A Practical Tool-Chain for the Development of Coordination Scenarios
Graphical Modeler, DSL, Code Generators and Automaton-Based Simulator

Eva Maria Kuehn(✉)

Faculty of Informatics, Compilers and Languages Group, TU Wien, Vienna, Austria
eva.kuehn@tuwien.ac.at
http://www.complang.tuwien.ac.at/eva/

Abstract. Coordination scenarios have high demands on concurrency and interaction. However, these are typical sources for flaws in both design and implementation. A modeling approach enables reasoning about distributed algorithms and finding deficiencies right from the beginning. The Peer Model has been introduced as a modeling tool for distribution, concurrency and blackboard-based collaboration and coordination, relying on known foundations like tuple spaces, Petri Nets and Actor Model. A runtime system exists that serves Java developers for prototyping, but still a feasible tool-chain was missing, like for most academic systems.

This paper presents a practical new tool-chain for the Peer Model consisting of a graphical modelling tool, building on a drawing program that exports XML. A translator parses the XML and translates it into a newly developed domain specific language that is the basis for code generation. One target is a new, formal automaton-based runtime written in the Go programming language. It allows systematic simulation runs of user models. The demo shows a peer competition scenario, where several players play a game, a global state holds the players' scores, and in addition each peer maintains a decentralized state. Before taking a move in the game, a peer asserts its current local state to be the same like the global one. If this is the case, it carries out its action and distributes the information about it to all other players for further verification. The scenario captures core coordination mechanisms found in blockchain systems.

Keywords: Peer Model · tool-chain · coordination modelling · coordination simulation

1 Introduction to the Peer Model Tool-Chain

The *Peer Model* [17,18] is a modelling notation that relies on know concepts from Petri Nets [5,14,23,24], tuple spaces [11,12] and the Actor model [7]. Distributed *Peers* collaborate with each other by asynchronous message sending (cf. Actor model). The behaviour of a Peer is indeterministic and modelled by

Published by Springer Nature Switzerland AG 2021
F. Damiani and O. Dardha (Eds.): COORDINATION 2021, LNCS 12717, pp. 239–254, 2021.
https://doi.org/10.1007/978-3-030-78142-2_15

means of *wirings* which relate to transitions in Petri Nets. All wirings of a Peer run concurrently and synchronize themselves via transactional operations on two tuple spaces: one is termed *PIC* (Peer input container) and the other one *POC* (Peer output container).

The Peer Model supports domain specific abstractions as higher-level modeling constructs which lead to easy to understand models, like automatic flow correlation [15,18], abstraction of asynchronous message sending [15,18], timing constraints for all resources [15], and transactions [16]. The original specification of the Peer Model can be found in [17].

Targeted application areas are scenarios with high concurrency and decentralized coordination. Some selected application examples are:

Smart Contracts which realize distributed application logic on a blockchain. They were made popular by Ethereum [1], which introduced a new language termed "Solidity". The result must be deterministic, meaning that eventually all blockchain nodes must achieve a consensus on the result. Also, the execution must terminate. A smart contract holds an internal state and is triggered by an external event. It forms an automatized, formalized contract among multiple parties based on complex conditions.

Complex coordination patterns [15] like distributed, heterogeneous transactions, distributed consensus algorithms, distributed voting, collaboration, load balancing, load clustering and peer clustering.

Embedded systems especially in the domain of traffic management systems (train [13], truck, and air), and other autonomous cyber physical systems.

So far, a Java-based runtime system has been developed for the Peer Model in the context of a diploma thesis [8] and a doctoral thesis [9] which also added a security model to the Peer Model. This system is open source and can be used by system developers who are experienced in Java programming. However there are the following drawbacks: There is no tool support yet for graphical modeling, and a formal analysis tool is still missing. The motivation of this work was therefore to provide a practical tool-chain for the modeling and analysis of complex coordination scenarios with the Peer Model. It consists of the components highlighted in gray color in Fig. 1.

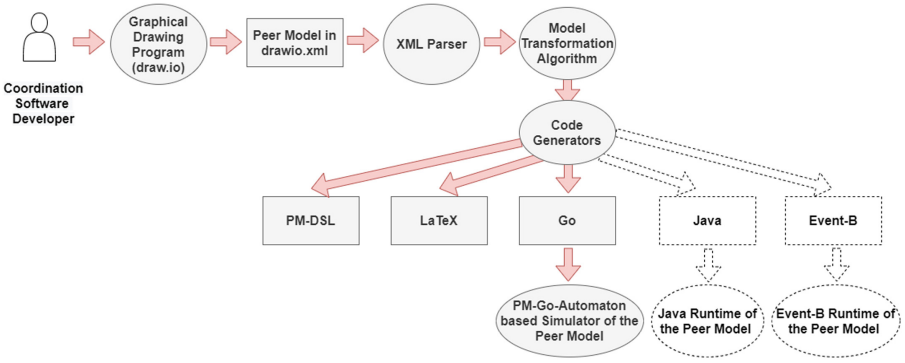

Fig. 1. Peer Model Tool-Chain.

As front end serves a graphical drawing program[1], for which dedicated shapes were designed for all Peer Model artefacts. An extended language notation is introduced that supports types for Peer Model artefacts to enable their reuse as patterns [15,19]. Moreover it provides context properties, a notation for range, for all and exists expressions, and the specification of configurations. Developers employ the new shapes to model coordination scenarios.

