A new Meta-Model for Story Diagrams

Christian Heinzemann*, Jan Rieke*,
Markus von Detten, Dietrich Travkin
Software Engineering Group,
Heinz Nixdorf Institute,
University of Paderborn, Germany
[c.heinzemann|jrieke|mvdetten|travkin]
@uni-paderborn.de

Marius Lauder†
Real-Time Systems Lab,
Technische Universität Darmstadt, Germany
marius.lauder@es.tu-darmstadt.de

ABSTRACT

Story-driven modeling (SDM) is a model-based specification approach combining UML activity diagrams and graph transformations. In recent years, the development in the SDM community led to many incompatible meta-models for story diagrams based on the same common concepts. The diversity of meta-models hindered the reuse of tools and limited synergy effects. In this paper, we introduce the new meta-model for story diagrams which was created in a joint effort of the SDM community. The new EMF-based model integrates the recent developments and paves the way for the interoperation of SDM tools with each other and with EMF-based tools.

1. INTRODUCTION

Story-driven modeling is a model-based specification approach which combines aspects from UML activity diagrams and graph transformations into an expressive and intuitive graph rewriting language, so-called *story diagrams* [3]. In the past years, story diagrams have received significant attention and have become the foundation of many different software engineering techniques and tools.

Ever since their inception, story diagrams have been used in widely different domains and for such different purposes as meta-model integration [1] or the specification of real-time systems [6]. Story diagrams can either be executed by generating appropriate code (e.g., [4]) or by interpretation [5]. These different approaches have led to a variety of extensions and specialized dialects of the original story diagram concept and were accompanied by a number of different tools for the specification, application and analysis of story diagrams. Unfortunately, due to this development, a number of different, incompatible meta-models for story diagrams have emerged which are all based on the same common concepts. Hence, reuse and the composition of tool chains is severely limited by these technical differences.

To cope with these problems, a new meta-model for story diagrams has been developed in a joint effort of the SDM community. The new meta-model integrates a number of useful concepts from the different dialects and provides an extensible framework for future developments. It is based on the Eclipse Modeling Framework (EMF) and thereby paves the way for the interoperation of SDM tools with EMF-based tools. In this paper, we present a slightly simplified version of the actual meta-model to allow for more concise explanations and the omission of technical details.

Before going into the details of the proposed meta-model, we briefly recall the concepts of story diagrams in Section 2. After introducing the meta-model in Section 3, we draw conclusions and sketch future work in Section 4.

2. STORY DIAGRAMS

Story diagrams allow to combine control flow with non-deterministic graph transformation rules. By means of graph grammars, they add a formal foundation to UML activity diagrams for the specification of behavior and, thus, enable their execution and analysis. A story diagram is a special activity diagram that specifies control flow by activity nodes and transitions (activity edges). In contrast to UML activity diagrams, activity nodes in story diagrams contain so-called *story patterns*.

A story pattern is a formal specification of a graph transformation and specifies an object structure (subgraph) that has to be matched in a model (host graph) as well as corresponding modifications of this structure. The structure is specified by special object diagrams in which the modifications, i.e., creation and deletion of elements as well as attribute value assignments, are designated accordingly. The object diagrams are typed over a set of classes.

3. THE NEW META-MODEL

In this section, we introduce the new meta-model package by package. Since the story patterns used for the specification of story diagrams are typed over a set of classes, a class model is required to specify story patterns. We model these classes by means of an Ecore model, thereby avoiding to define yet another meta-model for classes.

In Section 3.1, we give a short tour of the core elements. Then, we introduce the packages for story patterns and activities in Sections 3.2 and 3.3. Next, we discuss a simple example in Section 3.4. Finally, the new expressions and calls packages are presented in Sections 3.5 and 3.6, respectively.

*supported by the International Graduate School "Dynamic Intelligent Systems"

†supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt

3.1 Packages and Core Elements

The package structure of the new meta-model is outlined in Figure 1. The modeling package contains the base classes and an annotation mechanism. It also includes subpackages for patterns, activities, expressions, and calls. The package patterns contains the meta-model classes for specifying story patterns. These classes have been separated from the package activities to enable the reuse of story patterns in other pattern languages, e.g., TGGs [8]. The package expressions contains a set of basic expressions while the package calls comprises the classes for modeling invocations of other story diagrams or operations. A detailed introduction to the packages is given in the subsequent sections.

