Introducing Catch Arcs to Java Reference Nets

Lawrence Cabac, Michael Simon

University of Hamburg
Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
{cabac,9simon}@informatik.uni-hamburg.de
`http://www.informatik.uni-hamburg.de/TGI`

Abstract. Modeling plays an important role during design and development of systems and processes. Petri nets allow for well-defined models that can be executed. For the implementation of these systems, however, still *normal* programming languages are used. In contrast, modeling languages – also if executable, such as Petri net formalisms – are not deemed fit for implementation. Besides the pragmatic power, one thing that modern programming languages offer and which Petri net formalisms are missing, is exception handling.
In this paper we present an approach that includes exception handling for Java reference nets. Our goal is to make the designed systems more robust and reliable. As a consequence, such executable models can be cleanly integrated into real execution environments.
Our approach provides the information of an exception being thrown to the level of modeling. We are thus able to model the exception handling explicitly within the model as it is done in many modern programming languages. This extension is conservative and does not alter the *normal* behavior of the model, leaving the Petri net semantics untouched. We discuss several possible extensions to our approach with respect to the modeling possibilities, the ease of implementation and their pragmatic usefulness.

Keywords: High-Level Petri Nets, Reference Nets, Exceptions, Catch Arcs, Renew

1 Introduction

High-level Petri net formalisms have often been extended by new primitives. This has been done mostly to improve modeling possibilities. The aim has been to increase comprehensiveness and compactness as in the case of *test arcs* and *inhibitor arcs* (see [2]) or *flexible arcs* (see [8]). We introduce a *catch arc* as a new primitive, in order to raise the tightness of integration with the inscription language. By this we improve the robustness of our executable models. With the new construct we are able to handle exceptions that might occur during execution of the model.

Renew[1] has been missing the possibility to handle exceptions on the model level, since it has been created. In this paper, we introduce a new kind of arc – the *catch arc* – which fills that gap. Its functionality is straight forward: if an exception is thrown during the execution of a transition inscription (in our case Java code) while firing, the exception object is put into the connected place. From this point, the exception can be treated in the model in an appropriate way. This is the reason why we call this arc a *catch arc*. The arc initiates a sequence of code that – in analogy to a catch statement in Java – follows the catch statement. If no exception occurs, the normal Petri net semantics is followed and the *catch arc* does not produce any token. We discuss in detail why this arc is suitable for handling exceptions and how the firing of a transition should be aborted on encountering an exception. Furthermore, we discuss a way to implement handling of exceptions that are thrown in one net without having to catch them all separately.

The structure of the paper is as follows: we introduce the *catch arc* and discuss its behavior by the means of Petri net modeling in Section 2. This is the central concept of this paper. Section 3 presents the complex process of the firing of a transition in Renew. It shows the inability to fully abort a transition firing and the consequent limitations on extensions of the Java reference net formalism. These limitations motivate the *try arc* as a solution. Section 4 introduces the *try arc* and discusses how a transition firing can be reverted after an exception. Section 5 presents ways to isolate the tokens involved in an erroneous firing for exception handling. Using the *catch arc* alone is compared to using it in combination with a *try arc*. Section 6 discusses the (conservative) extension of the *catch arc* by an expression whose result gets returned on catching an exception. In Section 7 we extend the notion of exception handling from transitions to net instances.

2 The Catch Arc

We extend reference nets [6] by adding *catch arcs* that put an exception into a place as object reference, if one occurred. Avoiding uncaught exceptions in *action inscriptions* is very complicated: the modeler would need to make sure that all *action inscriptions* only call Java methods that do not throw exceptions. This is not feasible, as the standard Java methods and well-written Java classes rely heavily on throwing exceptions. We do not want to worry about exceptions in every *action inscription*, but would rather prefer to have a simple way to handle those in general without having to consider each possible error case.

Up until now, the possibility of an exception being thrown was simply not covered by the reference net formalism. Thus, such an event was outside the scope of well-defined behavior of the simulator. On encountering an exception, the ingoing arcs would consume the bound tokens, but no token would be put out by the outgoing arcs. The simulator would log the incident, but ignore it

[1] Renew: The Reference Net Workshop [7], `http://www.renew.de`

in any other regard. By adding the possibility to catch and handle exceptions we pull this behavior into the model level. The behavior of the Petri net models itself is not changed as long as no exception occurs.

Unfortunately, there is no easy and clean way to completely reset a transition that has already begun to fire, as we will discuss in Section 4.