The resulting XML model is parsed and transformed into an intermediate format on which a model transformation algorithm (MTA) is applied. The MTA interprets the newly introduced language concepts and statically resolves them for the code generators. These form the next step in the tool-chain. The here considered code generator translates the model into Go code that is understood by the Peer Model Concurrent State Automate System written in the Go programming language, termed the *PM-Go-Automaton* for short. This is a new formal analysis tool, based on state machines (see Sect. 3). Other code generators comprise the compilation into the XML-based PM-DSL (Peer Model Domain Specific Language), and also into LaTeX drawings that can be easily included into publications. Planned for the future are code generators for the existing Java runtime system, as well as for the Event-B-based [6] verification system for the Peer Model, which is currently under development [20].

According to our proposal of a hybrid verification approach [10] also further target systems will be considered in future work. Basically the hybrid verification approach suggests to integrate several verification tools in the Peer Model tool-chain. Each Peer Model is a complete model from which also production code for the Java runtime can be generated. It does not suppress important information, which is sometimes the case with abstract models. Therefore, one can formulate assertions and invariants based on the (respectively extended) Peer Model query language, as proposed in [21], and map them to the most suitable verification tool with a focus to verify certain properties of the system. For example, timed automate [4] will be used to reason about timing properties, for which Event-B is less suited, and Colored Petri Net tools [3] shall be exploited to verify deadlock and reachability properties.

2 Peer Model in a Nutshell

This section summarizes the already published concepts of the Peer Model (see Sect. 2.1) and describes the newly introduced features (see Sect. 2.2).

2.1 Core Concepts

The *Peer Model* serves for the modelling of coordination logic as concurrent, timed, distributed flows. Its design was influenced by the tuple space paradigm to achieve a high decoupling of components, by Petri Nets that inspired the modelling of concurrency by means of *wirings*, and by the Actor Model that

[1] The open source tool Draw.io [2] was selected, that can export XML.

motivated asynchronously communicating Peers that implement a behaviour. The main artefacts are: Peer, entry, container, wiring, link and service.

Peer. A peer is a named resource with behaviour. It possesses two tuple spaces, termed *containers*, that store tuples, termed *entries*.

Entry. An entry is a typed message with properties. A property has a name and a value. Entries consist of user and system properties (e.g., `ttl` is the time-to-live, `fid` is a flow identifier, and `type` denotes the entry type).

Container. The Peer's containers are termed peer-in-container (*PIC*) and peer-out-container (*POC*). Containers represent the system state. They support transactional operations to `read`, `take`, `create` and `delete` entries.

Wiring. *Wirings* model the Peer's behaviour. A wiring specifies a transaction on containers. It consists of *links* which are either *guards* or *actions*, and an internal container (*WIC*) to temporarily hold entries retrieved by guard links.

Link. A *link* issues space operations on containers. The source of a guard is either the PIC or the POC of the peer, and its target is the WIC; for actions it is the other way round. Link kinds are named after their operation on the source: READ and TAKE links select entries from source and write them to target, whereby the TAKE links also remove the entries from the target. DELETE links remove entries from the target. CREATE links create new entries and write them to the target. NOOP links do not access entries. The specification of which entries are to be selected or created consists of an entry type, an entry count (an exact number, an interval, or the keyword `ALL`), and a query on entry properties and variables. Assignments between entry properties and user variables serve to either set entry properties or to pass entry properties between links. The scope of user variables, which start with `$` and are written in lower case, is the current wiring instance. System variables start with `$$` and are written in upper case (e.g. `$PID` is the identifier of the local Peer, and `$FID` holds the current flow identifier). Finally, a link possesses properties; e.g., `ttl` defines how long a link shall wait until it can be fulfilled (default=`infinite`), `mandatory` defines if the link is obligatory (default=`on`), `flow` says if the flow correlation shall be applied (default=`on`), and `dest` on an action link asynchronously delivers all entries via an i/o Peer to the Peer denoted by the property value. All links are numbered.

Service. In between guard and action execution, wirings may call a service. It encapsulates application logic which is considered as "black box" from point of view of the Peer Model.

The operational semantics is that all wirings of a peer run concurrently. A wiring executes sequentially in the specified order: guards, the optional service, and actions. If a mandatory link cannot be fulfilled then the current wiring execution fails, a rollback takes place and depending on the `repeat_count` property of the wiring, a next instance of this wiring is started. Further properties of a wiring are `ttl` (maximum allowed execution time for this wiring instance) and `tts` (time to wait until the next wiring instance may start).

2.2 Newly Introduced Concepts

For more modelling convenience, in this paper, the following new modelling concepts are introduced. They are implemented by the Translator component of the tool-chain (see Sect. 3):

Types. Abstract types for entries, wirings, Peers and the Peer Model Meta Model (PMMM).

User-defined wiring, Peer and PMMM properties. So far, only entries had user-defined properties. This concept is extended to wirings, Peers and the PMMM. In type declarations, default property values can be specified.

Configuration. Instantiation of a PMMM, defining the required Peers, based on Peer types, and properties for all artefacts.

INIT entry. As a convention, a system entry of type `INIT` is written into the PIC of each instantiated Peer. This serves to signal a Peer that it has been started and is useful if it has to carry out initializations.

Reference data type. An array data type, using `#` as access operator, provides indirect referencing of user-defined properties.

RANGE, FORALL, and EXISTS. With `RANGE` it is possible to iterate within a range over the array type, using an `INDEX`. `FORALL` and `EXISTS` are to be used in queries; they also use an `INDEX` that iterates over an interval, and translate to AND respectively to OR expressions.

