Flexible Transactional Coordination in the Peer Model

Eva Kühn(✉)

Institute of Computer Languages, TU Wien, Vienna, Austria
eva.kuehn@tuwien.ac.at
http://www.complang.tuwien.ac.at/eva

Abstract. The Peer Model is a model for the specification of coordination aspects found in concurrent and distributed systems. It provides modeling constructs for flows, time, remoting and exception handling. The main concepts of the ground model are peers, wirings, containers, entries and services. Its intent is to introduce specific modeling abstractions of concurrency and distribution to make designs more readable and suitable for larger problems. However, there still exist coordination aspects that are not straight forward to model with it. In this paper, therefore the Peer Model is extended by modeling constructs for nested, distributed transactions based on the Flex transaction model. This approach eases the advanced control of structured and distributed coordination scenarios that have to cope with complex, dependent and concurrent flows. The evaluation introduces a coordination challenge that requires adaptive and transactional distribution of resources, dependencies between concurrent activities, error handling and compensation. It demonstrates the improvements that can be achieved with the new modeling concepts.

Keywords: Coordination model · Flexible transactions · Concurrent and distributed systems

1 Introduction

Cooperative information systems involve demanding coordination aspects. Separating coordination from application logic and providing precise models of the coordination is crucial in order to gain robust, distributed software systems [1]. The coordination pattern approach [2] suggests generalizing the aspects of how the participating and distributed processes interact. This enables coordination generics to become reusable among different applications and domains. A major benefit is that these complex parts of a distributed application need not be re-invented for each new application, thus contributing to more reliable systems.

Coordination requirements [3] extend far beyond routing information between processes: They comprise management of complex dependencies among many concurrent and distributed processes in real-time. Therefore, traditional coordination models quite often reach their limitations with regard to expressive power and usability of resulting models. Well known examples are Petri Nets [4], Actor Model [5] and Reo [6]. All are general and powerful and have mathematical foundations. The Actor Model abstracts asynchronous communication, but the behavior of an actor intertwines application logic with communication and synchronization logic. In contrast to Petri Nets and Actor Model, Reo provides clear separation of coordination from application logic; however, the abstraction level is similar to Petri Nets, where larger models tend to become unreadable [7,8]. An advantage of Petri Nets is that with the concept of transitions they provide a powerful modeling construct for atomic transactions.

The Peer Model is a coordination model that introduces specific assumptions for distributed and concurrent systems in order to make models less complex and easier to understand. The main abstractions comprise modeling concepts for local transactions, remoting, flow correlation, exceptions and timing. It separates coordination from application data and logic. Application data is represented as a "black box" and application logic is encapsulated into services that manipulate the application data and are called by the coordination layer.

All mentioned coordination models are able to model any scenario. However, if the complexity of a use case increases, the strict separation of application and coordination layer might not be maintained (as coordination logic slips into the application logic) or the model requires a lot of cumbersome work to specify details that are not directly related with the problem at hand.

A challenging coordination scenario is defined in Sect. 3 that lets also the original Peer Model reach its limitations with regard to modeling expressiveness. It comprises a factory, distributed and concurrent workers, and shops where resources can be ordered. The workers continuously execute tasks and compete for shared resources. Possible concurrency shall be exploited. A further complicating aspect is the distribution of processes. The specific challenges comprise: complex dependencies between activities, automatic compensation, error and timeout handling, and distributed transaction management.

This paper presents an extension of the Peer Model by a flexible, distributed, nested transaction approach termed flexible wiring transactions (FWTX) in order to ease the design of such advanced coordination scenarios. Also other coordination models can benefit from this concept. As proof-of-concept for the new modeling constructs the mentioned factory example is used.

The structure of the paper is as follows: Sect. 2 explains the original Peer Model. Section 3 introduces a coordination challenge and discusses a solution for it with the original Peer Model. Section 4 presents the new flexible wiring transaction concept (FWTX), inspired by Flex transactions [9]. Section 5 evaluates it with the coordination challenge, demonstrates that designs become leaner, and gives a comparison with related coordination models. Section 6 concludes the paper and gives an outlook for future work.

2 Peer Model

The Peer Model [2,8] is a coordination model with high-level modeling abstractions for concurrent and distributed systems: A *peer* relates to an actor in the

Actor Model [5]. It is an autonomous worker with ingoing and outgoing mailboxes, termed input and output *containers* (PIC and POC). The foundation of containers is tuple-space-based communication [10,11]. A container relates to a sub-space that maintains tuples (called *entries*) and supports transactional *queries* on them. The coordination behavior of the peer is explicitly modeled by *wirings* that are similar to Petri Net transitions [4]. It is triggered by events represented as *entries* written into containers. A wiring possesses *links* for the transport of entries between containers. Incoming links are termed *guards* and outgoing ones *actions*. A wiring stands for concurrent instances that actively and repeatedly execute the wiring specification. A wiring instance is one atomic local space transaction, termed wiring transaction (WTX) on the peer's space containers. As transaction mechanism we assume pessimistic locking and repeatable read isolation level. The operational behavior of a WTX is to execute in the specified sequential order first guard, then *service*, and finally action links. The WTX collects all entries retrieved by guards in an internal and non-transactional container that serves as a temporary, local entry collection for this wiring instance. Service links transport entries between the internal wiring container and the service and call the application method. Note that guard and action links set locks within the WTX on the space containers. All artifacts have *properties*. System properties have a pre-defined semantics, e.g. if a time-to-live (`ttl`) property on a WTX or on a link expires, this causes the current WTX to rollback and start a new instance.

