Exceptional Transformations

Eugene Syriani[1], Jörg Kienzle[1], and Hans Vangheluwe[1,2]

[1] McGill University, Montréal, Canada
[2] University of Antwerp, B-2020 Antwerp, Belgium

Abstract. As model transformations are increasingly used in model-driven engineering, the dependability of model transformation systems becomes crucial to model-driven development deliverables. As any other software, model transformations can contain design faults, be used in inappropriate ways, or may be affected by problems arising in the transformation execution environment at run-time. We propose in this paper to introduce exception handling into model transformation languages to increase the dependability of model transformations. We first introduce a classification of different kinds of exceptions that can occur in the context of model transformations. We present an approach in which exceptions are modelled in the transformation language and the transformation designer is given constructs to define exception handlers to recover from exceptional situations. This facilitates the debugging of transformations at design time. It also enables the design of fault-tolerant transformations that continue to work reliably even in the context of design faults, misuse, or faults in the execution environment.

1 Introduction

Model transformation is at the heart of model-driven engineering approaches, and it is therefore crucial to ensure that the transformations are safe to use: when a model transformation is requested to execute, any exceptional situations that prevent the transformation from executing successfully must be detected and the requester must be made aware of the problem. Informing the requester about the situation allows for possible reactions. What exactly needs to be done depends highly depends on the context in which the model transformation has been requested.

A model transformation can be seen as an operation on models, taking a model as input and producing a (possibly implicit) model as output. This is similar to operations in a programming language, which can have input and output parameters and in addition can affect the application state stored in objects or variables. In order to address exceptional situations that prevent the normal execution of an operation, modern programming languages introduced *exception handling* [1].

A programming language or system with support for exception handling allows users to signal exceptions and to define handlers [1]. To signal an exception amounts to detecting the exceptional situation, interrupting the usual processing sequence, looking for a relevant handler, and then invoking it. Handlers are defined on (or attached to) entities, such as data structures or contexts for one or several exceptions. Depending on the language, a context may be a program, a process, a procedure, a statement, an expression, etc. Handlers are invoked when an exception is signalled during the execution or the use of the associated or nested context. Exception *handling* means to put

L. Tratt and M. Gogolla (Eds.): ICMT 2010, LNCS 6142, pp. 199–214, 2010.

the system into a coherent state, *i.e.,* to carry out forward error recovery and then to take one of these steps: transfer control to the statement following the signalling one (resumption model [2]); or discard the context between the signalling statement and the one to which the handler is attached (termination model [2]); or signal a new exception to the enclosing context.

In model transformation, the transformation units (or *rules* in rule-based transformations) that compose a transformation have the notion of *applicability* (of a rule). In contrast to an operation at the programming language level, the model transformation may or may not be applied depending on the applicability of its constituting rules. We must from the beginning clearly distinguish *transformation failure* from *transformation inapplicability*, as we consider these as two distinct outcomes. In graph transformation for example, a rule r is said to be *applicable* if and only if an occurrence of its left-hand side (LHS) is found in the model (encoded as a typed attributed graph). When r also specifies a negative application condition (NAC), such a pattern shall *not* be found given the LHS match. In case of a successful match, the match is replaced by the right-hand side (RHS) of r. Thus the result of a *successfully applied rule* is the (possible) modification of the graph it received. If no occurrences of the LHS were found in the input model, the rule is said to be *inapplicable* and the resulting graph is identical to the input graph. Both a successfully applied rule and a rule that did not match (inapplicable) describe the regular execution of a transformation rule. However, as in the case of the execution of an operation in a program, it is possible that during a model transformation an exceptional situation is encountered in which it is impossible to continue normal execution. At run-time, there are situations in which neither an output model can be produced by applying the transformation in its entirety nor is it possible to determine the non-applicability of the transformation. In this case the rule is said to have *failed*. The definition of applicability, inapplicability, and failure of rules can also be extended to the level of the transformation. That is respectively, the transformation has at least one rule that was successfully applied, no rule in the transformation has been applied, and the last rule to be applied has failed.

Currently, no model transformation language offers means to reason about such exceptional situations encountered during model transformations (see related work section). This paper is a first attempt to motivate and define the notion of exception and exception handling in model transformations. Some may argue that it is not needed in a transformation language and that it is a tool or system issue instead. This contribution claims however that there are many kinds of exceptional situations that can arise while transforming a model, and that these should be modelled in the transformation language to give the modeller control over how such a situation is to be handled. If applied rigorously, exception handling leads to the design of safe model transformations.

The remainder of the paper is structured as follows. In Section 2, we analyse what kind of exceptions can occur in model transformations. Then, in Section 3, we elaborate on possibilities for handling such exceptions. We also outline the implementation of transformation exceptions and their handling in our transformation language. Finally, we put the presented work in perspective in Section 4 and conclude in Section 5.

