Temporal Graph Queries to Support Software Evolution*

T. Rötschke and A. Schürr

Real-Time Systems Lab
Darmstadt University of Technology
{rotschke|schuerr}@es.tu-darmstadt.de

Abstract. Graph transformation techniques have already been used successfully by several research groups to support re-engineering of large legacy systems. Where others often aim at transforming the system to improve it, we advocate an evolutionary approach that embeds transformations within the ordinary development process and provides tool support to monitor the ongoing progress regularly. In this paper, we discuss how temporal graph queries based on Fujaba story diagrams can provide a natural means to express trend-oriented metrics and consistency rules that we identified in our industrial case studies. To this end, we discuss a first-order logic rather than operational interpretation of a graph queries and show how well-known temporal logic operators can be added to express rules over consecutive states of the same instance graph.

1 Introduction

After successfully developing and enhancing a software intensive product for many years, developers often realize that it becomes harder and harder to add new features to the system. Subsequent reverse engineering usually shows that the software architecture needs some major restructuring of the existing code. In some cases, these restructurings can be quickly performed by automated re-engineering tools. In other cases, there are technical obstacles or rigid development processes prohibiting the use of transformation tools. Manual restructuring of the complete system however can delay the development of new features for several weeks or months, which is not acceptable from an economic point of view.

Under these circumstances, less invasive tool support can provide means to achieve long-term goals such as restructuring without sacrifying business-critical short term goals. The idea is to improve the software architecture in small steps as part of the ordinary development process and monitor the progress towards long-term goals with specially designed analysis tools. In our approach, such tools can be generated from domain-specific meta models describing architectural concepts, design tool data and source code structure as well as appropriate graph transformations and queries defining metrics and consistency rules.

* Work supported in part by the European Community's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis

Naturally, the initial situation is usually pretty bad in the sense, that it is quite different from the desired situation. When the existing problems cannot be solved at once or their dynamics are not yet fully understood, a first goal might be that things don't become worse than they already are. In later stages of the restructuring process, improvements should take place at a certain pace to ensure that the long-term goals are finally met. Hence, our analysis tools focus on trends in metric values and rule violations rather than absolute numbers.

Consequently, we came to the conclusion, that it would be beneficial to be able to specify rules with respect to analysis data of the same system sampled at different points of time. One could think of the following kind of temporal rules:

- A file with less than 1000 lines of code (LOC) *now* should have less than 1000 LOC *henceforth*.
- If a file includes a header file, but its block does not see an adequate exporting interface *now*, the block has *never* seen such an interface.
- *After* the next milestone (01-05-07), the number of invalid includes must be reduced by 100 every week.
- Each block must provide at least one interface *before* the next release.

During our research, we found out that temporal logic provides a means to express a number of these rules we had in mind. However, we had to investigate, how temporal logic and graph transformation fit together. As we point out in Section 5, some work had already be done to bring graph transformation and time together, but did not fully serve our purposes. The main contribution of this paper is the definition of a temporal graph query language derived from story diagrams used by Fujaba [1]. To define the semantics, we construct a pair grammar [2] mapping temporal graph queries on equivalent temporal first order logic formulae.

The rest of the paper is structured as follows: In Section 2, we introduce an example that is referred to throughout the paper. In Section 3, we introduce a subset of Fujaba story diagrams referred to as *graph queries*. Next, in Section 4, we show how well-established time operators from temporal logic can be added to graph queries so that they can be interpreted by temporal first order logic. Section 5 discusses related work with particular emphasis on graph transformation-based re-engineering and graph transformation considering time. We summarize our results in Section 6, and provide some insight into ideas to continue this work.

2 Running Example – Motivated by Case Study

Throughout the paper, we use a simple example motivated by a more complex industrial case study described in [3]. For this example, we assume that we start with an existing system programmed in C. The project started quite small more than a decade ago, with only a couple of developers. Back then, nobody anticipated the growth of the system and hence no-one cared about "architecture". In typical C-style, the system was divided into several libraries with globally

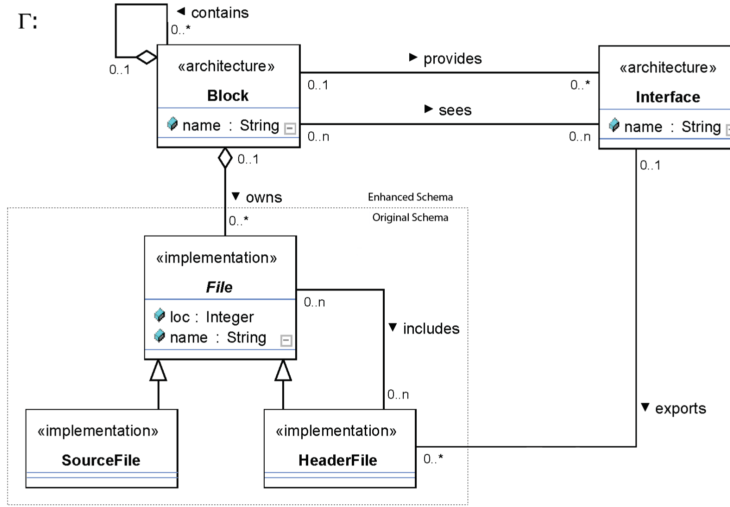

Fig. 1. Simple Graph Schema for Re-engineering Example.

available header files. Over the years, the system has become bigger and now consists of several million lines of code, which are maintained and improved by more than 200 developers.