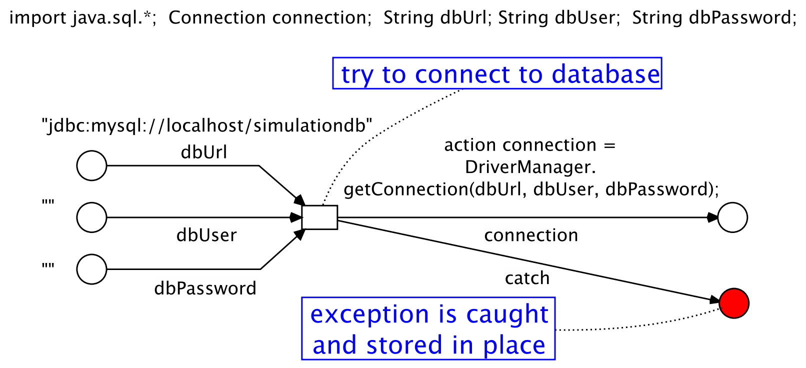

import java.sql.*; Connection connection; String dbUrl; String dbUser; String dbPassword;
try to connect to database
"jdbc:mysql://localhost/simulationdb"
dbUrl
""
dbUser
""
dbPassword
action connection = DriverManager. getConnection(dbUrl, dbUser, dbPassword);
connection
catch
exception is caught and stored in place

Fig. 1. Usage example.

Fig. 1 shows a typical use-case where *catch arcs* enable the modeler to express safe code with a few simple net elements, instead of modeling complex structures or exporting the error handling functionality to Java helper code. The *catch arc* can be identified by the *catch* inscription. The tokens in the places on the left are given, but they may also be dynamically derived by other net segments, for example through a user input dialogue. It is very difficult to ensure that all input tokens to the transition are valid. In this case, there are also external factors that determine the outcome of the *action inscription* as *dbUrl* is the URL of a database to open, which might not be available. In a scenario like this, catching exceptions is unavoidable to have a stable system. As already mentioned above, usually the handling of the possibly thrown exceptions is implemented in Java helper code that encapsulates the opening of the connection and catches exceptions on that level. With the *catch arc* we are not only able to pull the exception handling up to the model level, we also reduce the implementation of wrapper code. In fact, we are able to treat the exceptions as first order concepts within our models.

The reference net depicted in Fig. 2 illustrates the operational semantics of *catch arcs*. The transition holds an *action inscription* which converts a *String* into an *Integer* object. Naturally, this operation throws an exception, if the conversion cannot be achieved due to an invalid argument. Fig. 3 shows the result of executing the net in Fig. 2. The *String* `6` can be converted to an *Integer* which is put into **output**, while the attempt to convert `foo` throws *NumberFormatException* which is put out by the *catch arc*.

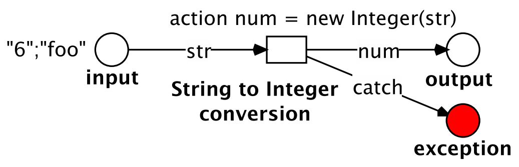


Fig. 2. Example of a safe *String* to *Integer* conversion with a *catch arc*.

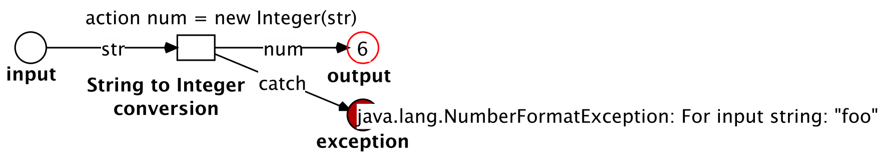
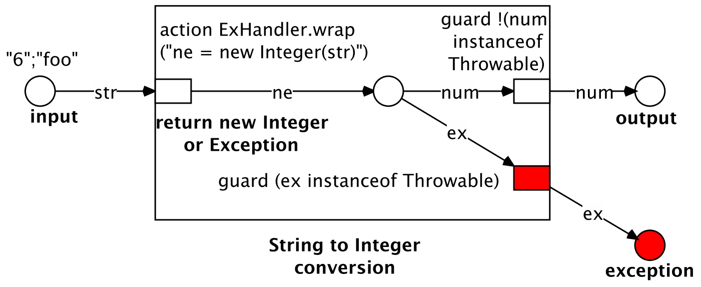

action num = new Integer(str)
str
num
6
input
String to Integer conversion
catch
output
java.lang.NumberFormatException: For input string: "foo"
exception

Fig. 3. Executed instance of the net in Fig. 2.