2 Classification of Exceptions in Model Transformations

Similar to exception class hierarchies used in object-oriented programming languages to distinguish between different kinds of exceptions, we propose in this paper a classification of the exceptions that may arise during a model transformation. In a transformation model, faults may originate from (1) the transformation design, (2) the model on which a transformation is applied, (3) or the context in which the transformation is executed. This section provides a non-exhaustive classification of potential exceptions that may arise during a transformation.

2.1 Terminology

In this subsection we define the terms *failure*, *error*, and *fault* that are used in fault-tolerant computing, in the context of model transformation. A *failure* is an observable deviation from the specification of a transformation. In other words, a failed transformation either produced a result that, according to the specification, is not a valid output model for the specified input, or produced no result at all. An *error* is a part of the transformation state that leads to a failure. The transformation state includes the input and output models, as well as potentially created temporary models and auxiliary variables. A *fault* is a defect or flaw that affects the execution of the transformation. A fault is thus typically present before the transformation execution, *e.g.,* when there is a flaw in the design of the transformation, or the fault arises from the fact that a transformation is applied to a model that it was not designed to work on, or finally the fault resides in the execution environment. At run-time, a fault can be activated and lead to an error, *i.e.,* an erroneous state in the transformation, which in turn may be detected if the transformation language supports it. If it is not handled, however, an error propagates through the system until the transformation fails.

We define an *exception* in the context of model transformation as a description of a situation that, if encountered, requires something exceptional to be done in order to resolve it. An *exception occurrence* at run-time signals that such an exception was encountered.

2.2 Execution Environment Exceptions

`Execution Environment Exceptions` (EEE) represent exceptional situations that typically originate from the transformation's virtual machine.

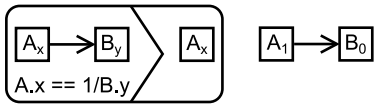

Fig. 1. A rule with attribute constraints written in an action language (on the left) applied to a specific input model (on the right). The rule is specified in **MoTif** [3] syntax where the left and right compartments represent the LHS and RHS respectively.

Action Language Exceptions
When the transformation language allows the use of an action language (which can contain a constraint language such as OCL), a complete exception tree may be provided for types of exceptions specific to the action language itself. Depending on the capabilities of the action language, these exceptions can come from arithmetic manipulations, list manipulations, de-referencing null references, etc. For

example, Fig. 1 illustrates a specific `Action Language Exception` (ALE), namely the case of a division by 0.

System Exceptions
During the execution of a transformation, the virtual machine executing the transformation can encounter exceptional situations, *e.g.,* it can run out of memory. There are many reasons that could lead to such a problem, one of which is a design fault in the transformation itself. Consider a transformation that contains an iteration over a monotonically increasing rule (that never deletes an element nor disables itself) as depicted in Fig. 2. A memory overflow will eventually occur if an infinite loop or recursion (like a recursive rule as described in [4]) is executed.

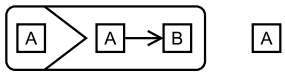

Fig. 2. A monotonically increasing rule (on the left) applied to a specific input model (on the right)

Other kinds of `System Exceptions` (SE) may arise, *e.g.,* I/O Exception when logging is used and the logging device is not writeable. Also, if the access to the model is provided via web-service functions, for example, the server may be down leading to communication or access errors.

2.3 Transformation Language-Specific Exceptions (TLSE)

Features specific to a particular transformation language can also be the source of exceptional situations. For example, **ProGReS** [5], **QVT** [6], and to a certain extent, **Fujaba** [7] allow rules to be parametrized by specific model elements that may be bound to matches from previous rules. In these languages, executing a rule with unbound parameters results in an exceptional situation that needs to be resolved. A similar exceptional situation arises when a pivot node is passed from one rule to another by connecting input and output ports in **GreAT** [8] or even through nesting in **MoTif**, if the rules are not appropriately connected.

But there are other kinds of exceptional situations that can arise due to a specific transformation language design. In languages such as **QVT-R**, for example, the creation of duplicate elements is semantically avoided by the concept of *key* properties. Two elements are logically the same if and only if their key properties are equal. The *key* is used to locate a matched element of the model and a new element is created (with a new *key*) when a matched element does not exist. However, if multiple keys with the same value are found in a model, this indicates that the model is faulty[1].

Moreover, exceptions proper to the implementation of the scheduling language can also be considered. In **MoTif**, for instance, since the underlying execution engine allows for timed transformations by specifying the duration of an application of a rule, bad timing synchronization may arise when *e.g.,* rules are evaluated at the same time (through conditional branching or parallel application). This typically happens due to the numerical error of floating point operations.

2.4 Rule Design Exceptions

`Rule Design Exceptions` (RDE) represent errors that stem from a fault in the design of the transformation model itself.

[1] We assume in this paper that the transformation engines are fault-free.

