraps" the original entry in a property termed `entry`. The type of the exception is stored in the `excType` property.

Link. A link transports entries between two containers c1 and c2. It selects entries from c1 using a query. Link operations are `move` (take entries from c1 and write them to c2), `copy` (read entries from c1 and write them to c2), `test` (read entries from c1), `delete` (read and remove entries from c1), `create` (create entries of given type and count and write them to c2), and no operation specified (no container access; this link serves solely to perform tests on variables). A link can set or get variables (see below) and set or get properties of entries (if an operation was used that selected entries from c1). In these expressions also system-defined functions like `head()` (get first element of a list), `tail()` (get rest of a list) can be used. Finally, a link may have system-defined properties, e.g., `dest` (peer to whose `PIC` all selected entries shall be sent), and `mandatory` (flag whether the link is obligatory).

Wiring. The coordination behavior of a peer is explicitly modeled with wirings, which have some similarity with Petri Net transitions [20]. All wiring specifications are executed concurrently in so-called wiring instances. Each instance has an internal container termed Wiring Internal Container (WIC), which serves as a temporary, local entry collection. Each wiring instance is a local transaction [16] that transports entries between containers with help of links. A link from a PIC or POC to the WIC is termed "guard" (G) and a link from the WIC to a PIC or POC is termed "action" (A). The operational behavior of a wiring instance is the sequential execution of guards, and actions. The arrival of entries

[3] There exists an Event-B-based model checker for the Peer Model [22].

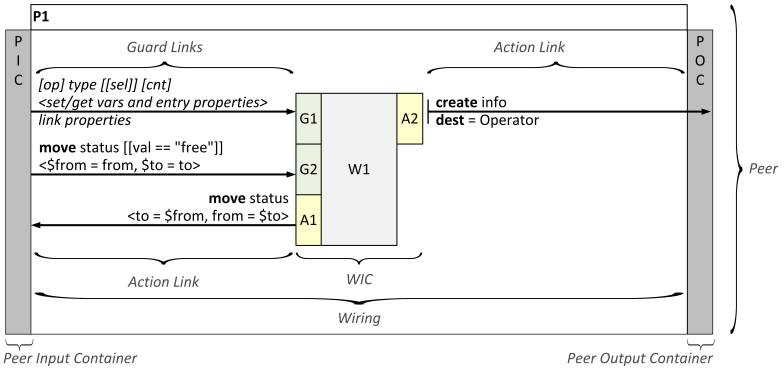

Fig. 1. Graphical notation for the main artifacts of the ground model.

in peer space containers triggers the execution of guards. A wiring is triggered if all of its guard links are satisfied. Wirings also have system-defined properties. Each instance of a wiring obtains a new WIC. Wirings can treat exceptions.

Query. A query consists of: (1) entry type (mandatory), (2) an integer number (default is 1) specifying how many entries shall be selected, or ALL (all entries in the container that can be read or taken, possibly 0), and (3) a selector expression. It is fulfilled when enough entries with specified type match the selector. It selects entries with same type and of compatible flows [15]. The first link of a wiring determines the flow. All further links select only entries with same `fid`, or not `fid` set. A selector can use `AND`, `OR` and `NOT` operators.

Variable. There exist application- and system-defined variables. The former are denoted with "$" and used in the scope of a wiring instance. The latter are denoted with "$$" and set by the runtime, e.g., $$PID (id of the current peer), $$WID (id of the current wiring), $$CNT (number of entries selected by a link query), and $$FID (current flow id of the wiring instance).

Fig. 1 shows the graphical notation of the core concepts. It depicts one peer with id P1 and one wiring termed W1. The wiring possesses two guard links (G1, G2), and two action links (A1, A2) – represented by arrows. G1 (connecting PIC and WIC) depicts the general notation: Operation, query and the access to variables and entry properties are located above the respective arrow and properties are denoted below. Guard links are directed towards the WIC, and action links are pointing in the other direction. If entries are read (and removed) from a container, then the arrow base is connected to it; if entries are written into a container, then the arrow head is connected to it. Otherwise, a vertical bar is used on the respective side to indicate that no entries "go through".

G2 shows a guard link that takes one entry of type status from the PIC, using a selector specifying that the val property of the entry must be set to free, and writes the entry into the WIC. It stores the from and to properties of the selected entry in local variables $from and $to. In action link A1 the wiring takes the above mentioned status entry from the WIC, swaps its to and from properties

with help of the local variables, and writes it back to the PIC. A2 creates one new entry of type info, and as the link property `dest` is set, the entry is sent to the PIC of the destination address, in this case the Operator peer. If `dest` is set, the arrow of the link goes through the POC. All system-defined terms are written in type writer style.

4 Introducing Peer Model Assertions

To improve the quality of specifications at the modeling level, the Peer Model is augmented with declarative invariant assertions. The goal of the assertion mechanism is to detect model inconsistencies at runtime. The trade-off between interference with normal execution, network traffic, and strict evaluation of assertions should be configurable. The mechanism shall be distributed, asynchronous and event-driven, assuming neither reliable nor ordered communication.

