Extending Renew's Algorithms for Distributed Simulation

Michael Simon and Daniel Moldt

Theoretical Foundations of Computer Science (TGI)
Department of Informatics, University of Hamburg, Germany
http://www.informatik.uni-hamburg.de/TGI/

Abstract Distributed Petri net simulations are still a challenge. A speed up of simulation can be reached, if large applications are distributed over several computers executing a common simulation. Besides technical implementation hinderances, concepts to implement the inherent simulation algorithms in a distributed manner are often missing. Former Renew simulations have been coupled via asynchronous mechanisms like RMI and by general communication frameworks, e.g. for multi-agent applications.
In this contribution, we present a simple prototype that supports the atomic synchronous firing of an arbitrary number of transition instances in Renew simulations, distributed over several computers. This is easily modeled with the help of a bidirectional *send channel* that can traverse simulation boundaries over a network. This channel type is very similar to the synchronous channel of the original Java reference net formalism. However, the unification algorithm is not distributed and cannot be used across send channel calls. The distribution of each algorithm in Renew's simulation cycle is discussed. The usage of our proposed concepts and tools for distributed simulations of this extended Java reference net formalism are discussed as well.

Keywords: Petri Nets, Distributed Simulation, Reference Nets, Synchronous Channels, Renew

1 Introduction

Scaling up the simulation of large applications requires the usage of distributed computing facilities. Regarding Petri nets, several attempts to implement distributed simulations have been made (see e.g. [4,6,8,11,12,13,15,21]). The former contributions concentrate either on modeling the distribution on the application level in combination with an underlying communication framework, or on a simple mechanism that transfers tokens (including time / timing information / colors etc.) in an unidirectional fashion. The goal of this contribution is to improve upon this by providing a bidirectional information exchange for an arbitrary number of synchronized transitions within a single binding. Our motivation originates from the practice of the Paose approach (*Petri net-based,* *Agent- and Organization-oriented Software Engineering*, see [4]): using our Petri net tool RENEW, we directly execute Petri nets together with Java code in distributed machines. The approach aims at a Petri net based system development methodology, which is enhanced by established software engineering concepts. Up to now, we had to handle the distribution on the application level (by using concepts of multi-agent systems; see [4] for more details), since there is no native integration between simulations on multiple machines. Synchronous communication approaches of the entities at the modeling level (agents, objects or nets) have been presented in [9], however, without the consideration of distributed simulations.

Synchronous channels that provide the powerful synchronization mechanism for our (Java) reference nets in the RENEW tool set (*The Reference Net Workshop*, see [16]), are realized by quite elaborated simulation algorithms. The RENEW algorithms cannot be executed across computers efficiently in general, due to the inherent data structures and algorithm design. Several attempts have been made to overcome this limitation. However, a general solution based on the current design seems impossible without changing the modeling power / expressiveness, since a distributed unification algorithm would generate a substantial communication overhead. Conducting this communication directly over a network connection leads to a slow down that conflicts with the goal of a speed up of the simulation. Furthermore, directly distributed implementations are highly complex and hindered by the limitations of Java and its Remote Method Invocation (RMI) system. For more details see Section 3.5.

In this contribution we address how RENEW can be provided with the possibility of distributed simulations by using a restricted kind of synchronous channels on the modeling level. First of all, we restrict the modeling power of the formalism that is used across system boundaries to reduce the number of messages that need to be exchanged. Then, we provide some extensions of the existing algorithms for the newly introduced modeling constructs. In order to demonstrate the applicability of the proposed modifications of the algorithms, a new RENEW plugin is presented. An extended description of the problem and the predecessor of the solution presented here, can be found in [20]. To provide a better insight into the internal structures of RENEW, we explain its undistributed simulation in Section 2. Multiple modeling constructs for the distributed simulation, the implemented approach and the justification for our choice of restrictions on the general synchronization concept are presented in Section 3. Our central contribution – the extension of the algorithms to atomically fire an arbitrary number of transition instances in multiple simulations via channels with bidirectional information exchange – is explained in Section 4. We present some simple examples in Section 5 to illustrate the different possibilities. We conclude with the main results and their possible further extensions.

2 Undistributed Simulations in Renew

This section provides an overview of Renew's undistributed simulation algorithms as a basis for the description of their distribution in Section 4. The algorithms are described in detail in [14, Chapter 14] (in German). A shorter English description is given in [20, Chapter 3]. The suitability for distribution was not taken into account when the algorithms were originally conceptualized and implemented. Remote observation of the simulation state was later added with the Remote plugin, but it is not designed for the communication between simulations and offers no possibility of synchronously firing transition instances across simulation boundaries.

2.1 Renew and Its Synchronous Channels

Renew is a simulator for Java reference nets (and other Petri net formalisms) (see [14,5]). Java reference nets are colored Petri nets, that may be inscribed with programming code. The inscription language is based on Java and can call methods of Java objects. Tokens are nested *net instances* or arbitrary Java objects. Transition instances in multiple net instances can be combined with *synchronous channels.* They may exchange information with channel parameters and in the end fire synchronously in an atomic event. The syntax for channels is `ni:ch(x,y)` for the downlink and `:ch(w,u)` for the uplink. `ni` is a variable that will get bound to a net instance, in which a transition is inscribed with the corresponding channel name `ch` and its parameters `x,y`. Both match the uplink channel `:ch` with its parameters `w,u`.

Renew follows an object-oriented design. In a simulation multiple *instances* of a single *net template* can exist.[1] Renew's Java reference net formalism allows net instances to be created and nested as tokens in other net instances. Thus, every transition/place of a net template has a corresponding *transition/place instance* in each of its net instances.

Figure 1 illustrates two simple channels `s()` and `t()`. Transition **t1** can always fire, since it does not have an input place. All other transitions have synchronous channels attached. Transitions **t2** and **t3** have a *downlink*, while **t4**, **t5**, **t6** and **t7** have an *uplink*. **t4** and **t6** can only fire together with **t2**. For this reason, they are in conflict with each other. The same holds for **t5** and **t7**, that require **t3** to fire. It is possible for transitions to have several downlinks to the same uplink (like **t3**). When **t3** is fired, it synchronizes either with **t5** and **t7**, **t5** and **t5**, or **t7** and **t7**. All three transitions are atomically fired in a single step. It is worthwhile to note, that the assignment of uplinks to downlinks is nondeterministic. For synchronous channels, the uplink and the downlink are always

[1] The relationship between a *net template* (often simply called *net*) and its *net instances* in the reference net formalism is analogous to the relationship between a *class* and its *class instances* in object-oriented programming languages. In the same way, the elements of a net correspond to the internals of a class, e.g. variables. The instances of variables of an object correspond, e.g., to places in net instances.

in the same simulation. This is also the case, if another net instance is used instead of `this`. Our goal is to have a channel similar to the synchronous channel, that may address transition instances in remote simulations. For a distributed example employing our *send channel*, see section 5.

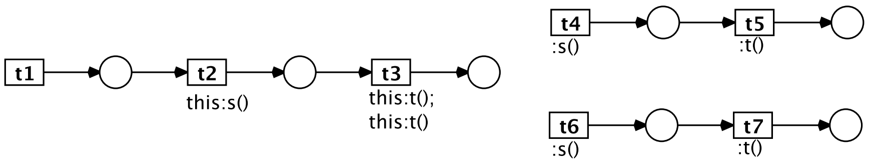

Figure 1. Simple synchronous channels.