3 Implementation of the Toolchain

Design Principles: The leading principles were: A graphical user interface shall be supported to make the Peer Model more usable. An existing tool shall be employed for implementing the graphical modeller. The tool-chain shall be practical and realizable with "one woman's power" in reasonable time, like the already developed PM-Go-Automaton. The PM-Go-Automaton should not have to be changed, i.e. all new concepts (see Sect. 2.2) must be translated to the core concepts (see Sect. 2.1) that the PM-Go-Automaton can execute. A domain specific language shall be provided, termed the PM-DSL, represented as XML.

Graphical Modelling Tool: The requirements on the drawing program were that it is easy to use, provides an open source license, allows the definition of shapes, and can export the designed user models as readable XML code where dedicated shapes can be identified. Eventually, draw.io was selected, where shapes can be tagged to make their recognition in the XML code possible. For each core and newly introduced Peer Model artefact a shape was designed. However, for practical reasons, the originally proposed graphical notation [17] of the Peer Model – where links are arcs connecting PIC and WIC, respectively WIC and POC, and where link features are specified as labels on these arcs – had to adapted, because this could not be realized by means of draw.io shapes. Instead, a link is now represented by a box that specifies the link's features as a form to be filled in. This form also denotes the PIC or the POC, and by means of an unlabelled arc it is connected to the WIC of the wiring to which it belongs.

All elements of the link are grouped and tagged, so they can be identified as belonging together in the XML.

One draw.io file represents one PMMM type including several configurations of it. It consists of several drawing sheets (tabs), following certain naming conventions. E.g., the name of a drawing containing a Peer Type must start with "PeerType:" followed by the type name of the Peer; analogously the prefixes "EntryTypes:", "WiringTypes:", "PmmmType:", and "Config:" were introduced. The layout with many drawing sheets contributes to structure the model.

Translator: The translator is written in Java and consists of three parts, where parsers respectively code generators can be exchanged by means of the builder design pattern.

XML Parser. There are two parsers supported so far: One reads the XML exported by draw.io, and the other one the XML-based PM-DSL and translates it to an internal data representation.

Model Transformation Algorithm (MTA). The MTA operates on the internal data representation and consist of several passes: recognition of tokens, type evaluation, and transformation of the newly introduced modelling concepts.

Code Generator. Finally, code for the target runtime system is generated. Three code generators have been developed so far: One compiles the model into Go code for the PM-Go-Automaton, one generates PM-DSL (which in turn can be parsed by the respective XML Parser), and one produces LaTeX code – representing the model in the original Peer Model graphical notation – that is useful for inclusion in publications.

The generated PM-DSL code of the Peer type Player and its GameOver wiring (see Fig. 6) is shown in Fig. 2 and the Go code (for one Peer instance, and with comments stripped) in Fig. 3; at "..." are the other wirings of this Peer (type) generated. Note that without the tool-chain, developers would have to write this Go code manually, but now they can either use the graphical interface or the PM-DSL to specify their models.

```
<PeerType name="Player">
 <Wiring name="gameOver" service="Watch">
  <Guard number="1" container="PIC" op="TAKE" entryType="matchball" count="1">
   <Query>EXISTS INDEX.1 IN 1..PMMM.nPlayers -> scores#INDEX.1 >= PMMM.max</Query>
  </Guard>
  <Action number="1" container="POC" op="TAKE" entryType="matchball" count="1">
   <VarPropsSetGet>gameOverFlag=true;</VarPropsSetGet>
   <PropsDefinition>dest=PMMM.gameController</PropsDefinition>
  </Action>
 </Wiring>
 ...
</PeerType>
```

Fig. 2. PM-DSL code snippet.

```
p = NewPeer("player")
w = NewWiring("gameOver")
w.AddServiceWrapper("SID_Watch", NewServiceWrapper(Watch,  "Watch"))
w.AddGuard("", PIC, TAKE,
  Query{Typ: SEtype("matchball"), Count: IVal(1),
  Sel: XValP(XVal(XVal(IArrayLabel(DynArrayRef("scores", IVal(1))), GREATER_EQUAL, IVal(10)),
    OR, XVal(IArrayLabel(DynArrayRef("scores", IVal(2))), GREATER_EQUAL, IVal(10))),
    OR, XVal(IArrayLabel(DynArrayRef("scores", IVal(3))), GREATER_EQUAL, IVal(10)))},
  LProps{},
  EProps{},
  Vars{})
w.AddSin(TAKE, Query{Typ: SVal("*"), Count: IVal(ALL)}, "SID_Watch", LProps{}, EProps{}, Vars{})
w.AddScall("SID_Watch", LProps{}, EProps{}, Vars{})
w.AddSout(Query{Typ: SVal("*"), Count: IVal(ALL)}, "SID_Watch", LProps{}, EProps{}, Vars{})
w.AddAction("", POC, TAKE,
  Query{Typ: SEtype("matchball"), Count: IVal(1)},
  LProps{"dest": SUrl("arbiter"), "commit": BVal(true)},
  EProps{"gameOverFlag": BVal(true)},
  Vars{})
...
p.AddWiring(w)
...
ps.AddPeer(p)
```

Fig. 3. Go code snippet.

PM-Go-Automaton: The PM-Go-Automaton is written in the Go programming language ("Golang[2]" for short). Golang supports convenient mechanisms to program concurrency, like go routines, mutexes, channels. This was the reason why it was chosen to implement the highly concurrent state machines of the PM-Go-Automaton.

The entire Peer Model specification [17] is mapped to formal state automata, so there is a separation into a framework that implements the concurrent state machines, the Peer Model specification and the application model. This way changes and extensions in either part can be carried out independently. A controller component coordinates the concurrency of the re-entrant machines, which support leave and enter mechanisms. At least if a machine waits for an event, e.g. an entry to arrive in a PIC, it gives up its control. The controller selects the next machine according to a configurable execution mode. E.g. it may take the machine that waits longest for execution; check specification-defined waiting conditions; repeatedly perform indeterministic simulation runs; or try out all possible inter-leavings defined by the possible leave-points up to a configurable bound. These different modi are helpful in finding bugs in the user model at an early design stage. At the moment it is possible to model run-time assertions [21] manually; in future work it is planned to support a declarative notation for assertions and invariants, based on the existing query syntax of the Peer Model.