Assertions are statements about container states, because the shared coordination state of the distributed system is manifested in them. They follow the same principles as the Peer Model concepts (see Section 3), e.g., they have properties, are flow sensitive, and cause exception entries to be treated by the application. The consideration of the flow is fundamental, because the scope of each assertion check is restricted to one flow. This has the advantage that the correlation of distributed events can fully rely on the Peer Model's native flow mechanisms. Peers are assumed to report about assertion violations to one or more coordinators, which are peers in charge of controlling the assertion checking for a particular flow. Reporting considers actual and past states; liveness is out of scope of this mechanism.

We differentiate between assertions referring to containers of one single peer only (intra-peer assertions) and assertions involving multiple (local or distributed) peers (inter-peer assertions). For the former, violations are checked autonomously by the peer itself, for the latter, violation checking is understood as a distributed coordination problem. The proposed mechanism includes a translation of the declarative assertions to existing Peer Model constructs by a model translation algorithm (MTA) (see Section 4.1). For intra-peer assertion checks, respective wirings are statically added to the meta-model of the peers that monitor their local container states and create assertion exception entries on violation. The inter-peer assertion mechanism relies in principle on the same wiring type, but instead of exceptions, report entries are created and sent to coordinators, which in turn decide if exceptions must be created based on the status of received reports. Therefore, the overall assertion check can be reduced to an analogous wiring that checks a derived and normalized intra-peer assertion about these reports, generated by the MTA.

The required overhead consists only of carrying out the required tests when a new relevant event is received, storing the current reporting state and (in the inter-peer case) sending one report message to each coordinator on a new assertion violation. However, not all violations can be detected in case of communication failures. To cope with these issues and to support assertions about past

states, a history of the relevant events can be maintained in a dedicated internal peer container termed `HIC`. This history is created by extending the existing application wirings to capture all local changes and deletions of entries referenced by assertions in the `HIC`, which creates a certain overhead. However, the overhead can be reduced by deleting events that cannot be referenced any more by active assertions (e.g. a flow has expired), assigning time-to-live properties and introducing bounds on the depth of the history.

The expressiveness of the mechanism can be further enhanced by introducing variables shared between different sub-assertions (see below) of inter-peer assertions. This produces additional message overhead, as reports about property changes in relevant entries must be sent to all respective wirings that reference these properties by means of these variables. Variable reports are only sent, if the corresponding assertion is not violated, because – like in failing optional links – variables can only be assumed to have meaningful value, if the expression that sets them was successful. In addition, assertion wirings depend on variable reports to be present and are activated by each new variable report. The mentioned trade-off between strictness and expressiveness versus overhead is controlled by the assertion model. Namely, avoiding inter-peer variables, and not using assertions about the past and not demanding a history reduces the overhead to a minimum.

The coordinator(s) for inter-peer assertions are specified by means of assertion properties. Default is no coordinator, meaning that each sub-assertion creates local exceptions instead of sending reports to coordinators; the logic of how to handle them can be explicitly modeled. Here, the trade-off is between many coordinators with a larger message overhead and fewer coordinators leading to a potential bottle-neck and single point-of-failure.

The syntax of assertions is given in Tab. 1 and their operational semantics is defined with existing Peer Model constructs. An assertion is either a simple one (SAss) or a complex one (CAss), i.e., sub-assertions connected with logic operators `AND`, `OR`, `NOT` and $\rightarrow$ (equivalent to $\wedge$, $\vee$, $\neg$ and $\implies$). Sub-assertions of inter-peer assertions are intra-peer assertions and sub-assertions of intra-peer assertions are container assertions. Container assertions are satisfied, if the container contains exactly *quantor* entries of *type* that satisfy the query selector *sel*. Note that *type sel* and *set/get* refer to the Peer Model's link notation (see Fig. 1). Intra- and inter-peer assertions are satisfied if the propositional formula corresponding to the assertion – in which the propositions are the sub-assertions – is satisfied.

4.1 Model Translation Algorithm (MTA)

Intra-peer assertions are translated into single wirings located in the peer(s) corresponding to the context. Each assertion corresponds to a wiring that checks its truth value and raises an exception on violation. Such assertion wirings define and use all variables used in the corresponding assertion and overhead variables to evaluate the truth value of the assertion. These overhead variables are: 1 variable for each container sub-assertion to store the evaluation value, 2 variables

Ass	::=	*CAss* '[' *properties* ']' | *SAss* '→' *Ass*	assertion
CAss	::=	*SAss* | *CAss* 'AND' *CAss* | *CAss* 'OR' *CAss* | ('NOT' *CAss*)	complex assertion
SAss	::=	*Ctx* '{' *PeerAss* '}'	simple assertion
Ctx	::=	*pid* | *QCtx* | *var*	context
QCtx	::=	'[' *quantor* ']' [*var* 'IN'] '<' *type* '>'	quantified context
PeerAss	::=	*PeerCAss* | *PeerSAss* '→' *PeerAss*	peer assertion
PeerCAss	::=	*PeerSAss* | *PeerCAss* 'AND' *PeerCAss* | *PeerCAss* 'OR' *PeerCAss* | ('NOT' *PeerCAss*)	peer complex assertion
PeerSAss	::=	*Container* '{' *ContainerAss* '}' | '⟦' *sel* '⟧'	peer simple assertion
Container	::=	'PIC' | 'POC' | 'HIC'	container
ContainerAss	::=	['[' *quantor* ']'] *type* ['⟦' *sel* '⟧'] ['<' *set/get* '>']	container assertion
properties	::=	(* list of properties of form label = value *)	property list
quantor	::=	(* ALL, NONE, or a number (default = 1) *)	quantification
var	::=	(* local variable referring to a peer *)	variable
pid	::=	(* id of a concrete peer *)	peer id

