TECHNISCHE UNIVERSITÄT BRAUNSCHWEIG

INFORMATIK-BERICHTE

93–07

Monitoring Temporal Preconditions in a Behaviour Oriented Object Model

Scarlet Schwiderski

Computer Laboratory
University of Cambridge
Cambridge, CB2 3QG, UK

Thorsten Hartmann
Gunter Saake

Abt. Datenbanken
Techn. Universität Braunschweig
Postfach 3329
D–38023 Braunschweig, Germany

Braunschweig
November 93

Monitoring Temporal Preconditions in a Behaviour Oriented Object Model[3]

Scarlet Schwiderski[1]
Thorsten Hartmann[2]
Gunter Saake[2]

[1]Computer Laboratory, University of Cambridge, Cambridge, CB2 3QG, UK (E-mail: Scarlet.Schwiderski@cl.cam.ac.uk)

[2]Abt. Datenbanken, Techn. Universität Braunschweig, Postfach 3329, D–38023 Braunschweig, Germany (E-mail: {hartmann|saake}@idb.cs.tu-bs.de)

[3]This work was partially supported by CEC under ESPRIT-III Basic Research Action Working Group No. 6071 IS-CORE II (Information Systems – COrrectness and REusability). The work of Thorsten Hartmann is supported by Deutsche Forschungsgemeinschaft under Sa 465/1-3. The work of Scarlet Schwiderski is supported by CEC under Human Capital and Mobility, ACID for Multimedia Applications, Grant No. ERB4001GT930471.

Abstract

Modern database applications require advanced means for modelling system structure and dynamics. Temporal logic has been proven to be a suitable vehicle for specifying the possible evolution of objects to be stored in databases. Past-directed temporal logic, as a means to describe the influence of the historical evolution of a database on applicable state changes, is one facet for the specification of object behaviour. The conceptual modelling language `TROLL` emphasizes the behaviour of objects over the course of time. Especially the restriction of events with preconditions in past-directed temporal logic has to be monitored, when a system specified in `TROLL` is implemented or prototyped.

In this report we introduce a technique for monitoring (past-directed) temporal preconditions during database runtime. This technique avoids storing the whole database history for evaluating temporal preconditions. Instead, minimal information about the database history is derived for specific temporal preconditions using transition graphs. This derived information is evaluated in later database states, when the temporal precondition is to be checked. We also describe a possible integrity monitor able to check temporal preconditions during database runtime. Such a monitor is specified locally to objects with advantages for distributed implementations. The dependency of the checking procedure on update operations leads to an optimized monitoring process that makes an efficient control of dynamic integrity constraints possible. The monitoring process itself is specified with the language `TROLL`. An implementation of an integrity monitor can therefore be tackled together with the language implementation. The advantages and future extensions of the proposed monitoring process and its modelling are briefly disussed.

Acknowledgements

For many fruitful discussions we are grateful to all members of IS-CORE, especially to Hans-Dieter Ehrich, Amílcar Sernadas, Cristina Sernadas who developed the basic ideas of the object model underlying `TROLL`, and to Udo Lipeck who gave valuable comments on the presentation of the algorithms. For discussions on the language `TROLL`-2 we thank Peter Hartel, Ralf Jungclaus, Jan Kusch, and all other colleagues in our group.

Contents

Chapter 1

Introduction

Modern database applications (e.g. office automation systems, banking systems, stock exchange systems) require advanced means for modelling system structure and dynamics. These new classes of database applications "exceed the capabilities of conventional (relational) database systems by far" [US90]. The underlying relational data model cannot fulfill the advanced design requirements, since it considers the static system structure only, not its dynamics. *Object-oriented data models* are, on the other hand, adequate for the design of these so-called information systems.

`TROLL` is a specification language for the object-oriented design of information systems [JSHS91]. Static and dynamic aspects of database applications are realized in an integrated manner by means of *objects*. The static structure of an object is given by its *attributes*. Attributes are typed and can be either of simple data types e.g. `bool, nat, ...` etc. or of complex data types (type constructors `set, list, tuple` applied to simple data types recursively). The dynamic structure of an object is given by the set of applicable operations (*events*). These events represent the only means for manipulating the object state. Events are, however, not applicable in an arbitrary order. `TROLL` supplies concepts which allow an explicit specification of the dynamic database behaviour. Among them are explicit process specifications, enabling conditions for events, and descriptive features to describe events that must necessarily be part of an objects life cycle.

Temporal preconditions constitute one of several `TROLL` concepts, which allows the specification of the dynamic object behaviour [Ser80, MP91, Ara91, Krö87, LEG85]. Temporal preconditions are formulae of *past-directed temporal logic (PTL)* and are used as enabling conditions for the occurrence of specific state-changing operations (events), as guards for communication requests, and to restrict attribute updates [JSHS91, HSJ+93]. If a certain event is to occur in a certain database state, we have to check the temporal preconditions associated with it and ensure that they are fulfilled. Checking temporal preconditions in a certain database state demands for appropriate monitoring algorithms for the stepwise evaluation of past-directed temporal formulae during database runtime.

Monitoring past-directed temporal formulae is not a trivial and straightforward venture. Past-directed temporal formulae refer to conditions and events in the history of the database. This is achieved through the usage of temporal operators, like **always_past**,

sometime_past, **previous**, and so on. One obvious possibility for monitoring past-directed temporal formulae is to store the whole database history and to evaluate it when requested. This, however, cannot lead to an efficient implementation, because of the permanently growing stored historical data and its therefore more and more complex evaluation. In this report we present a monitoring algorithm, which avoids storing the whole database history and, we claim, is minimal with respect to storage requirements (see also [Sch92, SS93]). The basic idea of the proposed method is that temporal preconditions are monitored preventively in the expectation of a later evaluation. Therefore, we monitor temporal preconditions from the beginning of the database life-cycle onwards, in parallel to the ongoing database development. In the final analysis, we have to check a simple non-temporal condition to evaluate, whether the temporal precondition is fulfilled or not. Our monitoring algorithm makes use of *transition graphs*, which are evaluation schemes constructed from future-directed, and past-directed temporal formulae respectively [LS87, Saa88, Lip89, Lip90, Saa91]. Transition graphs allow a stepwise evaluation of temporal formulae.

After the presentation of the monitoring algorithm for temporal preconditions in the general framework of temporal logic and transition graphs, we show, how it is integrated into a `TROLL` specification. We therefore sketch a procedure to derive *monitoring objects* (depending on some temporal formulae) that record necessary information about the evolution of an object. These monitoring objects are notified of *relevant events* that influence the evaluation of the specific temporal formulae and may be queried for its evaluation.

The report is structured as follows.

Chapter 2 gives an insight into the basic features of (first-order) predicate and temporal logic. We mainly introduce past-directed temporal logic (PTL), but also mention predicate logic as the kernel of temporal logic. Some basic knowledge of past-directed temporal logic is necessary, because temporal preconditions are formulae of this logic and the monitoring algorithms therefore base on its syntactical and semantical features.

In Chapter 3 we introduce the language `TROLL`. `TROLL` supports the conceptual design of object-oriented information systems. One characteristic of `TROLL` is that it supports temporal conditions as enabling conditions for event occurrences, communication requests, and restricted attribute updates. For implementation as well as for prototyping specifications, these conditions must be transformed into efficient monitoring procedures.

The monitoring algorithms for monitoring temporal preconditions during database runtime are developed in Chapter 4. The algorithms are general, that means, presented independently of the language `TROLL`. For a better understanding, the explanations are accompanied by a detailed example.

In Chapter 5 we consider the integration of temporal preconditions in `TROLL`. We transfer a `TROLL` specification with temporal preconditions (formulated in past-directed temporal logic) into a specification, which realizes the monitoring algorithms, and therefore reduce checking temporal formulae to checking simple non-temporal formulae.

Chapter 6 summarizes the achieved goals and gives an outlook on further work.

Chapter 2

Temporal Logic Framework

In this section we introduce predicate logic (2.1) and temporal logic (2.2). Predicate logic is also called classical logic and is, among other things, used to specify certain conditions in database systems, e.g. *static integrity constraints*. Conditions specified in predicate logic are characterized by the fact that they consider single database states only. Temporal logic is an extension of predicate logic. Conditions specified in temporal logic consider sequences of database states. *Dynamic integrity constraints* are such conditions [ELG84, LS87, SL89, Lip89, Ara91, Saa91, HS91]. Here we introduce past-directed temporal logic as a means to describe preconditions for database state changes that refer to the *history* of the database.

2.1 Predicate Logic (PL)

A logic consists of a syntax and a semantics. The syntax contains the rules for specifying conditions (so-called *formulae*). The meaning of these formulae is given in the semantics.

2.1.1 Syntax of PL

The used *symbols* of a logic are given by means of a signature, a set of variables and additional symbols for operators and quantifiers. These are put together to form *terms* and then *formulae*.

Definition 2.1 The *symbols of PL* are represented as

- a *signature* $\Sigma = (S, \Omega, \Pi)$ with
  - $S = \{s_1, s_2, \ldots\}$, a set of sorts
  - $\Omega = \{f_1 \colon s_{1_1} \times \ldots \times s_{1_{n1}} \to s_{1_0}, f_2 \colon s_{2_1} \times \ldots \times s_{2_{n2}} \to s_{2_0}, \ldots\}$, a set of function symbols
  - $\Pi = \{p_1 \colon s_{1_1} \times \ldots \times s_{1_{m1}}, p_2 \colon s_{2_1} \times \ldots \times s_{2_{m2}}, \ldots\}$, a set of predicate symbols

- a set of sorted *variables* $X = \{x_1 \colon s_1, x_2 \colon s_2, \ldots\}$
- *special symbols* like $=, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall, (,)$

This predicate logic is called *many-sorted*, because S consists of a set of sorts (e.g. `bool`, `integer`, `char`, `string`).

Definition 2.2 The *terms of PL* are built as follows:

- Every variable $x \colon s$ is a term of sort s
- If $t_1 \colon s_1, \ldots, t_n \colon s_n$ are terms and $f \colon s_1 \times \ldots \times s_n \rightarrow s_0$ is a function symbol, then $f(t_1 \colon s_1, \ldots, t_n \colon s_n) \colon s_0$ is a term of sort s_0
- Nothing else is a term of PL

Definition 2.3 The *formulae of PL* are built as follows:

- If $t_1 \colon s$ and $t_2 \colon s$ are terms of the same sort, then $t_1 \colon s = t_2 \colon s$ is a formula
- If $t_1 \colon s_1, \ldots, t_n \colon s_m$ are terms and $p \colon s_1 \times \ldots \times s_m$ is a predicate symbol, then $p(t_1 \colon s_1, \ldots, t_m \colon s_m)$ is a formula
- If φ and ψ are formulae and $x \colon s$ is a variable, then $(\neg\varphi), (\varphi \vee \psi)$ and $(\exists x \colon s)\varphi$ are formulae
- Nothing else is a formula of PL

$\wedge, \Rightarrow, \Leftrightarrow$ and $\forall$ are used as abbreviations for the following formulae:

$$\begin{array}{lcl}
\varphi \wedge \psi & \equiv & \neg(\neg\varphi \vee \neg\psi) \\
\varphi \Rightarrow \psi & \equiv & \neg\varphi \vee \psi \\
\varphi \Leftrightarrow \psi & \equiv & (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi) \\
(\forall x \colon s)\varphi & \equiv & \neg(\exists x \colon s)\neg\varphi
\end{array}$$

2.1.2 Semantics of PL

To determine the meaning of an arbitrary PL formula, that means, to evaluate an arbitrary PL formula, we have to identify the meaning of the signature Σ (the so-called *interpretation* of Σ), of the variables X (the so-called *substitution* of X) and of the operators and quantifiers.

Definition 2.4 An *interpretation of a signature* $\Sigma = (S, \Omega, \Pi)$ is a Σ-structure $A(\Sigma) = (A(S), A(\Omega), A(\Pi))$ where

- $A(S)$ a set $A(s)$ for each sort $s \in S$

- $A(\Omega)$ functions $A(f)\colon A(s_1) \times \ldots \times A(s_n) \to A(s_0)$ for each $f\colon s_1 \times \ldots \times s_n \to s_0 \in \Omega$
- $A(\Pi)$ predicates $A(p) \subseteq A(s_1) \times \ldots \times A(s_m)$ for each $p\colon s_1 \times \ldots \times s_m \in \Pi$

Definition 2.5 A *(local) substitution* β of variables X is an assignment $\beta(x\colon s) \in A(s)$ for each variable $(x\colon s) \in X$.

Definition 2.6 For a given interpretation A and substitution β, the terms of PL are evaluated as follows:

$$\begin{array}{lcl}
(A,\beta)[\![x]\!] & = & \beta(x) \\
(A,\beta)[\![f(t_1,\ldots,t_n)]\!] & = & A(f)((A,\beta)[\![t_1]\!],\ldots,(A,\beta)[\![t_n]\!])
\end{array}$$

Definition 2.7 For a given interpretation A and substitution β, the formulae of PL are evaluated as follows:

$$\begin{array}{lll}
(A,\beta) \models t_1 = t_2 & \text{iff} & (A,\beta)[\![t_1]\!] = (A,\beta)[\![t_2]\!] \\
(A,\beta) \models p(t_1,\ldots,t_m) & \text{iff} & ((A,\beta)[\![t_1]\!],\ldots,(A,\beta)[\![t_m]\!]) \in A(p) \\
(A,\beta) \models \neg\varphi & \text{iff} & \text{not } (A,\beta) \models \varphi \\
(A,\beta) \models \varphi \vee \psi & \text{iff} & (A,\beta) \models \varphi \text{ or } (A,\beta) \models \psi \\
(A,\beta) \models (\exists x\colon s)\varphi & \text{iff} & \text{there is a substitution } \beta'\text{, which is the same as} \\
 & & \beta \text{ except for the value of } x\text{, and } (A,\beta') \models \varphi
\end{array}$$

Therefore, having given an interpretation A and a substitution β, we can evaluate an arbitrary PL formula φ and find out that it is either **true** ($(A,\beta) \models \varphi$) or **false** ($(A,\beta) \models \neg\varphi$).