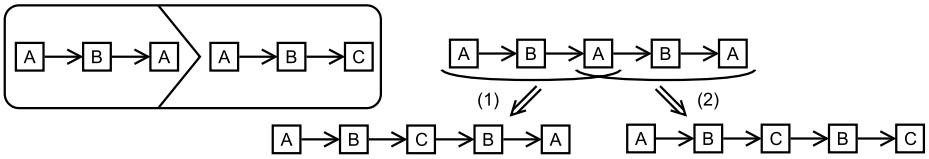

Fig. 3. An inconsistent use of an *iterated rule* (on the left) with respect to a specific input model (on the right)

Inconsistent Use Exception
One class of design faults that may happen in a transformation is when rules are conflicting with one another. We distinguish the case when a rule conflicts with itself from when several rules conflict with each other. The former occurs when a rule finds multiple matches on a given input model and is executed several times in a row. This typically happens during an iteration; *e.g.,* a rule executed in a `loop` in **ProGReS**, iterated in a for-loop or a while-loop in **QVT-OM** [6], or in case the rule is an FRule or an SRule in **MoTif**. This is the case in the example of Fig. 3, where the rule matches the input model twice, but depending on the order in which the matches are processed, two different output models are produced. Although the transformation itself is a valid transformation, the application of the transformation to this particular input model results in a non-deterministic result and as such is very likely to be incorrect. We consider such a situation as an *inconsistent use* of a transformation and propose that in these cases the transformation should be notified with an `Inconsistent Use Exception` (IUE).

Synchronization Exceptions
Another class of design fault can happen in the context of parallel execution of model transformations, a technique often used for efficiency reasons. Semantically, if a transformation designer specifies that two rules should be executed in parallel, this implies that the order of execution of the transformation rules is irrelevant. This optimization can, however, only work if the two rules are independent from one another. For example, the two rules in Fig. 4 are clearly not independent, as the application of one disables the application of the other. In fact, executing both rules in parallel yields two different models that cannot be trivially merged without knowledge of the application domain. We propose to signal such situations by raising a `Synchronization Exception` (YE).

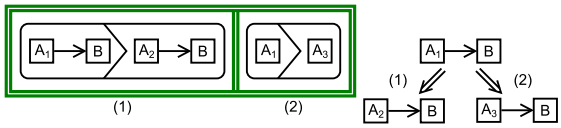

Fig. 4. Two conflicting rules to be applied in parallel (on the left) with respect to a specific input model (on the right). The two rules are specified in a PRule depicting that they will be executed concurrently.

2.5 Transformation-Specific Exceptions

Finally, we believe that a dependable transformation language should also support user-defined exceptions. Almost all programming languages with support for exception

handling support user-defined exceptions that allow the programmer to signal application-specific exceptional conditions to a calling context. Similarly, a transformation language that supports user-defined `Transformation-Specific Exceptions` (TSE) makes it possible for the transformation designer to check desired properties of the model being transformed at specific points during the transformation execution. These property checks can take the form of *assertions* as pre-/post-conditions on a (sub-)transformation by specifying a constraint on the current state of the model. In case the assertion fails, the corresponding TSE can be explicitly raised by the transformation model and signalled to the calling context.

2.6 Using Exceptions in Model Transformations

Fig. 5 summarizes the classification of potential exceptions that may arise during the execution of a transformation. Some classes of exceptions like ALEs and TLSEs can be empty for certain model transformation environments, if the design of the transformation language and action language allows the corresponding problems to be detected statically. In the domain of programming languages, for example, dynamically typed languages such as Python define certain types of exceptions (*e.g.,* NoSuchField Exception) that strongly typed languages do not need to provide. In C++, for instance, a compiler can always statically determine that the programmer was erroneously trying to access a field of a class that has not been declared.

We foresee that exceptions are going to be used in two different ways in the context of model transformation: during transformation development to help eliminate design faults (*debug mode*) and when the transformation is applied to different models in order to increase dependability of the transformation at run-time (*release mode*). Some exceptions are more likely to occur in debug mode while others are relevant only in release mode.

Debug Mode. When running a transformation in debug mode, the goal of applying the transformation to an input model is not so much to obtain an output model that is subsequently used for other purposes, but to validate that the transformation design is correct. Debugging a transformation is not trivial and exceptions are very helpful for *debugging*, namely to detect logical errors in the design of a transformation.

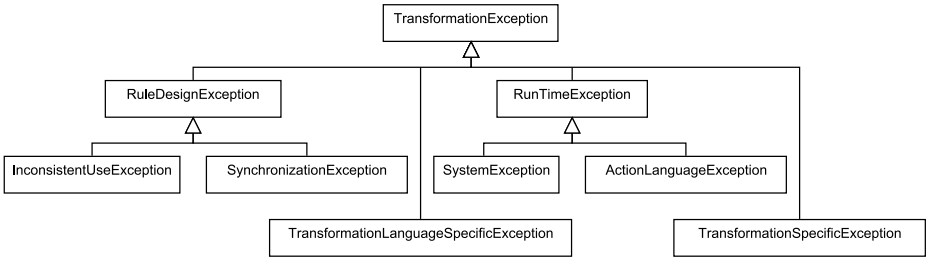

Fig. 5. The proposed classification of model transformation exceptions in UML class diagrams

If the generated output model does not correspond to what the transformation designer expects, then there must be a flaw in the transformation design that has to be found. In this case, the modeller can debug the transformation by adding "assertion" rules at intermediate points in the transformation that check that the previous rule achieved the desired effect. If not, a user-defined TSE is thrown.