2.1 Artifacts of the Ground Model

Property $prop = (label, val)$. *label* is a name, and *val* denotes a value. A label that defines a system property is written in typewriter style, otherwise it is an application property. The property is named after its label.

Entry $e = \mathbb{E}prop$. $\mathbb{E}prop$ is a set of properties $\{prop_1, prop_2, \ldots, prop_n\}$. Entry system properties are e.g., `type` (obligatory coordination type of the entry), `ttl` (time-to-live: if it expires the entry is wrapped into an exception entry[1]; default is infinite), `fid` (flow identifier), and `data` (application-specific data).

Container $c = (cid, \mathbb{E}, \mathbb{C}oord, \mathbb{C}prop)$. A container stores entries. *cid* is a unique name, $\mathbb{E}$ a set of entries, $\mathbb{C}oord$ a set of coordinators (see Query below), and $\mathbb{C}prop$ a set of system properties. A container relates to an XVSM container [11]. We differentiate between space containers and internal containers. The former ones support transactions and blocking behavior. Entries are retrieved by a query that necessarily requires the coordination type of the entry.

Query $q = (type, cnt, \mathbb{S}el)$. *type* is an entry coordination type. *cnt* is a number, a range, or the keyword `ALL` or `NONE`, determining the amount of entries to be selected; default is 1. $\mathbb{S}el$ is a sequence of AND/OR connected selectors. A selector is lent from the XVSM query mechanism [12]. It refers to a container

[1] In the assumed configuration exception entries are written into the peer's POC.

coordinator (e.g. `fifo`, `key`, `label`, `any`) or is a selection expression involving entry properties, variables and system functions.

Link $l = (c_1, c_2, op, q, \mathbb{E}xpr, \mathbb{L}prop)$. c_1 refers to a source and c_2 to a target container. $op \in \{$`create`, `copy`, `move`, `delete`, `test`, `noop`, `call`$\}$. `create` creates new entries and writes them to c_2. `copy` reads entries from c_1 and writes them to c_2. `move` reads and deletes entries from c_1 and writes them to c_2. `delete` reads and deletes entries from c_1. `test` checks entries in c_1. `noop` only executes q which must not refer to entries. `call` calls a service. All operations must fulfill the query q, if it is not empty, on c_1. $\mathbb{E}xpr$ is a sequence of expressions that set or get properties of selected entries and/or of variables [2]. $\mathbb{L}prop$ is a set of system properties, e.g.: `tts` (time-to-start: how long the link execution must wait to start; default is 0), `ttl` (time-to-live: how long the link execution may be retried until it succeeds; if it expires, a system entry of type exception is created that wraps the original entry and provides the type of the original entry in a property termed `ettl`; default is infinite), `dest` (specifies the id of a destination peer to which all selected entries on an action link are automatically transported via intermediary I/O peers [2]), `flow` (if `true` (default), the link transports only "flow-compatible" [2] entries – this means that the `fid` of all entries transported by links of this WTX must be the same or not set), and `mandatory` (if `true` (default), the fulfillment of the link is obligatory).

Wiring $w = (wid, \mathbb{G}, \mathbb{S}, \mathbb{A}, wic, \mathbb{W}prop)$. wid is a unique name, $\mathbb{G}$ is a sequence of guard links, $\mathbb{S}$ is a sequence of service links, $\mathbb{A}$ is a sequence of action links, wic is the id of an internal container, and $\mathbb{W}prop$ is a set of system properties, e.g., `tts` (time-to-start; time that the next instance of this wiring waits until its start; default is 0), and `ttl` (time-to-live; maximal execution time of one instance of this wiring; default is infinite). All links are numbered, specifying an execution order which has impact on concurrency and performance. Entries selected by guards are written into wic. Then w calls the service links in the specified sequence. Finally, the wiring executes the action links. c_2 of a guard and c_1 of an action link is wic. There is one dedicated wiring in a peer termed init wiring with its first guard having the identifier "*"; it is fulfilled exactly once, namely when the peer is activated.

Service $s = (sid, app)$. sid is the name of the service and app a reference to the implementation of its application logic (method). A service gets entries from its wiring's wic as input and emits result entries there (via service links). It has access to all entry properties including `data`.

Peer $p = (pid, pic, poc, \mathbb{W}id, \mathbb{S}pid, \mathbb{P}prop)$. pid is a unique name, pic and poc are the ids of incoming and outgoing space containers where p receives and delivers

[2] Application variables have the scope of the current wiring instance and start with a "\$". They are set by $\mathbb{E}xpr$. $\mathbb{E}xpr$ may involve system functions like "fid()" which generates a new unique flow identifier, as well as system variables (starting with "\$\$") that are set by the system, e.g., \$\$PID (name of the current peer), \$\$FID (actual flow id within the current wiring instance), and \$\$CNT (number of entries selected by the current link).

entries, $\mathbb{W}id$ is a set of wiring ids, $\mathbb{S}pid$ is a set of ids of sub-peers, and $\mathbb{P}prop$ is a set of system properties.