2.2 Temporal Logic (TL)

Predicate logic is not sufficient to state conditions concerning the temporal development of a database, e.g. *submitting a PhD thesis must be preceded by the matriculation as a PhD student.* Temporal logic on the other hand extends predicate logic in the appropriate way. Temporal logic formulae are evaluated in a sequence of database states, as opposed to predicate logic formulae, which are evaluated in a single database state. Consideration of several different database states within one formula can be achieved through the usage of additional *temporal operators.* Here, we confine to a *past-directed temporal logic (PTL).* A temporal logic can, however, also be future-directed or combined [Ara91, MP91, Krö87, Kun84, Aba88, Cho92].

2.2.1 Syntax of PTL

The syntax of PTL is the syntax of PL extended with temporal operators, like **previous, always_past, sometime_past, always**. . . **since_last**. Here, we focus on the differences to PL.

Definition 2.8 The *symbols of PTL* are represented as

- a *signature* $\Sigma = (S, \Omega, \Pi)$ with
  - $S = \{s_1, s_2, \ldots\}$, a set of sorts
  - $\Omega = \{f_1 \colon s_{1_1} \times \ldots \times s_{1_{n1}} \to s_{1_0}, f_2 \colon s_{2_1} \times \ldots \times s_{2_{n2}} \to s_{2_0}, \ldots\}$, a set of function symbols
  - $\Pi = \{p_1 \colon s_{1_1} \times \ldots \times s_{1_{m1}}, p_2 \colon s_{2_1} \times \ldots \times s_{2_{m2}}, \ldots\}$, a set of predicate symbols
- a set of sorted *variables* $X = \{x_1 \colon s_1, x_2 \colon s_2, \ldots\}$
- *special symbols* like $\neg, \wedge, \vee, \exists, \forall$, **previous, existsprevious, always_past, sometime_past, always**. . . **since_last, sometime**. . . **since_last**

Definition 2.9 The *formulae of PTL* are built as follows:

- Each PL formula φ is a formula.
- If φ and ψ are formulae, then $\neg\varphi$ and $\varphi \vee \psi$ are formulae.
- If φ is a formula, then **previous** φ and **existsprevious** φ are formulae.
- If φ and ψ are formulae, then **always_past** φ, **sometime_past** φ, **always** φ **since_last** ψ and **sometime** φ **since_last** ψ are formulae.

2.2.2 Semantics of PTL

Temporal formulae are evaluated in a sequence of database states. That means, we need a sequence of interpretations, one for each state of the sequence of database states, to evaluate temporal formulae. Accordingly, we also need a sequence of local substitutions. We insist that each of these local substitutions maps variables to the same elements of the corresponding sort as long as possible (*global substitution*) [Saa91].

Definition 2.10 A *state sequence* $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle$ is a non-empty sequence of Σ-structures $\sigma_j = A_j(\Sigma)$, where $0 \leq j \leq i$.
$\hat{\sigma}_{i-j}$, where $0 \leq j \leq i$, denotes the partial sequence $\langle \sigma_0, \sigma_1, \ldots, \sigma_{i-j} \rangle$ of $\hat{\sigma}$.

Definition 2.11 A *global substitution* $\hat{\beta}$ of variables X is a sequence $\langle \beta_0, \beta_1, \ldots, \beta_i \rangle$ of local substitutions, which satisfies the following condition for each $(x \colon s) \in X$ with $0 \leq m, n \leq i$.

$$\beta_m(x) \in (\sigma_m(s) \cap \sigma_n(s)) \Rightarrow (\beta_m(x) = \beta_n(x))$$

Definition 2.12 For a given state sequence $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle$ and a given global substitution $\hat{\beta} = \langle \beta_0, \beta_1, \ldots, \beta_i \rangle$, the formulae of PTL are evaluated as follows:

$$
\begin{array}{lll}
(\hat{\sigma}, \hat{\beta}) \models \varphi & \text{iff} & (\sigma_i, \beta_i) \models \varphi \text{ for PL formulae } \varphi \\
(\hat{\sigma}, \hat{\beta}) \models \neg\varphi & \text{iff} & \text{not } (\hat{\sigma}, \hat{\beta}) \models \varphi \\
(\hat{\sigma}, \hat{\beta}) \models \varphi \vee \psi & \text{iff} & (\hat{\sigma}, \hat{\beta}) \models \varphi \text{ or } (\hat{\sigma}, \hat{\beta}) \models \psi \\
(\hat{\sigma}, \hat{\beta}) \models \textbf{existsprevious } \varphi & \text{iff} & |\hat{\sigma}| > 1 \text{ and } (\hat{\sigma}_{i-1}, \hat{\beta}_{i-1}) \models \varphi \\
(\hat{\sigma}, \hat{\beta}) \models \textbf{previous } \varphi & \text{iff} & |\hat{\sigma}| > 1 \text{ and } (\hat{\sigma}_{i-1}, \hat{\beta}_{i-1}) \models \varphi \text{ or } |\hat{\sigma}| = 1 \\
(\hat{\sigma}, \hat{\beta}) \models \textbf{always\_past } \varphi & \text{iff} & (\hat{\sigma}_{i-j}, \hat{\beta}_{i-j}) \models \varphi \text{ for all } 0 \leq j \leq i \\
(\hat{\sigma}, \hat{\beta}) \models \textbf{sometime\_past } \varphi & \text{iff} & \text{there exists } j, \text{ where } 0 \leq j \leq i, \\
 & & \text{such that } (\hat{\sigma}_{i-j}, \hat{\beta}_{i-j}) \models \varphi \\
(\hat{\sigma}, \hat{\beta}) \models \textbf{always } \varphi \textbf{ since\_last } \psi & \text{iff} & (\hat{\sigma}_{i-j}, \hat{\beta}_{i-j}) \models \varphi \text{ for all } 0 \leq j < k, \text{ such that} \\
 & & k = \min \{\{m | (\hat{\sigma}_{i-m}, \hat{\beta}_{i-m}) \models \psi\} \cup \{i+1\}\} \\
(\hat{\sigma}, \hat{\beta}) \models \textbf{sometime } \varphi \textbf{ since\_last } \psi & \text{iff} & \text{there exists } j, \text{ where } 0 \leq j < k, \\
 & & \text{such that } (\hat{\sigma}_{i-j}, \hat{\beta}_{i-j}) \models \varphi, \text{ where} \\
 & & k = \min \{\{m | (\hat{\sigma}_{i-m}, \hat{\beta}_{i-m}) \models \psi\} \cup \{i+1\}\}
\end{array}
$$

The state sequence $\hat{\sigma} = \langle \sigma_0, \ldots, \sigma_i \rangle$ is evaluated backwards starting at the last state σ_i.

$(\hat{\sigma}, \hat{\beta})$ is called a *standard model* [Aba88], because the underlying notion of time is linear and discrete. This is expressed in the state sequence $\hat{\sigma}$, where we have an initial state σ_0 and a total ordering on the successing states.

Example 2.13 The following PTL formula states that a person is always a member of the college in which he/she matriculated sometime in the past.

$$
\begin{array}{ll}
\forall p : \mathit{Person}\ \forall c : \mathit{College} & \textbf{always\_past}(\neg\ \mathit{matriculation}(p, c)) \vee \\
 & \textbf{always } \mathit{member}(p, c) \textbf{ since\_last } \mathit{matriculation}(p, c)
\end{array}
$$

2.3 Evaluating PTL Formulae using Transition Graphs

The semantics of PTL provides no systematic procedure for evaluating PTL formulae. An evaluation method allowing a stepwise formulae evaluation is therefore needed. *Transition graphs*, whose construction and evaluation is discussed in this section, constitute such an evaluation method [LS87, Lip90, Saa88, Lip89, MW84].

2.3.1 Construction of Transition Graphs

Every PTL formula can be transformed into a “normal form”, which is the basis for the construction of a transition graph. This normal form is defined in the following lemma.

Lemma 2.14 *Disjunctive Past tense Temporal Normal-form (DPTN)*
Every PTL formula φ can be transformed into an equivalent formula $\varphi^\star$ of the form

$$\varphi^\star = \bigvee_k (\ \zeta_k \wedge \textbf{previous } \gamma_k [\ \wedge \ \ \textbf{existsprevious } \delta_k \ ] \ )$$

such that

- each ζ_k is a conjunction of basic PL sub-formulae of φ or their negations,
- each γ_k, δ_k is a conjunction of basic PL sub-formulae of φ, subformulae of φ bounded by temporal quantifiers or their negations.

The DPTN of a PTL formula can be built by applying the following two lemmas recursively, first to the whole formula, and then stepwise to all occurring subformulae. Therefore, we apply the rules from the outermost to the innermost parts of a formula.

Lemma 2.15 *Disjunctive Past tense Normalform (DPN)*
Every PTL formula φ can be transformed into an equivalent formula φ' of the form

$$\varphi' = \bigvee_k \chi_k$$

such that each χ_k is a conjunction of basic PL subformulae of φ, subformulae of φ bounded by temporal quantifiers or their negations.

The first step in building the DPTN of a PTL formula is building its DPN. This is simply done by applying the basic rules of PL, i.e., eliminating brackets using the distributive law and substituting $\Rightarrow$ and $\Leftrightarrow$ by $\wedge$, $\vee$ and $\neg$.

Lemma 2.16 *Temporal Recursion Rules*
For arbitrary PTL formulae φ and ψ, the following rules are valid:

$$\begin{aligned}
\textbf{always_past } \varphi &\iff \varphi \wedge \textbf{previous } (\textbf{always_past } \varphi) \\
\textbf{sometime_past } \varphi &\iff \varphi \vee \textbf{existsprevious } (\textbf{sometime_past } \varphi) \\
\textbf{always } \varphi \, \textbf{since_last } \psi &\iff \psi \vee (\varphi \wedge \textbf{previous } (\textbf{always } \varphi \, \textbf{since_last } \psi)) \\
\textbf{sometime } \varphi \, \textbf{since_last } \psi &\iff (\neg\psi \wedge \varphi) \vee (\neg\psi \wedge \\
& \qquad \textbf{existsprevious } (\textbf{sometime } \varphi \, \textbf{since_last } \psi))
\end{aligned}$$

We then apply these temporal recursion rules to the outermost temporal quantifiers of the DPN, and apply the two lemmas recursively, until all temporal quantifiers are bounded by **previous** and **existsprevious** respectively. The temporal recursion rules allow the decomposition of a temporally quantified formula into a temporal part and a non-temporal part.

Using the DPTN of a PTL formula, we can construct a transition graph, which has the following structure:

Definition 2.17 A *transition graph* $T = (G, \nu, \eta, m_0, F)$ consists of

1. a directed graph $G = (N, E)$ with nodes N and edges E,
2. a *node labelling* $\nu: N \to PTL$,
3. an *edge labelling* $\eta: E \to PL$,
4. a non-empty *initial marking* $m_0 \subseteq N$ and
5. a set of *final nodes* $F \subseteq N$.

Therefore, a transition graph is a directed graph, whose nodes are labelled with PTL formulae and whose edges are labelled with PL formulae. Moreover, certain nodes of the transition graph belong to the initial marking (in our case the initial marking consists of exactly one node) and certain nodes belong to the set of final nodes. The connection between a PTL formula and its corresponding transition graph is emphasized in the following definition (see [Saa88, Lip89] for a corresponding notion concerning future-directed temporal logic).

Definition 2.18 A transition graph is called an *evaluation schema* for a PTL formula φ iff it satisfies the following conditions:

1. The node label disjunction of the nodes in m_0 is equivalent to φ
2. For each node k the transition graph is *correctly labelled* such that

$$\begin{aligned}\nu(k) \iff & \left(\bigvee_{l \in F \ (k,l) \in E} (\eta((k,l)) \wedge \textbf{previous } \nu(l)) \right) \\ \vee & \left(\bigvee_{l \in (N-F) \ (k,l) \in E} (\eta((k,l)) \wedge \textbf{existsprevious } \nu(l)) \right)\end{aligned}$$

Example 2.19 The following PTL formula states that a person must have submitted a thesis sometime in the past.

$\forall p : Person \quad \textbf{sometime\_past}(submit\_thesis(p))$

The transition graph of this formula is presented in figure 2.1[1].

Algorithm 2.20 *Construction of a transition graph*
For a given PTL formula φ, the corresponding transition graph T_φ of φ is constructed as follows:

[1] Final nodes are indicated by a double line.

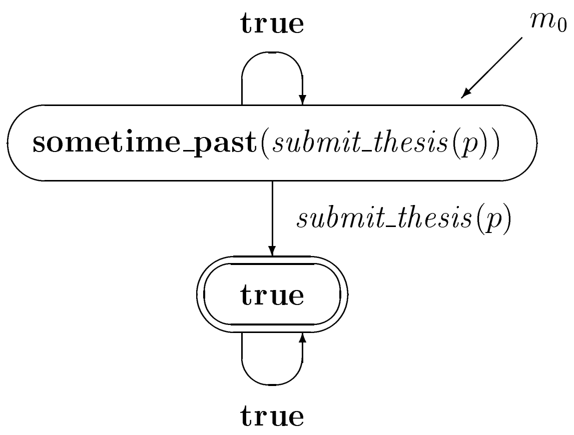

Figure 2.1: Transition graph of **sometime_past**(*submit_thesis*(*p*))

1. Initialize T_φ
   $E := \{\}$;
   $N := \{0\}$;
   $\nu(0) := \varphi$;
   $m_0 := \{0\}$;
   $F := \{\}$;

2. **for each** node $n \in N$ in T_φ
   do transform the node label $\nu(n)$ into its DPTN $\nu(n)'$;
       for each ($\zeta \wedge$ **previous** γ) in $\nu(n)'$
       do if $\nu(m) \neq \gamma$ for all $m \in F$
           then add a new node k with $\nu(k) = \gamma$ to N;
                   $F := F \cup \{k\}$;
           fi;
           let k be the node in F with $\nu(k) = \gamma$;
           add a new edge (n, k) with $\eta((n, k)) = \zeta$ to E;
       od;
       for each ($\zeta \wedge$ **previous** $\gamma \wedge$ **existsprevious** δ) in $\nu(n)'$
       do if $\nu(m) \neq \gamma \wedge \delta$ for all $m \in N - F$
           then add a new node k with $\nu(k) = \gamma \wedge \delta$ to $N - F$;
           fi;
           let k be the node in F with $\nu(k) = \gamma \wedge \delta$;
           add a new edge (n, k) with $\eta((n, k)) = \zeta$ to E;
       od;
       if there are several edges from n to k
       then substitute these edges by one labelled with
               the disjunction of the old edge labels;
       fi;
   od;

The construction of T_φ starts with a node 0, which is labelled with the starting formula φ and which is the node of the initial marking m_0. The labelling of node 0 always indicates, to which formula φ the transition graph corresponds. φ is then transformed into its DPTN. Every conjunctive subformula of this disjunction is considered separately. The temporal part of such a subformula (that means the subformulae bounded by **previous** and **existsprevious** respectively) represents the labelling of a successor node. The edge leading to this node is labelled with the non-temporal part of the considered subformula. This procedure is applied recursively until all nodes of the graph are examined.