Fig. 4 is a conceptual model showing the execution semantics of the *catch arc*, simulated by a reference net model and some possible Java code. This model illustrates the implementation of the *catch arc*.


action ExHandler.wrap ("ne = new Integer(str)")
guard !(num instanceof Throwable)
"6";"foo"
str
ne
num
num
input
return new Integer or Exception
ex
output
guard (ex instanceof Throwable)
ex
String to Integer conversion
exception

The *ExHandler.wrap(String)* method returns the same result as the expression given as *String*, if no exception is thrown. Otherwise, it returns the thrown exception.

Fig. 4. Net without *catch arcs* behaviorally equivalent to Fig. 2.

The transition **String to Integer conversion** in Fig. 4 is a refinement of the one in Fig. 2. The *ExHandler.wrap(String)* method is not implemented, but serves to illustrate how a wrapper that catches any error and returns it, or the result would be used to emulate the behavior of *catch arcs*.

The *catch arc* itself serves as a clear identifier of where the control flow for exception handling starts. All transitions that are dependent on the exception output token, are part of the exception handling control flow. By directing a *catch arc* to a place that is only used for exceptions, it is easy to separate the control flow. Even though it is made easy, it is up to the modeler to separate the exception handling parts of a reference net from the rest, like any other software architecture design decisions. This is analogous to the catch block of a try-catch-construct in a textual programming language: what is done inside is up to the programmer. There is no difference between code that can be executed inside the catch block or outside. Still it is good practise to separate exception handling code from other code. The place to which a *catch arc* leads should be regarded in the same way as a catch block in a textual programming language.

We implemented an extension to the *catch arc*, allowing it to be accompanied by an expression. This is further discussed in Section 6.

Another possible improvement of the *catch arc* would be the ability to declare the class of the exception to catch. This would deepen the analogy with the common *try-catch* programming language construct and would simplify the implementation of different ways to handle different exceptions. On the other hand, this behavior is easy to achieve with guards that check the exception types. These would be inscribed to the transitions which handle the exceptions. Because of the flexibility that guard inscriptions offer, there is no need to catch only some types of exceptions. This refinement of the *catch arc* could be implemented by an expression that is given as an argument to the *catch* inscription. It has to evaluate to a (sub)class of *java.lang.Throwable* (the exception rootclass).

A *finally statement* is not needed in the extended Java reference net formalism. In Java it represents a section of code that is inserted into the control flow of the erroneous as well as the normal execution. It is executed in all cases. In the Java reference net formalism the control flow is explicitly modeled. The *catch arcs* start the control flow of the erroneous case and the normal output arcs start the control flow of the normal case. In Fig. 2 there is either one token put out to **output**, or to **exception**. To introduce some code that gets executed as a *finally statement*, we introduce a transition that is fired for every token in both places. This can be done using only classic Java reference net elements.

3 Firing a Transition in Renew

Table 1 provides an overview of the different steps of Renew's internal algorithm which fires transitions (compare with [6, Sec. 14.7]). We did not change this algorithm in any arc implementations presented in this paper. Before a transition is fired, a valid binding has to be found in phase 0. During the actual firing, the *early executables* are applied first. In phase 1 the *early executables* which mostly represent the ingoing arcs and *test arcs*, need to lock the associated places. This ensures that the tokens can not be taken by another transition firing. Then they verify that they can still be applied, as concurrent changes to the net instance state could have invalidated the found binding (phase 2). After this, the *early executables* can finally be executed (phase 3). Then the locks are unlocked again (phase 4) and the firing of the transition is already reported as successful, since the firing can not be aborted anymore (phase 5). In the end, the *late executables*

0	Binding search (before firing)
1	Lock *early executables*
2	Verify 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 transition in Renew.

can be executed (phase 6). They mostly represent outgoing arcs and *action inscriptions*.

If an exception is thrown while searching for a valid binding of a transition, the corresponding search branch is discarded and the transition will never fire that binding. Phase 0 from Table 1 is never left. If an *early executable* leads to an exception in phase 2 or 3, the firing of the transition is aborted. In a case where Java code is in an *action inscription*, on the other hand, it is not evaluated in the binding search (phase 0), but in a *late executable* (phase 6) after a valid binding was found. The firing can not be reverted and the exception has to be thrown.