Peer Model $PM = (\mathbb{P}, \mathbb{W}, \mathbb{C})$. $\mathbb{P}$ is the set of all peers including sub-peers, $\mathbb{W}$ is the set of all wirings, and $\mathbb{C}$ is the set of all containers in the system.

2.2 Graphical Notation

The graphical representation of the Peer Model is shown in Fig. 1, outlining one peer with one wiring that has two links and calls one service (the depiction of service links is skipped). The guard link connects the peer's *pic* with the wirings's *wic*, and the action link connects the wiring's *wic* with the peer's *poc*. Note that the source space container of a guard can also be the peer's *poc* or the *poc* of a sub-peer. Analogously the target space container of an action link can also be the peer's *pic* or the *pic* of a sub-peer. A wiring can have many links that are numbered with $G_1, \ldots, G_k, S_1, \ldots, S_m, A_1, \ldots, A_n$ (the link ids are not depicted in Fig. 1). A peer can have many wirings.

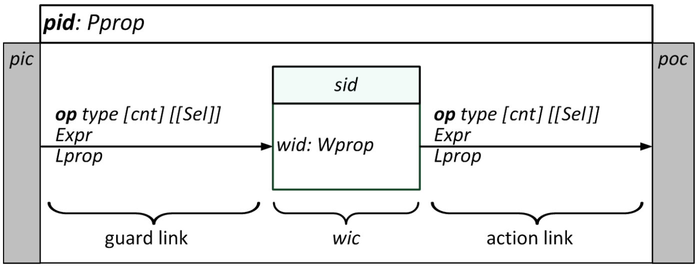

Fig. 1. Example peer.

3 Coordination Challenge

The contrived coordination example is a bakery with autonomous bakers who are specialized in producing certain products like bread, pizza, cake etc. Each product requires a defined amount of ingredients (eggs, flour, sugar etc.). The bakery operator provides the ingredients for the bakers which compete for them. If an ingredient runs short, the concerned bakers cannot proceed. The bakery tries to procure all missing ingredients at respective shops.

A baker's job is to produce doughs for the respective product as fast as possible and to send each dough immediately to the bakery. For this, he/she must first get hold of the needed ingredients. Another critical time is the stirring of the dough. If 5 pieces of dough are ready they form a "charge" that is baked in the oven. The baker informs the bakery when a charge must be sent to the oven. Only doughs of the same charge of a baker are baked together. However, the baker has a timeout for producing one dough. If it exceeds, the current charge – although incomplete – shall nevertheless be baked to avoid the risk of already produced dough to go off, provided the charge contains at least one dough.

The example is complicated to model, because of challenging **dependencies between the concurrent coordination steps**: The occurrence of failures (the lack of an ingredient) and timeouts (the obtaining of the ingredients or the dough stirring service takes too long) influence the completion of a charge. The baker must recognize in real-time whether a charge is complete or whether the production is stuck (due to **errors or timeouts**) and inform the bakery to start the delivery of the current charge to the oven at the right moment. The bakery is responsible to fill up its stock if resources run low. Let us assume a simple policy whereby in a defined time interval the bakery checks its stock to be below a certain boundary and tries to completely fill it up. It orders each kind of ingredient at a different shop. The **distributed procurement transaction** shall succeed only if all ingredients can be purchased; otherwise nothing is bought and the bakery retries the procurement process later on. Shops are autonomous and not willing to hold locks on items: They immediately remove the ordered ingredients from their stock and put them in a temporary container for the client. If the global transaction succeeds, they deliver the ingredients to the client; otherwise a **compensation** must take place that moves the reserved ingredients back to the shop's stock. This means that other clients might think that the shop has no items any more, albeit later on they are put back to the stock because the client has aborted the global transaction.

3.1 Bakery Without FWTX

Figures 2 and 3 model the use case with the original Peer Model, i.e. without FWTX. The three main peers types are shown: Baker, Bakery and Shop. Their behavior is represented by wirings as detailed below for each peer. The dough production must be split into two wirings to model the acquisition of ingredients before the dough stirring can start. All phases of the distributed procurement transaction between bakery and shops and the cleaning up of outdated entries used by the distributed transaction management must be modeled explicitly.

Baker Peer:

- *Init:* Create an entry for the next charge with a new `fid`, a `ttl`, and 0 doughs (property k), and set current phase to 1 (A1).
- *ProduceDough1:* If there is a charge with less than 5 doughs in phase 1 (G1), then set its phase to 2 (A1) and create a request in the current flow with a `ttl` and the baker's pid, and send it to the bakery, asking to send ingredients for the next dough (A2).
- *ProduceDough2:* If there is a charge with less than 5 doughs in phase 2 (G1), and if the needed ingredients are there (G2–G3), then call the dough stirring service, increment the dough count of the charge, and set its phase to 1 (A1), and send the dough, that was produced by the service, within the current