It can be shown that the transition graph T_φ of φ is an evaluation schema for the PTL formula φ (see [Saa88]).

2.3.2 Evaluation of Transition Graphs

The evaluation of a transition graph starts with the nodes of the initial marking (node 0 in our case). In our terminology we say, the node is marked, or the node belongs to the *current marking*. We then evaluate the given state sequence $\hat{\sigma} = \langle \sigma_0, \ldots, \sigma_i \rangle$ starting with state σ_i and decreasing the index stepwise from i to 0. In a certain state, we evaluate the edge labels of the outgoing edges of the nodes of the current marking, which are either **true** or **false** in this state. If the edge label is **true** (that is valid), we unmark the start-node of the edge and mark the end-node. If all edge labels of the outgoing edges of a certain node are **false** (that is invalid), we unmark the start-node only. Finally, φ is valid, if there is a marked final node in state σ_0.

Definition 2.21 A *current marking cm* of a transition graph T_φ is a non-empty set of its nodes, $cm \subseteq N$.

Algorithm 2.22 *Past-directed evaluation of a transition graph*
For the given transition graph T_φ of a PTL formula φ, a state sequence $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle$ and a global substitution $\hat{\beta} = \langle \beta_0, \beta_1, \ldots, \beta_i \rangle$, where $\beta_i(x : s) \in \sigma_j(s)$ for all x free in φ and $0 \leq j \leq i$, the transition graph is evaluated as follows:

begin
    $ind := i$;
    $cm := m_0$;   (* cm : current marking *)
    while $ind \geq 0$ **do**
        $new\_cm := \{\}$;
        for each $(n_1, n_2) \in E$ with $n_1 \in cm$ **and** $(\sigma_{ind}, \beta_{ind}) \models \eta(n_1, n_2)$
        do
            $new\_cm := new\_cm \cup \{n_2\}$;
        end
        $cm := new\_cm$;
        if $cm = \{\}$

```
        then return ((σ̂, β̂) ⊭ φ);            (* φ is invalid *)
        fi
        ind := ind − 1;
    od
    if ∃ n ∈ N : n ∈ {F ∩ cm}
    then return ((σ̂, β̂) ⊨ φ);                (* φ is valid *)
    else return ((σ̂, β̂) ⊭ φ);                (* φ is invalid *)
    fi
end
```

The algorithm above allows the stepwise evaluation of a PTL formula throughout the states of the database history, using the transition graph constructed from that formula.

Definition 2.23 A transition graph T_φ of φ *definitely accepts* a state sequence $\hat{\sigma}$ under a global substitution $\hat{\beta}$ iff the last current marking contains a final node.

It can be shown that a transition graph T_φ of φ definitely accepts a state sequence $\hat{\sigma}$ under a global substitution $\hat{\beta}$ (written $(\hat{\sigma}, \hat{\beta}) \vdash T_\varphi$) iff φ is valid in $\hat{\sigma}$ under $\hat{\beta}$ (see [Saa88]), that is

$$(\hat{\sigma}, \hat{\beta}) \vdash T_\varphi \Longleftrightarrow (\hat{\sigma}, \hat{\beta}) \models \varphi$$

This fact allows us to use transition graphs for the stepwise evaluation of PTL formulae in a state sequence under a global substitution. In each state of the sequence (going backwards from the current state σ_i to the initial state σ_0), we evaluate certain edge labels of the transition graph. This evaluation can be done locally within the considered state, because edge labels are (non-temporal) PL formulae. Finally, we check the nodes of the current marking. If one of them is a final node, the starting formula (which corresponds to the label of the node of the initial marking) is valid in the current state σ_i.

Example 2.24 The transition graph in figure 2.1 represents the evaluation schema for the PTL formula **sometime_past**(*submit_thesis*(p)). Checking this transition graph in a certain state σ_i means that we evaluate the database history from state σ_i backwards until we reach state σ_0. In every step of the evaluation, we check the edge labels of the outgoing edges of the currently marked nodes. In this case, node 0 (labelled with **sometime_past**(*submit_thesis*(p))) is the node of the initial marking. Node 0 stays marked throughout the whole evaluation (because of the circular edge labelled with **true**). When we reach a state, where *submit_thesis*(p) is true, we add node 1 (labelled with **true**) to the current marking, because the edge label *submit_thesis*(p) is valid. This node now also stays marked throughout the rest of the evaluation (because of the circular edge labelled with **true**). When we finish the evaluation in state σ_0, we can derive that the PTL formula is valid, because node 1 is a final node and is currently marked.

Chapter 3

The Language TROLL

In this chapter we will briefly introduce the basic language features of TROLL-2, the successor version of the language TROLL introduced in [JSHS91]. In the sequel we will refer to TROLL-2 by the name TROLL and to the former version by TROLL-1 where necessary. The basic ideas concerning specification logics [Jun93, Saa93, SSC92, FSMS91] and the semantic object model [SSE87, ESS89, ESS90] adopted for TROLL will not be repeated in this report. The interested reader can refer to [EGS90, ES91, EDS93] for recent developments of the semantic domain and [Jun93] for a comparison with related work.

A deeper analysis and motivation for language features not introduced in this report can be found in [JHS93] (high level relationships between dynamic objects), [HJS92] (composite objects), [HS91, HJS93] (transformation of high level language features to operational concepts and prototyping), [WJ91] (objects in different aspects or roles). [CGH92] introduces a dialect of TROLL simplified for exploration of formal verification of object properties. In [Wie91, SF91] work closely related to the concepts used in TROLL can be found.

The structure of TROLL objects is introduced by means of *templates* that consist of the following parts:

```
template Name
   < Various Declarations >
   < Components, Attributes, and Events Specification >
   < Object Behaviour Specification >
end template Name
```

A *template name* must be introduced, if we specify a template for later reuse in a class description. All template specification parts mentioned above are optional, but of course there are some constraints on the existence of some sections depending on other sections.

In the following sections we will describe the beforehand mentioned template parts as detailed as necessary for understanding the examples in Chapter 5. Primarily the attributes, events and components specification will be depicted. For the object behaviour specification see [HSJ+93].

3.1 Template Specification

3.1.1 Context Declaration

In Section 3.2.1 we will introduce classes as abstractions for sets of objects with similar structure and behaviour. On top of templates, classes are the basic structuring mechanism in TROLL. Inside template specifications, we can only refer to other *objects* by means of *components* being *objects* of *classes*.

The context section is thus used to describe the *visible* parts of an object society in terms of *classes*. This way, the specification of templates *associated* to classes as well as instances *contained* in classes may be accessed according to rules described in the section on classes and components.

For the discussion here it is sufficient to know that with a context declaration, we have *access to class objects as containers for objects* described by templates. This means that we may refer to a context class using its attribute and event interface. In the examples later on this feature is used to *create* objects.

A declaration like:

```
context
  Person, Buildings ;
```

makes available the class containers `Person` and `Buildings`. Components of an object can be taken from these containers as well as we can create new objects of these classes. We will provide an example at the end of this chapter.

3.1.2 Variable and Parameter Declarations

To describe various properties of objects, we introduce several kinds of formulae all based on sorted first order predicate calculus. Variables used in these formulae must be declared.

The following declaration declares three variables of type `nat`, `string`, and `|Person|` respectively:

```
variables
  Counter:nat; Name:string; aPersonIdentity:|Person|;
```

named `Counter`, `Name`, and `aPersonIdentity`. `|Person|` is a special data type denoting *identities* of objects (see below).

The meaning of such variable declarations is a universal quantification for each formula in a given block, e.g. an attribute or event description etc. (see below). Parameters of events and attributes are introduced in the same way (see the examples below).

3.1.3 Attributes

The attribute specification section of a template specification defines the *observable properties* of objects. Attributes in TROLL can be compared to instance variables of languages like Smalltalk. However, there is a difference in that attributes in TROLL are *visible at the object interface* (if not explicitly hidden) like proposed in other object oriented data models with a semantic data modelling background.

Attributes are specified with a name and data type. Optionally attributes may have parameters, thus introducing *sets of attributes*. Attribute parameters can be specified with an optional name. Parameterized attributes define one attribute for each possible value of the parameter sorts (data types). Parameters are named to make it possible to refer to them in the description part of attributes. As mentioned in the previous section, such declaration is an abbreviation of a universally quantified variable.

As an example for a parameterized attribute we specify the income of a person in specific years:

```
attributes
  IncomeInYear(Year:nat):money .
  ...
```

thus for each possile value of the declared parameter Year there exist an attribute value. Conceptually most of them will have an undefined value.

Attribute specifications may be refined in several ways. An attribute may be ***hidden***, i.e. not visible from outside the template, ***restricted***, i.e. constrained in its set of possible values or possible parameter values, ***initialized*** upon birth of an object, ***derived***, i.e. calculated from other attributes, or it may be ***constant***. The attribute IncomeInYear can be further refined writing:

```
attributes
  IncomeInYear(Year:nat):money
    hidden ;
    restricted Year>=1900 and Year<=2100 .
  ...
```

The attributes so defined are not visible outside of the object (***hidden***) and all attributes with parameter values outside the range 1900 to 2100 have an undefined value (***restricted***). We will present more examples that mention some of the above listed options and their syntactical representation later on.

3.1.4 Events

The most important part of an object behaviour description is the specification of *events* that can take place in an objects life and that *change its state*. According to approaches of object oriented languages in general, we identify three different aspects of events as abstractions of methods:

1. An event can be allowed in a given object state or be forbidden in a given state. In terms of functional specification we speak of *enabling conditions* (sometimes called *permissions* or *safety rules*).

2. An event can have effects on the local state of an object. In other words, we observe a new state after an event occurs. In more traditional terms events change state variables (attributes).

3. An event can involve other events in different objects or in the object at hand. This functionality is achieved with *communication* between objects and suitable *trigger mechanisms*.

These aspects are among others grouped together into an *event description* for a given event and a given list of formal parameters. In `TROLL-1`, the description of the three different aspects mentioned above were introduced in different sections of the template. For `TROLL` we propose an integrated view where an event is described by the necessary conditions that must be fulfilled, the local change of state, associations with other events (calling, derivation), and the binding of parameters to values (see below).

Events are classified into ***birth***, ***death***, or *simple* events, the latter are not especially marked with a keyword (see below). Birth and death events create and destroy objects respectively. More exactly they can only occur as the first or last events in an objects life. Creation and destruction is managed with class objects that will be introduced briefly later on.

3.1.5 An Example Template Specification

Instead of introducing the various options for attribute and event specification in more detail, we will now present a short example that lists the key features of a `TROLL`-template specification. In this example, we will introduce mainly the *operational language features* by means of an implementation directed example. It should be noted however, that `TROLL` is a *specification language* that offers also declarative features. The following template introduces a `BoundedNatStack`, a stack for natural numbers with a maximum size of elements to be stored.

```
template BoundedNatStack
  attributes
     Top:nat
       derived as
         if Empty then undefined else Array(Pointer-1) fi .
     Empty:bool
       derived as (Pointer = 0) .
     Full:bool
       derived as (Pointer >= UpperBound) .
     UpperBound:nat
```

```
      hidden ;
      initialised 100 default .
    Array(No:nat):nat
      hidden ;
      initialised Array(No) = undefined .
    Pointer:nat
      hidden ;
      initialised 0 .
  events
    create birth .
    create(UpperBound:nat) birth .
    destroy death .
    push(Elem:nat)
      enabled not Full ;
      changing
        Array(Pointer) := Elem;
        Pointer := Pointer + 1 .
    pop
      enabled not Empty ;
      changing
        Pointer := Pointer - 1 .
end template BoundedNatStack ;
```

The `BoundedNatStack` observation interface is represented as three attributes `Top`, the value on top of the stack, `Empty` and `Full` to denote the status of the stack. These attributes are derived, that means, their value is *computed* via data terms referencing other attributes, constants etc.

All other attributes mentioned in this example are *hidden*. Hidden attributes may be used only inside the specification of the template at hand. In this example, they are used to *implement the data storage* for the stack objects: Stack entries are stored in an array of naturals, the current top element is denoted by a pointer to this array.

The `Array` attributes, conceptually one for each `nat` value are initialized with the value *undefined* upon birth of the object. The data term after the keyword *initialised* is used to describe the values to be stored. Here it is simply the *value* `undefined` implicitly available for all data types, in case of the `Pointer` attribute it is the value zero. The attribute `UpperBound` denotes the maximal count of elements that can be stored. As opposed to the initial value of the `Array` and `Pointer` attributes, the initial value for the `UpperBound` attribute is denoted as *initialised/default*. It can be changed by a suitable birth event (see below) whereas the former attributes are strictly defined to take their initial values upon birth of the object.

In the event specification part, the first three events are used to create and destroy stack objects. Now we can also see the use of different birth events, the first one to use a default behaviour, the second one to use an additional parameter to set an upper bound of elements to be stored. The usage of the name `UpperBound` as parameter name is an abbreviation for an additional attribute update.

The event `push` is used to store naturals on the stack. This event has an *enabling condition* denoted after the keyword ***enabled***. An enabling condition in TROLL states, that the event can only occur in a given state if its condition is true. Otherwise it is rejected in the sense that an event that *calls* for it is *also not allowed* (a very strict condition for object communication). In general these conditions can be formulated as arbitrary historical queries on the objects life (past directed temporal formulae).

If the precondition is fulfilled, the event may occur and has effects on the visible state of the object in terms of new attribute values. These new values are described as *assignment statements*. The left hand sides of assignment statements denote attributes by means of *attribute terms*, whereas the right hand sides denote data values by means of *data terms*. All subformulae are evaluated in the state where the event occurs. This implies that the list of assignments in TROLL is not an *operational sequence*. The assignments occur in *parallel* and their effects become visible in the following state.