Implementation Notes: Some implementation facts: Currently, the Translator consists of 23K LOCs written in Java, and the PM-Go-Automaton (framework and Peer Model specification) has 24,5K LOCs written in Golang. The generated Go code for the Peer Competition example has 2,2K LOCs.

[2] https://golang.org/.

4 Use Case Example

As demonstrator, a game has been invented, where concurrent peers catch and throw a matchball. Whoever receives the ball next is indeterministic. Every ball catching/throwing gives a plus point and whoever reaches a defined high score first is the winner. Every move is broadcasted to all peers in the network who build up their own view of the network-wide "truth". The score of each peer is recorded on the matchball which represents the global state. In addition, each peer updates its own local statistics, based on the broadcasted game moves. This local view serves to verify, whether the global state of the matchball is correct.

This example reflects basic mechanisms found in blockchain applications: Every peer verifies the truth on its own and does the next game move only if its verification succeeds. For this it uses a runtime assertion [21] the violation of which causes the system to stop, as no move is possible any more[3]. Many games may run concurrently, using different matchballs. The correlation of Players' actions with the right game is accomplished with the flow concept of the Peer Model: Entries belonging to the same game are stamped with the same flow identifier (fid). A wiring transaction treats only entries with compatible flow.

Figures 4, 5, 6 and 7, which are drawn with the new draw.io shapes for the Peer Model, present the Player Peer. In addition there is an Arbiter Peer (see Appendix A2, Fig. 8), and an exchangeable Broadcaster Peer (not shown, as it is not part of the use case). The Arbiter Peer's responsibilities are to init itself (see wiring `init`), to start the game (see wiring `startGame`), to recognize the end of the game (see wiring `endGame`) and then to decide about the winner (see wiring `decision`). There is one decision wiring for each player that checks if this very player is the winner. Note that the fid is needed to correlate the matchball with the individual statistics of the Players and with the result.

Figure 4 declares entry and types entries that are used by and/or shared between the Peers: `matchball` (global game state that is passed around), `createGame` (created by the Arbiter Peer who starts the game and creates a flow id), `gameInfo` (info about a new game sent by the Arbiter Peer to all Player Peers), `winner` (result about who has won the game), `actionInfo` (sent by each Player Peer to all other Peers, saying is has taken a game move), `statistics` (local view of each Player Peer on the game, i.e. the current scores of all Peers), and `decide` (used by the Arbiter Peer for the decision about the winner).

Figure 5 and Fig. 6 specify the Player Peer type. Wirings are either directly modeled, or configured based on wiring types. As convention, each Peer receives a system entry termed `INIT` in its PIC at the beginning. The Player Peer just removes this entry (see `deleteInit` wiring) There is a concurrent and competing `doGameAction` wiring for each possible next player. This models the indeterminism of who will get the ball next. Specifically mentioned must also be guard 1 of this wiring, where the runtime assertion that verifies local versus global state is modeled. Note the usage of the flow id (fid) to correlate all components of a game. The wiring `playerInit` creates a local entry for the Peer's statistics. Every

[3] E.g., in case of Byzantine errors [22], the assertion might be violated.

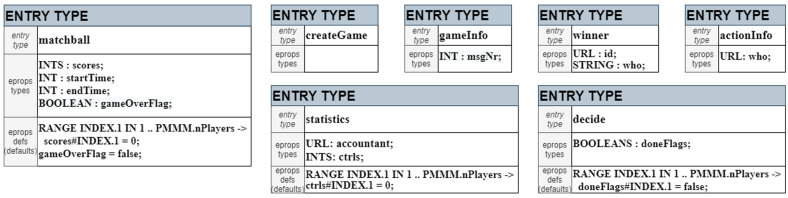

Fig. 4. Entry Types.

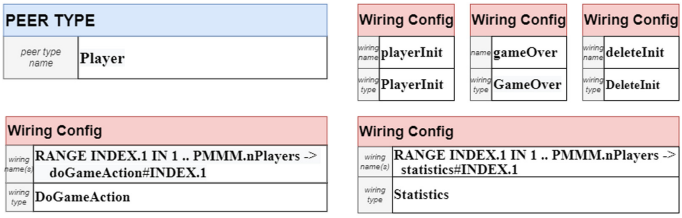

Fig. 5. Peer Type "Player".

time a Peer receives an `actionInfo` entry, it updates its bookkeeping (see wiring `statistics`). There is a concurrent `statistics` wiring for each Player, responsible to update the control counter for this very Player. Finally, the `gameOver` checks the matchball, if there exists one Player who has reached a certain high score, which is configured as property `max` in the Peer Model Meta Model.

Figure 7 shows the Peer Model Meta Model (PMMM) type. It defines a default `max` of 100. It also shows a PMMM configuration for up to 5 Player Peers and a high score of 5. The property `nPlayers` holds the actual number of players and is set to 3. For each Player an array property termed `players` is set in the PMMM.[4] The Arbiter Peer configuration defines the number of balls and the player who gets the matchball first.