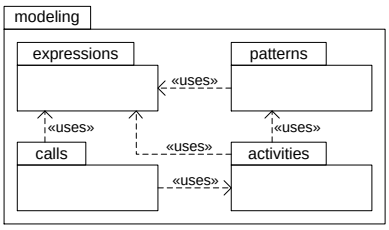

Figure 1: Package Structure

Figure 2 illustrates the core classes of our meta-model. The class ExtendableElement, being the super class of all meta-model classes, implements the annotation mechanism. Each element can be extended by subclasses of Extension. Additionally, we support to annotate our model elements (EModelElements) using EAnnotations.

The classes TypedElement, NamedElement, and CommentableElement are super classes of meta-model classes having a type, a name, or the ability to carry a comment, respectively. They are intended to be subclassed using multiple inheritance, if necessary.

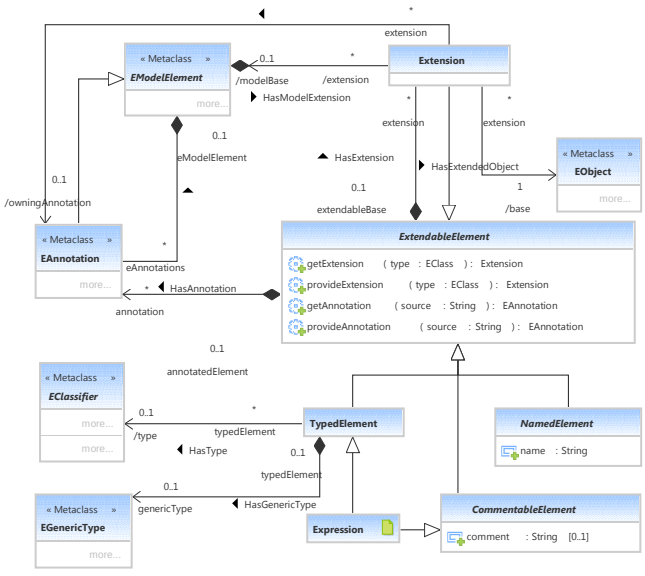

Figure 2: Core of the new Meta-Model

3.2 Story Patterns

The central role in story diagrams is played by *story patterns*, which are essentially in-place model transformation rules. The corresponding package patterns is depicted in Figure 3. Story patterns define object patterns and their modifications. Such structures are specified using ObjectVariables and LinkVariables. ObjectVariables are typed via the classifier attribute that points to an EClass of the underlying class model. LinkVariables are typed over the targetEnd attribute that points to an EReference. In case of bidirectional references, the derived attribute sourceEnd points to the according opposite reference.

Story patterns can be applied to models that contain objects and links which are instances of classes and references of the class model. First, the pattern is matched in the model. When there is a valid matching for the pattern (i.e., the matching is successful) modifications can be applied to the model. Otherwise, the matching fails and no modification is carried out.

The bindingOperator attribute of ObjectVariables and LinkVariables defines whether an element is to be created, deleted, or just matched. If the attribute is set to CHECK_ONLY or DESTROY, the according variables first have to be matched to objects and links in the model. As soon as all these variables have been matched, the model is modified: matched elements with the operator DESTROY are deleted and elements with the operator CREATE are produced. An AttributeAssignment alters an attribute value of an object represented by an ObjectVariable. This happens after the matching and the structural modification are completed.

In addition to the BindingOperator, variables have BindingSemantics. For a successful matching, MANDATORY variables have to exist in the model, while variables marked as NEGATIVE denote objects that must not exist. In contrast, OPTIONAL denotes objects that may exist. For example, a combination of OPTIONAL and CREATE is a compact way to express that an appropriate element will be created if it cannot be matched [7].