Due to the complexity of the dependencies, it has become more and more difficult to modify the existing code without breaking something. As a solution, the software architects defined, in their terms, architecture concepts – actually a proprietary module interconnection language – consisting of a nested building blocks and associated export interfaces. Fig. 1 presents a graph schema Γ[1] for these concepts and shows the dependencies between new *architecture* and already existing *implementation* concepts.

Given the right parsers, which are out of scope of this paper, it is possible to periodically create an instance graph of this schema representing the architecture and the related implementation at the time of analysis. Existing Graph queries (and transformations) can be defined to check (and transform) each single instance with respect to certain consistency rules. As one goal was to avoid *new* violations in the architecture, we found that we would like to have rules that enabled us to reason about certain objects in *different* instance graphs. The contribution of this paper is to propose the syntax and semantics of a temporal language extention for graph queries to make this possible. The exact implementation of these queries is out of the scope of this paper, but some basic ideas are sketched in Section 4.3.

The case study discussed in [3] involved several metrics and 23 consistency rules which used to be checked every night. The system consisted of 500 building blocks, 8000 files and 40000 include relationships. Most of the analysis tool

[1] A formal definition for graph schemas can be found in [4]

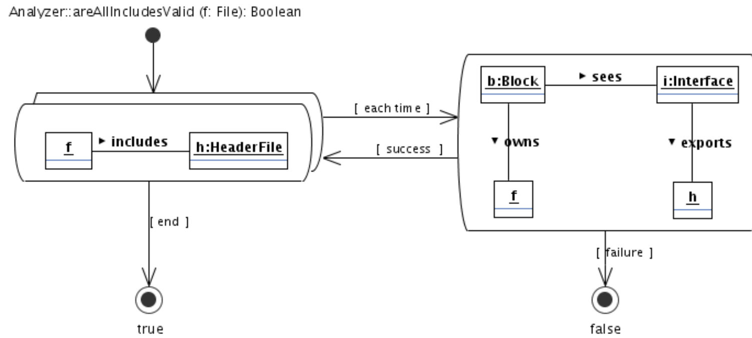

Fig. 2. Consistency Rule for All Included Header Files.

was hand-coded and implemented with the specific system in mind. The code for checking consistency rules however, was already generated from Fujaba 3 specifications, with a strong emphasis on changes in metrics and consistency violations, rather than trying to interpret absolute figures.

Ever since, our research effort has been put into generalizing the approach to make it easily available to other development projects as well. Recently started case studies indicate that our approach will be useful in other industrial application areas as well. This research is performed as part of the MOFLON project [5], which is a MOF 2.0-based meta modeling framework on top of the Fujaba Toolsuite [1]. Hence, our suggestions for temporal graph queries are based on the Story Driven Modeling (SDM) language, which is the graph transformation language featured by Fujaba.

3 First Order Logic Interpretation of Graph Queries

In the following, we describe the systax and semantics of a subset of SDM referred to as "graph queries". As we restrict ourselves to queries, the semantics can be descibed by well-known first order logic (FOL) formulae. First we discuss an example query that serves as consistency rule and discuss its logic interpretation in Section 3.1. Then we approach the issue more systematically in Section 3.2 and show by construction, how any FOL formula can be translated into a normalized graph query. We omit formulae for non-normalized graph queries, as they would be too complex considering the size restrictions of a paper. However, we discuss in Section 3.3, under which conditions more user-friendly story diagrams can be translated into FOL formula.

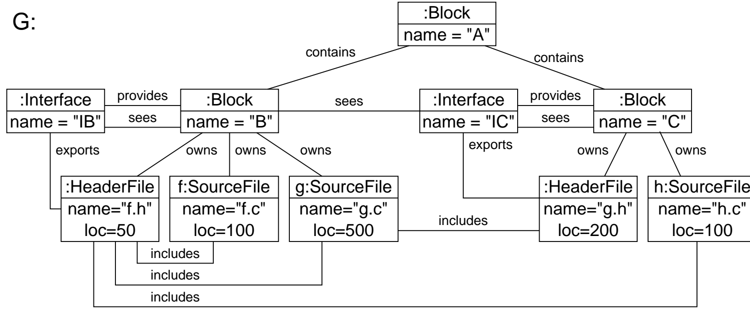

Fig. 3. Instance Graph for Schema in Fig. 1.