A more liberal specification of stacks can be described in the following way without a precondition for `push` events but an additional parameter. The `push` event is no more disabled for a stack that is full, but it has *no effect* on the observable state of the stack and returns an error status:

```
...
  events
    push(Elem:nat,!Status:bool)
      changing
        { not Full }
          Array(Pointer) := Elem,
          Pointer := Pointer + 1 ;
      binding
        Status = not Full .
...
```

This example is used to introduce the modelling of *locally instantiated parameters*. The parameter `Status` is preceded with a *!*, denoting that a value is determined locally if the event occurs. Such parameters can be used to model *return values*. For a uniform treatment, these return values are notated similar to attribute effects but using an equality sign to emphasize that no assignment but some kind of *unification* takes place. The data term on the right hand side is evaluated in the state where the event occurs. If the stack is not full ***true*** is "returned." In this example, however, the effects of the `push` event have to be specified according to the status of the stack. If it is full, no effects must be observable. Only the client is informed with the return value.

Besides these simple specification of possible object behaviour, TROLL introduces more sophisticated methods for describing the possible behaviour of objects, namely the *long* *term* behaviour in terms of events that are obliged to occur in an objects life. This can be done in an operational style using a process language or declaratively using first order and temporal logic conditions referring to the history of objects. Temporal logic is used for event preconditions as will be depicted in Chapter 4.

3.2 Class and Component Specification

3.2.1 Classes and Class Objects

To come from a prototype specification of objects (templates) to the objects themselves, we have to introduce a *naming mechanism*. There is more or less agreement in the "object orientation community" that objects must have unique identities. In fact, this feature distinguishes object oriented data models from so called *value-based* models. In `TROLL` unique identities are associated with objects, too. Identities are values of a special unstructured, not user defined data type. As already sketched in Section 3.1.2, these identifier data types are denoted by class names surrounded with vertical bars.

To introduce a class of stacks we have to specify for example:

```
object class Stack
  template BoundedNatStack
end object class Stack ;
```

and have implicitly defined a data type `|Stack|` with values usable as *identities* for stacks. For the `Stack` class, we only supply a template specification. This means that stack objects may be referenced only via their *identities* and that there may be an infinite number of stacks. For example `Stack(x)` denotes a stack object if `x` is declared as a variable `x:|Stack|` (and the object associated to `x` exists!). We can refer to the top value of stack `x` writing `Stack(x).Top`.

With the specification of an object class, an implicit *class object* is available. It is constructed out of the specification of the object class template and contains, for example, events derived from the *instance birth events* with additional parameters, e.g. attributes for the management of the class population etc. The class object specification has a special header (the keyword class is missing), namely stating that there exists *only one* (*single*) instance of this "class".[1] For the creation of objects, we have to use *class object events* of such implicitly generated class objects :

```
object StackCLASS
  ...
  events
    ...
```

[1] A single object is thus specified as a class with only one object. The identity data type thus has cardinality one, e.g. *card*(`|StackCLASS|`) = `1`.

```
    create(!OID:|Stack|)
      binding OID = ..... ;
      calling Stack(OID).create , .... ;
      ... .
    create(!OID:|Stack|,UpperBound:nat))
      binding OID = ..... ;
      calling Stack(OID).create(UpperBound), ... ;
      ... .
end object StackCLASS ;
```

that itself *call* the instance birth event. Calling is synchronous communication (see below). For convenience, we refer to these class objects via the *class name* specified. The expression

`Stack.create(anID)`

thus creates an object instance of class `Stack` and delivers an identity value `anID:|Stack|` back to the caller (some kind of a *handle* to the object that can be stored as an attribute value, be transferred during communication etc.). The notation `Stack.create(anID)` is thus an abbreviation for:

`StackCLASS.create(anID)`

On the language level the object `StackCLASS` is regarded as hidden as far as possible.

In the specification of the `StackCLASS` event `create`, the '`!`' prefixing the parameter `OID` denotes a parameter that is calculated locally inside the class object. *How* object identities are calculated is a matter of *implementation* and is not described here. The calling rule of the `create` event then initializes the *instances* with the appropriate *instance birth events* (also `create` in this example).

Similarly `Stack.create(anID,100)` is another creation event derived from the second *instance birth event* with an additional parameter that is then used for initialization as depicted above. The user of the language only has to specify the class `Stack`. The beforehand sketched specification of an object class as an object in its own right is completely derived from the `Stack` specification as are the creation and destruction events.

The *identity mechanism* introduced so far seems to be sufficient for anonymous objects like stacks, etc. Modelling real world contexts, we have to deal also with objects that have names which have *associated a meaning in the real world*. Modelling persons include an *identification* via their names for example besides their *identity* (that is used for specification purposes only). In `TROLL` we may introduce such names using *keys*:

```
object class Person
  identification
    ByName:(Name,BirthDate),
```

```
    BySSN:(SocialSecurityNo) ;
  attributes
    Name: string .
    BirthDate:date .
    SocialSecurityNo:nat .
  ...
end object class Stack ;
```

As keys, we declare *tuples of data values* taken from attribute domains. The attributes used have to obey special constraints. Attribute combinations used for identifying objects must be unique among all objects of this structure, here e.g. `(Name,BirthDate)` uniquely identifies persons. Clearly, this constraint cannot be formulated local to template or class specifications. We have to look at *all possible instances of a class*. This way we have to introduce a class as a *container for instances*. Again these *class container object features* are derived from the original (user) specification and are integrated into the already introduced *class objects*.

A class container is thus an object in its own right and we may declare the *key constraints* within these objects. Since the features sketched here have a lot of consequences that cannot be discussed briefly, we will refer the reader to the forthcoming language report on the revised version of TROLL [HSJ+93]. For this text, it is sufficient to know that we may refer to objects (using the class container) via identifying attributes and that we may also create objects this way. As an example look at he following expression:

```
Person(ByName("John",17-April-1964)).Hobbies
```

as a reference to the `Hobbies` attribute of person `"John"` born on `17-April-1964`. The class name `Person` used here may be substituted by a *component name* as we will see in the next section or by a name of a specialization class of class `Person`. We will, however, not deal with inheritance in this report. For all cases discussed, the notational conventions may be formally described using properties of implicitly derived class container objects.

3.2.2 Components

Nearly all objects found in the real world are constructed out of smaller part objects. TROLL supports the *part-of abstraction* by means of *components*. Components in TROLL may be *single*, *set*, or *list-valued*. For a component specification, we must refer to class containers that contain object instances we may use as components in the template at hand. To use for example `Person` instances as components of a *Company* object, we must declare the `Person`–class to be in the *context* of the `Company`. Then we may declare a *set-valued* component `Employees` or a single object valued component `Manager` in a company specification:

```
object class Company
  identification ByName:(CompanyName) ;
```

```
context Person, Buildings, ... ;
attributes
  CompanyName:string .
  ...
components
  Employees:Person set .
  Houses:Buildings set .
  Manager:Person .
...
```

In principle, a set component specification resembles a class specification in that we may use the component name as a *local class name* to refer to a *subset of instances* of the class. The single object component is just a special case. We can refer to instances of the `Person` class only if they became part of the composite object in the past. This is accomplished using implicitly defined events to alter the composition. For example, the events `Employees.INSERT(|Person|)` and `Employees.REMOVE(|Person|)` are implicitly defined with the set and single valued component specifications. The dot notation resembles the notation used to refer to instances of classes which emphasizes the idea of locally visible subsets of objects.

Object instances may thus be part of several different objects. For example, a "person object" may be a "component" of several companies. The part-of hierarchy does not necessarily form a tree which can be specified with `TROLL`, too. We will not deal with this language feature called *local classes* here.

If there exists an event `buildHouse` in a company, we may want to specify that a new `Building` is created and then incorporated into the composite object of class `Company` shown above. In the `Company` specification, this may be formulated by means of event calling. Suppose `Buildings` are identified by their `Position` of the data type `Coordinates` and they have a birth event `create`. We then have to specify:

```
object class Company
  ...
  events
    variables BuildingsOID:|Buildings| ;
    buildHouse(Position:Coordinates)
      calling
        Buildings.create(BuildingsOID,Position) ,
        Houses.INSERT(BuildingsOID) .
  ...
```

All events *called* by an initial event (transitively) occur as *one atomic state change* that is synchronously like already mentioned in the stack example. The first mentioned event called is an event of the *class container object* for class `Buildings`. This event binds the parameter `BuildingsOID` to a new identity after checking if there exists a building at the mentioned position and then initializes the object instance via calling the *instance* *birth event* as already described for the stack objects in the previous section. If the *key constraint* is violated, the creation event will be rejected.

The identity of the newly created object is also used to insert this instance into the component set `Houses` using the implicitly generated event `Houses.INSERT(...)`. All events called in this example conceptually occur *at once*. For an implementation of this mechanism, an analysis phase is needed that must decide about a suitable ordering of event *executions*. This is the case because the binding of parameters is not always obvious. For a discussion on this and related problems see [HS93].

Chapter 4

Monitoring Temporal Preconditions

Temporal preconditions are formulae of PTL. We can therefore use transition graphs for their evaluation. A transition graph is evaluated stepwise from the current database state backwards throughout the whole past state sequence of the database, that is, the whole database history. Hence, using this method directly implies storing the whole database history. This way, we cannot monitor temporal preconditions efficiently.

In this section we present a method for monitoring temporal preconditions efficiently during database runtime. This method is also based on the evaluation of transition graphs, although, in a different way from above.

We illustrate the single steps in our method by the following example.

Example 4.1 The following condition expresses that a person `p` can only start a PhD course at a university `u` (`start_PhD(p,u)`), if he/she first applied for admission (`apply_for_admission(p,u)`) and afterwards got admission (`get_admission(p,u)`). In `TROLL` it would be written in the following way:

```
events
  start_PhD(p,u)
    enabled
      sometime_past (apply_for_admission(p,u)) and
      sometime (get_admission(p,u))
          since_last (apply_for_admission(p,u)) .
  ...
```

The PTL formula φ after the keyword *enabled* is a temporal precondition of the event `start_PhD(p,u)`. The transition graph T_φ of φ is shown in figure 4.1.

4.1 Reversed Evaluation of Transition Graphs

The crucial step in our method is to evaluate a transition graph in the reverse direction. The evaluation starts in the initial database state and goes forward in time to the current

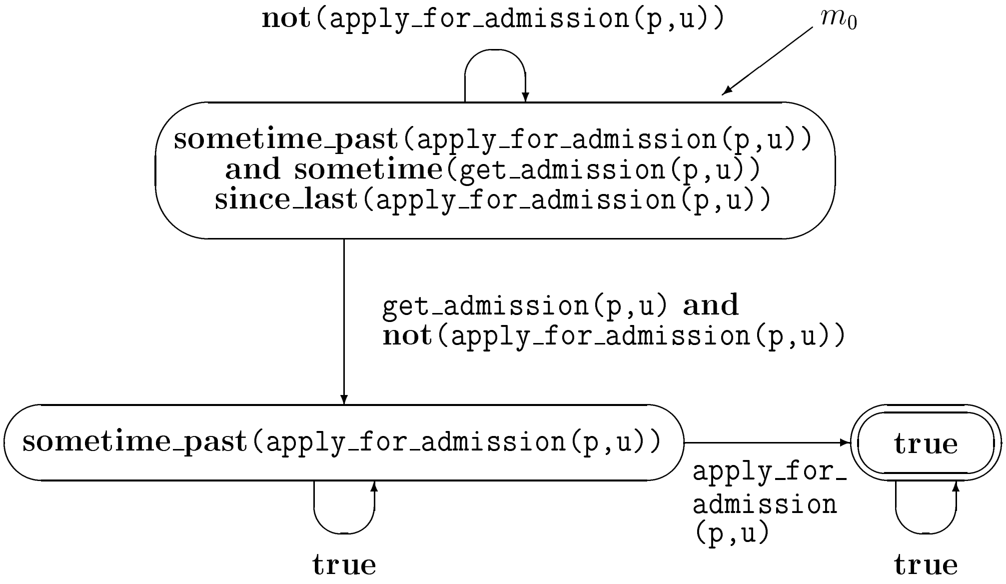

Figure 4.1: Transition graph T_φ

database state, evaluating the transition graph from the final nodes backwards to the node of the initial marking[1].

In the following, we want to formalize this step. We firstly define a *reversed transition graph*, which is a transition graph evaluable in the reverse direction. We then give the algorithm for evaluating a reversed transition graph correspondingly.

Definition 4.2 If $T_\varphi = \langle G = (N, E), \nu, \eta, m_0, F\rangle$ is the transition graph of a temporal precondition φ, then T_φ^{-1} is called *reversed transition graph* of φ, iff

$$T_\varphi^{-1} = \langle G' = (N, E'), \eta', m_0, F\rangle$$

where E' with $(n_2, n_1) \in E'$ iff $(n_1, n_2) \in E$ and
$\eta' \colon E' \to PL$ with $\eta'(n_2, n_1) = \eta(n_1, n_2)$

The nodes of a reversed transition graph are, as opposed to transition graphs, not labelled with corresponding PTL formulae. The reason is that a node labelling is not necessary for the actual evaluation. We can therefore do without it[2]. Moreover, the edges of a reversed transition graph T_φ^{-1} are reverse to the edges of the corresponding transition graph T_φ.

Example 4.3 The reversed transition graph T_φ^{-1} of φ is shown in figure 4.2.

[1]In our case there is always exactly one node in the initial marking of a transition graph. Hereafter, we refer to this node as the *start node*.

[2]This definition of a reversed transition graph is different from the one given in [Sch92, SS93], where we additionally used the node labelling ν. Here, however, we do without it, because the presented method for monitoring temporal preconditions is more obvious in this way.