Since story diagrams consist of interconnected story patterns, they allow for the reuse of previously matched elements. An ObjectVariable is referenceable by its name. If the bindingState is UNBOUND, the pattern matching algorithm is forced to find a new object, even if the variable was already matched earlier in the story diagram. A BOUND ObjectVariable must have been matched previously. In case a BOUND ObjectVariable was not matched before, the story pattern execution is considered unsuccessful. A MAYBE_BOUND variable is a combination of both: if the variable has been bound before, it is reused; otherwise a new match will be determined.

A special case of an ObjectVariable is an ObjectSetVariable that matches an arbitrary number of objects of the same type. The number of matched objects can be restricted by ObjectSetSizeExpressions.

Path and ContainmentRelation are special link variables. The former is used to denote a connection via a sequence of links determined by a pathExpression. The latter denotes that an object is contained in a collection.

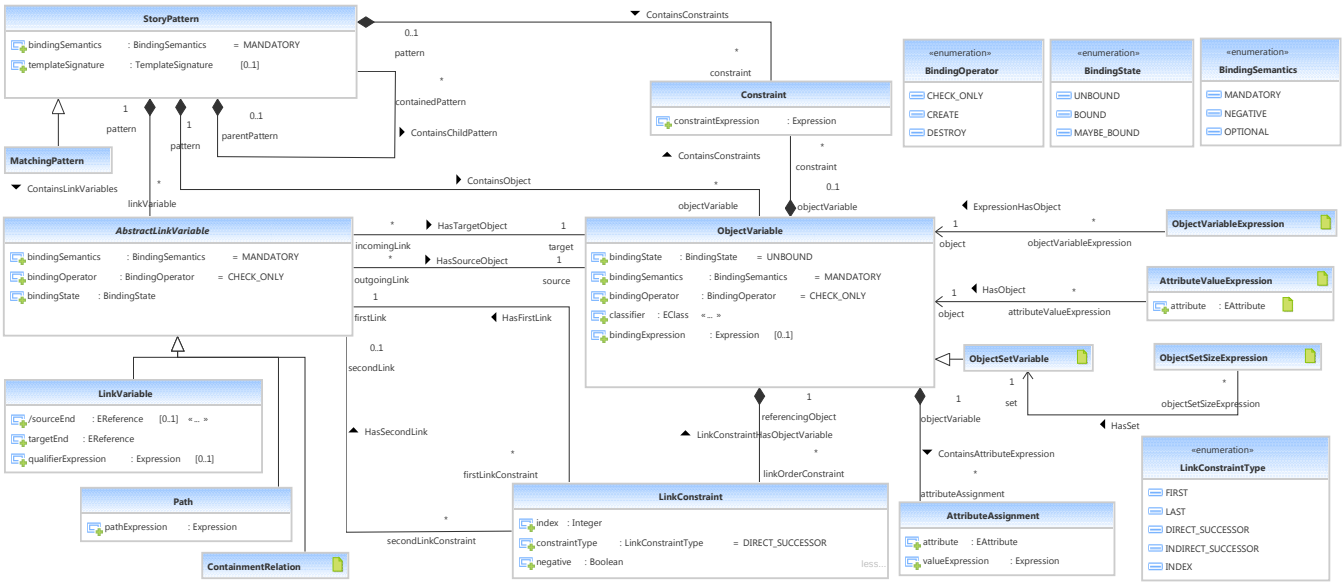

Figure 3: Story Patterns Meta-Model

The matching of a StoryPattern may be further refined using Constraints and LinkConstraints. A Constraint is a boolean expression that must evaluate to true for a matching to be successful. Its context is defined by its container, i.e., variable or pattern. For instance, within an ObjectVariable, you can directly access its attributes, while in a pattern, the ObjectVariable's name must be prefixed. The matching of a LinkVariable whose targetEnd is an ordered list can be constrained using a LinkConstraint. This way, it can be specified that the firstLink must either be the FIRST, LAST or a given INDEX in the list. Furthermore, given two links (firstLink and secondLink), the links' indices could be required to be DIRECT_SUCCESSORs or INDIRECT_SUCCESSORs in the list.

A MatchingPattern is a StoryPattern that is required to be non-modifying, i.e., it must only contain CHECK_ONLY variables and must not have AttributeAssignments. This allows creating side-effect-free story diagrams.