If unhandled, the exception halts the transformation execution and the transformation modelling environment informs the modeller of the exception kind and point of occurrence. Using this information, the modeller can more easily locate the rules that contain design faults.

When transformation rules are run distributed or in parallel to increase performance, a YE indicates a merging problem of the different output models. The problem occurs if the rules that are executed concurrently are not independent, *i.e.,* the intersection of the model elements modified by the rules is not empty. No transformation tool can provide an automated general merge operation, not only because general graph merging is undecidable, but also because the correct merging algorithm depends on the specifics of the transformation and its domain(s). Most likely a YE indicates that the modeller incorrectly assumed rule independence when he decided to instruct the transformation engine to use parallel execution.

The occurrence of an IUE on an input model, that the transformation under development should be able to handle, indicates that the iterated rule in which the exception was detected was incorrectly specified. The modeller needs to inspect the information carried with the exception such as the faulty matched model elements as well as the context of execution to then correct the faulty rule or revise the transformation design.

An EEE in debug mode can signal various problems to the modeller. It can signal design flaws, including flaws that are due to the incorrect use of a specific action language feature (*e.g.,* UnboundParameter Exception), incorrect expressions specified by the modeller using the action language (DivisionByZero Exception), or faulty transformation designs that result in infinite recursion or loops (MemoryOverflow Exception).

Release Mode. In release mode, a transformation that is assumed to work correctly is applied to an input model with the goal of producing an output model that is used for a specific purpose. Most likely it is essential that the transformation was applied successfully and did indeed produce the expected result, otherwise the output model is unusable. It is therefore important to design reliable transformations that can recover from exceptional situations and still provide a useful output.

In release mode, a SE such as `IOException` could signal that the device used for logging transformation related information is currently not writeable, for instance because the communication link broke down. Instead of immediately halting the transformation process, a reliable transformation could try to handle this situation. For instance, if the fault is assumed to be transient, the exception could be handled simply by waiting for some time and restarting the failed transformation. Alternatively, a different device could be used to store the log information.

The occurrence of an IUE in release mode signals that the transformation is being applied to an input model that the transformation was not designed to handle. An example of such a situation is given in Fig. 3. This does not mean, however, that the rule cannot produce a correct output model. Both possible outputs shown in Fig. 3 might be

correct, or maybe only one of them is. The problem is that the transformation system cannot guess what the correct behaviour of the transformation should be. One way of handling the exception could be to obtain *user (or external) input* from the transformation environment, *i.e.,* halt the transformation, prompt the user to designate the correct match or output, and continue with the transformation. Another transparent way of handling could be to apply a different set of rules instead that can produce an appropriate output model using different rules.

3 Exception Handling in Model Transformation

The previous classification identifies the exceptional situations that can occur during a model transformation. In order for a transformation to be dependable, the transformation designer should think of potential exceptions that could occur at run-time and design a way of addressing them in order to recover. We must therefore define a way that allows a modeller to reason about exceptions and express exception handling behaviour at the same level of abstraction as the model transformation itself.

3.1 Modelling Exceptions

In order to be able to reason about exceptions at the transformation level, exceptions should be treated as first-class entities, *i.e.,* just like any other model element that can itself be used as an input to a transformation. From a transformation language design point of view, a transformation exception can be considered as a model conforming to a distinct meta-model as shown in Fig. 6. An exception is identified by a name and has a status which can be: `active` (*i.e.,* the exception instance has not been addressed yet), `handling` (*i.e.,* it is currently being handled), or `handled` (*i.e.,* it has been addressed by a handler).

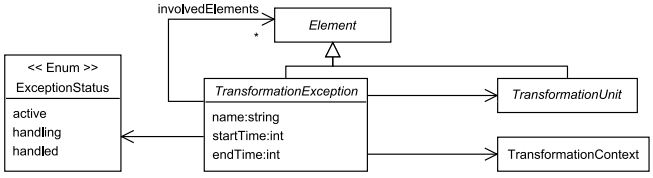

Fig. 6. The transformation exception meta-model

In order to enable proper handling, an exception must hold relevant information regarding its activation point context: where it happened, what happened, and when it happened. The transformation exception therefore references the transformation unit (the rule) that triggered its activation. The transformation context depicted in Fig. 6 contains all the information needed to effectively investigate the origin of the exception occurrence and allow the designer to model an appropriate handler. In our implementation, for instance, the context contains the stack frame and the state of the packet (see Section 3.3) at the activation of the exception. In compositional or hierarchical transformation languages such as **MoTif**, **GReAT**, or **QVT**, knowing the exact path to the rule helps locating the fault in the transformation design, especially if the handler is not in the same scope as the activation point. The activation point can be specified at the level of primitive transformation operations supported by the virtual machine instruction set

(*e.g.,* CRUD[2]) or at the transformation rule that triggered the exception. If the modelling language makes it possible to isolate the transformation operators (match, rewrite, iterate, etc.) from the virtual machine operations, such as in [9], then the activation point can be specified in terms of these operations. In addition to the point of activation, the transformation context should also indicate the state of the transformed (input) model at the time when the exception was thrown. For instance, in order to handle an RDE effectively, the input model elements involved in the matcher of the current rule should certainly be accessible to the handler.