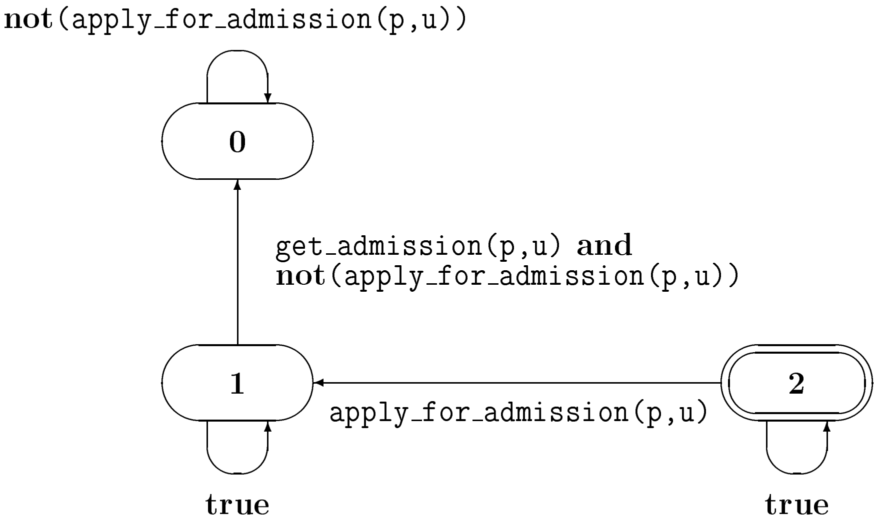

Figure 4.2: Reversed transition graph T_φ^{-1}

Algorithm 4.4 *Future-directed evaluation of a reversed transition graph*

For a given reversed transition graph T_φ^{-1} of a temporal precondition φ, a state sequence $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle$ and a global substitution $\hat{\beta} = \langle \beta_0, \beta_1, \ldots, \beta_i \rangle$, where $\beta_i(x : s) \in \sigma_j(s)$ for all x free in φ and $0 \le j \le i$, the reversed transition graph is evaluated as follows:

begin
    $ind := 0$;
    $cm := F$; (∗ cm : current marking ∗)
    while $ind \le i$ **do**
        $new\_cm := \{\}$;
        for each $(n_1, n_2) \in E'$ with $n_1 \in cm$ **and** $(\sigma_{ind}, \beta_{ind}) \models \eta'(n_1, n_2)$
        do
            $new\_cm := new\_cm \cup \{n_2\}$;
        end
        $cm := new\_cm$;
        if $cm = \{\}$
        then return $((\hat{\sigma}, \hat{\beta}) \not\models \varphi)$; (∗ φ is invalid ∗)
        fi
        $ind := ind + 1$;
    od
    if $\exists\, n \in N : n \in \{m_0 \cap cm\}$
    then return $((\hat{\sigma}, \hat{\beta}) \models \varphi)$; (∗ φ is valid ∗)
    else return $((\hat{\sigma}, \hat{\beta}) \not\models \varphi)$; (∗ φ is invalid ∗)
    fi
end

The following theorem shows that algorithm 2.22 and algorithm 4.4 are equivalent, that means, a temporal precondition φ is either valid in $(\hat{\sigma}, \hat{\beta})$ in both cases, using algorithm 2.22 and algorithm 4.4, or invalid.

Theorem 4.5 The past-directed evaluation of a transition graph T_φ and the future-directed evaluation of a reversed transition graph T_φ^{-1} are equivalent.

Proof Algorithm 4.4 presents an exact mirror image of algorithm 2.22. In $\hat{\sigma}$ under $\hat{\beta}$ the corresponding edges of T_φ and T_φ^{-1} are evaluated in the same states σ_{ind} under the same substitutions β_{ind}. The only point to consider is that algorithm 4.4 starts the evaluation with all final nodes. We have to show that there is a valid path from the final nodes to the start node using algorithm 4.4 iff there is a valid path from the start node to the final nodes using algorithm 2.22.

$\Longrightarrow$ a) There is a valid path from the final nodes to the start node using algorithm 4.4. That means, there is also at least one node in the final marking using algorithm 2.22. φ is valid.

b) There is no valid path from the final nodes to the start node using algorithm 4.4. That means, there is no node in the final marking using algorithm 2.22. φ is invalid.

$\Longleftarrow$ a) There is a valid path from the start node to the final nodes using algorithm 2.22. That means, the start node is in the final marking using algorithm 4.4. φ is valid.

b) There is no valid path from the start node to the final nodes using algorithm 2.22. That means, the start node is not in the final marking using algorithm 4.4. φ is invalid.

A given global substitution $\hat{\beta} = \langle \beta_0, \beta_1, \ldots, \beta_i \rangle$ is a precondition in algorithm 4.4. **This precondition cannot be fulfilled in a practical situation.** In the final analysis, we need the local substitution β_i in state σ_i, because the free variables of φ are bounded to these values throughout the whole evaluation and therefore in all states σ_j, where $0 \leq j \leq i$.

It follows that algorithm 4.4 cannot be used directly in this way. Instead, we have to find a possibility for evaluating the reversed transition graph *substitution-independent*. What does that mean? There may be edges in a reversed transition graph, whose labels do not contain variables. An example is an edge with the simple edge label **true**. These edges can be evaluated in the normal way. The other edges, whose labels contain variables, have to be evaluated in a different way. The solution is to divide the evaluation into two steps.

Preliminary evaluation Substitution-independent evaluation of an edge label in a current state σ_j, where $0 \leq j < i$, to derive a "conditional validity" or invalidity of the edge label.

Final evaluation Final evaluation of a "conditionally valid" edge label in state σ_i, when the local substitution β_i and therefore the global substitution $\hat{\beta}$ are given, to derive the validity or invalidity of the edge label.

4.2 Preliminary Evaluation

4.2.1 Substitution Independent Evaluation

The idea of the following algorithm is to substitute state-dependent elements of an edge label, namely attributes and events, by their interpretations in a current state σ_{ind} and to simplify this formula as far as possible. The resulting formula, the so-called *state information*, contains the minimal information about the current state σ_{ind} necessary to finally evaluate the edge label in any later state.

Algorithm 4.6 *Substitution-independent evaluation of an edge label*
For a given edge (n_1, n_2) of a reversed transition graph T_φ^{-1}, a current state $\sigma_{ind} \in \hat{\sigma}$ and a currently occurring event $event\_id_{cur}(par_{cur_1}, \ldots, par_{cur_n})$, the edge label $\eta'(n_1, n_2)$ of (n_1, n_2) is evaluated as follows:

begin
    $SI_{ind}(n_1, n_2) := \eta'(n_1, n_2)$;
    for each attribute $attribute\_id$ in $SI_{ind}(n_1, n_2)$ **do**
        substitute
            $attribute\_id$
        by
            $\sigma_{ind}(attribute\_id)$;
    end
    for each event $event\_id\,(par_1, \ldots, par_m)$ in $SI_{ind}(n_1, n_2)$ **do**
        if $event\_id = event\_id_{cur}$ **and** $m = n$
        then substitute
                $event\_id\,(par_1, \ldots, par_m)$
            by
                true $\wedge\ par_1 = par_{cur_1} \wedge \ldots \wedge par_m = par_{cur_n}$;
        else substitute
                $event\_id\,(par_1, \ldots, par_m)$
            by
                false;
        fi
    end
    Simplify $SI_{ind}(n_1, n_2)$;
end

Definition 4.7 $SI_{ind}(n_1, n_2)$ is called *state information of edge* (n_1, n_2) in σ_{ind}.

The state information of an edge constitutes an edge label partially evaluated in a current state. That means, state-dependent elements are interpreted and the resulting formula is simplified. In some cases, the derived formula is **true**, or **false** respectively (namely, if the edge label does not contain free variables). In the other cases, the derived formula contains free variables and gives requirements for their possible bindings.

Example 4.8 Algorithm 4.6 is presented for

- edge $(1,0)$ of T_φ^{-1} with edge label
  $\eta'(1,0) =$ `get_admission(p,u)` **and not**`(apply_for_admission(p,u))`
- current state σ_2 with currently occurring event
  `get_admission(Tim,Cam)`

begin
$SI_2(1,0) := \eta'(1,0)$;
`get_admission(p,u)` in $SI_2(1,0)$ substituted by
true and `p = Tim` **and** `u = Cam`;
`apply_for_admission(p,u)` in $SI_2(1,0)$ substituted by
false;
$SI_2(1,0) =$ **true and** `p = Tim` **and** `u = Cam` **and not(false)**
$=$ `p = Tim` **and** `u = Cam`;
end

The state information of an edge label can be **true** ($\equiv$ valid edge label), **false** ($\equiv$ invalid edge label) or a formula, which contains free variables of the temporal precondition φ ($\equiv$ "conditionally valid" edge label).

4.2.2 Managing State Information

In the last section we have shown, how single edge labels are partially evaluated without a given substitution of their free variables. A state information is derived, which contains the conditions about admissible substitutions of the free variables in the considered state σ_{ind}. In this section we show, how the derived state informations of all edges evaluated in the considered state σ_{ind}, are managed within the stepwise evaluation of a reversed transition graph. The idea is to collect the state informations of all incoming edges of a node and to store this combined state information with the node. The state information of a node can be **true** ($\equiv$ the node belongs to the current marking), **false** ($\equiv$ the node does not belong to the current marking) or a formula, which contains free variables of the temporal precondition φ ($\equiv$ the node belongs to the current marking under certain conditions). The state information contains the conditions about admissible substitutions of the free variables of φ for which the corresponding node belongs to the current marking.

Definition 4.9 $SI_{ind}(n)$ is called *state information of node n* in σ_{ind}.

Notation $ind(n)$ denotes the current index of the state information of node n, therefore the value of ind in $SI_{ind}(n)$.

Before we show, how the state informations of edges are collected and stored with the nodes of a reversed transition graph, we have to initialize the state informations of the nodes.

Algorithm 4.10 *Initialization of state information*
For a given reversed transition graph T_φ^{-1} of a temporal precondition φ, the state information is initialized as follows:

begin
    for each $n \in N$ **do**
        if $n \in F$
        then $SI_{-1}(n) :=$ **true**;
        else $SI_{-1}(n) :=$ **false**;
        fi
    end
end

Therefore, the state informations of all final nodes are initialized to **true**, that means, all final nodes belong to the initial current marking.

Example 4.11 Initialization of state information for T_φ^{-1}:

$SI_{-1}(0) :=$ **false**; $SI_{-1}(1) :=$ **false**; $SI_{-1}(2) :=$ **true**

In the following we present the algorithm for collecting the state informations of edges and deriving the state informations of nodes in a considered state σ_{ind}.

Algorithm 4.12 *Transfer of state information*
For a given reversed transition graph T_φ^{-1} of a temporal precondition φ, a current state $\sigma_{ind} \in \hat{\sigma}$ and a currently occurring event $event\_id_{cur}(par_{cur_1}, \ldots, par_{cur_n})$, the state information is transferred as follows:

begin
    if $ind = 0$
    then Initialize the state information (algorithm 4.10);
    fi
    for each $n \in N$ **do**
        $SI_{ind}(n) :=$ **false**;
        for each $(n_{pre}, n) \in E'$ **do**
            Derive the state information $SI_{ind}(n_{pre}, n)$
                of (n_{pre}, n) in σ_{ind} (algorithm 4.6);
            $SI_{ind}(n) := SI_{ind}(n) \vee (SI_{ind-1}(n_{pre}) \wedge SI_{ind}(n_{pre}, n))$;
        end
        Simplify $SI_{ind}(n)$;
    end
    for each $n \in N$ **do**
        Substitute $SI_{ind-1}(n)$ by $SI_{ind}(n)$;
    end
end

The fundamental step of algorithm 4.12 is done in the innermost **for each**-construct. All incoming edges of node n are considered there. First the state information of an incoming edge is derived in the current state σ_{ind} using algorithm 4.6 and then the derived formula $SI_{ind}(n_{pre}, n)$ is combined conjunctively with the "old" state information $SI_{ind-1}(n_{pre})$ of the corresponding predecessor node n_{pre}. The derived formulae are finally combined disjunctively.

Now we can integrate the derived results and present the main algorithm for the substitution-independent evaluation of a reversed transition graph, which is the revised version of algorithm 4.4 without a given global substitution $\hat{\beta} = \langle \beta_0, \beta_1, \ldots, \beta_i \rangle$. Using this algorithm we can derive the invalidity or conditional validity of a temporal precondition φ in a given state sequence $\hat{\sigma}$.

Algorithm 4.13 *Future-directed substitution-independent evaluation of a reversed transition graph*

For a given reversed transition graph T_φ^{-1} of a temporal precondition φ and a state sequence $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle$, the reversed transition graph is evaluated as follows:

begin
  $ind := 0$;
  $cm := F$; (* cm : current marking *)
  while $ind \leq i$ **do**
    $new\_cm := \{\}$;
    for each $(n_1, n_2) \in E'$ with $n_1 \in cm$ **and** $SI_{ind}(n_1, n_2) \neq$ **false**
    do
      $new\_cm := new\_cm \cup \{n_2\}$;
    end
    $cm := new\_cm$;
    if $cm = \{\}$
    then return $(\neg\exists\hat{\beta}\,(\hat{\sigma}, \hat{\beta}) \models \varphi)$; (* φ is invalid *)
    fi
    Transfer the state information of T_φ^{-1} w.r.t. σ_{ind} (algorithm 4.12);
    $ind := ind + 1$;
  od
  if $\exists\, n \in N : n \in \{m_0 \cap cm\}$
  then return $(\exists\hat{\beta}\,(\hat{\sigma}, \hat{\beta}) \models \varphi)$; (* φ is conditionally valid *)
  else return $(\neg\exists\hat{\beta}\,(\hat{\sigma}, \hat{\beta}) \models \varphi)$; (* φ is invalid *)
  fi
end

Definition 4.14 If $\exists\hat{\beta}\,(\hat{\sigma}, \hat{\beta}) \models \varphi$, then φ is called **conditionally valid in** $\hat{\sigma}$.

A conditionally valid PTL formula φ is valid, if the free variables of φ fulfill certain requirements. That means, φ is valid for certain variable bindings. Information about these variable bindings is given in the state information of the start node of the reversed transition graph T_{φ}^{-1}.