3.1 Example Interpretation

Fig. 2 shows a Fujabagraph query representing a consistency rule to check whether all[2] outgoing *include* relationships between a given file f and header files h are correct with respect to the architecture due to an adequate *sees* relationship between the related block b and the interface i.

$$
\begin{aligned}
&\text{areAllIncludesValid}(f) \Leftrightarrow \forall h(\\
&\quad (\text{type}(h, \text{HeaderFile}) \wedge \neg(f \equiv h) \wedge \text{rel}(f, \text{includes}, h)) \\
&\quad \rightarrow \exists b \exists i (\text{type}(b, \text{Block}) \wedge \text{type}(i, \text{Interface}) \\
&\quad \wedge \neg(b \equiv f) \wedge \neg(b \equiv h) \wedge \neg(b \equiv i) \wedge \neg(f \equiv h) \wedge \neg(f \equiv i) \wedge \neg(h \equiv i) \\
&\quad \wedge \text{rel}(b, \text{owns}, f) \wedge \text{rel}(b, \text{sees}, i) \wedge \text{rel}(i, \text{exports}, h))).
\end{aligned} \tag{1}
$$

Equation 1 is a FOL interpretation of this query, where type, rel and eval are predicate symbols and attr is a function symbol with the following interpretation:

- *type(x,T)*: object x is an instance of type T.
- *rel (x,E,y)*: between objects x and y exists a link of association E.
- *eval (x, a, Ω, e)*: the value of attribute a of object x compares to expression e by operator $\Omega \in \{=, \neq, <, \leq, \geq, >\}$.
- $x \equiv y$: x is the same object as y.

Let G_Γ be an instance graph consistent with the schema Γ. A valuation for a given G is a function $\mathcal{A}_G : \bar{A} \longrightarrow \{0, 1\}$, where $\bar{A}$ is the set of FOL formulae. As a shorthand, we write $G \models F$ for $\mathcal{A}_G(F) = 1$. Fig. 3 represents an instance graph G consistent with the schema Γ introduced in Fig. 1. As one can easily reproduce, the following holds: $G \models \text{areAllIncludesValid}(f)$, $G \models \text{areAllIncludesValid}(g)$, and $G \not\models \text{areAllIncludesValid}(h)$.

[2] Indicated by the doube-boxed "forall" activity.

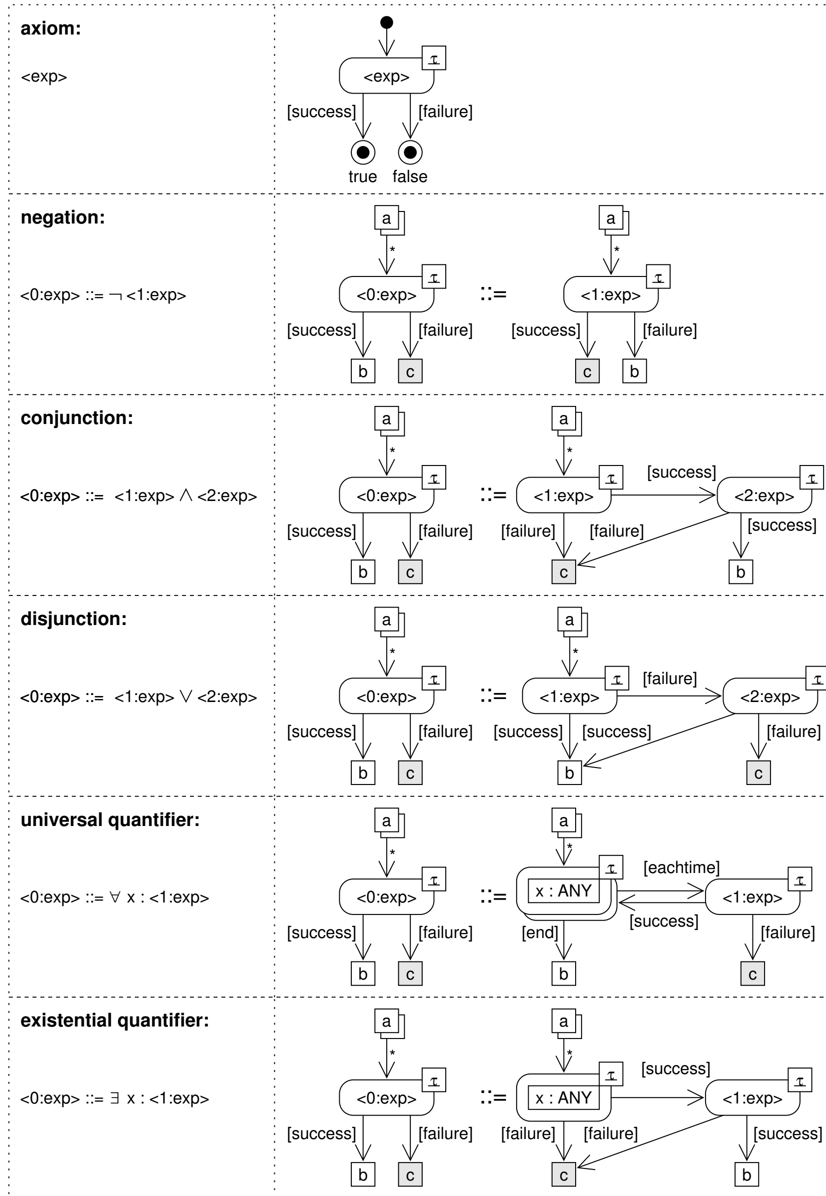

Fig. 4. Pair Grammar for First Order Logic Interpretation of Graph Queries.