The simulation can be split up into three algorithms: (a) the *search algorithm* that employs a depth-first search to find a valid binding for a transition instance. (b) the *firing algorithm* that fires the found binding and (c) the *triggering algorithm* that selects the next transition instances to be searched and inserts them into a search queue. This simulation cycle can be directly employed to facilitate a simulation, but Renew also allows for more concurrency. Once it is ensured that a firing will be successful, it can be detached and finished while the cycle continues concurrently. Furthermore, multiple instances of this cycle can be run concurrently, using a single search queue for triggered transitions.

Renew employs a unification algorithm underneath the search algorithm. Thus, the Java reference net inscription language can be seen as a logic programming language. This makes it very expressive and powerful. It is especially useful for synchronous channels that unify the parameters of both sides in a mutual manner. In this way information from both sides can be composed and matched to find the final binding. However, the unification algorithm was not distributed. Therefore, it is not considered in detail or modified for this paper, and the newly introduced channel constructs – the *send channels* – do not offer the same freedom of information flow as synchronous channels.

2.2 The Triggering Algorithm

In the binding search for a transition instance, Renew keeps track of the place instances it has already visited. If no binding can be found, the transition instance is not activated. However, in most cases not all place instances in the transition instance's preset have to be visited before the search fails. For example, if multiple place instances are empty, the search can fail after examining only one of them.[2] Before the transition instance is activated again, the searched place instances must have received at least one new token. Otherwise, the search

[2] Arcs that offer fewer choices of tokens to bind are examined first. This is integral to the design of the search algorithm described in Subsection 2.3.

is bound to fail again. The triggering algorithm maintains the sets of transition instances that the place instances have to trigger. Once a transition instance is (re)inserted into the search queue, either by being triggered or after a successful firing, it can be excluded from all of these sets. In the following search it will again be inserted into the visited place instances. Because these might not be the same place instances that were visited in the previous search, this reduces the number of triggering place instances, and thus the number of searches that are bound to fail.

The concept of single *representing places* that are sufficient to indicate that a transition is activated, originates from [7]. The concept of *triggering places* in RENEW is mainly inspired by [22]. The idea of using a *search queue* originates from [10]. For more information on the background of the development of the triggering algorithm see [14, Subsection 14.4.1].

As described above, it must be possible to trigger all transition instances in the set of a single place instance as well as to exclude a single transition instance from all of these sets. Thus, not only must the transition instances to be triggered be stored as a set at the place instance, but in reverse each transition instance must know all sets that include it, so it may exclude itself later. This is handled by the classes/interfaces shown in Figure 2.

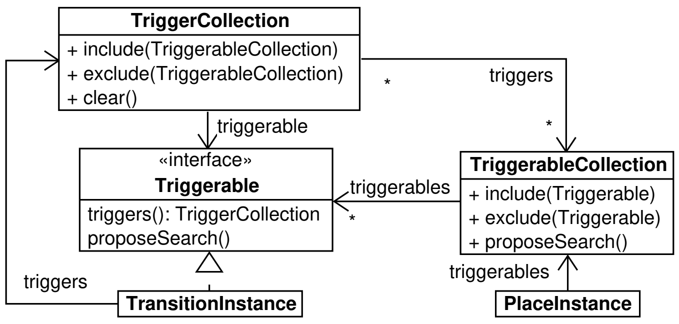

Figure 2. Class diagram of the triggering algorithm. (From [14, Figure 14.17].)

After a place instance changed, the `proposeSearch()` method of its `TriggerableCollection` is propagated to all transition instances in its `triggerables` set, and they insert themselves into the search queue.

When a transition instance inserts itself into the search queue, it is removed from all triggering place instances. This is done by the `clear()` method of its `TriggerCollection`, which calls both `exclude(...)` methods. The transition instance is excluded from all `TriggerableCollection` instances in `TriggerCollection.triggers` and `TriggerCollection.triggers` is emptied.

In the following search, the transition instance may again be registered at some place instances by being inserted into their `TriggerableCollection` instances. These TriggerableCollection instances are in turn inserted into the transition instance's TriggerCollection. This is done by the include(...) methods of both classes.

2.3 The Search Algorithm

Figure 3 shows the three most important classes/interfaces of the binding search algorithm. A depth first search is facilitated recursively by nested calls of the Searcher.search() and Binder.bind(Searcher) methods. Implementations of the Binder interface make choices, and thus narrow the search before calling Searcher.search(). Most importantly, an ingoing arc tries to bind one token at a time and a channel downlink tries to bind one uplink at a time. An uplink transition instance is added to the search by adding new binders, representing it. After each decision, the searcher tries another binder[3] by calling its bind(...) method until the search can no longer be narrowed. In that case, either a valid binding is found, or the search has to backtrack by ending recursive method calls until another choice can be tried in a binder. If a valid binding is found, the Finder.found(Searcher) method is called and decides what to do.[4] In the simulation cycle the finder stores the binding, so the firing algorithm can fire it in the next step.

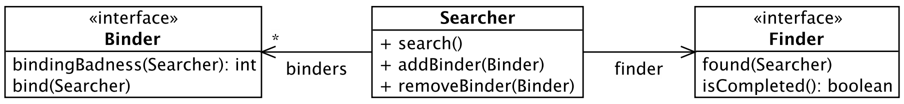

Figure 3. Class diagram of the basic classes of the search algorithm and their most important methods. (Based on [14, Figure 14.25].)

2.4 The Firing Algorithm

After a binding is found by the search algorithm, it is fired by the binding algorithm. Renew fires bindings atomically: either all transition instances occurring in a binding are fired synchronously, or none at all. Renew makes sure that the simulation behaves exactly as if the firing would have occurred at a single point in time. A firing is split up into multiple *executables* that represent different actions, that need to be taken in the firing. The exact set of executables is determined in the binding search by the explored transition inscriptions and arcs.

[3] The searcher always tries a binder with the minimal bindingBadness(...). This estimates its cost by the number of choices that it would try.

[4] Finder.isComplete() signals the searcher to end the search, if it returns true.

The executables are further split up into *early executables* and *late executables*. Early executables can be reverted and late executables cannot. Thus, to archive an atomic firing, its success must already be determined before the late executables are applied. The most important *early executable* implementation represents ingoing arcs. The associated action takes one token from a place. The most important *late executable* implementation represents outgoing arcs and puts out one token. Putting out a token always succeeds, but taking a token may fail, if another conflicting firing already took the token. The search algorithm is optimistic and assumes that explored tokens will still be present in the firing. The firing algorithm has to ensure the detection of missing tokens, revert all changes and let the firing fail. It locks place instances to prevent concurrent conflicting firings.

The individual steps taken to fire a binding are shown in Table 1. After the binding search (step 0), multiple steps are taken to execute an early executable. In step 1, ingoing arcs *lock* their place instances. To resolve conflicts between concurrent firings, the place instances are locked in a total order. This order is given by a unique ID assigned to each place instance. Following this rule, exactly one of multiple conflicting firings is executed at a time, while the others wait. Thus, the conflicts are resolved and deadlocks are prevented. This strategy is based on [21].

0	Binding search (before firing)
1	Lock *early executables* in locking order
2	Validate possibility of applying *early executables*
3	Execute *early executables*
4	Unlock *early executables*
5	Report success
6	Execute *late executables*

Table 1. Order of steps when successfully firing a binding in Renew. (Based on [3, Section 3].)