4.3 An Example

Example 4.15 Algorithm 4.13 is presented for the state sequence $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_5 \rangle$ where the following events occur:

σ_0 : `apply_for_admission(Tim,Cam)`
σ_1 : `apply_for_admission(Jon,Ox)`
σ_2 : `get_admission(Tim,Cam)`
σ_3 : `get_admission(Jon,Ox)`
σ_4 : `get_admission(Tim,Ox)`
σ_5 : `apply_for_admission(Jon,Ox)`

begin
$ind := 0$
$cm := \{1\}$

$ind := 0$ / **State** σ_0 after `apply_for_admission(Tim,Cam)`

First we evaluate the edge labels of the outgoing edges of nodes of cm in the current state σ_0:
$SI_0(2,2) =$ **true**;
$SI_0(2,1) =$ `p = Tim` **and** `u = Cam`;
The new current marking is:
$cm = \{2,1\}$
Then we derive the state informations of the nodes of the new current marking. These formulae are stored with the corresponding nodes.
$SI_0(1) =$ **true and** `p = Tim` **and** `u = Cam`
$\quad = $ `p = Tim` **and** `u = Cam`
$SI_0(2) =$ **true**

$ind := 1$ / **State** σ_1 after `apply_for_admission(Jon,Ox)`

$SI_1(2,2) =$ **true**
$SI_1(2,1) =$ `p = Jon` **and** `u = Ox`
$SI_1(1,1) =$ **true**
$SI_1(1,0) =$ **false**
$cm = \{2,1\}$
$SI_1(1) =$ (**true and** (`p = Jon` **and** `u = Ox`)) **or**
$\quad$ ((`p = Tim` **and** `u = Cam`) **and true**)
$\quad =$ (`p = Jon` **and** `u = Ox`) **or** (`p = Tim` **and** `u = Cam`)
$SI_1(2) =$ **true**

$\boxed{ind := 2 \text{ / } \textbf{State } \sigma_2 \text{ after } \texttt{get\_admission(Tim,Cam)}}$

$SI_2(2,2) = SI_2(2,2) = \textbf{true}$
$SI_2(2,1) = \textbf{false}$
$SI_2(1,0) = \texttt{p = Tim } \textbf{and} \texttt{ u = Cam}$
$cm = \{2,1,0\}$

$$
\begin{aligned}
SI_2(0) &= \texttt{((p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)) } \textbf{and} \\
&\quad\ \texttt{p = Tim } \textbf{and} \texttt{ u = Cam} \\
&= \texttt{p = Tim } \textbf{and} \texttt{ u = Cam} \\
SI_2(1) &= \texttt{(p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)} \\
SI_2(2) &= \textbf{true}
\end{aligned}
$$

$\boxed{ind := 3 \text{ / } \textbf{State } \sigma_3 \text{ after } \texttt{get\_admission(Jon,Ox)}}$

$cm = \{2,1,0\}$

$$
\begin{aligned}
SI_3(0) &= \texttt{(p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)} \\
SI_3(1) &= \texttt{(p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)} \\
SI_3(2) &= \textbf{true}
\end{aligned}
$$

$\boxed{ind := 4 \text{ / } \textbf{State } \sigma_4 \text{ after } \texttt{get\_admission(Tim,Ox)}}$

$cm = \{2,1,0\}$

$$
\begin{aligned}
SI_4(0) &= \texttt{(((p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)) } \textbf{and} \\
&\quad\ \texttt{p = Tim } \textbf{and} \texttt{ u = Ox) } \textbf{or} \\
&\quad\ \texttt{(((p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)) } \textbf{and true}\texttt{)} \\
&= \texttt{(p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)} \\
SI_4(1) &= \texttt{(p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)} \\
SI_4(2) &= \textbf{true}
\end{aligned}
$$

The event `get_admission(Tim,Ox)` has no effect on the state information of node 0, because the event `apply_for_admission(Tim,Ox)` had not occurred before.

$\boxed{ind := 5 \text{ / } \textbf{State } \sigma_5 \text{ after } \texttt{apply\_for\_admission(Jon,Ox)}}$

$cm = \{2,1,0\}$

$$
\begin{aligned}
SI_5(0) &= \texttt{(((p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)) } \textbf{and false}\texttt{)} \\
&\quad\ \textbf{or} \texttt{ (((p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)) } \textbf{and} \\
&\quad\ \textbf{not}\texttt{(p = Jon } \textbf{and} \texttt{ u = Ox))} \\
&= \texttt{p = Tim } \textbf{and} \texttt{ u = Cam} \\
SI_5(1) &= \texttt{(p = Jon } \textbf{and} \texttt{ u = Ox) } \textbf{or} \texttt{ (p = Tim } \textbf{and} \texttt{ u = Cam)} \\
SI_5(2) &= \textbf{true}
\end{aligned}
$$

end

φ is conditionally valid in $\hat{\sigma}$, because node $0 \in \{m_0 \cap cm\}$. $SI_5(0)$ contains the necessary information about admissible values of the free variables in φ.

4.4 Final Evaluation

In the previous sections we have shown, how a reversed transition graph is evaluated substitution-independent, that means, without a given substitution of its free variables. This way we can derive the invalidity or conditional validity of the corresponding temporal precondition. In this section we consider the final evaluation of a conditionally valid temporal precondition under a given global substitution $\hat{\beta}$. This global substitution is determined by the local substitution β_i in state σ_i, because we demand that the values of the free variables are fixed to these values throughout the evaluation in $\hat{\sigma}$.

Theorem 4.16 *Evaluation of a conditionally valid temporal precondition*
Given are φ, a conditionally valid temporal precondition in $\hat{\sigma} = \langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle$ and $\hat{\beta}$, a global substitution $\hat{\beta} = \langle \beta_0, \beta_1, \ldots, \beta_i \rangle$ where $\beta_i(x : s) \in \sigma_j(s)$ for all x free in φ and $0 \leq j \leq i$.
φ is valid in $\hat{\sigma}$ under $\hat{\beta}$, iff there is a state information $SI_i(n)$ with $n \in m_0 \subseteq N$ which is valid in σ_i under β_i, therefore

$$(\hat{\sigma}, \hat{\beta}) \models \varphi \Longleftrightarrow \exists\, n \in m_0\, (\sigma_i, \beta_i) \models SI_i(n).$$

Proof sketch
$\Longrightarrow$: $(\hat{\sigma}, \hat{\beta}) \models \varphi$ means that there is a path through T_φ^{-1} leading to a node $n \in m_0$, whose edges are evaluated to **true** in the corresponding states of $\hat{\sigma}$. Instead of evaluating an edge label all at once, the evaluation can be split into two steps:
1. The state-dependent elements of the edge label, namely attributes and events, are substituted by their current interpretations, and the resulting formula is simplified as far as possible.
2. The free variables of the derived formula are substituted by their current values, and the validity or invalidity of the edge label is derived.
The state information of the edge is obtained in the first step, before the edge label is finally evaluated in the second step. The conjunction of the corresponding state informations of the edges of the path is a subformula of $SI_i(n)$, which is valid in σ_i under β_i, because the free variables of φ are bounded to the same values in all β_j with $0 \leq j \leq i$.
$\Longleftarrow$: $\exists\, n \in m_0\, (\sigma_i, \beta_i) \models SI_i(n)$ means that there are paths through T_φ^{-1} leading to node $n \in m_0$, whose edges are conditionally valid in the corresponding states of $\hat{\sigma}$. $SI_i(n)$ is a disjunction of formulae. Each of these formulae is the conjunction of the corresponding state informations of a particular path. Since there is at least one of these conjunctions valid in σ_i under β_i and the free variables of φ are bounded to the same values in all β_j with $0 \leq j \leq i$, all edges of the corresponding path are evaluated to **true** and φ is therefore valid in $\hat{\sigma}$ under $\hat{\beta}$.

Theorem 4.16 expresses that we can determine whether a temporal precondition is valid or invalid in a state sequence $\hat{\sigma}$, by evaluating the state information of the start node of the corresponding reversed transition graph in state σ_i. Therefore, we have reduced the evaluation of a PTL formula (representing the considered temporal precondition) to the evaluation of a simple PL formula.

Example 4.17 The event `start_PhD(Tim,Cam)` is to occur in state σ_6.
`start_PhD(Tim,Cam)` can only occur, if the temporal precondition φ is valid under `p = Tim` and `u = Cam`.

φ is conditionally valid in $\hat{\sigma}$ (see above / example 4.15).
φ is valid in $\hat{\sigma}$ under `p = Tim` and `u = Cam`, because the state information

$SI_5(0) =$ `p = Tim` **and** `u = Cam`

is valid/**true** under `p = Tim` and `u = Cam`.

The event `start_PhD(Tim,Cam)` can occur.

Chapter 5

Integration with TROLL

According to the example in Chapter 4, we will now introduce the integration of the proposed algorithm for monitoring temporal conditions using the language TROLL.

5.1 Outline of this Approach

For every precondition incorporating a temporal formula, we have to introduce suitable additional attributes and events to *monitor* every state change that *may change* the validity of this formula. In the previous chapters, we introduced a mechanism for deriving transition graphs from temporal formulae and described an algorithm for recording the information necessary for the monitoring process.

Since the algorithm is based on the manipulation of arbitrary formulae, without specific consideration of implementation requirements, it is not suitable for a direct and efficient implementation. Therefore, we introduce another mechanism, namely extracting the historical information from the reversed transition graphs and storing it in objects, deriving the necessary operations to manipulate these objects, and integrating the monitoring operations into the description of the original object.

The transition graph with its nodes, and implicitly its node and edge labelling, are therefore described using a subset of TROLL without temporal formulae. The proposed technique can be sketched as follows (for some PTL formula F):

1. Derive the reversed transition graph of F.
2. From the nodes of the reversed transition graph, derive node classes to store state information.
3. Introduce a graph object incorporating objects from the node classes (a *monitor*).
4. Include the graph object into the original specification that contained the formula.
5. Expand the original specification with calling rules for events that influence the temporal condition F (notification to the monitor).

6. Transform the temporal condition into a first order condition using the derived state information (request to the monitor).

Step one to four describe the additional specification necessary for an operational control of temporal preconditions whereas step five and six describe the runtime behaviour of the monitoring process.

After introducing an example TROLL specification containing a temporal condition, we will introduce this technique in more detail.

5.2 Modelling the Example Context

The example in Chapter 4 deals with a world of universities and students, the latter trying to start a PhD course in some university. The process of starting a PhD course requires to firstly applying for admission at an university, and then getting admission from that university. A simple specification in TROLL takes the following form:

```
object AdmissionOffice
  attributes
    PhDStudents:set(string) .
  events
    bigBang birth .
    applyForAdm(Name:string,Univ:string) .
    getAdm(Name:string,Univ:string) .
    startPhD(Name:string,Univ:string)
      enabled
         ( sometime past(applyForAdm(Name,Univ)) and
           sometime(getAdm(Name,Univ)) since last(applyForAdm(Name,Univ)) ) ;
      changing PhDStudents := insert(Name,PhDStudents) .
    getPhD(Name:string,Univ:string)
      changing remove(Name,PhDStudents) .
end object AdmissionOffice
```

Here we introduced a simple (single) object `AdmissionOffice` containing one attribute representing a set of currently working PhD students and the events used in the original example. A more natural specification uses object classes for universities, persons, person roles for PhD-students etc., and suitable communication relationships between these object classes.

For clarity we only used a rather simple object representing only PhD-students as sets of strings. The event `applyForAdm` (formerly `apply_for_admission`) is restricted by a temporal condition stating that a `startPhD` event can only occur in this objects life (for given parameter values) if the person has applied for admission (`applyForAdm`) sometime in the past and that she/he got admission (`getAdm`) later on (see also figure 4.2).

5.3 Modelling the Monitoring Process in `TROLL`

5.3.1 Overview

The problematic part of the `AdmissionOffice`-specification for prototyping and the later implementation process is the temporal condition for the event `startPhD`. To achieve an efficient control of this precondition, the original specification has to be expanded and altered in several steps. These steps are derived from the discussion on the future-directed evaluation of reversed transition graphs during an objects life. Rather informally we can identify the following steps to be necessary:

Step 0. Derive the reversed transition graph G for the enabling condition to be monitored (here `startPhD`). We consider this step to be done using the algorithms in Chapter 4.

Step 1. For every node in graph G, derive *node classes* that allow the recording of state information. In principle, these classes must represent relations with attributes derived from the free variables of the node and edge formulae and some additional information. The objects of these classes are then used in an encapsulating graph object.

Step 2. After *relevant* state transitions the new state information stored in the node class objects must be derived. Relevant state transitions in our example are occurrences of the events `applyForAdm` and `getAdm`. In general, all events that change state dependent properties appearing in the formulae at hand are *relevant*.

After the occurrence of relevant events, the new current marking (see Alg. 4.13) has to be determined and the state information has to be updated in parallel to the occurrence of relevant events.

Step 3. Alter the temporal condition of event `startPhD` to a condition using the state information and the current marking of the start node m_0. This constitutes a new specification without temporal formulae and additional state information.

Step 2 will be introduced in two subtasks since the derivation of the new current marking explains the idea and is the basis for the state information transfer. Furthermore, it may be used for optimization purposes later on.

We will now informally sketch this process by means of an example *specification* using `TROLL`. The goal is to eliminate the temporal condition and transform it into first order conditions using the information recorded during the life of the `AdmissionOffice`-object.

5.3.2 Step 1: Derivation of Node Classes

The first class to be specified represents the state information stored with the nodes of the reversed transition graph.

```
object class NodeInfo
  identification NI:(NodeNo)
  attributes
     NodeNo:nat .
     CM:bool .                              /* current marking   */
     SI:set(tuple(Name:string, Univ:string)) /* state information */
       initialised emptyset .
     MarkedFor(Name:string, Univ:string):bool
       derived as ( in(<Name,Univ>,SI) ).
  events
     setup(Init:bool, NodeNo:nat) birth
       changing CM := Init .
     mark(Value:bool)
       changing CM := Value .
     insert(Name:string,Univ:string)
       changing
          SI := insert(<Name,Univ>,SI) .
     remove(Name:string,Univ:string)
       changing
          SI := remove(<Name,Univ>,SI) .
end object class NodeInfo ;
```

All nodes of the graph G are representable with the object class `NodeInfo`. Recall that the state information was formally introduced as a first order logic formula structured as a disjunction of and-connected equalities with parameters, or being simply *false* or *true* representing unconditional (in)validity. This is represented as a *relation* state information `SI` and an *attribute* `CM`. The bool-valued attribute `CM` and the derived bool-valued attribute `MarkedFor` are introduced as interface, the latter as derived from `SI`. The notation `<Name,Univ>` is used to describe *tuples* of data values.