3.2 From Formulae to Graph Queries

As we have seen, it is possible to interpret the graph query in Fig. 2 as a FOL formula, but is there an equivalent graph query for any given FOL formula? Indeed,

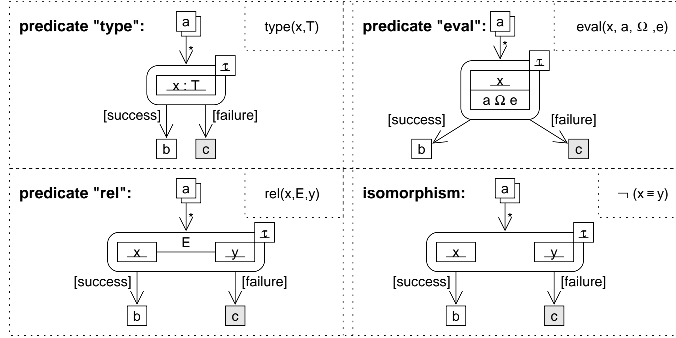

Fig. 5. Pair Grammar Right Hand Sides of Atomic Rules.

we were able to construct a graph query language comprising a useful subset of Story Diagrams defined in [1], which essentially are UML activity diagrams containing visually notated transformation rules of the instance graph. To this end, we defined the pair grammar shown in Fig. 4 and 5. Pair grammars have been invented by Pratt for the precise specification of text-to-graph and graph-to-text translations [2]. For this purpose, they combine a string and a graph grammar which derives corresponding string and graph sentences simultaneously. Note, that we define the pair grammar based on the concrete rather than the abstract syntax of SDM which would be correct, but less readable.

Each row in Fig. 4 contains one rule that is divided into two sub rules that are applied simultaneously. The left sub rule describes the construction of the FOL formula, while the right sub rule describes the construction of the related graph query. Non-terminals are denoted by angle brackets (e.g. <exp>). Corresponding non-terminals in both sub rules have the same name.

In the right sub rule, non-terminals are embedded in rounded boxes representing *activities*. We always consider the edge context of the activity extended by the rule and embed activities on the rule's right hand side in the context of the left hand side. Every activity may have any number of incoming transitions with arbitrary labels. The incoming edge context is marked by a box named a. Apart from *forall*-activities, each activity has exactly one outgoing *success* and *failure* transition. The success edge context is named b and the failure edge context c. As every word in this grammar is derived from the first rule, the resulting story diagram has one start activity, exactly one stop activity marked with *true*, and exactly one stop activity marked with $false$.

Fig 4 contains definitions of formulae containing the classical FOL operators negation, conjunction, disjunction, universal quantifier and existential quantifier. Further operators like implication and equivalence can be derived according to the usual substitutions. The *forall* activity is used to reflect the universal quantifier and must have exactly one outgoing *eachtime* and *end* transition instead of *success* and *failure*.

Fig 5 defines the atomic propositions that might occur in a graph query, namely node matching, edge matching, attribute evaluation and isomorphism. Rectangular boxed inside activies represent nodes of the instance graph. Note that this figure only contains the right hand side of each sub rule with the textual part in the upper right corner. The missing left-hand sides are the same as in Fig. 4 as e.g. for the rule "negation".

To save some space, the rules are already prepared for the temporal extensions proposed in Section 4. Therefore, each activity carries a temporal expression τ. In the case of static graph queries, i.e. without temporal operators, the temporal expression can be ignored.

Using this pair grammar, we can parse any FOL formula[3] using the context-free string grammar rule. We also can compute the equivalent depiction of an SDM graph using its corresponding graph grammar rules.

3.3 From Graph Queries to Formulae

As we have argued in Section 3.2, we can construct a graph query for every FOL formula. Obviously, we can do the reverse, and construct a FOL formula for any graph query that can be constructed by the grammar. However, Story Diagrams usually contain a relative small number of Story Patterns consisting of multiple objects, links and attribute-value pairs. We consider these story diagrams as shorthands for those that can be created by the pair grammar.

Because of size restrictions, we can not provide a tranformation system that is able to translate each "nice" story diagram into a normalized graph query to be able to find the FOL formula. This system would also translate additional features such as negative nodes, negative edges and optional nodes[4].