5 Evaluation

Simulation Runs: For the evaluation, a simulation run of the Peer Competition use case is shown in Appendix A1. The game was carried out with three players, three matchballs and a high score of five. Please note that there is no limit for the amount of players, matchballs and the maximum to be reached that the simulation is able to cope with. The wiring traces, produced by "Watch" services, show the Players' moves, and the final global state of the entire system, which consists of the PIC and POC containers of all Peers. This way, it is possible to verify, if the output reflects an allowed result. Due to the indeterminism of map access to data structures (e.g. like machine controls, container entries etc.) and

[4] `gameInfo` serves only to make game runs more appealing by giving the players real names.

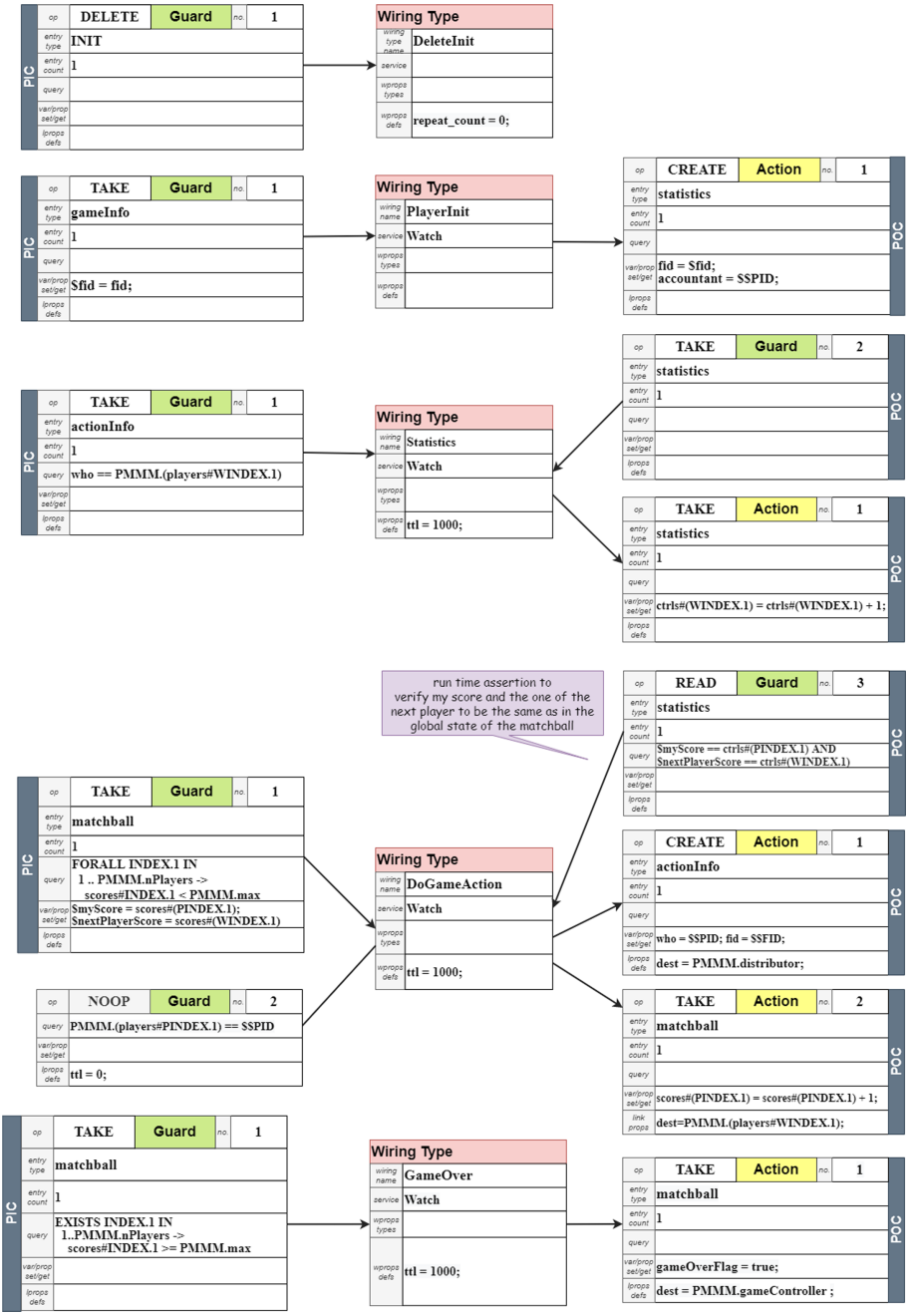

Fig. 6. Wiring Types for Player.

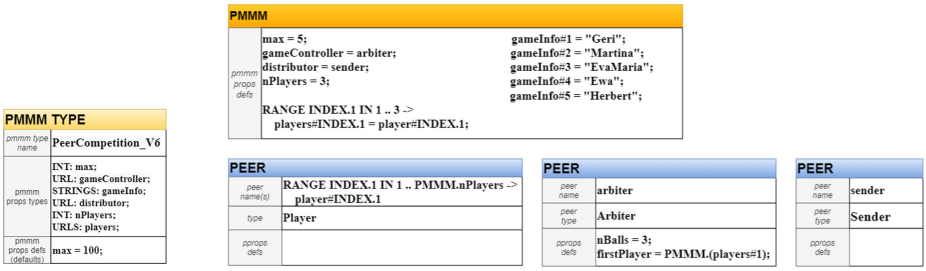

Fig. 7. PMMM Type (left) and Configuration (right).

the concurrent modelling of the machines in the PM-Go-Automaton, the result of a simulation run is not deterministic – another simulation run will result in other winners of the game. In addition, there exists a "model-checking" mode that systematically tries out all possible interleavings of the machines of the PM-Go-Automaton. These interleavings are determined by the enter/leave points of the re-entrant machines.

