Mag. Dipl.-Ing. Margareta Ciglič, Bakk.

Time Management in Cyclic Business Processes

DISSERTATION

zur Erlangung des akademischen Grades
Doktorin der Technischen Wissenschaften

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

Betreuer
O.Univ.-Prof. Dipl.-Ing. Dr. Johann Eder
Alpen-Adria-Universität Klagenfurt
Institut für Informatik-Systeme

Gutachter
O.Univ.-Prof. Dipl.-Ing. Dr. Johann Eder
Alpen-Adria-Universität Klagenfurt
Institut für Informatik-Systeme

Gutachter
Prof. Carlo Combi, Ph.D.
Università degli Studi di Verona
Dipartimento di Informatica

Klagenfurt a. W., August 2020

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich

- die eingereichte wissenschaftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe,
- die während des Arbeitsvorganges von dritter Seite erfahrene Unterstützung, einschließlich signifikanter Betreuungshinweise, vollständig offengelegt habe,
- die Inhalte, die ich aus Werken Dritter oder eigenen Werken wortwörtlich oder sinngemäß übernommen habe, in geeigneter Form gekennzeichnet und den Ursprung der Information durch möglichst exakte Quellenangaben (z.B. in Fußnoten) ersichtlich gemacht habe,
- die eingereichte wissenschaftliche Arbeit bisher weder im Inland noch im Ausland einer Prüfungsbehörde vorgelegt habe und
- bei der Weitergabe jedes Exemplars (z.B. in gebundener, gedruckter oder digitaler Form) der wissenschaftlichen Arbeit sicherstelle, dass diese mit der eingereichten digitalen Version übereinstimmt.

Mir ist bekannt, dass die digitale Version der eingereichten wissenschaftlichen Arbeit zur Plagiatskontrolle herangezogen wird. Ich bin mir bewusst, dass eine tatsachenwidrige Erklärung rechtliche Folgen haben wird.

Margareta Ciglič e.h. Klagenfurt a. W., August 2020

Vsem mojim najljubšim, ki jih vedno nosim v srcu
in mi bodo vedno blizu - tudi če so v drugi državi,
na drugem kontinentu ali na drugem svetu.

Zusammenfassung

In der heutigen globalisierten Welt müssen die Arbeitsabläufe rund um den Globus perfekt aufeinander abgestimmt werden. Zeitmanagement gewann sowohl im geschäftlichen wie auch im privaten Leben an Bedeutung. In vielen Geschäftsprozessen entscheidet ein gutes Zeitmanagement über den Erfolg des Prozesses und somit auch der prozessausführenden Organisation. In manchen Prozessen, wie z.B. medizinischen Behandlungen, kann ein gutes Zeitmanagement sogar den Unterschied zwischen Leben und Tod ausmachen.

Wichtige Bereiche im Zeitmanagement in Geschäftsprozessen sind die Modellierung von Zeiteinschränkungen sowie die Überprüfung von verschiedenen Eigenschaften wie z.B. Widerspruchsfreiheit der Zeiteinschränkungen, Erfüllbarkeit von Zeiteinschränkungen oder Steuerbarkeit eines Prozesses. Diese Bereiche sind bereits gut erforscht, allerdings nur für azyklische Geschäftsprozesse. Zeitmanagement in Geschäftsprozessen mit Schleifen stellt in der Forschung eine Lücke dar, mit der wir uns in der vorliegenden Dissertation befassen.

In dieser Arbeit präsentieren wir die *erweiterten Zeiteinschränkungen*. Diese machen es möglich, Zeiteinschränkungen zwischen zwei Aktivitäten in einem zyklischen Geschäftsprozess zu definieren. Weiters stellen wir die Funktion *atomize* vor, die beim Entfalten eines zyklischen Prozesses in einzelne Prozesspfade die erweiterten Zeiteinschränkungen in *atomare Zeiteinschränkungen* transformiert.