Though we do not provide the complete transformation, we at least want to provide an example how the graph query in Fig. 2 looks like in the normalized form. Fig. 6 represents this equivalent graph query. To keep the diagram layout clear, we merged some failure transitions using gray circles which are not actually part of the concrete syntax of our graph queries. The normalized form is obviously much more complex than the original form. On the other hand, it is easy to verify that the normalized form corresponds to formula 1.

Note, that the first order logic interpretation of graph queries does only work, if the operational semantics of story diagrams is modified with respect to the original definition in [1]: Atomic propositions contained in the same activity are connected by conjunction. Transitions between activities however effectively describe a *cut* as for instance in Prolog. Thus, if the selection of possible matches is non-deterministic, a story pattern might have different valuations even on the same instance graph, depending on the actual selection order.

---

[3] Rules for parentheses and precedence have been omitted due to lack of space.

[4] Set nodes may not be used in these graph queries, as second order logic is required to describe their semantics, which would complicate things unnecessarily and are of little use in graph queries: If a set node can be matched, so can be a single node from the set and the result of the formula is the same.

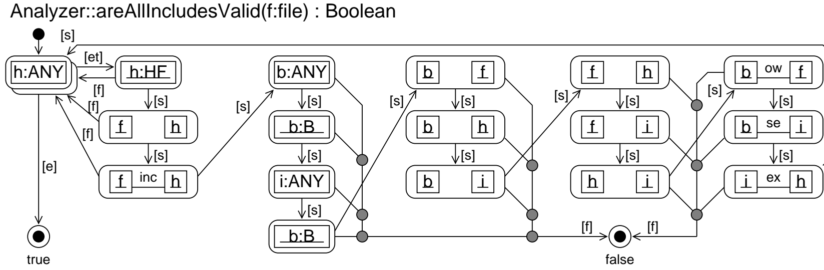

Fig. 6. Equivalent Normalized Graph Query for Fig. 2.

To solve this problem, the operational semantics of the graph queries proposed in this paper includes backtracking as in PROGRES [4]. Only then all possible matches are tried if necessary and the graph queries can be fully described by FOL formulae. Minor changes to the operational semantics are necessary to allow objects with unspecified type *ANY* and to permit nesting of forall-activities with outgoing success and failure transitions from enclosed activities. Syntactically, our graph queries are distinguished by the attached temporal expressions.

4 Adding Time to Graph Queries

Having defined how FOL formulae can be expressed as graph queries, we now extend the definition to include temporal operators. In Section 4.1, we elaborate on the concept of time used in our approach. Next, we enhance the pair grammar from Section 3 with unary temporal operators.

4.1 The Notion of Time

In this paper we have to deal with different concepts of time. As we make use of temporal logic we need to treat time as an infinitive sequence of states, i. e. instance graphs in our case. Naturally, these states coincide with measurements of the system under analysis. However, queries might be expressed in terms of "real" time, and the specificator might not be aware of the corresponding states. In the Unix-world, time is expressed as an 64-bit natural number $t \in \mathbb{T} = \mathbb{N} \cap [0; 2^{64} - 1]$, being the number of seconds that have passed since January 1st, 1970. This concept of time would serve our purpose well.

Fig. 7 provides an overview, how the different concepts of time correspond to each other. In this example, we assume that the initial measurement takes place on Monday, March 29th of any given year after January 1st, 1970. Consecutive measurements are scheduled once for each working day. Thus, the gaps between April 2nd and April 5th as well as April 9th and April 12th indicate weekends. On April 8th, there is an exceptional gap, e.g. because of a power failure. On

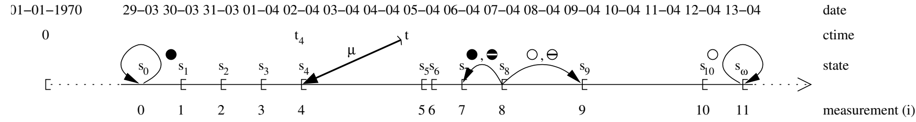

Fig. 7. Time line for Temporal Graph Queries.

April 5th, there is an additional measurement, e.g. because some changes had been applied to the analysis scripts this day.

Let $\mathcal{V}$ be a set of atomic propositions as defined in Fig. 5. Each measurement i taken at a certain point of time $t_i \in \mathbb{T}$ corresponds to a state s_i, which is a mapping $s_i : \mathcal{V} \longrightarrow \{0, 1\}$. The *Temporal Structure* for $\mathcal{V}$ is an arbitrary long, but finite sequence $\mathrm{T} = (s_0, s_1, \ldots, s_\omega)$.

As indicated by the brackets on the time axis, a state s_i is valid from the moment of the measurement to the second before the next measurement at t_{i+1}. The initial state s_0 is valid by definition between January 1st, 1970 and $t_1 - 1$. As queries may refer to any point of time $t \in \mathbb{T}$, but the corresponding state might be unknown a priori, we define a mapping function $\mu : \mathbb{T} \longrightarrow T$, where

$$\mu(t) = \begin{cases} s_0 & : \quad t < t_0 \\ s_\omega & : \quad t \geq t_\omega \\ s_i & : \quad t_i \leq t < t_{i+1} \end{cases}$$

As we are mainly interested in changes of instance graphs over time, pretending that the unknown situation before the first measurement was always the same as at t_0 is more useful than assuming that an empty instance graph existed before that time. With the same argument, we project the last known state s_ω for eternity into the future. As a consequence, queries referring to future states might produce different results depending on the current s_ω at query time.