In step 2, the early executables *validate* that they can be executed. This is necessary, because the simulation state could have changed since the binding search. Actions that must be reversible are also taken in this step. For ingoing arcs the required token is removed, if it is still present. Because the place instance is not yet unlocked, this does not (yet) have an impact on the rest of the simulation, and can thus be easily reverted by putting the token back. If the required token is not present, the validation step fails. All already validated early executables are *reverted* and the firing *fails*.

If the validation step was successful, the early executables are *executed* in step 3. At this point, the success of the firing is already determined. In this step, actions that cannot be reverted are taken. Ingoing arcs report the consumption of their token to the outside world.

In step 4, the early executables are *unlocked* (ingoing arcs unlock their place instances) and in step 5 the success of the firing is *reported* to the outside world. Finally, the late executables are *executed* in step 6. For outgoing arcs, this puts out the token.

3 Different Modeling Constructs for Synchronous Channels

This section presents different modeling constructs for distributed channels. They require different kinds of semantics for their implementation. The more powerful a modeling construct is, the more general the implementation has to be designed. We have experimented with several versions to test the feasibility of different constructs and the implementation of their intended semantics. They differed in expressiveness, complexity of the implementation, practical usefulness, the need for additional coordination structures and in the necessary technical support.

3.1 Integration Into the Simulation Algorithms

One of the most simple distribution approaches involves directly taking tokens from place instances, passing them to another simulation and putting them into a place instance there. This concept is very similar to virtual places (there are several instances on the drawing / visualization level, but only one is relevant for simulation). The integration into the simulation algorithms is minimal: the extraction place instance has to be locked to prevent conflicts with concurrent binding firings. All the rest has to be implemented by the distribution algorithm, but can be kept very minimal. The most simple methods of addressing the target place can be chosen. (See Subsection 3.2.)

Within Renew a tighter integration can be achieved by using the firing algorithm (Subsection 2.4), but not the search algorithm (Subsection 2.3). A binding could be constructed by a simpler algorithm and then be fired. This is most interesting in the source simulation, where the tokens are extracted, as it offers an easy way to implement the extraction from multiple places without conflicts. (See Subsection 3.3.)

For an even tighter integration, the search algorithm could also be employed. It offers all the flexibility of the Java reference net formalism, but may be restricted to limit the possible conflicts between found bindings, or simplify the implementation. See Subsection 3.4 for a discussion of the integration of a special kind of channel.

3.2 Addressing the Target

If tokens are to be send to a place instance in a remote simulation, there has to be a method of assigning the target place instance. The address of the target must be provided by the modeler of the source net to tell the algorithms in which place instance to insert the tokens. Analogously, if a remote transition instance should be added to a firing, it has to be addressed.

There are different possible methods of how to address a target place instance or transition instance. These differ in the flexibility they give to the modeler of the net: a source and a target place instance could be designated externally without being reflected in the net itself. In this case, the target is not given in the net, but must be decided elsewhere. E.g. it might be set as a parameter, when the simulation is started. Alternatively, place/transition instances could be statically addressed in the net by a simple inscription. Thus, the address may never change and is independent from the simulation state. Finally, the addressing could be done dynamically inside a binding search. One solution is the introduction of a special channel that can be bound on tokens, that represent net instances in a remote simulation. It is the most elegant and intuitive solution, because it is analogous to the way different net instances interact in a single simulation. This is further discussed in Subsection 3.4.

3.3 Synchronization of the Taking of Tokens

Based on the details of the distribution, conflict resolution / deadlock prevention strategies may need to be applied. Furthermore, the distribution may or may not guarantee an atomic execution depending on whether the movement of tokens occurs at a single point in time from the viewpoint of concurrent binding firings. If the distributed implementation fires multiple transition instances, the atomicity also guarantees that all are either successful, or fail.

3.4 Information Flow of Channels

If the distribution is facilitated in the net by a channel integrated into the binding search, it might be sufficient to require that its parameters are fully evaluated before the search crosses the simulation boundaries. This kind of channel is unidirectional: information can only flow from the calling net instance *down* to the called net instance. A bidirectional information flow could be archived by adding the possibility of a return value, analogous to a function/method call in most programming languages.

To offer even more flexibility, Renew's unification algorithm could be distributed. This way, it could have the same semantics as *synchronous channels* and information from both sides could aggregate freely and narrow the binding.

3.5 The Implemented Approach

For the distribution of Renew's simulations, a very tight integration in the existing algorithms was chosen. All three algorithms in the simulation cycle (Section 2) were distributed. In the search algorithm, transition instances in multiple Renew simulations are combined by a new kind of channel. In the firing algorithm they are fired as an atomic action without the possibility of deadlocks to occur. A return value offers bidirectional information exchange.

However, the unification was not distributed, and thus all values have to be fully unified before being sent to another simulation. Distribution of the unification algorithm was not possible, since the inherent unification algorithm has a strict dependence on the identity and comparison of every kind of Java object. Java only offers the possibility to compare certain kinds of objects in a distributed fashion.[5] Especially in complex binding scenarios, the number of comparisons by the unification algorithm can also become very high and amount to a significant overhead.

This contribution provides constructs for modeling and executing distributed synchronous channels that allow: (a) bidirectional exchange of parameters, however, all values are fully unified, before they are send to the remote simulation, (b) the number of involved transition instances to be arbitrarily large, (c) one transition to have an arbitrary number of downlinks, but (as usual in Renew) only one uplink, (d) the use of simple generally available technology and (e) a reasonable performance of the implementation.

4 Discussion of the Algorithm Extensions

This section discusses how each of the algorithms from Section 2 is distributed.

All communication between Renew processes is facilitated by the Java RMI system (see [23]). Interfaces that extend `java.rmi.Remote` are labeled as *remote interface* in class diagrams. All their implementing classes are automatically made available by RMI.

4.1 Distributing the Triggering Algorithm

The classes/interfaces of the triggering algorithm for transition instances and place instances inside a single Renew simulation are shown in blue in Figure 4. The only difference to those shown in Figure 2 is the separation of the classes into an interface and an implementing class. The fields are inscribed and not drawn as arcs to reduce the complexity of the diagram. The classes and interfaces in black augment those to distribute the triggering algorithm. Figure 4 was designed with a specific scenario in mind. In this scenario, one Renew simulation started a search (the *parent search*) and a channel of a net instance in a second simulation was bound (by spawning a local *child search*). The classes are arranged on the left or the right side, depending on where their instances are used.

The transition instance from the parent search is represented by a `TriggerableProxy` instance in the child search. The `TriggerableProxy` instance is added to the `triggerables` field of the place instance. (The `triggerables` field contains a `TriggerableCollectionImpl` instance.) Next, the `triggers` method of the `TriggerableProxy` instance is called to receive a `TriggerCollectionProxy` instance that

[5] In the general case, only *serializable* objects can be directly used for a distributed comparison in Java. The comparison of RMI *remote* objects must be explicitly implemented in further wrapping classes.