In our proposed meta-model of an exception we also included timing information, such as the timestamp at which the exception was generated (`active`) and has been handled (`handled`). This can be useful for profiling the transformation and gathering statistical measures on the handling policy. Moreover, in timed transformations such as in **MoTif**, the global (simulated) transformation time as well as the local time of the transformation rule operator may be useful.

3.2 Detection of Exceptions

When a transformation executes, the transformation run-time and the underlying virtual machine must monitor the transformation steps to detect the different kinds of exceptions presented in Section 2 and signal them appropriately.

For example, the transformation run-time of the **MoTif** framework is depicted in Fig. 7. It consists of several layers. **MoTif**, the language that a modeller uses to express transformations is a shortcut language of **MoTif-Core**, which consists of the core elements of the language. The former language simply defines a more user-friendly syntax encapsulating the different transformation operators provided in the latter language. **MoTif-Core** combines **T-Core** [9] and the *Discrete EVent system Specification* (a.k.a. DEVS), both running on a model-aware virtual machine. The different classes of exceptions relevant to the modeller presented in Section 2 are detected at different layers, but must all be propagated to the **MoTif-Core** layer (and conceptually to the **MoTif** layer) in order to allow the modeller to handle them explicitly within the transformation, if unhandled in the meantime.

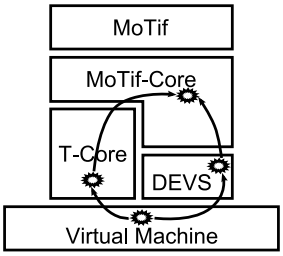

Fig. 7. The **MoTif** framework and the propagation of exceptions across different layers

Detection of ALEs, such as *null de-reference* or *division by zero*, are typically detected at the level of the virtual machine in the **MoTif** framework. Depending on whether the action language is interpreted or compiled, certain design faults can even be detected at compile-time, in which case the corresponding exception never occurs at run-time. Similarly, the transformation language may prevent the action language from accessing model elements that are not explicitly part of the LHS, RHS, or NAC patterns, in which case null de-referencing can never occur. This may, however, be considered as a restriction on the expressiveness of the transformation language used and may lead to excessively large rules.

[2] The commonly known Create, Read, Update, and Delete operations on model elements.

SEs are typically detected by the underlying operating system and the implementation language which is a Python interpreter in the **MoTif** case. To properly propagate the detected exception to the modeller, the exception needs to be caught at the virtual machine interface and transformed into the corresponding exception model instance shown in the previous section.

TLSEs are detectable at the level of **T-Core**, typically by checking pre-conditions before executing any language constructs. TLSEs are again an example of exceptions that can be rendered obsolete if the transformation language is compiled and strongly typed, in which case the compiler should be able to detect unbound parameters and similar situations. Bad timing synchronization of events can also be detected at the level of DEVS.

RDEs are also detected at the transformation language level. In algebraic graph transformation approaches, some RDEs can be detected statically. In grammar-like languages (a.k.a. unordered graph transformation), rule non-confluence can be detected through critical pair analysis [10]: verifying if a rule can disable another, *i.e.,* making it inapplicable. In such languages, this technique can assert parallel and sequential independence of the rules. Tools such as **AGG** detect these conflicts by overlapping the rules (all possible combination of the LHSs, taking NACs into consideration). However, their current approach is sometimes too conservative leading to false positives as it does not take into consideration the meta-model constraints (an example is given in [11]). Moreover, although containing critical pairs of rules, a transformation may still be semantically correct and avoid the conflicts depending on the matches selected at runtime. The occurrence of an IUE can usually also not be checked statically since, most of the time, the input model to which a transformation is applied is not known at compile time.

Controlled graph transformation languages—which are more general than algebraic graph transformation approaches—consist of (partially) ordered rules, where rule scheduling is not implicit but modelled explicitly by the transformation designer. In this case, critical pair analysis is not directly applicable. It must first be adapted to controlled transformations as it may consider a pair of rules in conflict although the conflict does not occur at run-time because of a particular rule scheduling. For instance, let r_1, r_2, r_3 be a sequence of rules to be applied in this order such that the critical pair analysis test fails on (r_1, r_3) because r_1 deletes an element that can be matched by r_3. If r_2 re-creates those deleted elements, r_3 may still be applicable. In our framework, **T-Core** primitives such as a `resolver` or a `synchronizer` can be customized to detect IUEs and YEs [9].