Early executables are designed to model actions that can be aborted and reverted. They are executed first, so that as much error cases as possible lead to a rollback. *Late executables*, on the other hand, model actions that can not be reverted. Semantically these can not fail, but they can in practice, if they are not modeled safely, e.g., if an action inscription throws exceptions. Safe modeling is very complex and almost never achieved in practice. Thus, it might be tempting to move as much unsafe actions as possible into the *early executables*, so that the classic reference net semantics are never violated. However, a transition firing attempt should only be reverted, if the current binding can not be found again. This is possible, if the state of the net instance has evolved after the binding search. An example is a token that is consumed by another thread between the binding search and the current attempt to fire. In this case, the *early executable* representing the input arc will fail and cause a rollback. On the other hand, if there would be a case, where the current binding can be found again, the Renew simulator would attempt to fire it in an infinite loop. Since in the existing Renew implementation without any new arc implementations all changes of this firing are reverted, this firing itself can not change the net instance's state.

For a modeler who wants to use *catch arcs*, only *action inscriptions* are of concern. The exceptions thrown by all other inscribed Java code are already dealt with in the binding search. Renew does not fire bindings that lead to exceptions in the binding search.

4 Resetting a Transition Firing after an Exception

It is possible to avoid infinite loops when resetting a transition. One way to achieve this, is to have a special token which is consumed if an exception occurs. If only one such token existed, this binding can not be found again until another transition returns it.

We created an experimental implementation of this exception input arc, called the *try arc*. The existence of a *try arc* does not change the behavior of the classic *late executables* after an exception has been thrown. This includes that no tokens are put out. Thereafter, the *early executables* are rolled back, as if the exception had been thrown inside one of them. The *try arc* does nothing when executed normally, but consumes the bound token when rolled back. This behavior is the reverse of the normal input arcs. With this extension of the simulator the *late executables* behave like the *early executables*, because we have implemented a way to revert them.

Unfortunately, the possibilities of the *try arc* have their limits. Action inscriptions with side effects are not handled correctly. In this case, the transition would look reverted, but the side effects could still have occurred.

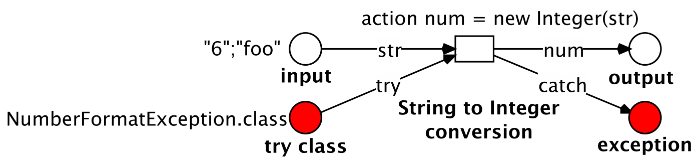

action num = new Integer(str)
"6";"foo"
str
num
input
output
try
catch
NumberFormatException.class
String to Integer conversion
try class
exception

Fig. 5. Net in Fig. 2 extended by a *try arc.*

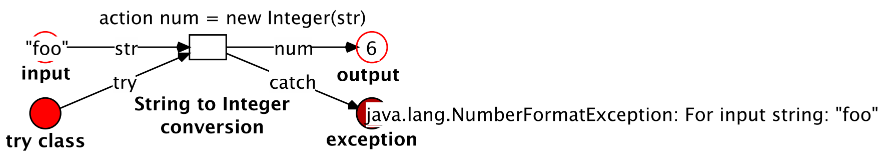

action num = new Integer(str)
"foo"
str
num
6
input
output
try
catch
String to Integer conversion
java.lang.NumberFormatException: For input string: "foo"
try class
exception

Fig. 6. One possible executed instance of the net in Fig. 5.

Fig. 5 extends the net in Fig. 2 by a *try arc*. Fig. 6 shows one of two possible executions of this net. In this execution the transition **String to Integer conversion** was first fired with the string `6` as input. Then `foo` was tried and the exception has been put out by the *catch arc*. The corresponding exception class token has been consumed and the rest of the transition firing has been reverted.

For this reason, the `foo` token is put back. The other possible execution of this net is that `foo` is tried first. In that case, the exception class is removed before `6` can be tried.