Um die Eigenschaften wie die Steuerbarkeit eines zyklischen Geschäftsprozesses mit erweiterten Zeiteinschränkungen untersuchen zu können, muss man zunächst prüfen, ob der gegebene Prozess terminieren muss, um alle Zeiteinschränkungen erfüllen zu können. Dazu präsentieren wir die *Terminierungsprüfung*, mit der wir eine solche Prüfung vornehmen können. Prozesse, die die Terminierungsprüfung positiv bestanden haben, können in azyklische Prozesse entfaltet werden und mit bereits vorhandenen Verfahren auf z.B. Steuerbarkeit überprüft werden.

Die genannten Forschungsbeiträge dieser Dissertation werden mit einem Prototyp abgerundet. Dieser kann die erweiterten Zeiteinschränkungen in einem gegebenen zyklischen Prozess interpretieren, diese in atomare Zeiteinschränkungen transformieren und die Terminierungsprüfung durchführen.

Abstract

In these days, the whole world needs to be clocked in a perfect rhythm more than ever. Time management became a crucial discipline in professional as well as in personal life. In many business processes, good time management is the key factor of (financial) success. In some cases, e.g. medical treatments, time management gains even more importance and can make the difference between life and death.

An important field of time management in business processes is modeling of time constraints as well as checking if they fulfill various properties like consistency, satisfiability, or controllability. There has been plenty of research in this field, however, the loops in business processes are constantly left out of focus. The intention of this thesis is to close this gap and to focus solely on loops.

In this thesis, we introduce *Extended Time Constraints* (ETCs) that enable us to define time constraints in a cyclic process. ETCs represent temporal bounds between two activities in a cyclic process. These activities can appear multiple times during the process execution due to the loops they are placed in. In this thesis, we define the syntax and the semantics of ETCs. Furthermore, we introduce a function that *atomizes* (instantiates) ETCs into *Atomic Time Constraints* (ATCs) if we decompose a cyclic process into *Instance Types* (process paths).

In order to check the consistency, satisfiability, or controllability of a cyclic process with ETCs, we first check if a cyclic process must terminate in order to satisfy all time constraints. We do this with the *Termination Check* that we introduce in this thesis. The Termination Check helps us to sort out the processes with loops that cannot be temporally bounded. Cyclic processes that pass the Termination Check can be unfolded into acyclic processes and checked for consistency, satisfiability, or controllability with existing algorithms.

We complete our contribution with a prototype that is able to interpret ETCs, transform them into ATCs, and to perform the Termination Check on any cyclic process with Extended Time Constraints.

Acknowledgments

I am very honored that I had the chance to work for and with Prof. Johann Eder in the Department of Informatics Systems. Prof. Eder constantly accompanied my work and enriched it with his brilliant ideas. I am very thankful for all the time he took for me, for his valuable constructive criticism, and for all discussions that we have had. They raised the quality of this work enormously.

I am very grateful for meeting Prof. Carlo Combi and I am very honored for having him as a reviewer. The meetings and discussions with him helped me to reflect on my research from other perspectives.

Furthermore, I want to thank Dr. Konstantin Schekotihin for all of the discussions that we had and for the hints regarding ASP tools.

It is a great blessing that I had the opportunity to study abroad at the University of Klagenfurt. I would like to thank my beloved parents and my sister Ema for making this possible and for all their patience and support. Without them, everything would have been much harder.

Being a PhD-student was a very unique experience that made me discover my own boundaries but was also a lot of fun. I am very lucky that I was accompanied in my journey by many helpful and understanding colleagues that became my friends. I would especially like to thank Dr. Julius Köpke, Dr. Patrick Rodler, Dr. Horst Pichler, and Marco Franceschetti for the interesting discussions, exchange of ideas, and their great company. A big thank you also goes to my former office mate, Dr. Stefanie Beyer, for being so uncomplicated, understanding, cheering, and funny. We could fill a book with our office stories.

I started a new job during the final PhD-phase in 2017 that made this journey even harder. I would like to thank all of the nice colleagues at Kelag for their understanding and support - especially my bosses Walter Penker and Michael Wieltschnig.

There were many more people involved into my PhD-journey – friends, colleagues at work, and PhD-colleagues that I met at the conferences. I am very thankful to each and every one of them. I am also grateful for my lovely cats, who kept me company and entertained me during long writing sessions.

I would like to acknowledge my gratitude to Andrew Moore, Ema Moore, and Dr. Christopher Schwarzlmüller for reading this thesis so carefully and correcting my English. Thank you for your patience and fast responses.