Table 1. EBNF for Assertion Notation.

for each container sub-assertion to store the number of entries that fit the type and query, 1 to store the overall assertion value and 1 to store whether a violation must be reported. The assertion and overhead variables are initialised on a dedicated mandatory guard. Each container sub-assertion is tested using 3 guards: (1) tests how many entries there are in the container that fulfil the query of the assertion and saves the number in a variable; this guard also sets the variables used by the assertion. (2) tests how many entries there are in the container that fulfil the type of the assertion. (3) compares the quantities for (1) and (2) according to the quantor in the container sub-assertion and sets the corresponding truth variable to true if it is fulfilled. (1) and (2) are mandatory, (3) is only executed if the sub-assertion is asserted. The truth value of the overall assertion is set to true on an optional guard according to the logic formula of the intra-peer assertion. If the overall intra-peer assertion is violated and no report has yet been issued, a report is created in the peer input container (PIC) and an exception is raised in the peer output container (POC).

Each intra-peer assertion creates an overhead of k wirings with $3n + 2$ variables and $3n + 5$ links each (n ...# of container sub-assertions and k ...# of peers in the context). For each violation, 2 additional entries are created.

Fig. 2 shows the translated wiring of an intra-peer assertion. G1 defines a variable $\$q_i$ for the result of each *ContainerAss* (see Tab. 1), a variable $q for the result of the entire *PeerAss*, a variable $report to reflect whether the assertion has already been reported, and initializes all application variables of the wiring, if any. G1 is created once per wiring. Gj.k (j = 2, ..., n+1, k = 1 , 2, 3) are generated for each of the n *ContainerAss*: Gj.1 tests all entries fulfilling *type* and *sel* (see Fig. 1) of the assertion query and saves their count in the variable $\$c_j$. Gj.2 counts all entries that fulfill the type of the assertion query and saves

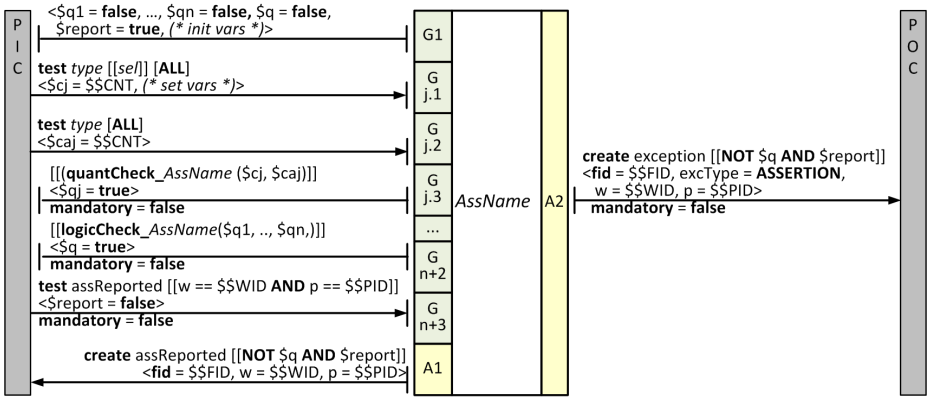

Fig. 2. Corresponding wiring for a intra-peer assertion.

the value in the variable $\$ca_j$ which is needed for the quantifier check. Gj.3 checks whether the right quantity of entries satisfies the *sel* by comparing $\$c_j$, $\$ca_j$ and the quantor in the *ContainerAss* as follows: (*quantor* == `ALL` `AND` $\$c_j$ = $\$ca_j$) `OR` (*quantor* == `NONE` `AND` $\$c_j$ == 0) `OR` ($\$c_j$ == *quantor*). Gn+2 sets the variable $q to the result of the overall assertion based on the values of $\$q_1$, ..., $\$q_n$. Gn+3 checks whether an assertion violation has already been reported for this wiring and flow, which is the case if an assReported entry exists in the `PIC`. If the assertion is violated and has not been reported yet, A1 creates an assReported entry for the wiring and flow in the `PIC`. If the assertion is violated and has not been reported yet, A2 creates an exception with type `ASSERTION` for the wiring and flow in the `POC`.

Inter-peer assertions are translated analogously: Sub-assertions (intra-peer assertions) are treated individually and the overall inter-peer assertions evaluated by coordinators based on the evaluation value of the sub-assertions.

The sub-assertions are translated into intra-peer assertion wirings. But instead of raising exceptions, they send violation reports to coordinators when the evaluation value might cause the overall assertion to fail. The differences are reflected in A2, which creates an assReport entry and sends it to all coordinators, and in the selector of Gn+2, which is inverted for negated assertions with quantor **ALL** and non-negated assertions with quantor **NONE** in the context.