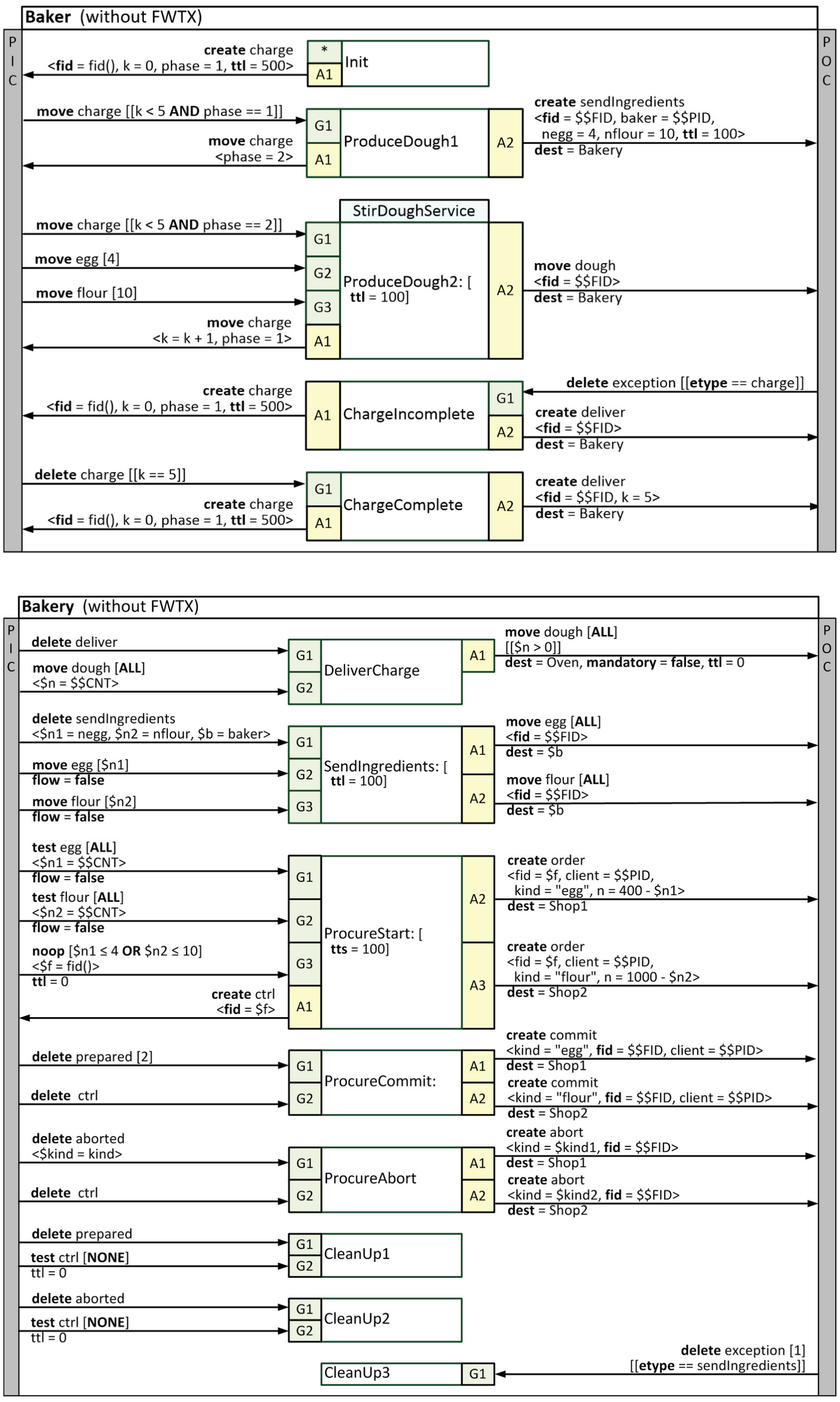

Fig. 2. Baker and Bakery (without FWTX).

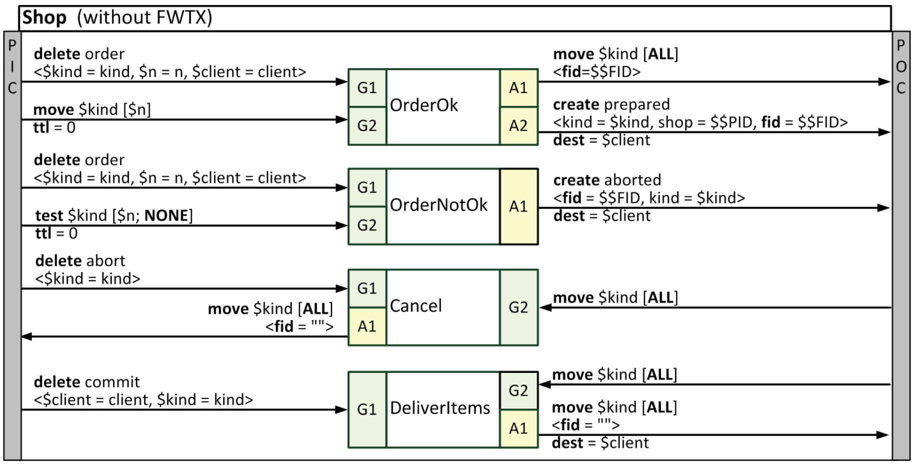
Shop (without FWTX)
PIC
POC
delete order <$kind = kind, $n = n, $client = client>
move $kind [$n] ttl = 0
OrderOk
move $kind [ALL] <fid=$$FID>
create prepared <kind = $kind, shop = $$PID, fid = $$FID> dest = $client
delete order <$kind = kind, $n = n, $client = client>
test $kind [$n; NONE] ttl = 0
OrderNotOk
create aborted <fid = $$FID, kind = $kind> dest = $client
delete abort <$kind = kind>
move $kind [ALL] <fid = "">
Cancel
move $kind [ALL]
delete commit <$client = client, $kind = kind>
DeliverItems
move $kind [ALL]
move $kind [ALL] <fid = ""> dest = $client

Fig. 3. Shop (without FWTX).

charge's flow to the bakery (A2). The WTX is bounded by a timeout (`ttl`). If it expires, it performs a rollback, releases all locks on entries, and retries.
- *ChargeIncomplete:* If the charge has expired and turned into an exception (G1), then create a new charge with a new flow id (A1), and tell the bakery to deliver the incomplete charge, referred to by its flow id, to the oven (A2).
- *ChargeComplete:* If the current charge is complete with 5 doughs (G1), then start a new charge within a new flow id (A1), and tell the bakery to deliver the complete charge, referred to by its flow id, to the oven (A2).