Last but not least, I would like to thank my dear Christopher, who always believed in me and was proud of me on every step that I made. He encouraged me to do more than I thought I could. He was never 'jealous' of my work, even during our very rare times together. Thank you for your strong belief and your support.

Contents

List of Figures

List of Tables

"The clock, not the steam-engine, is the key-machine of the modern industrial age."

— Lewis Mumford

Chapter 1

Introduction

The modern world was revolutionized by the time measurement that came with the concept of universal time-scale after the first industrial revolution in the mid-19th century. Philosopher of technology Lewis Mumford once famously wrote it: "The clock, not the steam-engine, is the key-machine of the modern industrial age."

In the fourth industrial revolution that is unfolding before our very eyes, the internet (of things) and digitalization are the driving forces of the revolution. However, time remains a key factor. The entire world needs to be synchronized in a perfect rhythm now more than ever. Time management became a crucial aspect of both professional and personal matters. In many business processes, good time management is the key factor of (financial) success. In some cases, e.g. medical treatments, time management is even more important and can make the difference between life and death.

In the 1990's, soon after the rise of business process management in the 1980's, time management in business processes became a matter of research. Many fields of time management evolved in the last few decades, e.g. modeling of temporal information [EP00, BWJ02b, CP03, CKGJ13, CGJ+07, MO99], computation of time plans [EPR99, EEP06, LNCY11], scheduling [EPGN03, AF08, BWJ00, Bus98, CP06, LKR13], time patterns [LWR09, LWR14], exception handling [PWE09, vdARD05], temporal prediction [vdASS11], and others. A short overview of the development of the field time management in business processes is given in [EPR13].

An important field of time management in business processes is modeling of time constraints, as well as checking if they fulfill various properties like consistency,

satisfiability, or controllability. There has been plenty of research in this field, however, the loops in business processes are constantly left out of focus.

In the literature, time constraints suffer a lack of a specification that is adapted to business processes with loops and the nature of some loops (uncertain repetition of activities).

In the process of property checking or in the computation of time plans for a business process, the loops are mostly not handled at all (e.g. [LSPG06, EGP00]), handled as a complex activity (e.g. [Mar00, BWJ00]), or rolled out into a sequence (e.g. [SKK05]).

Combi *et al.* propose a more advanced approach of loop handling - a translation of loops and related temporal constraints into conditional blocks (XORs) [CGPP12, CGMP12, LPCR13, CGMP14]. Rewriting loops into conditional blocks offers an adequate handling of loops and related temporal constraints, however, the authors make several restrictions to be able to handle the complexity that arises with loops. They limit the maximum number of loop iterations already in the process model as well as the variety of temporal constraints to only such constraints that consider cyclic elements between two directly succeeding iterations [LPCR13], or the same iteration [CGPP12].

In [Pic06], Pichler introduced an advanced loop handling approach that allows the handling of unbounded loops. The author assigns branching probabilities to workflow graphs and uses this information to transform a cyclic workflow graph into an acyclic graph called a *Probabilistic Unfolded Workflow Graph*. To prevent an infinite growth of the graph, graph expansion stops when the probability of missing cases is below a certain threshold.

Pichler *et al.* introduce another interesting approach that considers loops in [PEC17]. They introduce temporal splits and temporal loops that use temporal conditions to decide which branch in a split will be taken, or if a loop can be entered. An example of such a temporal condition attached to a temporal loop is $elapsed < 100$. Such a loop can only be entered if less than 100 time units passed since the process instance was started.

In the literature, loops are mostly mentioned as a part of the business process that is not the primary focus of the work. The intention of this thesis is to close this gap and to focus solely on loops.

1.1 Problem Definition

Time management begins with the definition of time constraints. Time constraints that we know from the literature, e.g. [EPR99, BWJ02b, CGJ+07] cannot be used in processes that contain loops. Let us consider the example from figure 1.1 to understand the problem. The process in figure 1.1 contains 2 Upper Bound Constrains (UBCs): $ubc(A, D, 30)$ and $ubc(B, D, 14)$.