Variety of Application Possibilities: The Peer Model is a useful and feasible methodology that – as a proof-of-concept – has been employed to model a variety of use cases, reaching from embedded systems to enterprise scenarios in research as well as industrial scenarios. The Peer Model examples published in our papers so far, were verified with the PM-Go-Automaton, which has proven to be helpful to detect bugs in the designs. The graphical modeller of the new tool-chain has been tested already in several scenarios, like a simplified triage scenario of a hospital, a client/server pattern, some factory 4.0 scenarios, an armed gate, a producer/consumer pattern, and currently a blockchain use case is under development.

Graphical Modeller versus PM-DSL or Go Code: Whether developers prefer a graphical notation over a code-based one, depends on his/her educational background. In [20] we found out, that there was a slight preference for the (original) graphical notation of the Peer Model over an Event-B notation, which addressed rather mathematically oriented developers.

In any case, a graphical notation is a contribution that developers can discuss their models with their end users (clients). Even if a non-expert, like a client, will not be able to model a use case by him/herself, he/she can understand graphical models to a certain degree so that the communication between developer and end user is eased this way: For sure, this will not be possible with PM-DSL or Go code or any other notation that requires programming experience.

Usability and Scalability of Designs. The usability and scalability of models primarily depends on the semantics of the Peer Model modelling constructs themselves. The objective of its domain specific abstractions was to gain models without unnecessary, low-level "ballast" (see [18]). This means that models are

compact and therefore it was possible to depict the *entire* and complete model of the peer competition use case including its configuration in this paper. E.g., the correlation of data belonging to one workflow can be simply achieved by stamping the related entries with a flow identifier. A wiring will only consider entries with compatible flows, i.e. belonging to the same workflow. For example, in the peer competition, many matchballs can be around, each representing a separate game that does not interfere with the other ones: The Statistics wiring in Fig. 6 correlates the current action info with the right local book-keeping entry termed statistics; The DoGameAction wiring in Fig. 6 correlates the matchball with the local statistics and the next action carried out by the player (by setting the flow id property on the actionInfo entry in Action 1).

An analysis how good designs with the Peer Model's (original) graphical notation scale is given in [18]. Especially when it comes to more dynamic scenarios, the Peer Model is advantageous. The here introduced types for Peer Model artefacts also contribute to the scalability of designs, as well as the fact that the model is structured into many diagrams in the draw.io file. In the future, also the pattern-based concept proposed in [19] will be implemented.

With regard to usability, the number of concepts that a developer must learn play a major role [25]. There are only 14 draw.io shapes that a developer must comprehend, whereby, however, for convenience the single concept of a link has been explicitly "flatted" into 6 separate shapes for (PIC/POC/NOOP $\times$ action/guard), which gives 6 shapes instead of 1.

6 Conclusion and Future Work

The contribution of this work is a new tool-chain and an improved modelling notation for the Peer Model, which is translated by a model transformation algorithm to code for the simulator. The objective is to support the design of complex coordination algorithms involving high concurrency and many Peers. The new tool-chain has been demonstrated by means of an example whose full specification is given in the paper. Currently we apply the tool-chain for the verification of smart contracts of blockchain applications, as well as for consensus protocols on the infrastructure layer.

In future work the methodology shall be extended by invariants, hybrid verification tools and more sophisticated pattern support. Also planned are the visualization of the results of simulation runs, and in the long-term of the runtime behaviour of a Peer Model by showing the states of the containers and how entries are moved between them.