4.2 A Pair Grammar for Temporal Graph Queries

As shown in Fig. 8, we extend the pair grammar with additional rules to cover unary temporal logic operators. As in Fig. 5, we only provide the right hand sides of both sub rules. The left column defines future time operators, while the right column defines past time operators. Binary operators like "until", "unless", "atnext", "before" are not yet supported but could be added easily. An nice summary of first order temporal logic and related operators can be found in [6].

When designing the extensions, we tried to keep as close to ordinary story diagrams as possible. The basic idea is to provide each activity with a time object, i.e. a state $\tau \in T$, where T is the temporal structure defined in Section 4.1. The time object is treated in a way similar to regular objects.

The *henceforth* and *hitherto* operators can be seen as universal quantifiers over unbound time variables and hence are visualized by double boxes around the time expression. Therefore, we use the same pattern with *eachtime* and

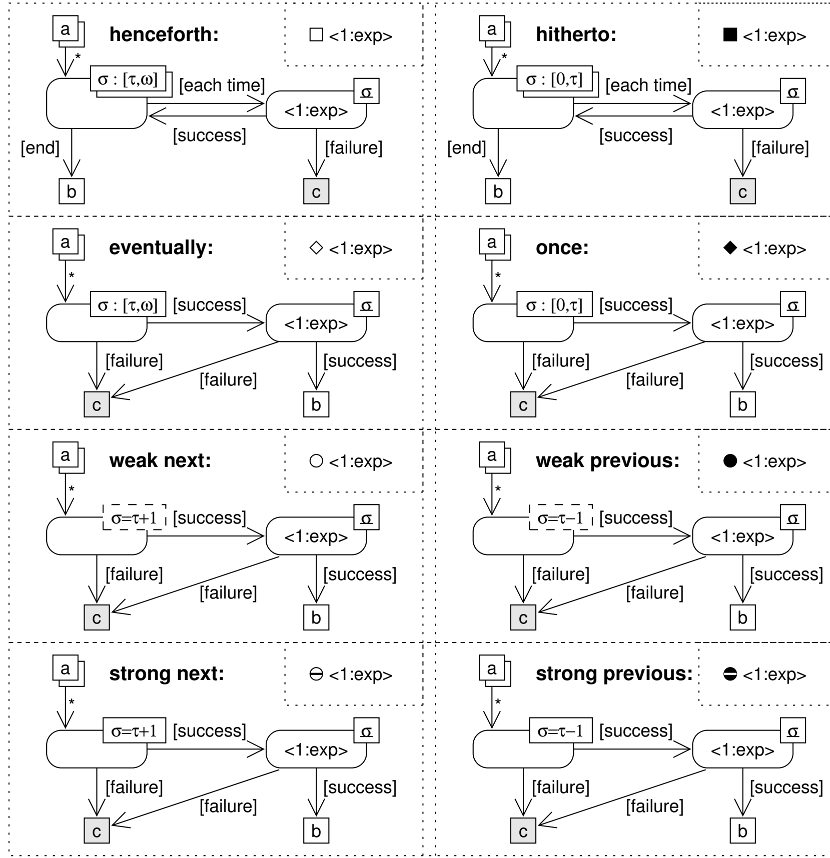

Fig. 8. Pair Grammar Right Hand Sides for Temporal Operators.

end transitions as for universal quantifiers. The quantified time variable σ is successively bound to τ and, depending on the operator, to every state defined in T before or after τ.

The *eventually* and *once* operators are consequently treated as existential quantifier over unbound time variables. The time variable σ is successively bound to every state in T. If a consecutive story pattern fails, backtracking is used to assign another state. If there is no state so that consecutive story patterns match, the rule finally fails.

The *next* and *previous* operators also define a new time variable σ, but immediately assign a value based on the enclosing variable τ. As shown in Fig. 7, the *next* and *previous* operators refer to states rather than points in time. Note that our temporal structure is not infinite as usual, because there is always a last state σ_ω. The original definition of the *weak previous* operator, i.e. that $\bullet A = 1$ at the inital state s_0 for any given formula A, is also problematic: The value

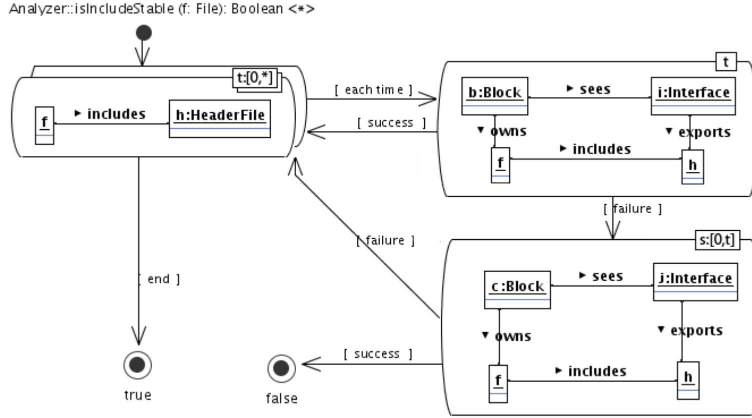

Fig. 9. Example for Temporal Graph Query.