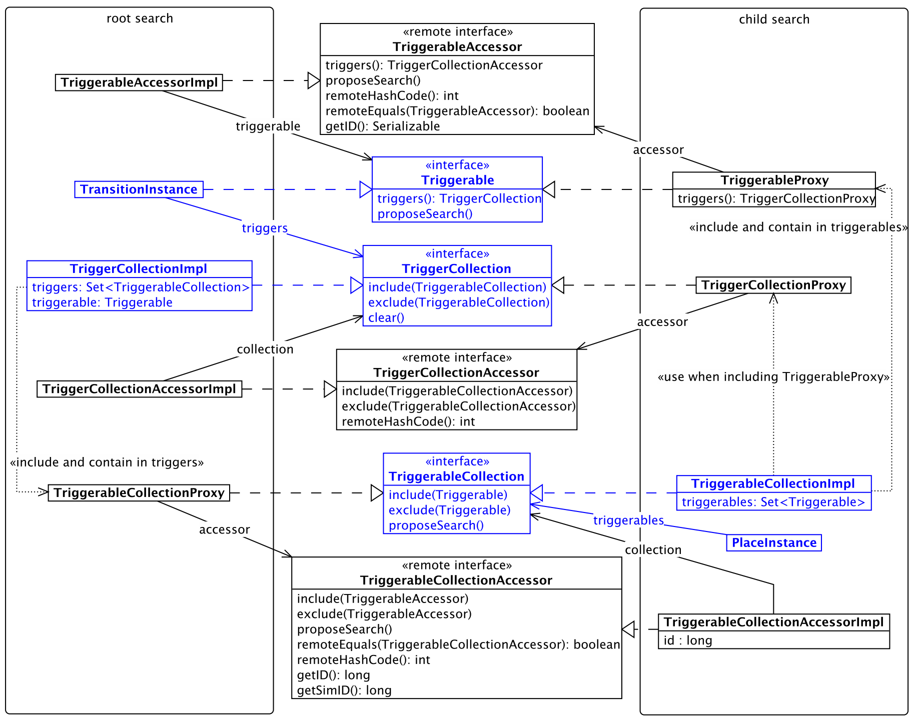
root search
child search
«remote interface»
TriggerableAccessor
triggers(): TriggerCollectionAccessor
proposeSearch()
remoteHashCode(): int
remoteEquals(TriggerableAccessor): boolean
getID(): Serializable
TriggerableAccessorImpl
triggerable
accessor
«interface»
Triggerable
triggers(): TriggerCollection
proposeSearch()
TransitionInstance
TriggerableProxy
triggers(): TriggerCollectionProxy
triggers
«include and contain in triggerables»
TriggerCollectionImpl
triggers: Set<TriggerableCollection>
triggerable: Triggerable
«interface»
TriggerCollection
include(TriggerableCollection)
exclude(TriggerableCollection)
clear()
TriggerCollectionProxy
accessor
collection
«remote interface»
TriggerCollectionAccessor
include(TriggerableCollectionAccessor)
exclude(TriggerableCollectionAccessor)
remoteHashCode(): int
TriggerCollectionAccessorImpl
«use when including TriggerableProxy»
«include and contain in triggers»
TriggerableCollectionProxy
«interface»
TriggerableCollection
include(Triggerable)
exclude(Triggerable)
proposeSearch()
TriggerableCollectionImpl
triggerables: Set<Triggerable>
triggerables
PlaceInstance
accessor
collection
«remote interface»
TriggerableCollectionAccessor
include(TriggerableAccessor)
exclude(TriggerableAccessor)
proposeSearch()
remoteEquals(TriggerableCollectionAccessor): boolean
remoteHashCode(): int
getID(): long
getSimID(): long
TriggerableCollectionAccessorImpl
id : long

Figure 4. Class diagram on the extension of the triggering algorithm with some additional explanations. (Compare with Figure 2.)

represents the `triggers` field of the transition instance. (The `triggers` field contains a `TriggerCollectionImpl` instance.) This `TriggerCollectionProxy` instance is used to add the place instance's `triggerables` field to the transition instance's `triggers` field. The `triggers` field is represented by a `TriggerableCollectionProxy` instance in the parent search.

Each proxy object uses a corresponding accessor remote object on the other side. Additional information, including hash codes and internal IDs, is transmitted by the remote objects, where needed, to recognize proxies of the same original as equal. The distribution of the triggering algorithm is completely hidden from the original triggering algorithm code, which was not changed at all.

4.2 Distributing the Search Algorithm

Figure 5 shows the most important classes/interfaces of the distributed search algorithm.[6] The classes/interfaces of the undistributed binding search, presented in Subsection 2.3, are colored blue. The implementation uses all three to tie into

[6] The distribution of the search algorithm changed fundamentally since the description in [20, Section 4.6].

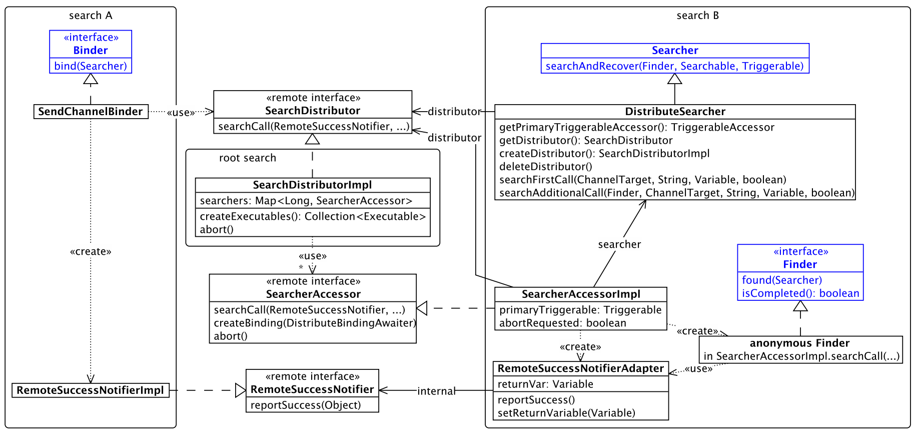
search A
«interface»
Binder
bind(Searcher)
SendChannelBinder
«use»
«create»
RemoteSuccessNotifierImpl
«remote interface»
SearchDistributor
searchCall(RemoteSuccessNotifier, ...)
distributor
distributor
root search
SearchDistributorImpl
searchers: Map<Long, SearcherAccessor>
createExecutables(): Collection<Executable>
abort()
«use»
*
«remote interface»
SearcherAccessor
searchCall(RemoteSuccessNotifier, ...)
createBinding(DistributeBindingAwaiter)
abort()
«remote interface»
RemoteSuccessNotifier
reportSuccess(Object)
internal
search B
Searcher
searchAndRecover(Finder, Searchable, Triggerable)
DistributeSearcher
getPrimaryTriggerableAccessor(): TriggerableAccessor
getDistributor(): SearchDistributor
createDistributor(): SearchDistributorImpl
deleteDistributor()
searchFirstCall(ChannelTarget, String, Variable, boolean)
searchAdditionalCall(Finder, ChannelTarget, String, Variable, boolean)
searcher
«interface»
Finder
found(Searcher)
isCompleted(): boolean
SearcherAccessorImpl
primaryTriggerable: Triggerable
abortRequested: boolean
«create»
«create»
anonymous Finder
in SearcherAccessorImpl.searchCall(...)
RemoteSuccessNotifierAdapter
returnVar: Variable
reportSuccess()
setReturnVariable(Variable)
«use»

Figure 5. Class diagram on the distribution of the search algorithm. (Compare with Figure 3.)

and extend every aspect of the binding search. The interaction of the classes is shown in Figure 6. Both diagrams are separated into the classes/interfaces that handle one of three local searches, that make up the distributed search. The *root search* is the local search that was first started at the original transition instance. *search A* is the local search, where we observe the binding of a send channel downlink and *search B* is the local search of the corresponding uplink. Either the *root search* and *search A* are the same, or there has been at least one other send channel binding before. However, *search B* is always in a different RENEW simulation than *search A*, because the shown classes only handle the distributed case.