Bakery Peer:

- *DeliverCharge:* Upon receipt of a deliver request (G1), take all doughs of the same charge (correlated by their flow id) and remember in the local variable $n how many were taken (G2), and send them to the oven (A1) if there exists at least one dough.
- *SendIngredients:* If a sendIngredients request is received (G1) and if the requested ingredients (G2–G3) are there, then send them to the requesting baker (A1–A2).
- *ProcureStart:* In a defined interval (modeled as `tts` of the wiring) check how many ingredients are still there (G1–G2). If one of them has fallen below a defined threshold create a new `fid` (G3). If ingredients are missing, then create a ctrl entry within this flow and store it in the PIC in order to control the distributed procurement transaction (A1), and send an order request for each ingredient to the corresponding shops (A2–A3).
- *ProcureCommit:* The information that all shops are in prepared state has received (G1), and the corresponding ctrl entry (G2) for this flow exists: Create commit entries within this flow carrying the id of this peer and send them to all shops with information about the confirmed order (A1–A2).

- *ProcureAbort:* A shop has sent an aborted entry (G1), and the corresponding ctrl entry (G2) is found: Create abort entries within this flow and send them to all shops (A1–A2).
- *CleanUp1:* Remove an outdated prepared entry (G1) for which no ctrl entry exists any more (G2).
- *CleanUp2:* Remove an outdated aborted entry (G1) for which no ctrl entry exists any more (G2).
- *CleanUp3:* Remove an sendIngredient exception (G1).

Shop Peer:

- *OrderOk:* If an order request arrived (G1), and the required amount of the ingredient can be taken from the shop's stock (represented by its PIC) (G2), then temporarily move these ingredients to the POC with the same `fid` as the order (A1), and send a prepared entry to the requesting client within this flow indicating what has been reserved for it and by which shop (A2).
- *OrderNotOk:* If an order request is received (G1), but the required amount of this ingredient is not in stock (G2), then send aborted to the client in this flow (A1). Note the counter expression "[$n; NONE]" on G2: It models a range with the meaning "not at least $n entries".
- *Cancel:* A client has aborted the distributed transaction (G1), and the reserved amount of ingredients is therefore withdrawn from the intermediate storage (G2): Write these ingredients back to the shop's stock (A1).
- *DeliverItems:* The client has issued a commit for the distributed transaction (G1), and the ingredients are therefore removed from the intermediate storage (G2): Send them to the respective client (A1).

4 Flexible Wiring Transactions (FWTX)

The Flex transaction model [9,13,14] defines nested transactions that allow the early commit of sub-transactions, thus relaxing the isolation property of transactions. The tradeoff is that so-called compensate actions must be supported. Compensate actions are motivated by Sagas [15]. They are application defined logic that carries out a compensation of the effects of committed sub-transactions, however they cannot really "undo" in the strict sense an effect that was already seen by others, but only perform a "semantic" compensation. They are activated by the transaction manager if a sub-transaction has committed and then one of its parent transactions fails. No cascading compensation is done, i.e. if a sub-transaction commits, it is responsible for the compensation of its sub-transactions. The Flex transaction model supports compensatable as well as non-compensatable sub-transactions. The former perform an early commit, the latter delegate their commit to the caller (cf. nested transactions [16]).

The idea to use a flexible transaction model to coordinate distributed processes in heterogeneous systems was firstly used by the coordination kernel [14], implementing a distributed virtually shared object space. The coordination kernel extends the Flex transaction model by on-commit and on-abort actions that are called if a transaction commits respectively aborts. It was the basis for the later CORSO (coordinated shared objects) coordination system [17] that demonstrates that the Flex transaction model can be implemented efficiently.

We adapt here this concept for the Peer Model. The local transaction of a wiring (WTX) is extended towards flexible wiring transactions (FWTX). A WTX locally executes all links in one atomic step (see Sect. 2.1). A FWTX in addition supports nested flexible transactions, as well as compensate actions (optionally cascading or not), on-commit actions, on-top-commit actions, and on-abort actions. The definition of a wiring is enhanced by introducing passive wirings which are not actively executing instances, but must be activated by other FWTXs. Passive wirings therefore may take input parameters so that the calling FTWX can pass local variables values (by value). Note: the communication between peers must be carried out by exchanging entries.