The overall assertion is evaluated in dedicated wirings in coordinators that collect the corresponding sub-assertion values and combine them according to the propositional formula of the overall assertion. The resulting wiring is analogous to an intra-peer assertion wiring where the sub-assertions are checks on the number of assertion reports sent by other peers. Variable initialization, evaluation of the propositional skeleton, keeping track of violations and raising of exceptions is analogous to intra-peer wirings. For each sub-assertion, the number of reports sent by peers is counted (G2.1) and the numbers are compared ac-

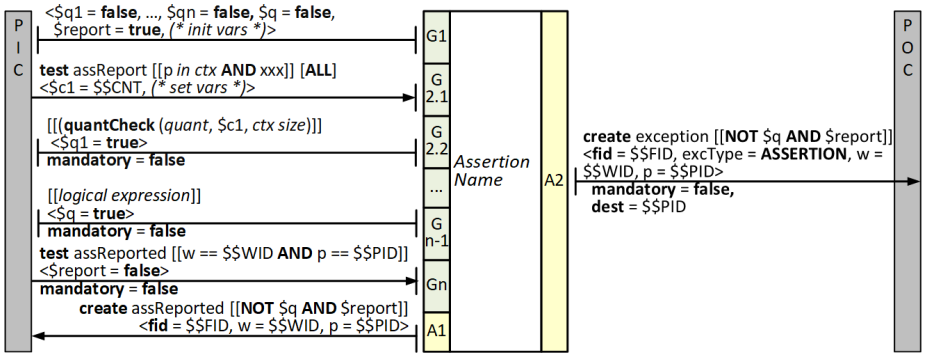

Fig. 3. Corresponding wiring for the evaluation of inter-peer assertions.

cording to the quantor and the variable set to true if the sub-assertion is fulfilled (G2.2). See Fig. 3 for the corresponding wiring.

Inter-peer assertions introduce c overall evaluation wirings ($c \ldots$ # of coordinators) with $2m+5$ links and $2m+2$ variables each ($m \ldots$ # of sub-assertions) and $\sum_1^m k_i$ intra-peer assertion wirings ($k_i \ldots$ # of peers in the context for each sub-assertion) with $3n_i+5$ links and $3n_i+3$ variables ($n_i \ldots$ # of container sub-assertions in the respective sub-assertions). For each violation at most $\sum_1^m k_i$ messages are sent and $2c + \sum_1^m 2k_i$ entries are created.

The use of `HIC` adds additional links to each wiring that modifies or deletes entries for which a `HIC` entry is assumed: The wiring gets additional links to retrieve, update and write back sequence numbers from the `HIC` for an overall sequence number (`os`) and a type sequence number (`ts`). (1) if the entry is taken from the container, modified and written back or written to another container, a copy of the entry as seen in the `WIC` is written to the history with `ts` and `os` set, and (2) if the entry is deleted, the delete is replaced by a `move` and it is subsequently moved into the `HIC` with `ts` and `os` set. All history entries of a wiring instance are assigned the same sequence numbers. $l+4$ additional links and 2 new variables are added to each wiring that modifies or deletes entries that must be kept in the history ($l \ldots$ # of links that modify or delete entries relevant for the history).

Variables can be shared between sub-assertions. In that case, the wiring for the assertion that sets the variable, must get additional logic to report the values. The wirings for assertions that read the variables, must treat the variable reports and set their local wiring variables accordingly. For positive sub-assertions in the CNF, reports are sent if the corresponding assertion did not fail, for negated ones, they are sent if the assertion failed. Variables shared between contexts introduce an overhead of $\sum_1^m (2o_i * k_i + k_{2j})$ additional links and at most $\sum_1^m k_i \sum_1^{o_i} k_{2j}$ additional messages (m $\ldots$ # sub-assertions, $k_i \ldots$ size of context setting a variable, $k_{2j} \ldots$ size of context reading a variable).

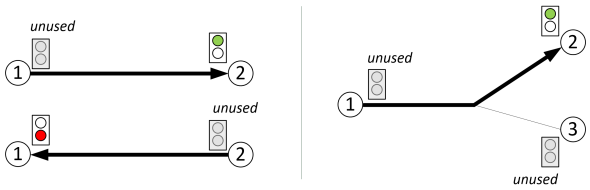

Fig. 4. Two tracks (left) and one point (right) with set direction and signals.

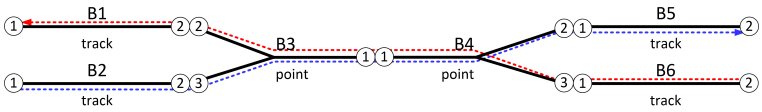

Fig. 5. Sample configuration. Two routes: red and blue.

5 Proof-of-Concept: Railway Use Case

The selected use case refers to the reservation of rails (cf. [1], [23]). A *block* is either a track or a point (see Fig. 4). It has a unique identifier (id). A *track* has 2 connectors. A *point* has 3 connectors. A *connector* references a connector of another Block. A *signal* refers to a main signal; its value is red or green. It is valid at the exit of the Block and associated with a connector (dependent on the direction). A *route* is a sequence of connected Blocks. A *reserved route* is a route where all Blocks are exclusively reserved for this route.