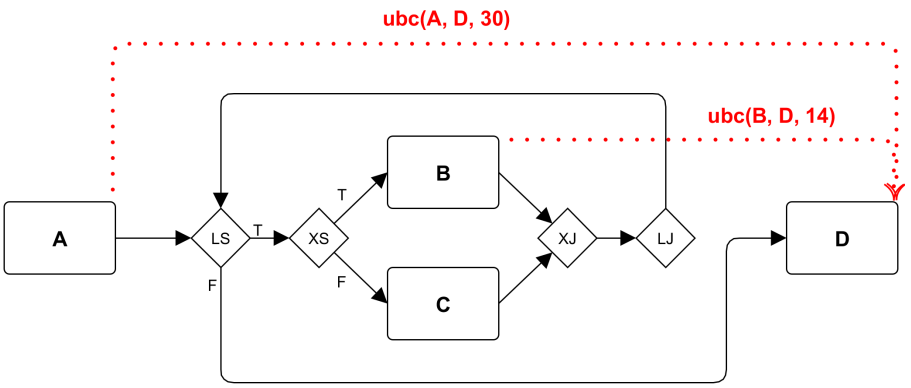


Figure 1.1: A process with a loop and conventional time constraints

The first Upper Bound Constraint $ubc(A, D, 30)$ limits the maximal duration between the ending point of the source activity A and the ending point of the destination activity D to 30 time units (e.g. days). In other words, activity D must end within 30 days after activity A has ended. The second Upper Bound Constraint $ubc(B, D, 14)$ limits the maximal duration between the ending point of the source activity B and the ending point of the destination activity D to 14 days. Since activity B appears in a loop and can therefore occur multiple times, the UBC $ubc(B, D, 14)$ should have specified not only the source activity, but also the occurrence(s) of that particular activity - which it does not. It is not clear if the UBC applies to the first occurrence of the activity B, to each occurrence, or maybe only to the last occurrence. This problem leads us to the first research question that we deal with in our research:

RQ1: How can we define time constraints in a cyclic process?

The next step in time management is to check if a process and the defined time constraints satisfy particular properties, e.g. temporal consistency, satisfiability of time constraints, or process controllability. Out of those properties, process controllability[1], introduced by Combi *et al.* in [CP09], is the strongest one. However, existing algorithms for controllability checking [CP09, CHP13] are not able to check cyclic processes. This brings us to the next challenge stated in the research question RQ2:

RQ2: How can we check the controllability of a cyclic process?

In order to check the controllability of a cyclic process with existing algorithms, the process must first be unfolded to an acyclic process. The problem that we face in the unfolding process, are unbounded loops that can be unfolded endlessly. The idea is to bind the unfolding process of unbounded loops with the given time constraints that temporally limit the duration of a loop block. However, not every unbounded loop involves such time constraints that are able to bind the unfolding process. Due to this issue, we define a sub-research question RQ2a that we deal with in this thesis:

RQ2a: How can we check if a cyclic process must terminate in order to satisfy all time constraints?

The answer to this research question will help us to sort out the processes with loops that cannot be temporally bounded in a pre-step before the controllability check. Therefore, the answer to RQ2a brings us a step closer to the controllability check of cyclic processes.

[1] In general, in a controllable temporal process it is possible to satisfy all temporal constraints for any possible duration of tasks that cannot be influenced by the agent (contingent links).[CP09][CP10][CGMP12]

1.2 Outline of the Thesis

In this thesis, we address the research questions RQ1 and RQ2a. The thesis is structured as follows:

Chapter 2 delivers an overview of business process management in general and narrows the focus to business process modeling and business process analysis that accompany us through the thesis.

Chapter 3 digs deeper into one particular aspect of business process management - time. First, time patterns give us an overview of which time information is modeled in business processes and how it is modeled. Afterwards, we describe the main modeling and verification approaches of temporal aspects that can be used as a starting point for the pattern *TP9: Cyclic Elements*, which describes time lags between activities in a loop.

Our contribution to the research field of business process time management is described in chapters 4, 5, and 6, which are the essence of this thesis. They add new knowledge to the field modeling and analysis of time perspective of business processes with loops.