Detection of TSEs cannot be done by the transformation framework automatically, since those situations depend on the semantics of the specific transformation. As mentioned in Section 2, they represent user-defined exceptions. Just like in programming languages that support user-defined exceptions where the programmer is responsible for detecting the exceptional situation using `if` statements or assertions, TSEs have to be detected by the transformation modeller. Fortunately, expressing a condition that needs to be satisfied by a model (or a condition that should never be satisfied by a model) is trivial in a transformation language: the condition can simply be expressed as a query on the input model. In graph transformation systems, this query can be modelled by a transformation rule consisting solely of a LHS. Depending on the query, either a match

being found or the fact that no matches are found depicts a violation of a constraint. To signal that, the rule must have the ability to `throw` an exception, which is described in the following subsection.

3.3 Extending Rules with Exceptions

In order to allow rules to signal exceptions to the enclosing transformation, we propose to add exceptional outcomes to rules. Therefore, such an *exceptional rule* receives a model as input and has three possible outcomes: a successfully transformed model (in case of a successful match and execution of the transformation), an unmodified model (in case the rule is inapplicable on the model), or an exception (when an exceptional situation occurred). If the rule outputs an exception, there are two possibilities: either (1) if the error took place in the matching phase, then the input model is not modified, or (2) if the error took place during the application phase of the rule, then the input model may have been partially modified. The latter outcome seems to defeat the *atomicity* property of a rule. However, as expressed in [12], a partial output may sometimes be desirable. This feature is certainly very helpful in debugging mode, as the modeller would like to see a partial result even if not complete to understand at what point the transformation execution failed in terms of the input model. Nevertheless, if backward recovery is desired, the transformation language should offer a mechanism to roll-back to the previous "safe" state of the model, *i.e.,* to the state that was valid before the rule was applied.

We have integrated the notion of exception in **MoTif** following the ideas mentioned above. **MoTif** is a graph transformation language whose semantic domain is the **DEVS** formalism. Among the elements composing the language, **MoTif** *rule blocks* represent transformation rules. They exchange *packets*[3] through *input* and *output ports* connected via *channels*. Upon receiving a packet from the *packet inport*, an ARule (atomic rule block as illustrated by **`Faulty`** in Fig. 8) finds the potential matches and then applies the transformation depicted by the rule it encodes. If the rule is applicable, the transformed graph is output through the *success outport* (depicted by a tick); otherwise the input graph is output through the *inapplicable outport* (depicted by a cross). Being an event-based system, we model exceptions as events since they allow interruption. Transformation exceptions are output through an *exception outport* (depicted by a zigzag line). Therefore a rule has three possible outcomes: a new graph when the transformation is successful, the unmodified graph when the rule is not applicable, or an exception (modelled as in Section 3.1) if an exceptional situation is detected. In the latter case, the packet may have been partially modified. On the one hand in the matching phase, information concerning the matched elements may have already been stored in the packet. On the other hand, if the rewriting phase was already initiated, the input model may have been modified. In any case, the packet is in an *inconsistent state* with respect to the atomicity property of graph transformation rules application. The modeller can choose to output either the recovered packet or the packet as is in the exception. This is done if the exceptional rule is an XRule, supporting backward recovery of the packet through checkpointing.

[3] A packet consists of the graph to be transformed as well as matching information from previous rules, all embedded in DEVS events.

3.4 Modeling the Handler

Unlike current transformation tools such as **ATL** [13] or **Fujaba** where exception handling is available only at the level of the code of the implementation of the transformation language, we believe again that the most appropriate level of abstraction at which exception handling behaviour should be expressed is at the transformation language level. This is similar to what is done in programming languages, *e.g.,* in Python, where exception constructs are provided in Python and not in C (the language in which Python is implemented). This could be achieved by specifying exception handlers either (1) at the level of transformation rules (in which case an exception handler would take the form of a transformation rule that is only applied in exceptional situations), (2) at the level of the transformation operators (*e.g.,* at the level of **T-Core** primitives such as the `matcher` or the `rewriter`), (3) or at the level of the primitive model manipulation implementation provided by the virtual machine level (CRUD). For the same reasons that we detailed in Section 3.1, we consider specifying exception handlers at the same level of abstraction as the transformation rules themselves as the optimal choice.

We propose two alternatives for how transformation exception handlers can be specified in a model transformation language. From a pure model transformation point of view, an exception can be seen as an ordinary model, although its semantics distinguishes it from a "normal" model. Hence when a rule emits an exception (model), this model can serve as input for other rules whose pre-condition looks for a specific exception type. Given an appropriate meta-model for modelling an exception (such as partly given in Fig. 6), the meta-model of the LHS pattern of an exception handling rule (*i.e.,* its domain) would need to involve multiple meta-models: the meta-model of the input model to transform as well as the meta-model of the transformation exception. This can be easily accomplished if the transformation language is itself meta-modelled as in [14].