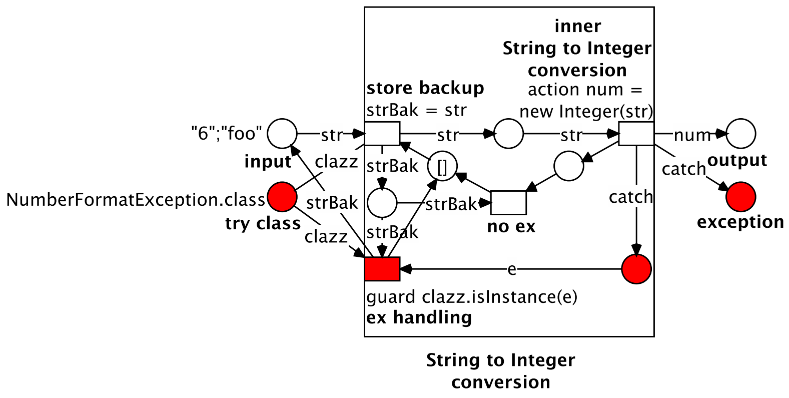


Fig. 7. Net without *try arcs* behaviorally equivalent to Fig. 5.

In Fig. 7 you can see that the behavior of *try arcs* can be emulated in the classic Java reference net formalism, extended only by *catch arcs*. In contrast to the net in Fig. 4, which was used to illustrate the semantics of *catch arcs*, we do not need any special, difficult to implement wrapper functionality in the Java inscriptions. In fact, this net is executable. First, one of the two strings is taken from **input** by **store backup** and passed on as *str*. The same string is put to the place below as *strBak*. The places in the triangle between **store backup**, **inner String to Integer conversion** and **no ex** can only hold one token in total at any time. The transition **inner String to Integer conversion** takes the string, attempts to convert it to an integer and returns the integer to **output**, or the thrown exception to **exception**. On the inside, it either returns a generic token to the white place in the lower left, or the *catch arc* returns another Java reference of the exception to the lower red place. Depending on this outcome, either **no ex** fires, consumes the just processed input string as *strBak* and returns the generic token to the place in the triangle, where it was originally, or **ex handling** fires and puts the input string back to **input**. In this process, it consumes the class token in **try class**. If this token is no longer present, **store backup** can not fire, because of the *test arc* to **try class**. This models the inability of **String to Integer conversion** in Fig. 5 to fire, if the *try arc* can not bind to an exception class token. Like **no ex**, **ex handling** also puts back the generic token. This would enable **store backup** to fire again, if **try class** contained another exception class token.

The model in Fig. 7 is only behaviorally equivalent to Fig. 5, if there are only Java references to one exception class in **try class**. It is more flexible, if we want to use more than one class of exceptions as possible token for the *try arc*. In our current experimental *try arc* implementation, the *try arc* is first bound to an exception class token, before the transition starts firing (in phase 0 of Table 1). In every firing the *try arc* is bound to only one exception class. Thus, the *try arc* may not reset the transition, even if the class of the thrown exception exists as a token. One scenario, where this problem occurs, is a modification of Fig. 5 with a number of exception classes in **try class**. In Fig. 7, on the other hand, the **ex handling** transition can try every class in **try class**. The Renew simulator would need to be changed considerably to implement this behavior for *try arcs*.

5 Handling Exceptions

In this chapter, we compare using the *catch arc* alone and along with a *try arc* to model exception handling. We show that the *try arc* does not provide a real advantage in this situation.

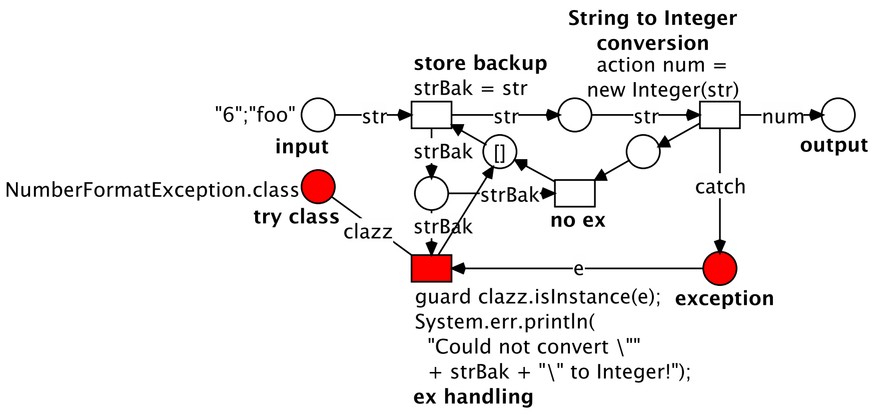


Fig. 8. Net in Fig. 7 modified to print out error.