An instance of a passive wiring is activated by a parent FWTX either (i) via a guard link, or (ii) as a compensate, on-commit, on-top-commit or on-abort action (which in turn are FWTXs). For (i) the link definition (see Sect. 2.1) is extended in that *op* can also be **wiring**, denoting the sub-wiring to be called in a new sub-FWTX. For (ii) new wiring system properties ($\mathbb{W}prop$) are introduced: `on-top-commit`, `on-commit`, `on-abort`, and `compensate` to specify a passive local or remote wiring; and a boolean property termed `cascading` to define whether a compensation action is cascading or not. Parameters to be passed to the sub-wiring activation are modeled as part of $\mathbb{E}xpr$ as variables, where the i-th parameter is referred to by \$i. A sub-FWTX is activated exactly once. It inherits the flow id of the parent-FWTX, and vice versa, if at the time of its activation the flow id of the parent-FWTX is not yet determined, it can set it.

A parent FWTX is only dependent on synchronous sub-FWTXs that are called via guard links, provided that the property `mandatory` of this link is not turned off. The link execution must wait until the sub-FWTX – which can be a remote one – has finished. This concept extends the expressiveness of guards in that it becomes possible to send a request to a remote peer and wait in a subsequent guard for entries that the peer sends back. Otherwise this would require two or more wirings – implying that the flow of control becomes more complicated – as well as the explicit treatment of possible errors.

Let a FWTX X have an on-commit (OC), an on-top-commit (OTC), an on-abort (OA), and a compensate (COMP) action. OC is called immediately after X has committed. OTC is called immediately after the top-level-FWTX of X has committed. If X is the top-level-FWTX then OTC is called immediately after OC. OA is called immediately after X has aborted. COMP is called if X has committed and later on a parent-FWTX of it aborts. It runs asynchronously to X. If `cascading` is true, then the compensation is recursively propagated to all sub-FWTXs of X that were called via guard links. X waits with the execution of its next wiring instance (X') until all OC, OTC or OA executions have completed. The time how long it waits can optionally be configured by respective `ttl` wiring system properties for on-(top-)commit and on-abort actions. X is neither dependent on OC, OTC, COMP nor OA. On-commit, on-top-commit, on-abort and compensate actions are automatically committed.

The distributed transaction managers jointly control the execution of FWTXs: A FWTX must persist the information about each called sub-FWTX. If FWTX itself is nested, it must store its parent-FWTX and top-level-FWTX. It passes the id of its own FWTX and the top-level FWTX to each called sub-FWTX. If a sub-FWTX commits or aborts, it reliably sends this decision to its parent-FWTX, i.e. it repeats the sending until an acknowledgment is received. If it commits, it stores its compensate action until it receives the final decision of the top-level-FWTX. The necessary assumptions are that a crashed site eventually will recover and that eventually each pair of sub-FWTX and its direct parent-FWTX is available at the same time.

The model avoids that resources are locked for a long period of time or forever. The interesting error cases are caused by dependent sub-FWTX activations via guards. Breaking it down to the pair of a parent-FWTX and its sub-FWTX these situations comprise: *) A committed sub-FWTX must wait for its parent-FWTX's decision whether to compensate or not. During this time, because the relaxation of the isolation property allowed the early commitment of the sub-FWTX, no data need to be locked. The compensation is a semantic one; it is standalone and may run even a long time after the commitment of the sub-FWTX. *) A parent-FWTX cannot commit because its sub-FWTX did not answer yet. In this case it is recommended that the parent-FWTX uses a `ttl`. If the `ttl` fires then eventually the sub-FWTX will be aborted, too or needs to compensate. *) A sub-FWTX has committed, but the commit did not reach its parent-FWTX. Either the parent-FWTX waits until it can communicate again with sub-FWTX, or it aborts meanwhile. In the former case parent-FWTX can proceed, in the latter case eventually sub-FWTX learns about parent-FWTX's abort and will compensate.

In the graphical notation, the declaration of a passive wiring has a box with a dotted line and a parameter list enclosed by "()" brackets. The activation of a sub-FWTX via a guard uses the `wiring` operation.

5 Proof-of-Concept

As a proof-of-concept for the new FWTX concepts of the Peer Model, we present a solution with it for the bakery example. The number of wirings could be reduced from 17 to 10 (i.e. by ca. 41%) and the total number of links from 66 to 28 (i.e. by ca. 58%).

5.1 Bakery with FWTX

The baker uses a sub-FWTX to get ingredients from the bakery. If it fails, an on-abort action sends the incomplete charge immediately to the oven and starts a new charge. The distributed procurement transaction of the bakery uses sub-FWTXs with compensation to order ingredients at shops. If one shop fails, the other one is automatically aborted or compensated. If both succeed, their commit is implicitly triggered by the commit of the Procure FWTX; on-top-commit-actions at the shops start the goods delivery to the client.

The improvements of the version with FWTX (see Sect. 3.1) over the one without FWTX are summarized in the following.

Baker Peer (with FWTX):

- *Init:* No difference.
- *ProduceDough:* Consolidates ProduceDough1 and ProduceDough2 in one wiring where G2 calls a sub-wiring at the bakery termed SendIngredients. The definition of an on-abort action is added to the wiring to call a local sub-wiring termed StartNewCharge.
- *ChargeComplete:* Has only one guard (G1) that tests if the charge is complete and if so, calls the sub-wiring StartNewCharge as on-commit action.
- *StartNewCharge:* Is a new passive wiring. It takes the current charge entry (G1), resets both its fid and k and writes it back to the PIC (A1). It calls the DeliverCharge sub-wiring of the bakery as on-commit action and passes it the number of doughs in this charge as parameter.