In this example it is sufficient to introduce only one `NodeInfo` class since all information (tuples of data values) to be stored are of the form `<Name:string,Univ:string>`. For ease of discussion, the final node `(2)` is represented in the same way, although its marking can only be *true* or *false*. For a more general translation process we would introduce a generalized class `Node` and specialize [JSHS91] it for the appropriate graphs respectively nodes. For now we will only sketch the general ideas.

To integrate the node class objects and to make creation and manipulation more transparent, we introduce an object `Graph` that contains node class objects as components and supports initialization:

```
object Graph
  context NodeInfo
  attributes
```

```
    N(i:nat):|NodeInfo|                /* abbreviation, see text */
      derived as Nodes(NI(i)).OID .
  components
    Nodes:NodeInfo set .
  events
    variables I:nat, Val:list(bool), OID:|NodeInfo|;
    create
      birth ;
      calling
        { I>=0 and I<=2 and Val=[false,false,true] } createNode(I,Val[I]) .
    createNode(No:nat, Val:bool)
      calling
        NodeInfo.setup(OID,No,Val) ;
        Nodes.INSERT(OID) .
end object Graph ;
```

The attributes `N(i:nat):|NodeInfo|` are a convenient way for introducing short names or abbreviations. Here these attributes denote identifiers for `NodeInfo` instances. It should be noted, however, that these attributes are only *defined* in a given state for objects being in the component set `Nodes`! The creation of `NodeInfo` objects is done as depicted in Chapter 3. The creation event `setup` is called in the *class object* `NodeInfo`. Compared to the (instance) birth event `setup` this (class) event `setup` has additional parameters for an object identity (`OID`) to be created when the event occurs and key attributes (`NodeNo:nat`).

5.3.3 Step 2: Monitoring Update Operations

The substitution-independent evaluation of reversed transition graphs is performed during runtime of an object. This implies that the state information stored in the node class objects must be updated everytime relevant events occur. Therefore, the node class objects are incorporated into the graph object for this rule, which itself becomes part of the original object.

Initialization

The first step of this process (according to algorithm 4.4) is to initialize the current marking and the initial state information. The necessary information is encapsulated in the specific graph object as shown above. The connection to the `AdmissionOffice` object is achieved by enriching the `bigBang` event with an additional calling rule that initializes the graph object:

```
object AdmissionOfficeTransformed
```

```
  context Graph ;
  components
    Monitor:Graph  hidden .
  events
    variables OID:|Graph|
    bigBang
      calling
        Graph.create(OID) ;              /* (1)
        Monitor.INSERT(OID) .            /* (2)
  ...
end object AdmissionOfficeTransformed ;
```

The calling rule introduced here creates a new `Graph` instance when the event `bigBang` occurs (1), and inserts the new instance into the component `Monitor` (2). The numbering of nodes (see the graph specification) is obtained from the example in Chapter 4.

Update of Current Marking

The next step is to integrate the computation of the new current marking. Therefore, the events `applyForAdm` and `getAdm` must be taken into account. In this example only these two operations have an effect on the state information to be stored in the node classes, because they are the only events changing state dependent properties used in the original formula.

According to algorithm 4.13 we have to decide which formulae are false *in all cases*. Only an edge label that is constructed without event-occurs-predicates for events with parameters may be surely *false* in this sense. If event-occurs- predicates with parameters are present (like in this example) we have to supply additional rules (see below).

With the occurrence of one of the events (`applyForAdm` or `getAdm`), the mark-update events in the node components (end nodes) are called with bool expressions obtained from the edge label formulae of the reversed transition graph. With some optimizations using first order equalities we informally infer the following specification of the graph object modelling the update of the current marking (algorithm 4.13):

```
object Graph
  ...
  events
    ...
    updateFor(Evt:string)
      calling
        { Evt = 'applyForAdm' }
          Nodes(N(2)).mark( Nodes(N(2)).CM ),
          Nodes(N(1)).mark( Nodes(N(2)).CM or Nodes(N(1)).CM ),
          Nodes(N(0)).mark( Nodes(N(0)).CM ) ;
        { Evt = 'getAdm' }
```

```
        Nodes(N(2)).mark( Nodes(N(2)).CM ),
        Nodes(N(1)).mark( Nodes(N(1)).CM ),
        Nodes(N(0)).mark( Nodes(N(1)).CM or Nodes(N(0)).CM) ) .
  ...
end object Graph ;
```

Which can be further optimized to

```
object Graph
  ...
  events
    ...
    updateFor(Evt:string)
      calling
        { Evt = 'applyForAdm' }  Nodes(N(1)).mark( true ) ;
        { Evt = 'getAdm' }  Nodes(N(0)).mark( Nodes(N(1)).CM ) .
  ...
end object Graph ;
```

taking into account that some of the `mark`-update events do not change the state of the destination nodes. For the time being we can then use the `updateFor`-event without parameters (still a preliminary specification of the final monitoring process) to *notify* the state change to the *monitoring* graph object:

```
object AdmissionOfficeTransformed
  ...
  events
    ...
    applyForAdm(Name:string,Univ:string)
      calling
        Monitor.updateFor('applyForAdm') .
    getAdm(Name:string,Univ:string)
      calling
        Monitor.updateFor('getAdm') .
  ...
end object AdmissionOfficeTransformed ;
```

Only the previous current marking together with the occurring events, which are mentioned in the edge labels, determine the new current markings. This is not true in general. If the edge labels mention other state dependent properties, like attributes, then the events changing these attributes must also trigger the node information update.

After this first step only the invalidity of the precondition for `startPhD` can be decided, namely, if the marking becomes empty:

$$(\texttt{Nodes(N(0)).CM}\ \textit{or}\ \texttt{Nodes(N(1)).CM}\ \textit{or}\ \texttt{Nodes(N(2)).CM}) \equiv \textit{false}$$

Then the object can never reach a state, where the precondition of `startPhD` is valid. Taking the parameter values into account we can refine the monitoring process.

Update of State Information

To decide on the conditional validity of the given PTL formula, the transfer of state information is the next step before the final evaluation can be performed. With the state change induced by one of the events `applyForAdm` or `getAdm` the current state information must be transferred.

Since the algorithms introduced in Chapter 4 rely on evaluation of arbitrary first order formulae, this step requires some additional work. The evaluation of edge labels leads to either ***true*** or ***false*** if only simple formulae are used, but they deliver parameter bindings if events with parameters are involved. In the `TROLL` representation, the state information transfer will be done in parallel with the original events for all nodes of the graph.

The algorithm for the state information transfer combines the incoming edge evaluation of a node with the corresponding state information (formulae) of its predecessor nodes, i.e. the corresponding formulae are used *as a whole*. Clearly this is not always necessary since the state information stored in the nodes as relations is in some sense "*continuous*". Tuples of parameter values are inserted respectively deleted depending on the event predicates contained in the edge labels.

This means that the *tuple transfer* or *state information transfer* can be performed in the same way as the calculation of the new marking introduced in the last section. Consider the graph G represented as a matrix with the columns representing the incoming and the rows the outgoing edges of the nodes. Now the current parameter values (`p:string,u:string`) are taken into account:

	end-nodes of an edge		
start-	2	1	0
2	true	applyForAdm(p,u)	
1		true	getAdm(p,u) and not applyForAdm(p,u)
0			not applyForAdm(p,u)

The occurrence of events – forgetting about parameter values – has been considered in the previous step. If parameters are involved, *insert* respectively *remove* operations for the node class objects must be generated. These operations will be performed using calling rules to the graph and node objects.

In principle, the calling rules can be inferred like the rules for the update of the current marking. Some edge labels are evaluated to ***false*** in every case, e.g. `applyForAdm` from node 2 to 1 if the event `getAdm` occurs. Again we assume that only one event occurs at a time. Other events like ***not*** `applyForAdm(Name,Univ)` from node `0` to `0` *may be* evaluated to ***true*** for parameter combinations other than the ones delivered with the occurring event, that is, they will lead to *remove* operations of the actual parameter tuple if `applyForAdm` occurs (see below).

A condition like

`getAdm(Name,Univ)` *and not* `applyForAdm(Name,Univ)` (node 1 to node 0)

is transformed to (Alg. 4.6):

false and not `(` *true and* `p = Name` *and* `u = Univ )`

and is always evaluated to *false* when event `applyForAdm` occurs since the first part of the *and* is evaluated to *false*. Nothing has to be inserted or removed in this case. On the other hand, with the occurrence of `applyForAdm` the condition *not* `applyForAdm(p,u)` (node 0 to node 0) is transformed to (Alg. 4.6):

not `(`*true* `And p = Name` *and* `u = Univ )`

> *which is valid for all substitutions of the free variables* ***p*** *and* ***u*** *different from the ones of the currently occurring event* ***applyForAdm(Name,Univ)****. It is not valid for the substitution* ***p->Name,u->Univ*** *which leads to a removal of this tuple from the state information stored in node* ***0****.*

The rather complicated explanation of the derivation of insert and remove operations is necessary, because the original algorithm works with formulae as state information, that are transferred *as a whole* from one node to another and combined with the edge label evaluation and the predecessor state information. Here we have to work with relations that in some sense represent these formulae, but updates are made locally within the nodes. No transfer of `SI`-relations in the nodes takes place directly (as a whole).

The formula construction has been replaced by intuitively derived manipulation operations of the node objects. For the example used here, this explanation seems to be enough. A formal proof of the equivalence of the formalization in `TROLL` and the original algorithms presented in Chapter 4 should be done for a complete description of an automatic transformation tool. Especially, the *continuity* of the state information and the formal derivation of insert and remove operations has to be shown.

The following calling rules (together with the ones from the previous specification fragment) perform the state information update inside the graph objects by means of calling `insert` and `remove` events of the `NodeInfo` objects. The `updateFor` events are therefore expanded with necessary parameter information and the rules are expanded with additional conditions modelling the dependency of the insert/remove operations from the state information stored in the relevant predecessor nodes:

```
object Graph
  ...
  events
    ...
```

```
  updateFor(Evt:string,Name:string,Univ:string)
    calling
      { Evt = 'applyForAdm' }
         Nodes(N(1)).mark( true ),           /* or: mark( Nodes(N(2)).CM)
         Nodes(N(0)).remove(Name,Univ) ,     /* node 0 to node 0 */
         Nodes(N(1)).insert(Name,Univ) ;     /* node 2 to node 1 */
      { Evt = 'getAdm' }
         Nodes(N(0)).mark( Nodes(N(1)).CM ) ;
      { Evt = 'getAdm' and Nodes(N(1)).MarkedFor(Name,Univ) }
         Nodes(N(0)).insert(Name,Univ) .     /* node 1 to node 0 */
```

As we can see now the mark-events resemble the insert events in that a *true*-marking conforms to an insert event. The `mark` events are however not very fine grained and may only be used for optimization of the monitoring process. In this example a further optimization is not possible since the temporal condition at hand can never become *invalid for all continuations of life cycles*.

5.3.4 Step 3: Checking the Transformed Constraint

The final rewriting of the original specification considers the original temporal condition for the event `startPhD`. Now the node class object, which represents the formerly called start node m_0, together with the state information stored in this node (node 0) can be used to derive a new (first order) condition that replaces the temporal condition.

```
object AdmissionOfficeTransformed
  ...
  events
    ...
    startPhD(Name:string,Univ:string)
        enabled Monitor.Enabled(Name,Univ) .
  ...
end object AdmissionOfficeTransformed
```

where the `Enabled` attribute is derived from the node class objects in the following way:

```
object Graph
  ...
  attributes
    Enabled(Name:string,Univ:string)
      derived as Nodes(N(0)).MarkedFor(Name,Univ) .
  ...
end object Graph
```

In this case it is enough to see, whether node 0 is marked for the parameter values of the currently occurring event. We only need the information that the parameter combination is part of the state information, then the node is in the current marking automatically: `Node(0).CM` $\equiv$ *true*.

5.4 Specification of the Monitoring Process

To conclude this chapter we will now present the final modelling of the transformed admission office object with the specification of the monitoring objects:

```
object AdmissionOfficeTransformed
  context Graph ;
  components
    Monitor:Graph  hidden .
  attributes
    PhDStudents:set(string) .
  events
    bigBang
      birth ;
      calling
        Graph.create(OID) ,
        Monitor.INSERT(OID) .
    applyForAdm(Name:string,Univ:string)
      calling
         Monitor.updateFor('applyForAdm',Name,Univ) .
    getAdm(Name:string,Univ:string) .
      calling
         Monitor.updateFor('getAdm',Name,Univ) .
    startPhD(Name:string,Univ:string)
      enabled Monitor.Enabled(Name,Univ) .
    getPhD(Name:string,Univ:string)
      changing remove(Name,PhDStudents) .
end object AdmissionOffice ;
```

The `Graph` specification that incorporates the structure and interconnection of the graph respectively the node objects:

```
object Graph
  context NodeInfo ;
  attributes
    Enabled(Name:string,Univ:string)
```

```
      derived as Nodes(N(0)).MarkedFor(Name,Univ) .
    N(i:nat):|NodeInfo| derived as Nodes(NI(i)).OID .
  components
    Nodes:SET(NodeInfo) .
  events
    variables I:nat, Val:list(bool), OID:|NodeInfo|;
    create
      birth
      calling
        { I>=0 and I<=2 and Val=[false,false,true] } createNode(I,Val[I]) .
    createNode(No:nat, Val:bool)
      calling
        NodeInfo.setup(OID,No,Val) ;
        Nodes.INSERT(OID) .
    updateFor(Evt:string,Name:string,Univ:string)
      calling
        { Evt = 'applyForAdm' }
            Nodes(N(1)).mark( true ),
            Nodes(N(0)).remove(Name,Univ) ,
            Nodes(N(1)).insert(Name,Univ) ;
        { Evt = 'getAdm' }
            Nodes(N(0)).mark( Nodes(N(1)).CM ) ;
        { Evt = 'getAdm' and Nodes(N(1).MarkedFor(Name,Univ) }
            Nodes(N(0)).insert(Name,Univ) .
end object Graph ;
```

The `NodeInfo` specification incorporating state information storage for nodes `0` and `1`. We used only one kind of nodes for the monitoring graph, namely one that manages simple information about a node to be marked or not (node 2) and information about parameter values of events that occurred in the past (nodes `0` and `1`).

```
object class NodeInfo
  identification NI:(NodeNo)
  attributes
     NodeNo:nat .
     CM:bool .                                /* current marking   */
     SI:set(tuple(Name:string, Univ:string)) /* state information */
       initialised emptyset .
     MarkedFor(Name:string, Univ:string):bool
       derived as ( in(<Name,Univ>,SI) or CM ).
  events
     setup(Init:bool, NodeNo:nat) birth
```

```
    changing CM := Init .
  mark(Value:bool) changing CM := Value .
  insert(Name:string,Univ:string)
    changing
      SI := insert(<Name,Univ>,SI) .
  remove(Name:string,Univ:string)
    changing
      SI := remove(<Name,Univ>,SI) .
end object class NodeInfo ;
```

The discussed transformation process can be done automatically up to this point. It is not clear, however, to what extend this specification can be optimized automatically. Some calling rules can be eliminated, for example, if they describe transferring the marking from true initialized nodes to the same node as shown above. For instance, in this example after the initialization, the current marking of node 2 remains *true*.