A send channel is bound by the `SendChannelBinder.bind(...)` method. It tries to bind it to an uplink transition instance that typically resides in another RENEW simulation and is addressed by a `DistributeNetInstance`[7] and a channel name. First, it is ensured that a `SearchDistributor` exists. Each distributed search has exactly one such object. It keeps track of all the local searches that are involved by maintaining a map from the simulation IDs to the `SearcherAccessor` objects, that may reside in other RENEW simulations. If needed, a new `SearchDistributorImpl` is created with a `SearcherAccessorImpl` for the local search.

The `SearchDistributorImpl.searchCall(...)` method in the *root search* is called to propagate the request to search a suitable uplink. The distributor makes sure a `SearcherAccessorImpl` for the RENEW simulation with the uplink exists. Then, it propagates the request to *search B* by calling `SearcherAccessorImpl.searchCall(...)`. If *search B* is not currently involved in the distributed search

[7] A `DistributeNetInstance` object is a proxy for a net instance, that may reside in another RENEW simulation. The example in Section 5 shows how `DistributeNetInstance` objects may be exchanged between simulations.

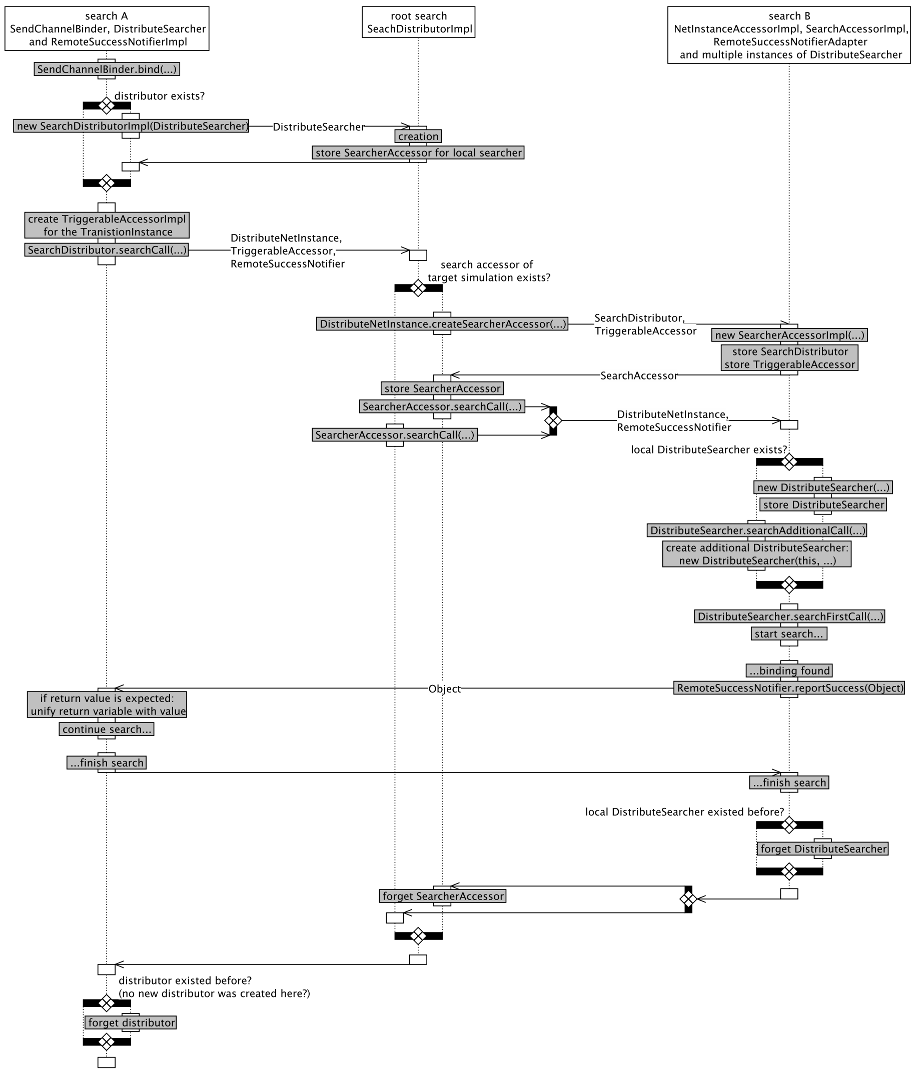
search A
SendChannelBinder, DistributeSearcher and RemoteSuccessNotifierImpl
root search
SeachDistributorImpl
search B
NetInstanceAccessorImpl, SearchAccessorImpl, RemoteSuccessNotifierAdapter and multiple instances of DistributeSearcher
SendChannelBinder.bind(...)
distributor exists?
new SearchDistributorImpl(DistributeSearcher)
DistributeSearcher
creation
store SearcherAccessor for local searcher
create TriggerableAccessorImpl for the TranistionInstance
SearchDistributor.searchCall(...)
DistributeNetInstance, TriggerableAccessor, RemoteSuccessNotifier
search accessor of target simulation exists?
DistributeNetInstance.createSearcherAccessor(...)
SearchDistributor, TriggerableAccessor
new SearcherAccessorImpl(...)
store SearchDistributor
store TriggerableAccessor
SearchAccessor
store SearcherAccessor
SearcherAccessor.searchCall(...)
SearcherAccessor.searchCall(...)
DistributeNetInstance, RemoteSuccessNotifier
local DistributeSearcher exists?
new DistributeSearcher(...)
store DistributeSearcher
DistributeSearcher.searchAdditionalCall(...)
create additional DistributeSearcher: new DistributeSearcher(this, ...)
DistributeSearcher.searchFirstCall(...)
start search...
...binding found
RemoteSuccessNotifier.reportSuccess(Object)
Object
if return value is expected: unify return variable with value
continue search...
...finish search
...finish search
local DistributeSearcher existed before?
forget DistributeSearcher
forget SearcherAccessor
distributor existed before? (no new distributor was created here?)
forget distributor

Figure 6. Interaction diagram of the creation of remote child searches.

with another uplink to a send channel, it creates a fresh DistributeSearcher. Otherwise, a DistributeSearcher for the previous (outer) send channel binding already exists. In that case, a new DistributeSearcher instance is created that shares part of its internal state with all other DistributeSearcher instances in this Renew simulation (*search B*), that belong to the same distributed search. This way, a single local binding can be found that incorporates all uplinks bound as send channels. It also allows the search at one uplink to respect the state of the search at another uplink in *search B*. Otherwise, it would be possible that a binding is found, that tries to consume a single token by two uplink transitions.

While binding the send channel, a RemoteSuccessNotifier object is passed along. The original RemoteSuccessNotifierImpl is created in the SendChannelBinder.bind(...) method and can be used to report the return value back to *search A*. This is done once a valid binding is found by the *Finder* in SearcherAccessorImpl.searchCall(...). At this point, the return value is guaranteed to be determined. The distributed search then continues normally at *search A*.

If a valid binding is found at the *root search*, a DistributeBinding is created in every local search to prepare for a synchronous firing as described in Subsection 4.3. The distributed search is then ended by unraveling the recursive method call structure across Renew simulation boundaries. Finally, the binding can be fired from the *root search* simulation.