Fig. 7 illustrates the behavior of the *try arcs* in Fig. 5. The fact that it is very complex, suggests that *try arcs* simplify error handling greatly. However, when handling the exception, one would normally want to ensure that it does not occur again. For this purpose, the involved tokens usually need to be identified and separated from the rest for special treatment. An example would be to generate feedback to calling code or the user. In Fig. 7 there is already a transition for exception handling present: **ex handling**. In order to generate feedback, instead of resetting the firing, we need to remove the output arc from **ex handling** to **input**, so no token is put back. We also need to change the arcs from **try class** to prevent the class token from being consumed. The Java code generating the output can be inscribed to **ex handling**. This is demonstrated in Fig. 8.

If we change the net in Fig. 5 to give feedback, we have more work to do. A possible implementation is presented in Fig. 9, which is very similar to Fig. 8. In both models it is important to reconstruct which input string has induced an exception. For this purpose, the capacity of the input place is restricted to 1. In Fig. 8 there can only be one token in the triangle between **store backup**, **inner String to Integer conversion** and **no ex**. In Fig. 9 there can only be one token in the input place and the place below. In contrast to Fig. 8, there is no need to store another Java reference of the input token, because this token is returned if an exception occurs. For this reason, a transition such as **no ex**, which is fired if no exception has occurred, is also not needed. Another difference is that the exception class token in **try class** gets consumed in the event of an exception and has to be replaced by **ex handling** when the exception is handled.

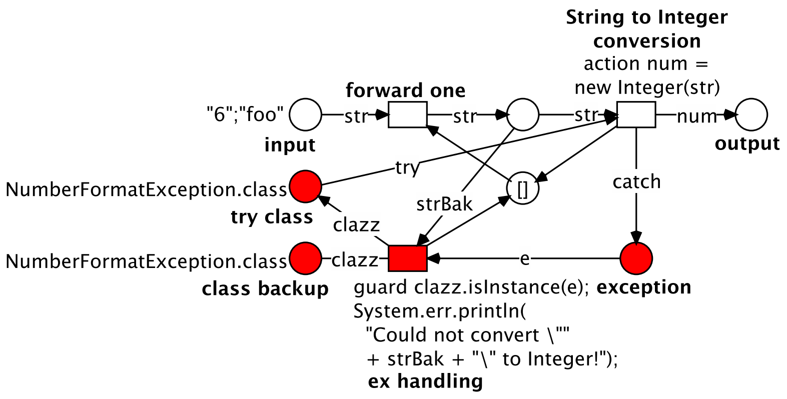

String to Integer conversion
action num = new Integer(str)
forward one
"6";"foo"
input
str
str
str
num
output
try
catch
NumberFormatException.class
try class
clazz
strBak
[]
NumberFormatException.class
clazz
e
class backup
guard clazz.isInstance(e);
exception
System.err.println(
"Could not convert \""
+ strBak + "\" to Integer!");
ex handling

Fig. 9. Net with *try arc* modified to print out error (behaviorally equivalent to Fig. 8).

6 Catch Arc with an Expression

In order to further the expressiveness and thus simplify the scenarios, in which we would like to retain the tokens involved in an erroneous transition firing, we extended the *catch arc* by an expression whose result is produced alongside the exception. Similar to input arc inscriptions, this expression has to be fully bound before firing, and can thus not be dependent on any *action inscriptions*.

Fig. 10 emulates the *try arc* from Fig. 5. Unlike the model in Fig. 7, there is no need to limit the capacity of the input place of the transition, which can throw an exception. The involved input token can be reconstructed from the result of the *catch arc*'s expression, which gets returned to **exception** as tuple alongside the exception.

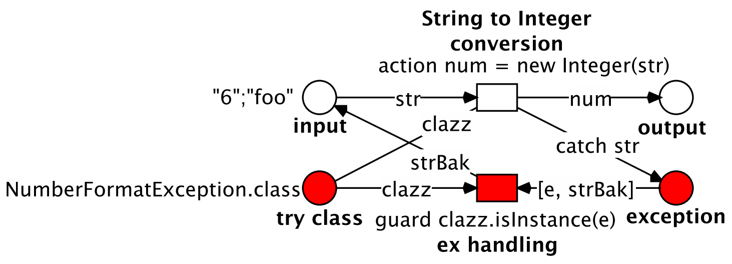

String to Integer conversion
action num = new Integer(str)
"6";"foo"
input
str
num
output
clazz
catch str
strBak
NumberFormatException.class
clazz
[e, strBak]
try class
guard clazz.isInstance(e)
exception
ex handling

Fig. 10. Net without *try arcs*, but with a *catch arc* with an expression, behaviorally equivalent to Fig. 5 and 7.