Three scenarios are selected: (1) Reserve Route: Mark all Blocks $\text{Block}_1, \ldots, \text{Block}_N$ for a route in the specified order. Check that the neighbor Blocks are physically connected via connectors, and set their direction appropriately. (2) Set Signals: Set signals on Blocks of a reserved route.

An example for a physical configuration of tracks is shown in Fig. 5. Each Block knows to which neighboring Blocks it is connected. The challenge is that there may exist many train Operators who try to reserve routes concurrently (see red and blue routes in Fig. 5). It must not happen that a safety constraint is violated (see informal specification of required assertions in Tab. 2). These assertions for the example use case were provided by a railway expert.

We assume that the Operator defines a `ttl` within which the route must be reserverd. If not, the reservation fails, and the Operator issues the freeing of the already reserved Blocks of the route. The reservation of a route, the setting of signals of Blocks of the reserved route, and the freeing of the route belong to the same flow.

5.1 The Model

The Peer Model based representation is presented in the following using the graphical notation.

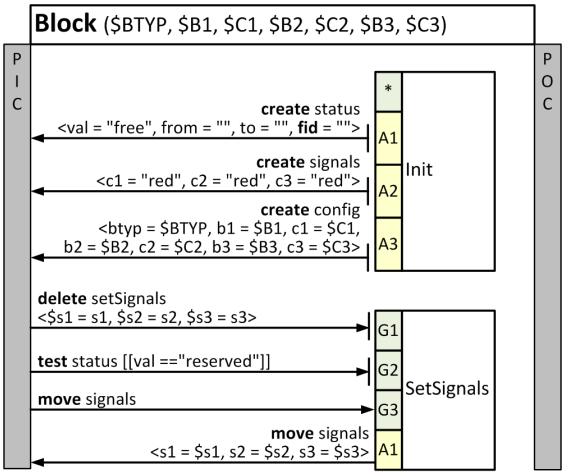

Fig. 6. Block Peer: Initialization, and setting of signals.

Entries: The entries used in the example are categorized into entries that represent the state of the distributed system, and entries that represent events which are sent between the peers as requests and answers. "bid" stands for Block id, and "cid" for connector id.

State Entries:

- config:
  - btyp: track or point
  - b1: bid of neighbor 1
  - c1: cid of neighbor 1
  - b2: bid of neighbor 2
  - c2: cid of neighbor 2
  - b3: bid of neighbor 3 (if point)
  - c3: cid of neighbor 3 (if point)
- status:
  - val: free or reserved
  - from: connector name
  - to: connector name
- signals: red or green
  - s1: signal at cid 1
  - s2: signal at cid 2
  - s3: signal at cid 3 (if point)

Request/Answer Entries:

- setDirection:
  - from: connector id
  - to: connector id
- setSignals:
  - s1: red or green
  - s2: red or green
  - s3: red or green
- markRoute:
  - route: sequence of Block ids
  - operator: peer id of operator
  - done: flag used by route marking
  - action: reserve or free
  - sender: peer id of sender (Operator or Block)
- ackRoute
- nackRoute

Peers: We define two kinds of peer roles: *Block* and *Operator*. Each track and point is modeled as a Block with unique block id. It accepts commands to change status, set signals and direction. The task of an Operator is to issue the setting of Block directions, the reservation and freeing of routes, and the setting of signals on reserved routes. All Operators act concurrently.

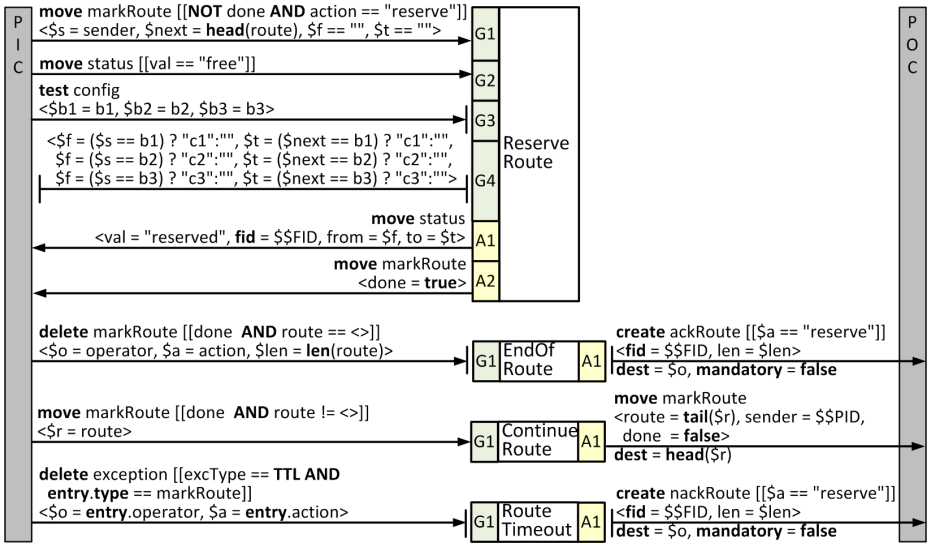

Fig. 7. Block Peer: Marking a route as reserved.