4.3 Distributing the Firing Algorithm

Figure 7 shows the classes of the distributed firing algorithm and Figure 8 shows the coordination of two local firings in a distributed firing. *root firing* denotes the Renew simulation, where the search originally started. This simulation also coordinates the firing of the distributed binding, found in that distributed search. The class DistributeBinding represents a local binding in one Renew simulation. The *root firing* is executed in the DistributeBinding.execute(StepIdentifier, boolean) method by the undistributed firing algorithm, as described in Subsection 2.4. The DistributeBinding of the *root firing* is omitted from both figures, because the coordination with the *child firing* is completely contained in a single RemoteBindingExecutable instance. Thus, the original firing algorithm is employed to coordinate multiple local firings and is in this way transparently distributed.

The RemoteBindingExecutable distributes the conflict resolution strategy of the undistributed algorithm by defining a total order on all place instances in all Renew simulations. It also globally enforces the same order of steps as shown in Table 1. Thus, the same mechanisms used for a local firing are employed for the distributed case. The atomicity is preserved and conflicts are resolved.

In the *locking* step 1 from Table 1, the lock() method of the RemoteBindingExecutable is called. The order in which RemoteBindingExecutable instances are locked, is determined by an unique ID given to each Renew simulation. First, all RemoteBindingExecutable instances with a lower ID than the *root firing* simulation are locked in this order. Then, all other early executables are locked in the order given by the place instance IDs. Finally, the RemoteBindingExecutable instances with a higher ID are locked. By this strategy, all place instances are locked in a global total order. In the lock() method the RemoteBindingExecutable starts the *child firing* by remotely calling BindingAccessor.execute(RemoteBindingSynchronizer), which calls the DistributeBinding.execute(StepIdentifier, RemoteBindingSynchronizer) method. It then calls RemoteBindingSynchronizerImpl.waitUntilLocked() to wait until the *locking*

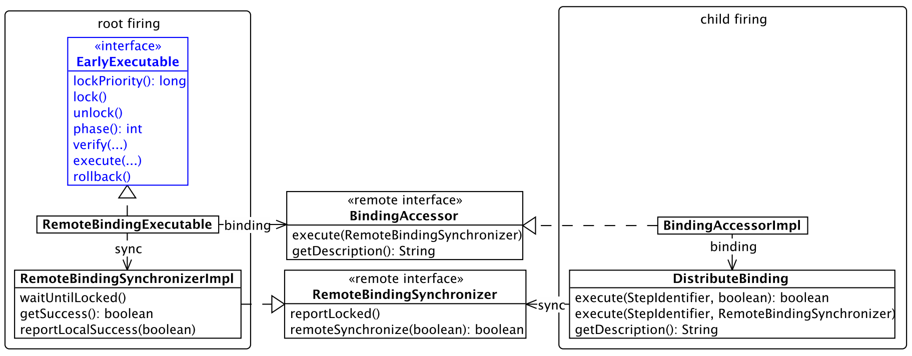

Figure 7. Class diagram on the extension of the binding firing algorithm. (Based on [20, Figure 4.9].)

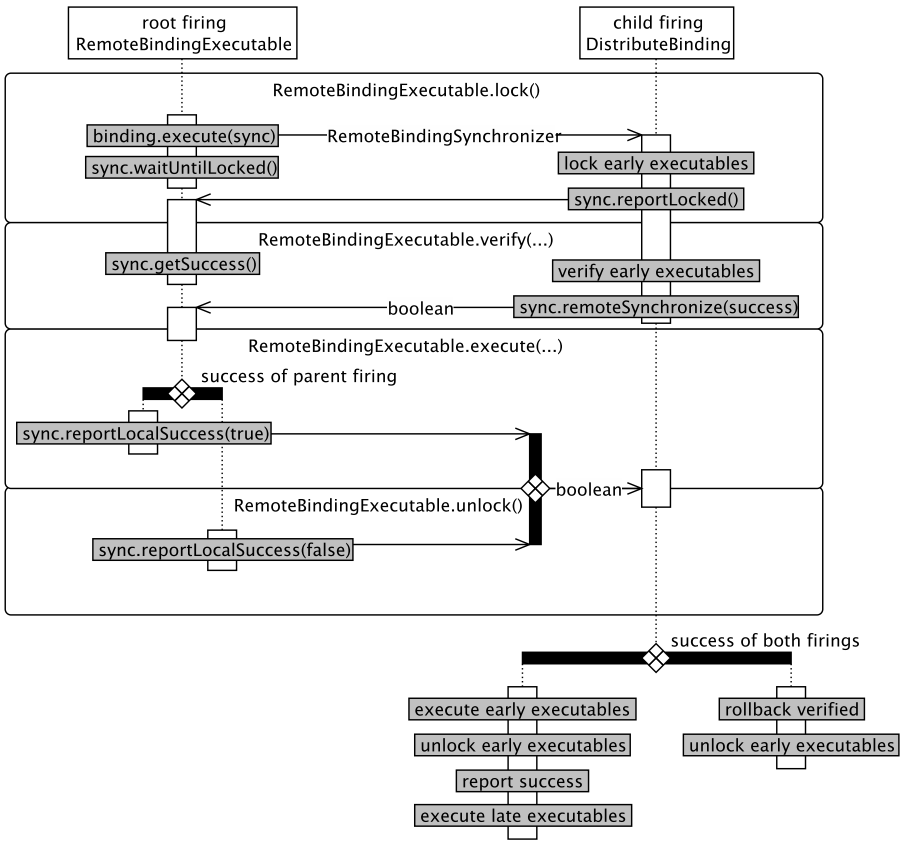

Figure 8. Interaction diagram on the extension of the binding firing algorithm. Interaction of the RemoteBindingExecutable class with the remote DistributeBinding class from Figure 7 in steps 1 to 4 from Table 1.

step (1) is finished in the *child firing*. The *child firing* signals this by remotely calling `RemoteBindingSynchronizer.reportLocked()`.

In the *validation* step (2), the `RemoteBindingExecutable.verify(...)` method waits for the *child firing* to complete the validation step by calling `RemoteBindingSynchronizerImpl.getSuccess()`. Its return value reflects whether the validation step was successful. If it was not, the validation of the `RemoteBindingExecutable` fails, and thus the *root firing* fails as well.

After the validation step, the *child firing* synchronizes itself with the *root firing* by calling `RemoteBindingSynchronizer.remoteSynchronize(boolean)`. This signals the result of the validation step, and thus allows the *root firing* to continue, but also waits for the corresponding result from it. If the *root firing* completed the validation step successfully, it will eventually call `RemoteBindingExecutable.execute(...)` and propagate a positive result. If it was not successful, it will eventually call `RemoteBindingExecutable.unlock()` and propagate a negative result. This concludes the synchronization and the *child firing* now also knows whether the distributed firing succeeded. It can then conclude the firing autonomously and either execute all *executables*, or revert.

The execution of the *late executables* is not synchronized in the current version of the DISTRIBUTE plugin. They are executed in different *phases* that are only enforced locally. Late executables from different simulations cannot have a dependency on each other, because they could only be linked by the unification algorithm, but that is not used across simulation boundaries. This does, however, lead to diverging finishing times for the local firings and may seem odd to the user. Additional synchronization points could be added between all or specific execution phases to fix this, but it is not necessary semantically. Furthermore, when *late executables* throw exceptions, one simulation might abort the further execution of the local firing, while the others are unaffected. This is acceptable, because the occurrence of exceptions is outside of the Java reference net semantics anyway. However, further research might be necessary in the context of exception handling extensions as introduced by [3].