Appendix

A1: Simulation Run

```
player#1_playerInit_SID_Watch_SIC____M8: WATCH:
    <Id=e90, type=gameInfo, fid=‘‘f1">
player#1_doGameAction#3_SID_Watch_SIC____M16: WATCH:
    <Id=e86, type=matchball, fid=‘‘f1", gameOverFlag=false, scores#1=0, scores#2=0, scores#3=0, startTime=46>
    <Id=e123, type=statistics, accountant=‘‘player#1", ctrls#1=0, ctrls#2=0, ctrls#3=0, fid=‘‘f1">
player#2_playerInit_SID_Watch_SIC____M22: WATCH:
    <Id=e92, type=gameInfo, fid=‘‘f1">
player#3_playerInit_SID_Watch_SIC____M31: WATCH:
    <Id=e94, type=gameInfo, fid=‘‘f1">
player#3_doGameAction#3_SID_Watch_SIC____M30: WATCH:
    <Id=e126, type=matchball, fid=‘‘f1", gameOverFlag=false, scores#1=1, scores#2=0, scores#3=0, startTime=46>
    <Id=e204, type=statistics, accountant=‘‘player#3", ctrls#1=0, ctrls#2=0, ctrls#3=0, fid=‘‘f1">
player#1_playerInit_SID_Watch_SIC____M8: WATCH:
    <Id=e158, type=gameInfo, fid=‘‘f2">
player#1_doGameAction#1_SID_Watch_SIC____M11: WATCH:
    <Id=e103, type=matchball, fid=‘‘f2", gameOverFlag=false, scores#1=0, scores#2=0, scores#3=0, startTime=58>
    <Id=e217, type=statistics, accountant=‘‘player#1", ctrls#1=0, ctrls#2=0, ctrls#3=0, fid=‘‘f2">
player#2_playerInit_SID_Watch_SIC____M22: WATCH:
    <Id=e160, type=gameInfo, fid=‘‘f2">
player#3_playerInit_SID_Watch_SIC____M31: WATCH:
    <Id=e162, type=gameInfo, fid=‘‘f2">
player#1_playerInit_SID_Watch_SIC____M8: WATCH:
    <Id=e175, type=gameInfo, fid=‘‘f3">
player#1_doGameAction#2_SID_Watch_SIC____M12: WATCH:
    <Id=e115, type=matchball, fid=‘‘f3", gameOverFlag=false, scores#1=0, scores#2=0, scores#3=0, startTime=73>
    <Id=e254, type=statistics, accountant=‘‘player#1", ctrls#1=0, ctrls#2=0, ctrls#3=0, fid=‘‘f3">
player#2_playerInit_SID_Watch_SIC____M22: WATCH:
    <Id=e177, type=gameInfo, fid=‘‘f3">
player#3_playerInit_SID_Watch_SIC____M31: WATCH:
    <Id=e179, type=gameInfo, fid=‘‘f3">
player#2_doGameAction#2_SID_Watch_SIC____M25: WATCH:
    <Id=e257, type=matchball, fid=‘‘f3", gameOverFlag=false, scores#1=1, scores#2=0, scores#3=0, startTime=73>
    <Id=e364, type=statistics, accountant=‘‘player#2", ctrls#1=0, ctrls#2=0, ctrls#3=0, fid=‘‘f3">
player#3_doGameAction#3_SID_Watch_SIC____M30: WATCH:
    <Id=e207, type=matchball, fid=‘‘f1", gameOverFlag=false, scores#1=1, scores#2=0, scores#3=1, startTime=46>
    <Id=e391, type=statistics, accountant=‘‘player#3", ctrls#1=1, ctrls#2=0, ctrls#3=1, fid=‘‘f1">
player#1_doGameAction#2_SID_Watch_SIC____M12: WATCH:
    <Id=e220, type=matchball, fid=‘‘f2", gameOverFlag=false, scores#1=1, scores#2=0, scores#3=0, startTime=58>
    <Id=e418, type=statistics, accountant=‘‘player#1", ctrls#1=1, ctrls#2=0, ctrls#3=0, fid=‘‘f2">
player#2_doGameAction#2_SID_Watch_SIC____M25: WATCH:
    <Id=e367, type=matchball, fid=‘‘f3", gameOverFlag=false, scores#1=1, scores#2=1, scores#3=0, startTime=73>
    <Id=e550, type=statistics, accountant=‘‘player#2", ctrls#1=1, ctrls#2=1, ctrls#3=0, fid=‘‘f3">
...
player#3_doGameAction#2_SID_Watch_SIC____M34: WATCH:
    <Id=e1934, type=matchball, fid=‘‘f1", gameOverFlag=false, scores#1=4, scores#2=4, scores#3=4, startTime=46>
    <Id=e2026, type=statistics, accountant=‘‘player#3", ctrls#1=3, ctrls#2=4, ctrls#3=4, fid=‘‘f1">
player#2_gameOver_SID_Watch_SIC____M18: WATCH:
    <Id=e2029, type=matchball, fid=‘‘f1", gameOverFlag=false, scores#1=4, scores#2=4, scores#3=5, startTime=46>

*** SYS INFO ------ SYSTEM TTL 100000 exceeded ------------------------------------------------------- CLOCK=101189
*** SYS INFO ------ Controler EXIT ------------------------------------------------------------------- CLOCK=101189

    -------------------- SPACE at CLOCK=101189 --------------------
arbiter_POC updateEvtTime=2083 = {
  <Id=e1540, type=matchball, endTime=1735, fid=‘‘f3", gameOverFlag=true, scores#1=2, scores#2=5, scores#3=2, startTime=73>,
  <Id=e1730, type=matchball, endTime=1920, fid=‘‘f2", gameOverFlag=true, scores#1=5, scores#2=3, scores#3=2, startTime=58>,
  <Id=e2029, type=matchball, endTime=2080, fid=‘‘f1", gameOverFlag=true, scores#1=4, scores#2=4, scores#3=5, startTime=46>,
  <Id=e1825, type=winner, fid=‘‘f3", id=‘‘player#2", who=‘‘!!! the winner is Martina !!!">,
  <Id=e1976, type=winner, fid=‘‘f2", id=‘‘player#1", who=‘‘!!! the winner is Geri !!!">,
  <Id=e2127, type=winner, fid=‘‘f1", id=‘‘player#3", who=‘‘!!! the winner is EvaMaria !!!">}
player#1_POC updateEvtTime=2029 = {
  <Id=e1944, type=statistics, accountant=‘‘player#1", ctrls#1=5, ctrls#2=3, ctrls#3=2, fid=‘‘f2">,
  <Id=e1766, type=statistics, accountant=‘‘player#1", ctrls#1=2, ctrls#2=5, ctrls#3=2, fid=‘‘f3">,
  <Id=e2089, type=statistics, accountant=‘‘player#1", ctrls#1=4, ctrls#2=4, ctrls#3=5, fid=‘‘f1">}
player#2_POC updateEvtTime=2044 = {
  <Id=e1949, type=statistics, accountant=‘‘player#2", ctrls#1=5, ctrls#2=3, ctrls#3=2, fid=‘‘f2">,
  <Id=e1771, type=statistics, accountant=‘‘player#2", ctrls#1=2, ctrls#2=5, ctrls#3=2, fid=‘‘f3">,
  <Id=e2096, type=statistics, accountant=‘‘player#2", ctrls#1=4, ctrls#2=4, ctrls#3=5, fid=‘‘f1">}
player#3_POC updateEvtTime=2057 = {
  <Id=e1958, type=statistics, accountant=‘‘player#3", ctrls#1=5, ctrls#2=3, ctrls#3=2, fid=‘‘f2">,
  <Id=e1778, type=statistics, accountant=‘‘player#3", ctrls#1=2, ctrls#2=5, ctrls#3=2, fid=‘‘f3">,
  <Id=e2103, type=statistics, accountant=‘‘player#3", ctrls#1=4, ctrls#2=4, ctrls#3=5, fid=‘‘f1">}
    ------------------------------------------------------------
```