Finally, patterns are allowed to contain subpatterns. Whenever such a subpattern is found in a story pattern, it is matched as a whole. A subpattern may also be NEGATIVE or OPTIONAL. In the former case, the subpattern as a whole must not be found in the model, allowing more expressive negative application conditions. In the latter case, the subpattern is not required to be found. NEGATIVE subpatterns are matched before OPTIONAL subpatterns, but after matching the core (MANDATORY) pattern.

3.3 Activities

We developed a simplified meta-model for activity diagrams which is closely related to the corresponding UML specification. Our meta-model is depicted in Figure 4.

An activity diagram is represented by the Activity class. ActivityEdges connect ActivityNodes to specify the control flow. JunctionNodes are used to split and join the control flow. StructuredNodes are used to build a hierarchical activity diagram by embedding other activity nodes. StatementNodes offer the opportunity to textually specify algorithms with the help of expressions (see Section 3.5). Other activities can be called by means of ActivityCallNodes.

StoryNodes embed a story pattern using the storyPattern reference. To simplify analyses of graph transformations, we distinguish MatchingStoryNodes and ModifyingStoryNodes. While the former are only allowed to match a specified structure, the latter ones are also allowed to perform modifications. A MatchingStoryNode can for example be used to specify an Activity's precondition.

ActivityEdges can have guards, given by the guard attribute and the enumeration EdgeGuard. ActivityEdges with the guards SUCCESS and FAILURE distinguish the cases of (a) successfully executing the story pattern in the source activity node, i.e., completely match and modify the specified structure, and (b) missing to match the complete structure. NONE enforces to choose the ActivityEdge in either case.

Loops can be defined using an activity's forEach attribute. An ActivityEdge with an EACH_TIME guard is chosen for each match of the preceding forEach activity, while an END ActivityEdge is chosen if no such matching can be found anymore.

Boolean guard conditions (BOOL) are specified using the attribute guardExpression. ActivityEdges can also be chosen if an exception is thrown (EXCEPTION). In this case, the exception specified by the ExceptionVariable can be handled by following activity nodes. The activity node that is reached via the FINALLY edge is executed whether an exception is thrown or not.

A story diagram can be used to specify the behavior of an EOperation. We specify this with an OperationExtension which connects an EOperation to an Activity.

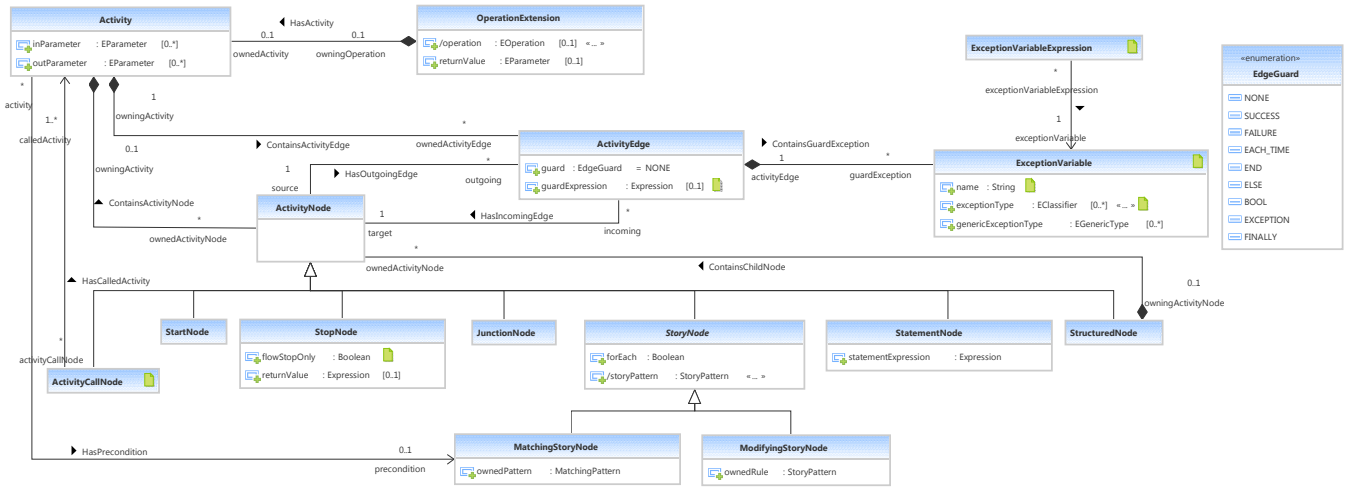

Figure 4: Activities Meta-Model

3.4 Example

Figure 5 shows the concrete syntax of an exemplary story diagram.