5 An Example of a Distributed Simulation

An example is given to present the capabilities of the DISTRIBUTE plugin[8]. It involves three net instances that may each be simulated on a different computer. Every net instance represents a different role in a sales negotiation: the *manufacturer* in Figure 9 offers items it produced, the *dealer* in Figure 10 offers a bundle of two items sold by the manufacturer and the *buyer* in Figure 11 buys a specific bundle from the dealer under certain conditions.

The simulation of the *manufacturer* has to be started first. In the **register** transition it registers a `DistributeNetInstance` object of itself at the registry. After this step, its **sell** transition can be fired from other simulations by binding its

[8] A special version of RENEW including the DISTRIBUTE plugin and its User Guide (distribute.pdf) can be found at `https://paose.informatik.uni-hamburg.de/paose/wiki/distributed-simulation`

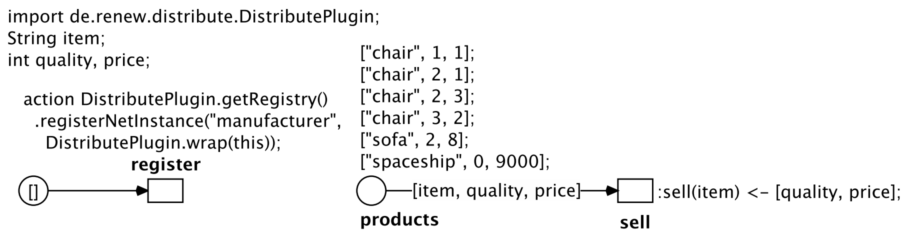
import de.renew.distribute.DistributePlugin;
String item;
int quality, price;
action DistributePlugin.getRegistry()
.registerNetInstance("manufacturer",
DistributePlugin.wrap(this));
register
[]
["chair", 1, 1];
["chair", 2, 1];
["chair", 2, 3];
["chair", 3, 2];
["sofa", 2, 8];
["spaceship", 0, 9000];
[item, quality, price]
products
:sell(item) <- [quality, price];
sell

Figure 9. The *manufacturer* net.

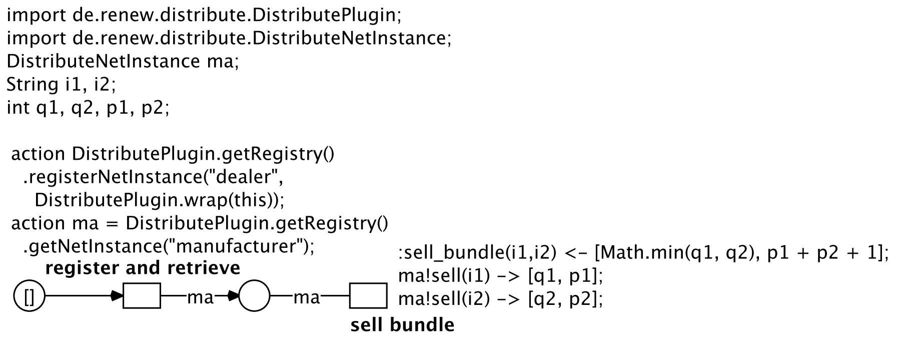
import de.renew.distribute.DistributePlugin;
import de.renew.distribute.DistributeNetInstance;
DistributeNetInstance ma;
String i1, i2;
int q1, q2, p1, p2;
action DistributePlugin.getRegistry()
.registerNetInstance("dealer",
DistributePlugin.wrap(this));
action ma = DistributePlugin.getRegistry()
.getNetInstance("manufacturer");
register and retrieve
[]
ma
ma
:sell_bundle(i1,i2) <- [Math.min(q1, q2), p1 + p2 + 1];
ma!sell(i1) -> [q1, p1];
ma!sell(i2) -> [q2, p2];
sell bundle

Figure 10. The *dealer* net.

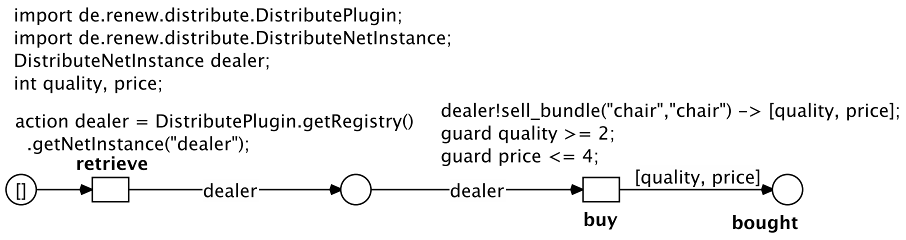
import de.renew.distribute.DistributePlugin;
import de.renew.distribute.DistributeNetInstance;
DistributeNetInstance dealer;
int quality, price;
action dealer = DistributePlugin.getRegistry()
.getNetInstance("dealer");
retrieve
[]
dealer
dealer!sell_bundle("chair","chair") -> [quality, price];
guard quality >= 2;
guard price <= 4;
dealer
[quality, price]
buy
bought

Figure 11. The *buyer* net.

:`sell`(...) uplink with a send channel. It takes the string representation of an item that is offered and returns its quality and price. If it is fired, the corresponding tuple is removed from the **products** place to indicate that it is sold.

Second, the simulation of the *dealer* has to be started. It first retrieves the manufacturer and registers itself. After this step, the **sell bundle** transition can bind the manufacturers :`sell`(...) channel, if it is bound by a send channel. Because the manufacturer's `DistributeNetInstance` is not consumed when it is fired, but only accessed by a test arc, it could fire multiple times. Each time the two items specified by its uplink parameters are bought from the manufacturer, and the quality and price attributes of the items are combined. The quality is the lowest of the two items' qualities and the price is added and increased by one unit.

Last, the *buyer* simulation is started. It retrieves the dealer and binds its `sell_bundle(...)` uplink in an attempt to buy two *chairs*. Guards are applied to the return values to ensure that the quality of the bundle is at least 2 and the price is not greater than 4 units. When the **sell bundle** transition of the dealer is bound, it in turn binds the **sell** transition of the manufacturer twice to buy two chairs.[9] In the binding search every possible combination of quality and price is tried and returned to the **buy** transition, until the guards are satisfied. Once one is found, the distributed binding is fired. As an atomic action, the items are removed in the manufacturer and the tuple of quality and price is added to the **bought** place in the buyer net. The only possible value in **bought** is `[2,4]`. If the *buyer* net simulation is restarted, the **buy** transition cannot fire, because the items are already consumed.

6 Conclusion

The Renew plugin presented in the paper allows the atomic firing of transition instances in multiple reference net simulations across a network. The *send channel* is introduced, which adapts most of the functionality of the *synchronous channel* for distributed net instances, namely bidirectional synchronous communication. However, it does not support the distributed unification of data structures. The extension of the Java reference net formalism is not only conservative, but send channels can be freely mixed with synchronous channels and all other features.

The three central algorithms of Renew's simulation cycle were presented and different distribution approaches were discussed. The implementation of the Distribute plugin was motivated as being well integrated into the Java reference net formalism, while still being practical. For each algorithm the most important aspects of its distribution were explained with class diagrams and interaction diagrams.