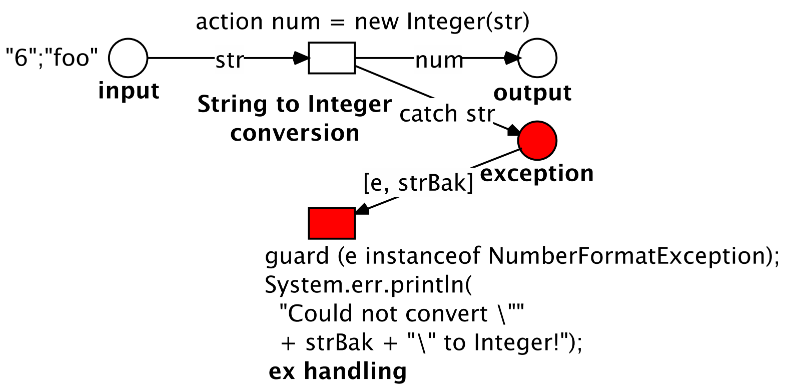

action num = new Integer(str)
"6";"foo"
input
str
num
output
String to Integer conversion
catch str
exception
[e, strBak]
guard (e instanceof NumberFormatException);
System.err.println(
"Could not convert \""
+ strBak + "\" to Integer!");
ex handling

Fig. 11. Net in Fig. 10 modified to print out error (behaviorally equivalent to Fig. 8).

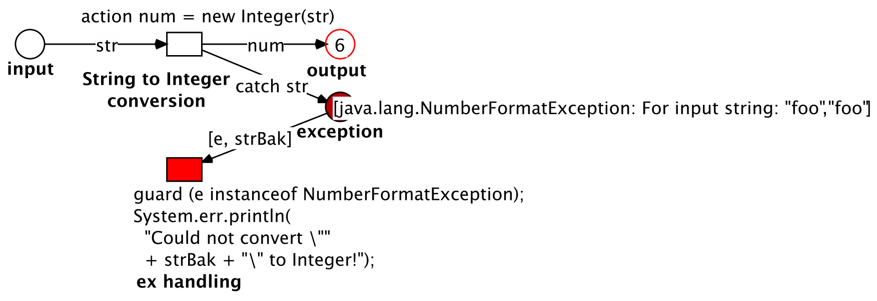

action num = new Integer(str)
input
str
num
6
output
String to Integer conversion
catch str
[java.lang.NumberFormatException: For input string: "foo","foo"]
[e, strBak]
exception
guard (e instanceof NumberFormatException);
System.err.println(
"Could not convert \""
+ strBak + "\" to Integer!");
ex handling

Fig. 12. Instance of the net in Fig. 11 after firing.

Fig. 11 prints out an error message, exactly like the nets in Fig. 8 and 9. The token put out by the *catch arc* can be seen in Fig. 12, which shows an executed instance of the former net after firing the transition **String to Integer conversion** twice. It is a 2-tuple that consists of the thrown exception and the result of the expression.

ex handling is a transition that fires after **String to Integer conversion** in the event of an exception. Because the transition **ex handling** takes in the string together with the exception, it behaves as if it had the same preset as **String to Integer conversion** and took the same token. If one wanted to fire a exception handling transition with all the same input token as the original transition, one could accompany the *catch arc* with a tuple of all input arc inscriptions.

7 Uncaught Exceptions and Exception Propagation

In classic programming languages exceptions propagate outward and escape all code sections, until they are finally caught and handled. If they are not handled in program code, the program crashes on the occurrence of an exception. In RENEW's classic Java reference net formalism exceptions are logged, but otherwise ignored. All ingoing tokens are consumed and no tokens are written out. Since dependency is explicitly modeled in reference nets, it is reasonable to allow those parts of the simulation that are not dependent on the part where the exception occurred, to continue to run.

However, a concept of propagating exceptions upwards through a net hierarchy, can be realised. One step of this propagation can be achieved by a transition that binds exception tokens to an uplink, so another net instance which knows the current instance, can extract the exception. Binding to a downlink would also be possible, but then the exception handling net instance must be known to the current instance. (This would more accurately be described as *propagating downwards.*)

It is also possible to have a specific uplink channel to pass on exceptions that are not caught locally (for example *:catch(e)*). This can be done through a normal net channel that gets bound to all these unhandled exceptions. To achieve this, there can be one place in every net for unhandled exceptions and one transition that binds every token of this place to the uplink of the channel. All transitions without a *catch arc* receive one to this place. The only exception from this rule is at a transition with an uplink. The exceptions thrown by a firing that involves this transition, can be caught at the transition with the downlink. The place and the transition for unhandled exceptions could be hidden to the modeler, so they are only accessible through the channel. The *catch arcs* could be created automatically, where there does not already exist one in the model. If the modeler would like to manually mark an exception as unhandled (maybe because the class is not expected), she can add an uplink transition to the channel herself.