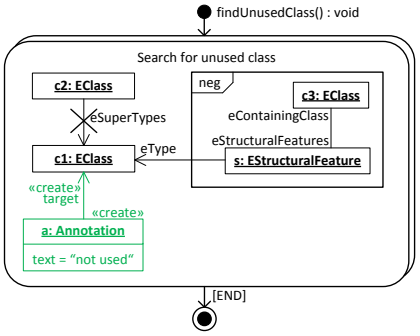

Figure 5: Usage of a Negative Subpattern

The "double" border of the StoryNode denotes a forEach node, i.e., it matches once for every possible matching in the model. The story pattern inside matches when there is a class `c1` which is not used by other classes.

The negative subpattern, denoted by the rectangle labeled with `neg`, is a negative application condition that has to be satisfied for a successful matching. In this case there must not be a structural feature of another class `c3` that references `c1`. Another constraint is given by the crossed-out link which specifies that `c1` must not be subclassed by any class `c2`. If a valid matching for this story pattern is found, the class `c1` is marked as "not used" by a newly created annotation `a`.

The story pattern is applied to each class that satisfies these constraints. Thus, after the execution of this story diagram, all classes that do not have another class using them are marked with a "not used" annotation.

3.5 Expressions

Although story diagrams are an expressive language, in some cases textual languages are better suited and more compact, e.g., for complex calculations or regular expressions.

Therefore, recent SDM tools allow to embed Java code in story diagrams. When generating code for a story diagram, the embedded Java code is included in the resulting code. As a consequence, the embedded code cannot be checked at modeling time (e.g., no type checking or model checking) and interpreting story diagrams that contain arbitrary Java code is hardly possible.

To improve this situation, the SDM community decided to explicitly model textual expressions in story diagrams. The expressions, on the one hand, still allow to embed textual languages like Java and OCL and, on the other hand, enable interpretation and type checking for most of them.

Our meta-model separates two cases: Either an arbitrary textual expression is represented as String in the class TextualExpression or the expression is modeled explicitly by building an abstract syntax model of the expression. In the former case, arbitrary code can be embedded for code generation, but comes with the cost of missing opportunity to analyze the expression. In the latter case, the expression model is more complex, but can be type-checked.

With our meta-model, we try to cover most common expressions in story diagrams and propose to explicitly model these to enable type checking at least for these cases. Examples for such expressions are matching constraints in story patterns or assignments of a certain value to an object's attribute.

Our story diagrams meta-model supports literals like `7`, `3.1`, `true`, or `"xy"` whose type is explicitly given (EDataType). Furthermore, we support logical expressions, arithmetic expressions, and comparing expressions. The expressions with an operator combine other expressions to build more complex expressions.

In addition, we allow for building expressions that represent an object variable in a story pattern, the value of one of its attributes, or the number of objects matched to an object set variable. Furthermore, method calls (MethodCallExpression, Figure 6), which are explained in the next section, can be modeled, too.