Wirings: Figures 6–7 show the wirings of Block peers.

Init: The initialization wiring (Fig. 6) is enabled exactly once when the Block peer is created. It creates entries to hold the local status (A1), signals (A2), and config (A3) and writes them to the peer's PIC. Initially the status is free and does not belong to a flow, all signals are set to red, and the configuration of the Block is set based on the given peer's configuration parameters.

SetSignals: This wiring (Fig. 6) sets the signals, provided that the Block is reserved for the current flow. It is triggered by receipt of a setSignals entry which it deletes from the `PIC` (G1). In (G2) it tests that its status says that it is reserved for this flow. In (G3) it takes the signals entry from the `PIC` and in (A1) writes the updated signals entry back to the `PIC`.

The marking of a route consists of the following wirings. Depending on the action of markRoute, the Blocks are either reserved of freed within a flow.

ReserveRoute (see Fig. 7) reacts on the receipt of a markRoute event in its `PIC`. We assume that a markRoute entry sent by an operator contains a valid route and has its `ttl` (time-to-live) property set. It checks that it has not been treated yet (i.e., the done flag is `false`), that the action is to reserve the route, remembers the sender in the variable $s, the next Block of the route in the variable $snext (G1), and initializes two variables $f and $t to empty. It takes its status entry (G2) from `PIC`, tests that in its configuration the val property equals free (G2). G3 tests the existence of one config entry in the `PIC` and remembers its properties in local variables termed $b1, $b2 and $b3. G4 initializes the local variables $f and $t to contain the connector id of the sender (if it was not the operator, i.e., sender is empty), and to the connector id of the next block. It updates the value of the status entry to be reserved, sets its flow id

to the current flow, and the from and to properties to the respective connector ids (A1). Finally, it sets the done flag on the markRoute entry to `true` (A2) to denote that the status has been updated.

FreeRoute frees a reserved route. Its model is similar to ReserveRoute, with the only difference that it needs not check that the block was free before and updates the signals entry. It is therefore not shown.

EndOfRoute (see Fig. 7) fires after the Block's status has been set by ReserveRoute (i.e., done is `true`) and if the route has been completely treated, i.e., the route is empty (G1). It sends the reply entry ackRoute to the requesting Operator, provided that the action was to reserve the route. Otherwise no reply is sent (A1). Therefore the action link is not mandatory.

ContinueRoute (see Fig. 7) fires after the Block's status has been set by ReserveRoute and recursively processes the rest of a not yet empty route (G1). It sends the directive to mark the rest of the route to the next Block found in the route (A1), i.e., the head of the current route.

RouteTimeout (see Fig. 7) treats the `ttl` exception of the markRoute entry (G1), and sends as reply nackRoute to the Operator (A1), if the action was to reserve the route, using the flow id of the markRoute entry, which is inherited by the exception entry.

CleanUp (not shown) deletes outdated markRoute entries that want to free the Block.

5.2 Assertions

Table 2. Railway use case assertions (informal description). N...assertion number, D...distributed assertion, P...refers to past

N	Assertion	D	P
1:	There must exist exactly one status entry at each Block.	–	–
2:	There must exist exactly one config entry at each Block.	–	–
3:	There must exist exactly one signals entry at each Block.	–	–
4:	markRoute is received from "from" neighbor or from Operator.	–	–
5:	The next Block of the route is the "to" Block neighbor.	–	–
6:	A Block receiving setSignal is reserved within the same flow id.	–	–
7:	If a Block is free then all its signals are red.	–	–
8:	If a Block status is free then the flow id of its status must be null.	–	–
9:	Reserved Blocks were free before.	–	x
10:	Block freeing requires Operator to have received ack/nackRoute.	–	x
11:	nackRoute is only sent after TTL of markRoute has expired.	–	x
12:	If Operator got ackRoute, all Blocks of the route are reserved.	x	–
13:	Signals are set after the Operator has received ackRoute for the flow.	x	x

Tab. 2 describes the major assertions for the use case in an informal way. All of them were successfully modeled with the presented assertion mechanism,

Table 3. Railway use case assertions (formal specification).