7 Discussion and Future Works

The Distribute plugin matured greatly since its early development, documented in [20]. Nonetheless, it has still not experienced much use. An integration into other projects based on Renew would be very beneficial to identify worthwhile improvements. At the same time, the Distribute plugin's synchronous communication capabilities offered to Renew projects are very hard to implement in other ways. It could be integrated into the Mulan agent infrastructure

[9] The distributed version of the search algorithm (Subsection 4.2) makes sure that no binding can be found that would attempt to take the same chair token twice.

to realize synchronous communication between distributed agents, as it was proposed, e.g. in [9]. The application to other communication in the Mulan context is also interesting, such as the persistence (see [18]) or mobility / deployment of agents and whole platforms (see [17]).

The Distribute plugin might also help to improve the ideas of complex distributed workflow execution (see e.g. [19]): it can be used for simple communications between simulations that are distributed to make use of the accumulated computational power and storage capacity of multiple computers. In this scenario, the context of a channel call can often be designed less complex as it is used in some of our earlier prototypes or the preceding works of others (see the slightly different approaches following token, place or transition centered designs: [2,6,8,12,21]), and thus does not interfere with the rest of the simulation or impede its performance. More recently, we integrated some results from Renew-related Cloud computing (see [1]) with the plugin presented here. Several technical issues need to be solved to have an efficient implementation, that is Cloud based.

For future work it will be relevant to check, if the functionality or the modeling power of the here introduced send channel can be extended without substantial performance loss. As already mentioned, we experienced difficulties with the performance of more powerful synchronous mechanisms. Another challenge is to transfer this kind of algorithm to a software package, that could be used in other Petri net simulation formalisms / tools.

References

1. Bendoukha, S., Moldt, D., Bendoukha, H.: Building cloud-based scientific workflows made easy: A remote sensing application. In: Marcus, A. (ed.) 4th International Conference on Design, User Experience and Usability. LNCS, vol. 9187, pp. 277–288. Springer-Verlag (2015)
2. Briz, J., Colom, J.M.: Implementation of weighted place/transition nets based on linear enabling functions. In: Valette, R. (ed.) Application and Theory of Petri Nets 1994, 15th International Conference, Zaragoza, Spain, June 20-24, 1994, Proceedings. LNCS, vol. 815, pp. 99–118. Springer (1994), http://dx.doi.org/10.1007/3-540-58152-9_7
3. Cabac, L., Simon, M.: Introducing catch arcs to Java Reference Nets. In: Moldt, D., Rölke, H. (eds.) PNSE'13, Milano, Italia, June 2013. Proceedings. CEUR Workshop Proceedings, vol. 989, pp. 155–168. CEUR-WS.org (Jun 2013)
4. Cabac, L.: Modeling Petri Net-Based Multi-Agent Applications, Agent Technology – Theory and Applications, vol. 5. Logos Verlag, Berlin (2010)
5. Cabac, L., Haustermann, M., Mosteller, D.: Renew 2.5 – towards a comprehensive integrated development environment for Petri net-based applications. In: Kordon, F., Moldt, D. (eds.) 37th Int. Conference on Application and Theory of Petri Nets, Toruń, Poland. LNCS, vol. 9698. Springer-Verlag (Jun 2016), (to be published)
6. Chiola, G., Ferscha, A.: Distributed simulation of Petri nets. IEEE Parallel Distrib. Technol. 1(3), 33–50 (Aug 1993), http://dx.doi.org/10.1109/88.242441
7. Colom, J., Silva, M., Villarroel, J.: On software implementation of Petri nets and colored Petri nets using high-level concurrent languages. In: Seventh European

Workshop on Application and Theory of Petri Nets. pp. 207–241. Oxford, England (06/1986 1986)

8. Ferscha, A.: Concurrent execution of timed Petri nets. In: Proceedings of the 26th Conference on Winter Simulation. pp. 229–236. WSC '94, Society for Computer Simulation International, San Diego, CA, USA (1994), `http://dl.acm.org/citation.cfm?id=193201.194025`
9. Fix, J., Duvigneau, M., Moldt, D.: Bereitstellung eines Synchronisationsmechanismus für MULAN basierte Agenten. In: Bergenthum, R., Desel, J. (eds.) 18th Workshop AWPN 2011, Hagen, September 2011. pp. 8–14 (2011), `http://www.fernuni-hagen.de/sttp/download/tagungsbandawpn2011.pdf`
10. Haagh, T.B., Rudmose, T., Contents, H.: Optimising a coloured Petri net simulator. Tech. rep., University of Aarhus, Department of Computer Science (1994)
11. Hartung, G.: Programming a closely coupled multiprocessor system with high level Petri nets. In: Rozenberg, G. (ed.) Advances in Petri Nets 1988, LNCS, vol. 340, pp. 154–174. Springer Berlin Heidelberg (1988), `http://dx.doi.org/10.1007/3-540-50580-6_28`
12. Hauschildt, D.: A Petri net implementation. Fachbereichsmitteilung FBI-HH-M-145/87, University of Hamburg, Department of Computer Science (1987)
13. Kaim, W.E., Kordon, F.: An integrated framework for rapid system prototyping and automatic code distribution. In: Proceedings of IEEE 5th International Workshop on Rapid System Prototyping, RSP 1994, Grenoble, France, June 20-23, 1994. pp. 52–61. IEEE (1994), `http://dx.doi.org/10.1109/IWRSP.1994.315909`
14. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002), `http://www.logos-verlag.de/cgi-bin/engbuchmid?isbn=0035&lng=deu&id=`
15. Kummer, O., Moldt, D., Wienberg, F.: Symmetric communication between coloured Petri net simulations and Java-processes. In: Donatelli, S., Kleijn, J. (eds.) Application and Theory of Petri Nets 1999, 20th International Conference, ICATPN'99, Williamsburg, Virginia, USA. LNCS, vol. 1639, pp. 86–105. Springer-Verlag (Jun 1999)
16. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L., Haustermann, M., Mosteller, D.: Renew – the Reference Net Workshop. Available at: `http://www.renew.de/` (Jun 2016), `http://www.renew.de/`, release 2.5
17. Möllers, K.S.M.: Entwicklung eines P*AOSE-Werkzeugs zur Dynamisierung von Verteilungsdiagrammen. Bachelor thesis, University of Hamburg, Department of Informatics (2014)
18. Mosteller, J.: Persistierung in agentenorientierten Anwendungen – Prototypische Implementierung im Kontext von RENEW/MULAN. Master thesis, University of Hamburg, Department of Informatics (Apr 2016)
19. Reese, C.: Prozess-Infrastruktur für Agentenanwendungen, Agent Technology – Theory and Applications, vol. 3. Logos Verlag, Berlin (2010)
20. Simon, M.: Concept and Implementation of Distributed Simulations in RENEW. Bachelor thesis, University of Hamburg, Department of Informatics (Mar 2014)
21. Taubner, D.: On the implementation of Petri nets. In: Rozenberg, G. (ed.) Advances in Petri Nets 1988. LNCS, vol. 340, pp. 418–439. Springer-Verlag (1988)
22. Valette, R., Bako, B.: Advances in Petri Nets 1991, chap. Software implementation of Petri nets and compilation of rule-based systems, pp. 296–316. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
23. Wollrath, A., Riggs, R., Waldo, J.: A distributed object model for the java system. Computing Systems 9(4), 265–290 (1996)