In chapter 4, we first introduce some basic models (*Process Graph*, *Loop Instance Type*, and *Instance type*) that we use throughout the thesis. The core of this chapter is the introduction of *Extended Time Constraints* (ETCs) for modeling time information in processes with loops. We introduce the syntax and the semantic of ETCs and define the atomization function that translates ETCs in a cyclic process into *Atomic Time Constraints* (ATCs) in an Instance Type (workflow path). This chapter delivers an answer to the research question RQ1.

In chapter 5, we introduce the *Termination Check*, which tests whether a given cyclic process must terminate in order to satisfy all specified Extended Time Constraints or not. Termination Check consists of three steps: 1) process transformation, 2) time constraints inference, and 3) termination check. This chapter describes and formalizes each of these steps and delivers several examples with diverse process structures and ETCs. The Termination Check for cyclic processes is an answer to the research question RQ2a.

Chapter 6 delivers a proof of concept for the concepts and formalisms introduced in chapters 4 and 5. We present a prototype that can take a cyclic process with a set of Extended Time Constraints as an input and checks if the input process must terminate in order to satisfy all ETCs or not. The prototype with an input example is attached in Appendices A, B, C, D, and E.

We draw our conclusions in chapter 7 and give some directions for future work.

Chapter 2

Business Process Management

Throughout history, mankind has become more specialized and organized. In the last few centuries, the 1st, 2nd, and 3rd industrial revolution substantially sped up this evolution. In the 3rd industrial revolution, computers and other electronic devices became accessible and led to digitalization and automation of work as well as completely new business models. The division of work across organizational units and different people, together with evolving information systems in the 80's led to the early beginnings of business process management. Workflow management systems (WfMS) and enterprise resource planning (ERP) systems [CBS04] were introduced to support business processes and functions. Both systems played an important role for (partial) automation of business processes. Today, business process management covers much more than only automation of business processes and has become an essential part of many organizations.[vdALRS16]

Business process management (BPM), as defined by Weske, "includes concepts, methods, and techniques to support the design, administration, configuration, enactment, and analysis of business processes"[Wes07].

A business process consists of activities, events, and decision points and involves actors, data, and other objects to reach a particular process outcome. An activity is a unit of work that has a duration and is assigned to and completed by an actor. An actor can be a person as well as an organization or a software. Events, in contrast to activities, have no durations and no actors. Decision points are used to navigate through the process.[DRMR13]

According to Weske, business process management typically consists of four phases that complement and build on each other. These phases, as shown in figure 2.1, are collectively called the business process lifecycle. A brief description of each business process lifecycle phase, as depicted in [Wes07], is given below.

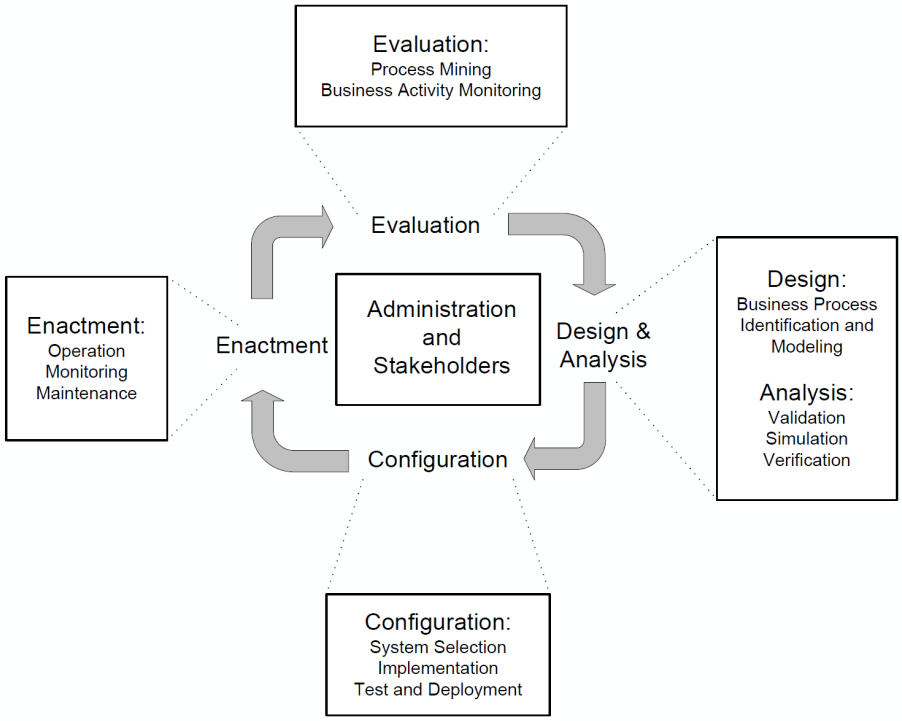


Figure 2.1: Business process lifecycle [Wes07]