Although this solution is elegant, our experience showed that it is not very practical to let the modeller specify rules that match elements from the transformation domain and simultaneously from the domain of exceptions. A more pragmatic solution is to let the exception produced by a rule be used to influence the control flow of the transformation, redirecting it to rules designed for handling specific exceptions. To support this, **MoTif** introduces an explicit *handler block* where the modeller may access the information of the exception model and specify subsequent rules to pursue the exception flow. The handler block acts as a dispatcher sending the packet contained in the transformation context of the exception through the outport corresponding to the exception name. Fig. 8 shows the use of the handler block. The handler block associates the packet with the appropriate exception outport given a predefined exception tree. Since it is possible that not all the exceptions that can occur during a model transformation execution have a specific handler rule designed

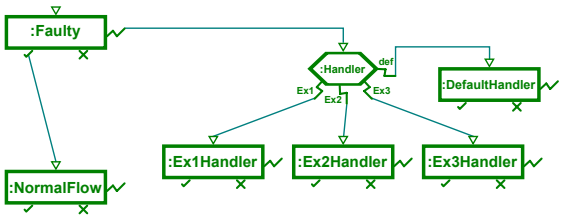

Fig. 8. Handling exceptions in the transformation model

to address them, a default exception port is provided with the handler block (which is linked to the top-most exception class in the classification tree presented in Fig. 5, *e.g.,* `TransformationException`).

Note that in both models the handling part may itself produce an exception which can be in turn handled. For example, since the handling process involves further pattern matching, memory overflows are likely to occur and hence it is necessary to properly handle such exceptions.

3.5 Control Flow Concerns

When a rule emits an exception, the control flow is redirected to a handler component which, in release mode at least, handles the exception with the goal of continuing the transformation. After the exception is handled, there are three options: the enclosing transformation may *resume*, *restart*, or *terminate*.

Resuming the transformation means to return the flow of control to the place where it was interrupted by the exception. As depicted by channel (1) in Fig. 9, the transformation continues in the "normal" flow *after* the rule that activated the exception. Such a resumption model allows to express an alternative execution of the transformation. However, care should be taken if the input model (or even the packet) was modified. As outlined in Section 3.3, the modeller may choose to recover the model to a state that was valid before the rule started applying its modifications, if desired.

Restarting the transformation means to re-run the enclosing transformation from the beginning. As depicted by channel (2) in Fig. 9, the transformation restarts the "normal" flow *before* the rule that activated the exception. This is certainly an interesting way to tolerate transient faults. However, restarting the transformation induces a loop in the control flow which may lead to dead-locks, in case the fault is of a permanent nature.

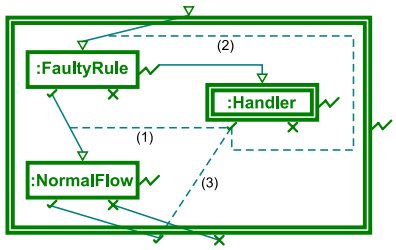

Fig. 9. Modelling possible control flows after handling an exception locally at a sub-transformation level: (1) resume after the activation point, (2) restart at the beginning of the enclosing context, and (3) terminate enclosing context

Terminating the transformation means to skip the entire flow of the transformation in which a rule raised the exception. As depicted by channel (3) in Fig. 9, the enclosing transformation exits the scope of the occurrence of the exception seamlessly. It ends in a "normal" flow, *i.e.,* in success or not applied mode, and not in generating an exception. As a result, the outer scope is not aware that an exception occurred.

Exception Propagation
Up to now, we have considered that it is at the level of a transformation rule that an exception is generated and subsequently handled at the level of the enclosing transformation. As mentioned previously, **MoTif** transformation models are hierarchical, in the

sense that transformations can be nested. Constructs such as a CRule modularly define scoped sub-transformations, allowing to compose transformations models.

Just like in programming languages, it is recommended to handle exceptions in a scope that is as close as possible to the point of activation. In other words, local exception handling is preferred. However, it is possible that handling an exception locally is not possible, because the necessary context information that is needed to define a useful handler is only available in a more global context.

Therefore, unhandled exceptions must be propagated up the transformation hierarchy as long as no corresponding handler can be found. Only if an exception propagates unhandled out of the topmost CRule, the transformation execution must be halted and debugging information displayed. Nevertheless, at each level, a handler could be specified to "clean up" any local state before the exception is propagated outside. Note that once an exception is handled, it can no longer be propagated. If propagation is needed, the handler must create a new user-defined exception which can refer to the previously handled exception in its `TransformationContext`.

4 Related Work

The concept of exceptions exists in programming languages since the 1970s [2]. Several approaches have been proposed for modelling exceptions in workflow languages [15] and event-driven languages [16]. However, there has not been any work on modelling exceptions in model transformation languages. Current tools mostly rely on exceptions triggered from the underlying virtual machine. As a matter of fact, debugging in tools such as ATL [17], Fujaba [7], GReAT [8], QVTo [18], SmartQVT [19], or VIATRA [20] is specific to their respective integrated development editors (IDE).