"1" means that the related story pattern succeeds, but the new time variable σ cannot be bound although it is used in consecuting story patterns.

We solve this problem by the following deviations from ordinary temporal logic: We define *strong* and *weak previous* and *next* operators with symmetric semantics. To distinguish between *weak* and *strong* operators, we adopted the notation for *optional* and *obligate* nodes of Fujaba story diagrams. For instance, *strong next* can only be true, if there really is a next state, i.e. $s_\tau \neq s_\omega$. *Weak next* in contrast, is also defined for $s_\tau = s_\omega$. In this case, σ is bound to the last known state ω. Weak operators are visualized by dashed boxes.

For convenience, real time expressions like "τ = '01-05-07'" or "$\sigma = \tau +$ 20 days" may be used. They are implicitly transformed into states using the mapping function μ defined in Section 4.1. Besides, normalized temporal graph queries as defined by our pair grammar can be condensed by defining activities containing more than one atomic proposition. Again, we have to skip the exact transformation rules.

4.3 Example Revisited

Fig. 9 represents a non-normalized temporal graph query based on the static query in Fig. 2. Rather than checking whether there is a valid *sees* relationship for every *include* relationship, the rule ensures that valid include relationships never become invalid. As a result, only changes after the initial measurement are taken into account and *deterioration* of the system is detected as soon as it occurs so that it can be quickly countered.

The temporal graph query contains the equivalent of the future time operator *henceforth* and the past time operator *once*. The free time variable t is sequentially bound to all states in T. For each t, the remaining part of the first story pattern is evaluated. For every match, the second story pattern is evaluated for

the current state t. If the second story pattern matches for the given t, the next iteration begins with the following state in T. If the second story pattern fails, the third story pattern is evaluated. It has the same body as the second, but represents a *once* operator. If s can be bound to any state up to the current t so that the third pattern succeeds, the whole query fails. If there is no possible match for any s, the outer iteration continues.

Note that this procedure does not necessarily describe the actual implementation. Our idea is to equip every graph element with a *created* and *deleted* timestamp so that we are able to avoid iterations in many cases. However, a detailed discussion of the implementation is out of the scope of this paper.

5 Related Work

The idea of applying graph transformation in the field of software re-engineering has already been proposed by others. There have been several approaches to re-engineer industrial software systems based on PROGRES [7]. Another example is the Varlet project [8] that deals with re-engineering of information systems. To our knowledge, all graph transformation-based approaches deal with snapshots of the analyzed systems and hence do not require language features that allow to consider time.

There are some approaches that add time aspects to graph transformations. In [9], a notion of time is added to graph transformations which is adapted from time environment-relationship nets. The approach introduces logical clocks and allows the user to specify the age of nodes in the instance graph and durations for transformations so that the age of involved nodes can be updated. Time can be used to determine the order of productions in transformation sequences so that the age of related nodes does never increase in the course of its transformations. However, this approach allows to reason about *durations* rather than to reason about multiple instance graphs at different times as we do in this paper. The same holds for real-time automatons as described for instance in [10].

The contribution related closest to this paper is described in [11, 12]. These documents introduce graphically denoted temporal invariants to graph transformations. In this approach, states are the result of runs, i.e. sequences of atomic graph transformations. Among other things, the temporal invariants allow to specify the correct order of transformations if the choice of transformations is non-deterministic. The main differences are, that we completely embed temporal operators into graph queries while they are expressed as constraints in the context of concrete matches in [11]. Before the extension, the former already require first order logic to describe. Hence we need to support temporal first order logic, while propositional logic suffices for the latter. Besides we provide a more explicit and complete definition of the syntax of temporal graph queries.

There are other approaches that allow to visually define constraints. For instance, [13] provides a visual representation of arbitrary OCL expressions. The operators "not", "implies", "or", etc. are represented as nested sub diagrams with the corresponding keywords as attachments, i.e. the structure of OCL ex-

pression (first order logic expression) is preserved, in contrast to our approach that avoids nesting of boxes in favour of the usage of "flat" control flow diagrams. There is no support for temporal logic expressions. An alternative for graphical FOL constraints are spider diagrams [14] that provide no support for temporal operators either.