Bakery Peer (with FWTX):

- *DeliverCharge:* Is a passive wiring called by the baker every time it starts a new charge via StartNewCharge. Therefore the original G1 is not needed.
- *SendIngredients:* Is a passive wiring called by the baker's wiring ProduceDough in G2. On start it calls Procur. The original G1 is not needed.
- *Procure:* Consolidates ProcureStart, ProcureCommit, ProcureAbort, CleanUp1, CleanUp2 and CleanUp3 in one wiring. G1–G3 correspond to G1–G3 of the original ProcureStart wiring. Instead of sending order entries to the shops it calls the passive Order wiring of each shop as a sub-FWTX (G4–G5). This models the distributed transaction with compensation.

Shop Peer (with FWTX):

- *Order:* Consolidates OrderOk and OrderNotOk. It is a passive wiring that is called by the Procure wiring of the bakery (G4–G5). It has a compensate action that cancels the reservation and an on-top-commit action that delivers the reserved ingredients if the top-level-FWTX commits.
- *Cancel:* Passive wiring called as compensate action of the Order wiring.
- *DeliverItems:* Passive wiring called by Order upon top-level commit.

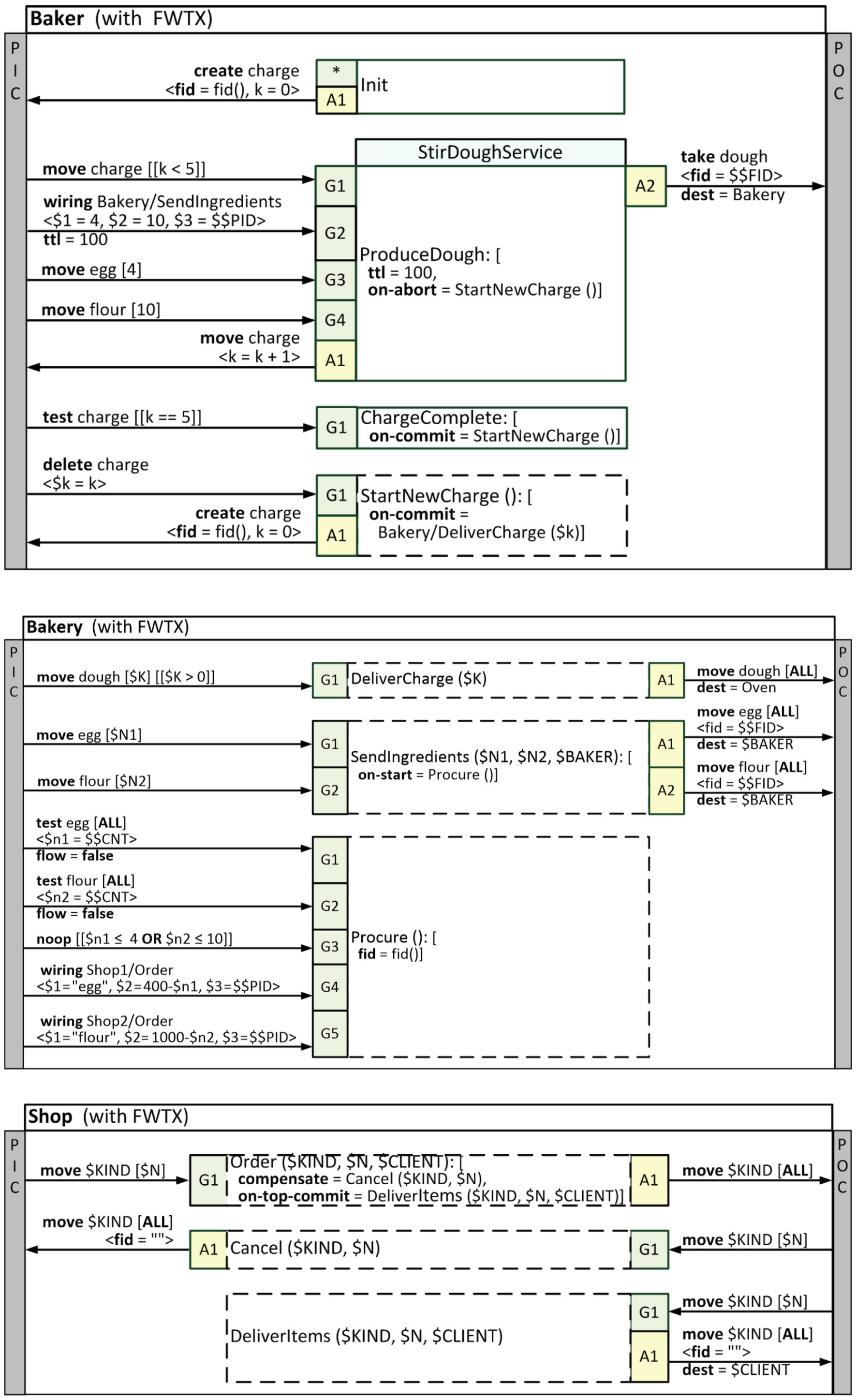

Fig. 4. Baker, Bakery and Shop (with FWTX).