A2: Arbiter Peer Type

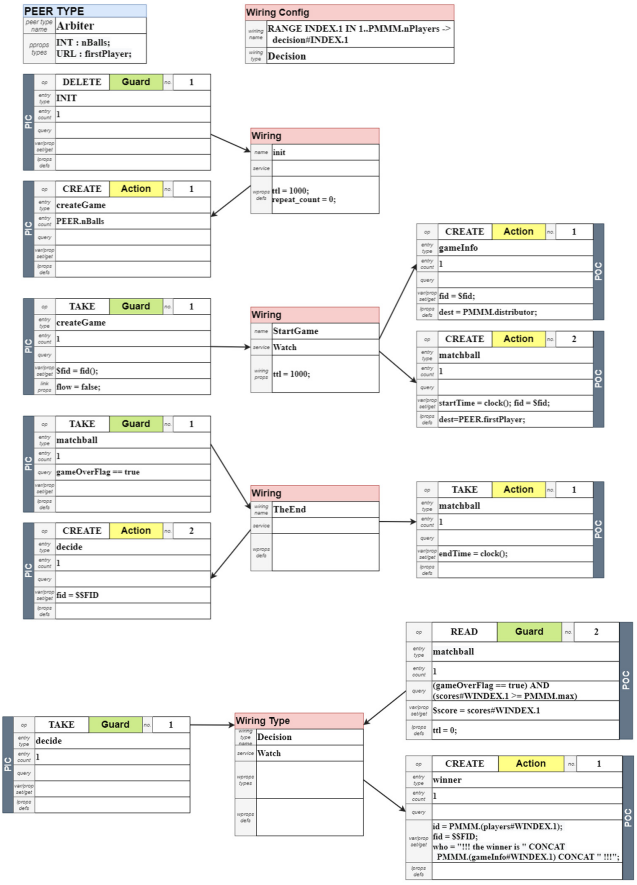

Fig. 8. Peer Type "Arbiter" (above) and Wiring Types for Arbiter (below).

References

1. Ethereum. https://www.ethereum.org/. Accessed 01 Apr 2019
2. Flowchart marker and online diagram software (draw.io). https://app.diagrams.net/. Accessed 20 Feb 2020
3. GreatSPN 3, Universita di Torino, http://www.di.unito.it/greatspn/
4. UPPAAL 4.015, Uppsala University Sweden (2019). https://uppaal.org/
5. High-level Petri Nets - Concepts, Definitions and Graphical Notation. Tech. rep., Final Draft International Standard ISO IEC 15909, V. 4.7.1 (2000)

6. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
7. Agha, G.A.: ACTORS: A Model Of Concurrent Computation in Distributed Systems. MIT Press (1990)
8. Cejka, S.: Enabling scalable collaboration by introducing platform-independent communication for the Peer Model. Master's thesis, TU Wien (2019)
9. Craß, S.: Secure coordination through fine-grained access control for space-based computing middleware. Ph.D. thesis, TU Wien (2020)
10. Elaraby, N., Kühn, E., Messinger, A., Radschek, S.T.: Towards a hybrid verification approach. In: Mazzara, M., Ober, I., Salaün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 367–386. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_27
11. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
12. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM (CACM) **35**(2), 96–107 (1992)
13. Group, E.E.U.: Hybrid ertms/etcs level 3: principles. Technical Report, Ref: 16E042, version 1A, Hybrid-ERTMS-ETCS-Level-3 (2017)
14. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Int. J. Softw. Tool Technol. Transfer (STTT) **9**, 213–254 (2007)
15. Kuehn, E.: Reusable coordination components: reliable development of cooperative information systems. Int. J. Cooperative Inf. Syst. **25**(4), 1740001:1–1740001:32 (2016)
16. Kühn, E.: Flexible transactional coordination in the peer model. In: Dastani, M., Sirjani, M. (eds.) FSEN 2017. LNCS, vol. 10522, pp. 116–131. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68972-2_8
17. Kuehn, E.: Peer Model: Agile Middleware and Programming Model for the coordination of parallel and distributed flows. Tech. rep, TU Wien, Institute of Computer Languages (2012)
18. Kühn, E., Craß, S., Joskowicz, G., Marek, A., Scheller, T.: Peer-based programming model for coordination patterns. In: De Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 121–135. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38493-6_9
19. Kuehn, E., Craß, S., Schermann, G.: Extending a peer-based coordination model with composable design patterns. In: 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP. IEEE (2015)
20. Kühn, E., Radschek, S.T.: An initial user study comparing the readability of a graphical coordination model with event-B Notation. In: Cerone, A., Roveri, M. (eds.) SEFM 2017. LNCS, vol. 10729, pp. 574–590. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74781-1_38
21. Kuehn, E., Radschek, S.T., Elaraby, N.: Distributed coordination runtime assertions for the Peer Model. In: Di Marzo Serugendo, G., Loreti, M. (eds.) COORDINATION 2018. LNCS, vol. 10852. pp. 200–219, Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92408-3_9
22. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
23. Petri, C.A.: Kommunikation mit automaten. Ph.D. thesis, Technische Hochschule Darmstadt (1962)

24. Ratzer, A.V., et al.: CPN tools for editing, simulating, and analysing coloured petri nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 450–462. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44919-1_28
25. Scheller, T., Kuehn, E.: Automated measurement of API usability: the API concepts framework. Inf. Softw. Technol. **61**, 145–162 (2015)