In [15, 16], temporal extensions for OCL are described. The former work allows to specify the behavior of business software components using linear time logic in OCL syntax including operators such as initially, until, etc. It is a straightforward extension of OCL with just the temporal operators added. The latter provides a computational tree logic extension of OCL used for the definition of temporal invariants of state charts to model the behavior of real-time systems. Both approaches rely on a textual definition of temporal logic formulas and could not be integrated with story diagrams without destroying the essential property of story diagrams of being a visual notation.

While many papers deal with textual notations for propositional temporal logic, there are some that also discuss first order temporal logic. The main problem is that first order logic might lead to an infinite number of models for the formulae that have to be considered. In our case, the number of models is always finite, as our temporal structure is defined by an arbitrary large though finite sequence of measurements. We would like to mention the work described in [6], where a normal form for first order temporal logic is suggested. We are currently investigating, if this normal form provides a better means to reason about the differences in expressive power of temporal graph queries and first order temporal logic.

6 Conclusions and Future Work

In this paper, we have motivated why temporal graph queries can be beneficial in the area of evolutionary software restructuring. As main contributions of this paper, we provide a temporal first order logic interpretation of a well-defined subset of temporal graph queries by means of a pair grammar. We further define a mapping function to translate real time expressions into states of the underlying temporal structure. Equipped with these new concepts, we are able to specify visual consistency rules over a temporal sequence of instance graph, i.e. models of the analyzed system.

There are several possible continuations of the work described in this paper: First, we need to broaden the set of graph queries, that we are able to construct FOL formulae for, ideally so that we can support at least all features that are already available in story diagrams. Next, we have to integrate the temporal graph queries into our meta modeling framework, MOFLON, and therefore enhance our editor and code generator. While doing so, we will have to turn special attention on the compact representation of graph instances, probably based on versioned graphs, as we have to deal efficiently with large amounts of data. Finally, we intend to applying our approach to new industrial case studies and learn how we can benefit from temporal graph queries in practice.

References

1. Zündorf, A.: Rigorous Object Oriented Software Development. University of Paderborn (2001) Habilitation Thesis.
2. Pratt, T.W.: Pair Grammars, Graph Languages and String-to-Graph Translations. Journal of Computer and System Sciences **5** (1971) 560–595
3. Rötschke, T.: Re-engineering a Medical Imaging System Using Graph Transformations. In: Applications of Graph Transformations with Industrial Relevance. Volume 3062 of LNCS., Springer (2003) 185–201
4. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2. World Scientific Publishing (1999) 487–550
5. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In: Proc. European Conference on Model-Driven Architecture. (2006) Accepted for publication.
6. Dixon, C., Fischer, M., Barringer, H.: A Graph-Based Approach to Resolution in Temporal Logic. In Gabbay, D.M., Ohlbach, H.J., eds.: Temporal Logic, ICTL '94. Number 827 in LNAI, Springer (1994) 415–429
7. Cremer, K., Marburger, A., Westfechtel, B.: Graph-Based Tools for Re-Engineering. Journal of Software Maintenance **14**(4) (2002) 257–292
8. Jahnke, J., Wadsack, J.: Integration of Analysis and Redesign Activities in Information System Reengineering. In: Proc. of the 3rd European Conference on Software Maintenance and Reengineering (CSMR'99), IEEE Press (1999) 160–168
9. Gyapay, S., Heckel, R., Varró, D.: Graph Transformation with Time: Causality and Logical Clocks. Fundamenta Informaticae **58**(1) (2003) 1–22
10. Burmester, S., Giese, H., Hirsch, M., Schilling, D., Tichy, M.: The FUJABA Real-Time Tool Suite: Model-Driven Development of Safety-Critical Real-Time Systems. In: Proc. 27th ICSE, ACM Press (2005) 670–671
11. Koch, M.: Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems. PhD thesis, TU Berlin (1999)
12. Gadducci, F., Heckel, R., Koch, M.: A Fully Abstract Model for Graph-Interpreted Temporal Logic. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Proc. 6th International Workshop on Theory and Application of Graph Transformation (TAGT'98). Number 1764 in LNCS, Springer (2000) 310–322
13. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: A Visualization of OCL Using Collaborations. LNCS **2185** (2001) 257–271
14. Howse, J., Molina, F., Taylor, J.: On the Completeness and Expressiveness of Spider Diagram Systems. In: [17]. (2000) 26–41
15. Conrad, S., Turowski, K.: Specification of Business Components Using Temporal OCL. In Favre, L., ed.: UML and the Unified Process, IRM Press/IDEA Group Publishing (2003) 48–65
16. Flake, S., Müller, W.: An OCL Extension for Real-Time Constraints. In Clark, T., Warmer, J., eds.: Object Modeling with the OCL, The Rationale behind the Object Constraint Language. Volume 2263 of LNCS., Springer (2002) 150–171
17. Anderson, M., Cheng, P., Haarslev, V., eds.: Theory and Application of Diagrams, First International Conference. In Anderson, M., Cheng, P., Haarslev, V., eds.: Diagrams 2000. Volume 1889 of LNCS., Springer (2000)