1: [ALL] ⟨Block⟩ { PIC { status } }
2: [ALL] ⟨Block⟩ { PIC { config } }
3: [ALL] ⟨Block⟩ { PIC { signals } }
4: [ALL] ⟨Block⟩ { PIC { markRoute <$s = sender> } → (PIC { status <$f = from> } AND PIC { config ⟦label($f) == $s⟧ }) }
5: [ALL] ⟨Block⟩ { PIC { markRoute <$next = head(route)> } → (PIC { status <$t = to> } AND PIC { config ⟦label($t) == $next⟧ }) }
6: [ALL] ⟨Block⟩ { PIC { setSignal } → PIC { status ⟦val == "reserved"⟧ } }
7: [ALL] ⟨Block⟩ { PIC { status ⟦val == "free"⟧ } → PIC { signals ⟦s1 == "red" AND s2 == "red" AND s3 == "red"⟧ } }
8: [ALL] ⟨Block⟩ { PIC { status ⟦val != "free" OR fid == ""⟧ } }
9: [ALL] ⟨Block⟩ { HIC { status ⟦val == "reserved"⟧ <$ts = ts> } → HIC { status ⟦ts == $ts-1 AND val == "free"⟧ } }
10: [ALL] ⟨Block⟩ { PIC { markRoute ⟦action == "free"⟧ } } → [1] ⟨Operator⟩ { HIC { ackRoute } OR HIC { nackRoute } }
11: [ALL] ⟨Operator⟩ { PIC { nackRoute } → HIC { exception ⟦type == TTL AND entryType == markRoute⟧ } }
12: [ALL] $O IN ⟨Operator⟩ { PIC { ackRoute } } → ($O { PIC { markRoute <$r = route> } } AND [ALL] ⟨Block⟩ { ⟦$$PID IN $r⟧ → PIC { status ⟦val == "reserved"⟧ } }) [**coordinators** = { [ALL] ⟨Operator⟩ }]
13: [ALL] ⟨Block⟩ { PIC { setSignal } } → [1] ⟨Operator⟩ { HIC { ackRoute } } [**coordinators** = { [ALL] ⟨Block⟩ }]

translated with MTA, and tested with the runnable specification of the Peer Model. The resulting Peer Model assertions are shown in Tab. 3. Of these, assertions 12 and 13 show the specification of the assertion property `excType`. For assertion 12 we specify that all Operators shall act as coordinators, and in assertion 13 all Blocks serve as coordinators.

MTA Application.

As an example for applying the developed model translation algorithms to the defined assertions, the resulting implementation of assertion 7 is shown in Fig. 8, which is an intra-peer one with two sub-assertions.

Overhead Evaluation.

We calculate the overhead by assuming one distributed railway application that has all shown example assertions. The introduced overhead is categorized by number of additional wirings and links. In average per assertion 1,08 more wirings and 11,15 links per Block, and 0,46 more wirings and 6,08 links per operator were needed as shown in Tab. 4.

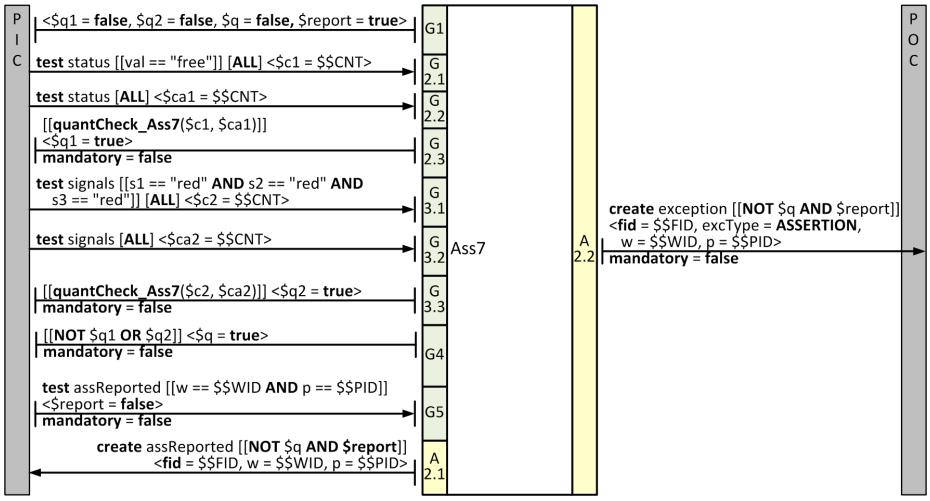

Fig. 8. MTA applied to assertion 7 (see Tab. 2 and Tab. 3).

6 Conclusion

In this paper we have presented an asynchronous runtime assertion mechanism for the Peer Model, which is an event-driven coordination model. Assertions are formulated in a declarative notation that relies on the query mechanism of the Peer Model. While their concept and syntax are new, they are closely related

Table 4. Calculation of the overhead.

assertion number	# wirings per Block	# wirings per Operator	# links per Block	# links per Operator
1	1	0	8	0
2	1	0	8	0
3	1	0	8	0
4	1	0	14	0
5	1	0	14	0
6	1	0	11	0
7	1	0	11	0
8	1	0	8	0
9	1	0	24	0
10	1	1	8	24
11	1	1	0	17
12	1	3	12	30
13	2	1	19	8
Total	14	6	145	79
Average	1,08	0,46	11,15	6,08
Median	1,00	0,00	11,00	0,00

those of the link and do not pose an entirely new mechanism as the use of a separate tool or language would.

The proposed mechanism allows for lazy and strict assertion checking. In the former case, it might happen that certain errors are not detected due to race conditions. In the latter case, all errors can be detected during runtime, however, with the trade-off that a history of certain events must be kept.

Without history and shared variables, overhead is linear in the number of assertions to be checked and an the sizes of context they must be applied to. History imposes an overhead linear in the number of modifications to each entry type to be kept in history and shared variables increase the overhead quadratic in the size of the contexts and linear in the number of variables and sub-assertions.

In a first step, the application model is transformed so that the event history can be maintained, if required by assertions. In a second step, the assertions are implemented by mapping them to the concepts of the Peer Model, so that they become an integral part of the application model and can be understood as a coordination solution themselves.