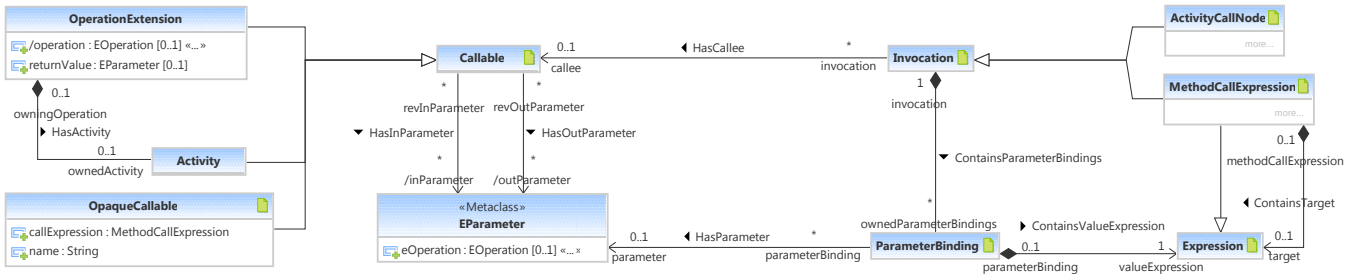

Figure 6: Calls package of the Meta-Model

3.6 Activity and Method Calls

The calls package of the new meta-model supports the invocation of so-called Callables directly from story diagrams (cf. Figure 6). Callables are Activities (i.e., other story diagrams), operations (represented by the wrapper class OperationExtension, which references an EOperation), and OpaqueCallables. EOperations are part of the model while OpaqueCallables are not represented in the model, but may, for example, be part of a library. A Callable can have in- and out-parameters as indicated by the two references from Callable to EParameter. While the number of parameters is unbounded in general, OperationExtensions and OpaqueCallables may only have one out-parameter. In contrast, Activities can have arbitrarily many out-parameters. The same object may be used as in-parameter and out-parameter, thereby emulating the in-out-parameters from other transformation languages like QVT.

Callables can be invoked by Invocations which can either be ActivityCallNodes or MethodCallExpressions. ActivityCallNodes are special ActivityNodes which can be used in story diagrams to represent the call of another story diagram. MethodCallExpressions represent the invocation of a method, i.e., either an EOperation or an OpaqueCallable. The target of a MethodCallExpression can be determined by an Expression which can, for example, be a variable or the result of another method invocation. Every Invocation must have a number of ParameterBindings that assign concrete arguments to the callee's parameters.

The calls package also provides a concept for the polymorphic dispatching of calls which is omitted here due to space restrictions. Details can be found in [2].

4. CONCLUSIONS AND FUTURE WORK

We presented the new common meta-model for story diagrams which was developed in a joint effort of the SDM community. It is the foundation for future projects as it provides a common basis for developments and facilitates the interoperation of SDM tools. In comparison to the different previous models, it especially simplifies static type checking due to the explicit modeling capability for expressions.

To facilitate the execution of story diagrams, it is necessary that the existing code generation and interpretation approaches are adapted to the new meta-model. This will be imperative for the development of SDM tools. In addition, all existing editors and tools will have to be adapted accordingly.

In this paper, we focused mostly on the abstract syntax of the story diagram meta-model and the semantics of some of the newly integrated features. While the concrete syntax of ActivityCallNodes has been defined in [2], a concrete syntax for other new elements still has to be defined in future works.

Acknowledgments

We would like to thank all other participants of the SDM unification task force for their ideas and contributions to the new meta-model: Steffen Becker, Stephan Hildebrandt, Ruben Jubeh, Elodie Legros, Carsten Reckord, Andreas Scharf, Christian Schneider, Gergely Varró, and Albert Zündorf.

5. REFERENCES

[1] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based tool integration with MOFLON. In *ICSE '08 Proceedings*, pages 807–810. ACM, 2008.

[2] S. Becker, M. von Detten, C. Heinzemann, and J. Rieke. Structuring Complex Story Diagrams by Polymorphic Calls. Technical Report tr-ri-11-323, University of Paderborn, Mar. 2011.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT '98 Selected Papers*, volume 1764 of *LNCS*, pages 296–309. Springer, 2000.

[4] L. Geiger, T. Buchmann, and A. Dotor. EMF Code Generation with Fujaba. In *Fujaba Days '07 Proceedings*, 2007.

[5] H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In *GT-VMT '09 Proceedings*, volume 18 of *Electronic Communications of the EASST*, 2009.

[6] C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch, and W. Schäfer. Fujaba4Eclipse Real-Time Tool Suite. In *MBEES '07 Revised Selected Papers*, volume 6100 of *LNCS*, chapter 12, pages 309–315. Springer, 2009.

[7] S. Rose, M. Lauder, M. Schlereth, and A. Schürr. A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In *Model-Driven Domain Analysis and Software Development: Architectures and Functions*, pages 90–113. IGI Publishing, 2011.

[8] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *WG '94 Proceedings*, volume 903 of *LNCS*. Springer, 1994.