Chapter 6

Conclusion and Outlook

`TROLL` as a language for the conceptual modelling of information systems goes beyond most of the so-called semantic data models in that the *system dynamics* in terms of long term object behaviour is explicitly integrated into the object model. As one means for the specification of dynamic evolution of objects `TROLL` uses a *past temporal logic* to restrict object evolution. State changes of the modelled system (called events) are often dependent on the *history* of the objects involved. Thus, the history restricts the possible future evolution. One (naive) solution to monitor such dependencies is to explicitly *store the history* of events of such a system. For a practical solution this approach is infeasible. For implementation (or prototyping of conceptual models) in case of more than toy examples, only a *history-less* approach [Cho92] is imaginable.

The method for monitoring temporal constraints and its integration into the language framework introduced in this report is different from the naive solution of storing the whole history in that only state changes *relevant for the validity of temporal conditions of some specification* are monitored. This is possible because in database terms `TROLL` integrates *data* and *operations* within one formal framework. The work by [Cho92] resembles our approach. In contrast however we use an explicit dependency of temporal conditions and *update operations* so that the monitoring can be optimized in several ways.

Temporal formulae restricting the occurrence of events are first structurally transformed into *(reversed) transition graphs* where edges are labelled with subformulae of the original formula. In case of state transitions these subformulae are evaluated and control the transfer of *markings* between graph nodes. The nodes with their markings represent the validity of temporal subformulae. One node (or a set of nodes), the end node(s) of the graph (which represents the original formula) can then be used to decide on the possibility of the state change that is restricted by conditions on the object history. Since state changes induced by operations may be *parameterized*, information about theses parameters has to be stored and transferred between the graph nodes in the same way.

We presented how this technique can be integrated into the specification of some system specified in `TROLL` by means of *adding* objects that store *necessary* information about *relevant state changes* during the life of an object. The original specification has to be altered in three places:

1. *calling rules* for events that *may influence* the validity of the temporal condition to be monitored (relevant events),
2. *additional objects* storing information about relevant events derived from the temporal condition, and
3. a *simplified condition* referencing data stored in the objects derived in step 2.

The advantages of this approach can be summarized as follows:

- The stored information about the history is minimized.
- Monitoring is *localized* in (database) objects. The monitor for some object is only triggered if the object changes. This feature of the proposed monitoring process leads to an efficient procedure.
- The language framework can be retained, there is no need for implementing the evaluation of (past) temporal formulae in a different language.
- The mechanism is closely related to the update framework of `TROLL` leading to an optimized checking procedure.
- The original specification has to be altered minimally. Although complex classes for the monitoring process have to be derived, this step can be done independently of the rest of the specification.
- Additional update operations can be integrated incrementally. If they are classified as *relevant* for some temporal condition, additional calling rules to the monitoring objects have to be generated.

In the future several problems concerning the work described here have to be solved:

- Formalizing the specification of necessary insert and remove operations. A formal proof of equivalence of the algorithms and the proposed modelling in `TROLL` is necessary.

  The procedure to derive additional classes monitoring the update operations introduced in this report has been described only intuitively. Also, the construction of the set of *relevant events* was only sketched.
- The modelling of node classes and the graph representing the temporal formula was done straightforwardly. For an automated procedure, we have to supply *generic classes* for transition graphs and nodes, that can then be specialized for each condition to be monitored.
- Derivation of additional optimization rules if some monitoring operations are redundant or the monitoring process can be stopped. The latter may be the case if the marking of nodes becomes empty in the course of object evolution.

- `TROLL` makes it possible to describe state changes where *sets* of events take place at one time (snapshots). This concept is used throughout the specification of Chapter 5 where all monitoring operations take place within *one* state transition. Concurrently occurring events on the application level must also be handled.

  Also, events in the original application domain may occur concurrently. For example the monitoring objects introduced here cannot handle *two* ***applyForAdm****-events occurring in parallel*. The generic monitoring framework must be liberal enough to handle this `TROLL` feature

- Integration with future-directed temporal logic. `TROLL` also supports a future temporal logic for the specification of *possible attribute evolutions*. Similar techniques for the monitoring of future-directed constraints, also based on *transition graphs*, have been worked out in [Lip89, Lip90, Saa88, Saa91].

Bibliography

[Aba88] Abadi, M.: The Power of Temporal Proofs. Research Report 30, Digital Equipment, 1988.

[Ara91] Arapis, C.: Temporal Specification of Object Behaviour. In: Thalheim, B.; Demetrovics, J.; Gerhardt, H.-D. (eds.): *Proc. 3rd Symp. Mathematical Fundamentals of Database and Knowlede Base Systems (MFDBS).* Springer-Verlag, LNCS 495, 1991, pp. 308–324.

[CGH92] Conrad, S.; Gogolla, M.; Herzig, R.: TROLL *light*: A Core Language for Specifying Objects. Informatik-Bericht 92–02, TU Braunschweig, 1992.

[Cho92] Chomicki, J.: History-less Checking of Dynamic Integrity Constraints. In: Golshani, F. (ed.): *Proc. of the Eight IEEE Conference on Data Engineering*, 1992, pp. 557–564.

[EDS93] Ehrich, H.-D.; Denker, G.; Sernadas, A.: Constructing Systems as Object Communities. In: Gaudel, M.-C.; Jouannaud, J.-P. (eds.): *Proc. TAPSOFT'93: Theory and Practice of Software Development.* LNCS 668, Springer, Berlin, 1993, pp. 453–467.

[EGS90] Ehrich, H.-D.; Goguen, J. A.; Sernadas, A.: A Categorial Theory of Objects as Observed Processes. In: deBakker, J.W.; deRoever, W.P.; Rozenberg, G. (eds.): *Proc. REX/FOOL Workshop*, Noordwijkerhood (NL), 1990. LNCS 489, Springer, Berlin, pp. 203–228.

[ELG84] Ehrich, H.-D.; Lipeck, U. W.; Gogolla, M.: Specification, Semantics, and Enforcement of Dynamic Database Constraints. In: *Proc. Int. Conf. on Very Large Databases VLDB '84*, Singapore, 1984. pp. 301–308.

[ES91] Ehrich, H.-D.; Sernadas, A.: Fundamental Object Concepts and Constructions. In: Saake, G.; Sernadas, A. (eds.): *Information Systems – Correctness and Reusability.* TU Braunschweig, Informatik Bericht 91-03, 1991, pp. 1–24.

[ESS89] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: Objects, Object Types, and Object Identification. In: Ehrig, H.; Herrlich, H.; Kreowski, H.-J.; Preuß, G. (eds.): *Categorical Methods in Computer Science.* LNCS 393, Springer, Berlin, 1989, pp. 142–156.

[ESS90] Ehrich, H.-D.; Sernadas, A.; Sernadas, C.: From Data Types to Object Types. *Journal on Information Processing and Cybernetics EIK*, Vol. 26, No. 1-2, 1990, pp. 33–48.

[FSMS91] Fiadeiro, J.; Sernadas, C.; Maibaum, T.; Saake, G.: Proof-Theoretic Semantics of Object-Oriented Specification Constructs. In: Meersman, R.; Kent, W.; Khosla, S. (eds.): *Object-Oriented Databases: Analysis, Design and Construction (Proc. 4th IFIP WG 2.6 Working Conference DS-4, Windermere (UK))*, Amsterdam, 1991. North-Holland, pp. 243–284.

[HJS92] Hartmann, T.; Jungclaus, R.; Saake, G.: Aggregation in a Behavior Oriented Object Model. In: Lehrmann Madsen, O. (ed.): *Proc. European Conference on Object-Oriented Programming (ECOOP'92)*. Springer, LNCS 615, Berlin, 1992, pp. 57–77.

[HJS93] Hartmann, T.; Jungclaus, R.; Saake, G.: Animation Support for a Conceptual Modelling Language. In: Mařík, V.; Lažanský, J.; Wagner, R.R. (eds.): *Proc. 4th Int. Conf. on Database and Expert Systems Applications (DEXA), Prague.* LNCS 720, Springer, Berlin, 1993, pp. 56–67.

[HS91] Hülsmann, K.; Saake, G.: Theoretical Foundations of Handling Large Substitution Sets in Temporal Integrity Monitoring. *Acta Informatica*, Vol. 28, No. Fasc 4, 1991, pp. 365–407.

[HS93] Hartmann, T.; Saake, G.: Abstract Specification of Object Interaction. Informatik-Bericht 93–08, Technische Universität Braunschweig, 1993.

[HSJ+93] Hartmann, T.; Saake, G.; Jungclaus, R.; Hartel, P.; Kusch, J.: TROLL–2 report. Informatik-bericht, TU Braunschweig, 1993. *In preparation.*

[JHS93] Jungclaus, R.; Hartmann, T.; Saake, G.: Relationships between Dynamic Objects. In: Kangassalo, H.; Jaakkola, H.; Hori, K.; Kitahashi, T. (eds.): *Information Modelling and Knowledge Bases IV: Concepts, Methods and Systems (Proc. 2nd European-Japanese Seminar, Hotel Ellivuori (SF))*. IOS Press, Amsterdam, 1993, pp. 425–438.

[JSHS91] Jungclaus, R.; Saake, G.; Hartmann, T.; Sernadas, C.: Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91-04, TU Braunschweig, 1991.

[Jun93] Jungclaus, R.: *Modeling of Dynamic Object Systems—A Logic-Based Approach.* Advanced Studies in Computer Science. Vieweg Verlag, Braunschweig/Wiesbaden, 1993.

[Krö87] Kröger, F.: *Temporal Logic of Programs.* Springer-Verlag, Berlin, 1987.

[Kun84] Kung, C.H.: Temporal Framework for Database Specification and Verification. In: *Proc. 5th Int. Conf. on Very Large Databases (VLDB)*, 1984, pp. 91–99.

[LEG85] Lipeck, U. W.; Ehrich, H.-D.; Gogolla, M.: Specifying Admissability of Dynamic Database Behaviour Using Temporal Logic. In: Sernadas, A. et al. (eds.): *Proc. IFIP Working Conf. on Theoretical and Formal Aspects of Information Systems*. North-Holland, Amsterdam, 1985, pp. 145–157.

[Lip89] Lipeck, U. W.: *Zur dynamischen Integrität von Datenbanken: Grundlagen der Spezifikation und Überwachung*. Informatik-Fachbericht 209. Springer, Berlin, 1989.

[Lip90] Lipeck, U. W.: Transformation of Dynamic Integrity Constraints into Transaction Specifications. *Theoretical Computer Science*, Vol. 76, 1990, pp. 115–142.

[LS87] Lipeck, U. W.; Saake, G.: Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, Vol. 12, 1987, pp. 255–269.

[MP91] Manna, Z.; Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Springer-Verlag, New York, 1991.

[MW84] Manna, Z.; Wolper, P.: Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 6, 1984, pp. 68–93.

[Saa88] Saake, G.: *Spezifikation, Semantik und Überwachung von Objektlebensläufen in Datenbanken*. PhD thesis, Technische Universität Braunschweig, 1988.

[Saa91] Saake, G.: Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, Vol. 6, No. 1, 1991, pp. 47–74. North-Holland.

[Saa93] Saake, G.: *Objektorientierte Spezifikation von Informationssystemen*. Teubner, Leipzig, 1993. Habilitationsschrift, *To appear Autumn 1993*.

[Sch92] Schwiderski, S.: Realisation von Objekten in einem Relationalen Datenbanksystem. Diploma thesis, TU Braunschweig, 1992.

[Ser80] Sernadas, A.: Temporal Aspects of Logical Procedure Definition. *Information Systems*, Vol. 5, 1980, pp. 167–187.

[SF91] Sernadas, C.; Fiadeiro, J.: Towards Object-Oriented Conceptual Modelling. *Data & Knowledge Engineering*, Vol. 6, 1991, pp. 479–508.

[SL89] Saake, G.; Lipeck, U.W.: Using Finite-Linear Temporal Logic for Specifying Database Dynamics. In: Börger, E.; Kleine Büning, H.; Richter, M. M. (eds.): *Proc. CSL'88 2nd Workshop Computer Science Logic*. Springer, Berlin, 1989, pp. 288–300.

[SS93] Schwiderski, S.; Saake, G.: Monitoring Temporal Permissions using Partially Evaluated Transition Graphs. In: Lipeck, U.; Thalheim, B. (eds.): *Proc. 4th International Workshop: Modelling Database Dynamics, Volkse 1992*. Workshops in Computing, Springer, Berlin, 1993, pp. 196–217.

[SSC92] Sernadas, A.; Sernadas, C.; Costa, J. F.: Object Specification Logic. Research report, INESC/DMIST, Lisbon (P), 1992. *To appear in Journal of Logic and Computation.*

[SSE87] Sernadas, A.; Sernadas, C.; Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. In: Stoecker, P.M.; Kent, W. (eds.): *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*. VLDB Endowment Press, Saratoga (CA), 1987, pp. 107–116.

[US90] Unland, R.; Schlageter, G.: Object-Oriented Database Systems: Concepts and Perspectives. In: Blaser, A. (ed.): *Database Systems of the 90s, International Symposium*. Springer, Berlin, LNCS 466, 1990.

[Wie91] Wieringa, R. J.: A Conceptual Model Specification Language (CMSL Version 2). Technical Report IR–248, Vrije Universiteit, Amsterdam, 1991.

[WJ91] Wieringa, R.; Jonge, W. de: The Identification of Objects and Roles—Object Identifiers Revisited. Technical Report IR–267, Vrije Universiteit, Amsterdam, 1991.