In contrast to model checking, the proposed assertion mechanism is useful at both model definition and runtime. It makes no assumptions about the environment and can serve models of arbitrary size and distribution without encountering the state explosion problem.

The lazy mechanism is completely orthogonal to the application. The communication overhead depends on the distribution and number of the assertions. The strict mechanism causes additional computation in the wirings, however, only those events need to be captured in the history container that are referred to by (sub)assertions, that are newly obtained in a peer container, or updated or deleted by guards.

While the MTA is specific to the Peer Model, its concepts can be applied to assertion checking in coordination models or distributed systems in general. I.e., states are evaluated locally and evaluations compared instead of collecting global states, only evaluations and variable assignments that influence the overall evaluation of the assertion are shared and the mechanism is highly configurable.

As a proof-of-concept we have specified the major assertions for a use case from the railway domain and demonstrated that all could be captured.

7 Future Work

In future work, the MTA will be implemented and integrated into Peer Model implementations. Its correctness will be proven and its performance evaluated systematically. We will work on optimizations for overhead minimization produced by the model translation algorithm, investigate assertions that also refer to future events and dynamic changes of the meta model to support dynamic injection of assertions.

Acknowledgements. The authors would like to thank Anita Messinger for the discussions on train systems, and Stefan Craß for commenting this paper.

References

1. Abrial, J.R. In: Train Systems. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 1–36
2. Agha, G.A.: ACTORS: A Model Of Concurrent Computation In Distributed Systems. MIT Press (1990)
3. Arbab, F.: Reo: A Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science **14**(3) (2004) 329–366 Cambridge University Press.
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology **20**(2) (2011)
5. Chao, H., Li, H., Song, X., Wang, T., Li, X.: On evaluating and constraining assertions using conflicts in absent scenarios. In: 26th IEEE Asian Test Symposium, ATS 2017, Taipei City, Taiwan, November 27-30, 2017. (2017) 195–200
6. Din, C.C., Owe, O., Bubel, R.: Runtime assertion checking and theorem proving for concurrent and distributed systems. In: MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014. (2014) 480–487
7. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2017, ACM (2017) 125–135
8. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the bip framework with formally-proved sound and complete instrumentation. Software & Systems Modeling **14**(1) (Feb 2015) 173–199
9. Faruk Polat, R.A.: A multi-agent tuple-space based problem solving framework. The Journal of Systems and Software **47**(5) (1999) 11–17
10. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems (TOPLAS) **7**(1) (1985) 80–112
11. Gelernter, D., Carriero, N.: Coordination Languages and Their Significance. Commununications of the ACM (CACM) **35**(2) (1992) 96–107
12. Kamburjan, E., Din, C.C., Chen, T.C.: Session-based compositional analysis for actor-based languages using futures. In Ogata, K., Lawford, M., Liu, S., eds.: Formal Methods and Software Engineering, Springer (2016) 296–312
13. Kasuya, A., Tesfaye, T.: Verification methodologies in a tlm-to-rtl design flow. In: DAC 2007, ACM (2007) 199–204
14. Khosravi, R., Sirjani, M., Asoudeh, N., Sahebi, S., Iravanchi, H.: Modeling and analysis of reo connectors using alloy. In: Coordination Models and Languages, Springer (2008)
15. Kühn, E.: Reusable Coordination Components: Reliable Development of Cooperative Information Systems. International Journal of Cooperative Information Systems **25**(4) (2016)
16. Kühn, E.: Flexible transactional coordination in the peer model. In: International Conference on Fundamentals of Software Engineering (FSEN). Volume 10522 of LNCS., Springer (2017) 116–131
17. Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C.: Introducing the Concept of Customizable Structured Spaces for Agent Coordination in the Production Automation Domain. In: 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS (2009) 625–632

18. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of ltl specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium. (May 2015) 494–503
19. Newsham, Z., d. Oliveira, A.B., Petkovich, J.C., Rehman, A.S.U., Tchamgoue, G.M., Fischmeister, S.: Intersert: Assertions on distributed process interaction sessions. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). (2017)
20. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Technische Hochschule Darmstadt (1962)
21. Pham, L.H., Thi, L.L.T., Sun, J.: Assertion generation through active learning. In: 39th IEEE International Conference on Software Engineering Companion, IEEE/ACM (2017) 155–157
22. Radschek, S.T.: A usable formal methods tool chain for safety critical concurrent systems design. Master's thesis, TU Wien (2018) submitted.
23. Reichl, K., Fischer, T., Tummeltshammer, P.: Using formal methods for verification and validation in railway. In: Tests and Proofs, Springer International Publishing (2016) 3–13
24. Scheller, T., Kühn, E.: Automated measurement of API usability: The API Concepts Framework. Information & Software Technology **61** (2015) 145–162 Elsevier.
25. Tjang, A., Oliveira, F., Martin, R.P., Nguyen, T.D.: A: An assertion language for distributed systems. In: Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems. PLOS'06, ACM (2006)
26. Wang, C., He, F., Song, X., Jiang, Y., Gu, M., Sun, J.: Assertion recommendation for formal program verification. In: 41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1. (2017) 154–159