8 Related Work

There are many attempts to model the behavior of exceptions with various modeling languages. This kind of exception modeling includes concepts for expressing the behavior of exceptions that occur in systems. We call these exceptions model intrinsic. Examples are the *exception handler* in current versions of UML (Unified Modeling Language, current version 2.4.1, see [9, Sec. 12.3.23]) and the attempts to include exception handling to (hierarchical) Petri net formalisms (cf. [3] and [4]). While the above mentioned examples model the behavior of exceptions or errors, we strive for the treatment of exceptions that occur during execution of these models. We call these (execution) extrinsic exceptions.

Jannach and Gut [5] discuss the possibilities of exception / error handling in current modeling languages in detail and also point out the difference between modeled exception behavior (intrinsic) and exception handling during model execution (extrinsic). On the side of exception handling of executable models they discuss (among others) the possibilities in workflow execution (e.g., offered by YAWL, compare also with [1]) and WS-BPEL. The focus lies in both cases on the cancellation of processes / workflows and the compensation of undesired results. Cancellation of processes (or process regions) as by the *cancel arc* in YAWL is tightly related to *clear arcs* (*reset arcs*) in Petri net formalisms (i.e. Reset Nets). However, the cancellation / exception trigger comes from within the model – a cancel task has to be modeled explicitly and triggered. In our approach we tighten the integration of the executed model and the underlying expression language.

9 Conclusion and Future Works

We presented an approach and a first implementation of the exception handling as an extension of the Java reference net formalism and the Renew simulation engine. The *catch arc* behavior can be expressed by a combination of net refinement and code wrapper implementation. Our first approach which has been implemented within Renew, constitutes already a powerful and also conservative extension to the execution semantics of Java reference nets. The *try arc* was discussed as a possible further extension. It was introduced and motivated by the idea of resetting a erroneous transition firing. We have shown that it does not add much to the expressiveness of the formalism in the context of exception handling. We do not plan to incorporate it in our practical implementation. The possibility of adding an expression to a *catch arc* whose result is returned alongside the exception, is a conservative extension of the original *catch arc* concept. In difference to the *try arc*, it greatly adds to the expressive power. It can be used to provide relevant details of the binding, in which a firing of a transition failed and allows for concise net models for detailed exception handling. This concept can also effectively emulate the resetting of an erroneous transition firing and thus, supersedes the *try arc* concept. Especially the questions of exception propagation in reference net systems and the adequate compensation modeling is, however, not satisfyingly resolved and needs to be further investigated.

References

1. Michael James Adams. *Facilitating dynamic flexibility and exception handling for workflows.* PhD thesis, Queensland University of Technology, 2007.
2. Søren Christensen and Niels Damgaard Hansen. Coloured petri nets extended with place capacities, test arcs and inhibitor arcs. In Marco Ajmone Marsan, editor, *Application and Theory of Petri Nets*, volume 691 of *Lecture Notes in Computer Science*, pages 186–205. Springer, 1993.
3. W.L.A. de Oliveira, N. Marranghello, and F. Damiani. Exception handling with petri net for digital systems. In *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, pages 229–234, 2002.
4. M. Doligalski and M. Adamski. Exceptions handling in hierarchical petri net based specification for logic controllers. In *Systems Engineering (ICSEng), 2011 21st International Conference on*, pages 459–460, 2011.
5. Dietmar Jannach and Alexander Gut. Exception handling in web service processes. In Roland Kaschek and Lois M. L. Delcambre, editors, *The Evolution of Conceptual Modeling*, volume 6520 of *Lecture Notes in Computer Science*, pages 225–253. Springer, 2008.
6. Olaf Kummer. *Referenznetze.* Logos Verlag, Berlin, 2002.
7. Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. Renew – the Reference Net Workshop. Available at: `http://www.renew.de/`, March 2012. Release 2.3.
8. Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets.* Springer-Verlag New York, October 1997.
9. UML. Unified modeling language: Superstructure. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`, August 2011.