QVT Operational Mappings (**QVT-OM**) supports exception handling at the action language level, an imperative extension of OCL 2.0 [6]. The language allows to handle exceptions in a *try ... except* statement in the same way as in modern programming languages such as Java, It is however unclear where, when, or how an exception occurs in **QVT-OM**. User-defined exceptions can be declared and raised arbitrarily[4] in the *main* operation of a transformation. Moreover, an exception is also raised when a *fatal assertion* is not satisfied. However, it is unclear what information exceptions carry and whether they can be propagated outside the scope of the transformation. Implementations of **QVT-OM** such as SmartQVT and QVTo (an Eclipse plug-in into EMF) have different interpretations from the standard, *e.g.,* allowing *map* to raise an exception if the pre-condition is not fulfilled. The advantage of our approach is to (1) explicitly model the raising of exceptions and (2) explicitly model the control flow subsequent to the handling of an exception.

Fujaba is a model transformation tool based on graph transformation combined with Story diagrams [7]. There, exceptions are also not modelled, although present at the code level. The *maybe* statements in Story diagrams can be used to handle exceptions in the transformation, but they are only available for statement activities (*i.e.,* Java code). The same argument can be used as for the choice of allowing exception handling at the level of **T-Core**.

More elaborate details of the presented approach can be found in [21].

[4] This fits in the *Action Language Exceptions* category according to our classification.

5 Conclusion

In this paper, we have motivated the need for providing the concepts of exception and exception handling at the level of transformations. We have outlined a classification of potential exceptions that can occur in the context of model transformation. Though having different uses at different steps of the development of a transformation model, these transformation exception must be handled by the transformation model itself. We have discussed the different issues related to the handling of these exceptions.

We have implemented the main concepts of this approach in **MoTif**. As the prototype is still in an early stage, we are working on a system which will allow for user friendly debugging of model transformation.

The same exercise this paper presented for the **MoTif** framework can be done for the **ATL** or **QVT-OM** languages. We are confident that it is also applicable to transformation languages at different levels of abstraction such as relational transformations (*e.g.,* **QVT-Relational** or **Triple Graph Grammars**).

Exception handling can become handy when designing a higher-order transformation. For example in **ATL**, static verification of well-formed higher-order transformation rules is quite limited [14]. In this case, with an exception handling mechanism at the transformation level, the designer may safely rely on the engine to design arbitrarily complex higher-order transformations.

References

1. Dony, C.: Exception handling and object-oriented programming: Towards a synthesis. In: ECOOP. SIGPLAN, vol. 25, pp. 322–330. ACM Press, New York (1990)
2. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Communications of the ACM 18, 683–696 (1975)
3. Syriani, E., Vangheluwe, H.: DEVS as a Semantic Domain for Programmed Graph Transformation. In: Discrete-Event Modeling and Simulation: Theory and Applications, CRC Press, Boca Raton (2009)
4. Guerra, E., de Lara, J.: Adding recursion to graph transformation. In: GT-VMT 2007, ECEASST, Braga, vol. 6 (2007)
5. Zündorf, A., Schürr, A.: Nondeterministic control structures for graph rewriting systems. In: Schmidt, G., Berghammer, R. (eds.) WG 1991. LNCS, vol. 570, pp. 48–62. Springer, Heidelberg (1992)
6. Object Management Group: Meta Object Facility 2.0 QVT Specification (2008)
7. Fischer, T., Niere, J., Turunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on UML and Java. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
8. Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. SoSym 5, 261–288 (2006)
9. Syriani, E., Vangheluwe, H.: De-/re-constructing model transformation languages. In: GT-VMT, ECEASST, Paphos (2010)
10. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
11. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach. In: ICSE 2002, pp. 105–115. ACM, Orlando (2002)

12. Gardner, T., Griffin, C., Koehler, J., Hauser, R.: A review of QVT submissions and recommendations towards the final standard. In: MetaModelling for MDA, pp. 178–197 (2003)
13. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
14. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Systematic transformation development. In: ECEASST, vol. 21 (2009)
15. Brambilla, M., Ceri, S., Comai, S., Tziviskou, C.: Exception handling in workflow-driven web applications. In: WWW 2005, Chiba, pp. 170–180 (2005)
16. Pintér, G., Majzik, I.: Modeling and analysis of exception handling by using UML statecharts. In: Guelfi, N., Reggio, G., Romanovsky, A. (eds.) FIDJI 2004. LNCS, vol. 3409, pp. 58–67. Springer, Heidelberg (2005)
17. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming, Special Issue on EST 72, 31–39 (2008)
18. Dvorak, R.: Model transformation with operational QVT. In: EclipseCon 2008 (2008)
19. France Telecom R&D: SmartQVT (2008)
20. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming 68, 214–234 (2007)
21. Syriani, E., Kienzle, J., Vangheluwe, H.: Exceptional transformations. Technical Report SOCS-TR-2010.2, McGill University, School of Computer Science (2010)