5.2 Related Coordination Models

A realization of the bakery example with the Actor Model is quite straight forward, but mixes application and coordination logic. On the other side, models like Petri Nets and Reo [18] are very general and therefore powerful enough to also model complex coordination scenarios, however, such designs will become complex and exhibit deficiencies and/or become verbose and unreadable (compare with [7,8] who demonstrated this fact with even less demanding coordination and collaboration problems like split/join and leader election without the assumption of failures etc.). The problem is that "lack of appropriate modeling primitives has often resulted in descriptions with either reduced concurrency or increased complexity of the net structure and/or the net inscriptions" [19].

The Transactor Model [20] follows a similar goal like our approach, i.e. to provide language constructs that ease the management of distributed states. It introduces the following concepts: stabilize, checkpoint, dependent test, and rollback. With stabilize an actor guarantees that its state will not change any more (it refers to the prepared phase in a two-phase-commit protocol). A checkpoint is successful if the transactor is not dependent on any other actor that is in a volatile state. Otherwise it will either perform a rollback or is equivalent to a noop (if there have not yet been enough messages received to determine the dependency). A successful checkpoint stores the state of the actor so that a rollback to this state is possible. The dependent test checks whether the actor is dependent on another one. As the entire protocol is asynchronous, this test does not block and therefore the user must take care of this situation explicitly.

A major difference of FWTX is that they support multiple concurrently running flows and automatic execution of user defined actions at certain points in time, namely on-top-commit, on-commit and on-abort of transactions. In addition, the isolation property is relaxed and sub-transactions may early commit. Semantic compensation is used in contrast to the Transactor Model that carries out a rollback. The advantage of compensation is that distributed processes stay autonomous and need not hold locked states over a long period of time.

6 Conclusion

Coordination requirements are challenging and lack of adequate modeling primitives leads to unusable models. We presented an extension of the Peer Model by distributed "flexible wiring transactions (FWTX)" to make wirings more powerful. FWTX enable the control of complex distributed interactions in a very flexible way. The new modeling concepts are on-commit, on-top-commit, on-abort and compensation actions that are designed as passive wirings. With help of FWTX also coordination situations where complex dependencies between concurrent distributed interactions take place or where multi-direction interactions are demanded, can be modeled straight ahead. The treatment of failure situations is easy because the distributed transaction management automatically coordinates the activation of the on-commit, on-top-commit, on-abort and compensation actions. As evaluation, a proof-of-concept is given that shows the design of the selected coordination scenario whereby the separation of application and coordination data and logic could be preserved. The model with FWTX is significantly leaner: the total number of wirings could be reduced by 41% and the number of links by 58%. We believe that also other coordination models can benefit from the introduction of a Flex transaction based coordination mechanism. In future work we will use FWTX to bootstrap other distributed transaction models and implement a simulation tool for automatic analysis.

Acknowledgment. Many thanks to Stefan Craß, Geri Joskowicz, Martin Planer, Matthias Schwayer, Jörg Schoba, Peter Tilian, and the anonymous reviewers for their comments on this paper.

References

1. Astley, M., Sturman, D.C., Agha, G.A.: Customizable middleware for modular distributed software. Commun. ACM **44**(5), 99–107 (2001)
2. Kühn, E.: Reusable coordination components: reliable development of cooperative information systems. Int. J. Coop. Inf. Syst. **25**(4), 1740001 (2016). World Scientific Publishing Company
3. Malone, T.W., Crowston, K.: The interdisciplinary study of coordination. ACM Comput. Surv. (CSUR) **26**(1), 87–119 (1994)
4. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Technische Hochschule Darmstadt (1962)
5. Agha, G.A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1990)
6. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004). Cambridge University Press
7. Börger, E.: Modeling distributed algorithms by abstract state machines compared to petri nets. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 3–34. Springer, Cham (2016). doi:10.1007/978-3-319-33600-8_1
8. 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). doi:10.1007/978-3-642-38493-6_9
9. Bukhres, O., Elmagarmid, A.K., Kühn, E.: Implementation of the flex transaction model. IEEE Data Eng. Bull. **16**(2), 28–32 (1993)
10. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. (TOPLAS) **7**(1), 80–112 (1985)
11. 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 International Conference on Autonomous Agents and Multiagent Systems (AAMAS), IFAAMAS, pp. 625–632 (2009)
12. Craß, S., Kühn, E., Salzer, G.: Algebraic foundation of a data model for an extensible space-based collaboration crotocol. In: International Database Engineering and Applications Symposium (IDEAS), pp. 301–306. ACM (2009)
13. Elmagarmid, A.K.: Database Transaction Models for Advanced Applications. Morgan Kaufmann, San Francisco (1992)

14. Kühn, E.: Fault-tolerance for communicating multidatabase transactions. In: 27th Annual Hawaii International Conference on System Sciences (HICSS), pp. 323–332. IEEE (1994)
15. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Record **16**(3), December 1987
16. Moss, E.B.: Nested Transactions: An Approach to Reliable Distributed Computing. Technical report, Cambridge, MA, USA (1981)
17. Kühn, E.: Virtual Shared Memory for Distributed Architecture. Nova Science Publishers, New York (2001)
18. Meng, S., Arbab, F.: A model for web service coordination in long-running transactions. In: Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE), pp. 121–128 (2010)
19. Christensen, S., Hansen, N.D.: Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 186–205. Springer, Heidelberg (1993). doi:10.1007/3-540-56863-8_47
20. Field, J., Varela, C.A.: Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. In: 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 195–208 (2005)