Design and Analysis

In this initial stage (on the right in figure 2.1), business processes first get identified. Business domain experts, process designers, and other business process stakeholders exchange the information about possible process instances. This information basis is used to model the processes, which can be done in a workshop with representing stakeholders. There are different graphic notations that can be used for modeling business processes in this phase, e.g. Business Process Model and Notation (BPMN). Once the processes get identified and modeled, their behavior gets analyzed and verified. If the behavior satisfies the expectations, the next phase in the business process lifecycle can be approached.

Configuration

In the configuration phase, the identified processes get implemented. The implementation can be done without software support in the form of organizational rules and policies that are communicated among the targeted organizational units, or it can be done with software support. In the latter case, processes get configured in the chosen business process management system (BPMS), integrated in the organizational software landscape, and tested.

Enactment

After the processes have been configured in a BPMS, processes can be started in this phase, as well as have their running behavior monitored. The starting event of the process could be, for example, an incident or a customer order. The completion of the process activities, compliance to defined constraints, and other status information is constantly monitored in this phase and thus delivers important decision information for possible actions, e.g. inform the customer that the order will have a delay.

Evaluation

The logged process execution information from the previous phase can lead to interesting discoveries in the evaluation phase. Process models, unknown process behaviors, and many more can be deducted from the logs. In recent years this phase gained a lot of attention and became popular under the name process mining. A good overview of this discipline can be found in [vdA16].

Each of the described phases of the business process lifecycle has an impact on the other phases. The process evaluation phase, for example, impacts the stakeholders to adapt the processes, remodel, and then reimplement them. It is very important that the stakeholders take care with adequate representation and storage of all process information and efficient possibilities to search for it, as well as other related information, e.g. information technology landscape.

The research work described in this thesis is settled in the first phase of the business process lifecycle - design and analysis. More details on this phase are given in the following two sections.

2.1 Business Process Modeling

Over time, many formal and conceptual modeling languages for business processes were introduced, e.g. Petri nets, UML activity diagrams, Event-driven Process Chains (EPCs), Business Process Model and Notation (BPMN), etc. A comparison of different modeling languages for business processes can be found in [ZMI10]. The most commonly used modeling language currently is the Object Management Group (OMG) standard BPMN. Conceptual business process models in this thesis are based on BPMN, therefore BPMN will be described in more detail.

BPMN in version 2.0.2 from January 2014 [OMG13] consists of many modeling elements, from which only the core elements are being used in most cases [ZMR13]. A process in BPMN consists of four categories of elements: flow objects, connecting objects, swimlanes, and artifacts [KIG+15, WM08].

Flow objects

Flow objects affect the process and thus its outcome. There are three groups of flow objects: activities, events, and gateways. Activities (shown in figure 2.2) represent the actual work that is done in a process. The simplest activity is a task that can be understood as a single unit of work. A sub-process and a transaction are activities that consists of a set of activities. Transaction additionally defines rollback and compensation in case something goes wrong.

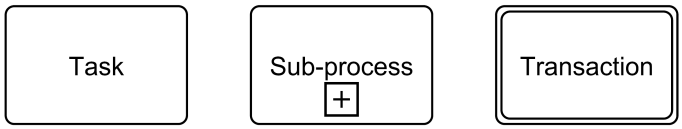


Figure 2.2: Activities

Events signalize that something happened in a process. In contrast to activities, they have no duration. There are three groups of events: start, intermediate, and end events, as represented in figure 2.3. A start event starts the process, an end event ends it, and an intermediate event can occur anywhere in the process between start and end events. Additional markers in events can give them more meaning, e.g. a clock marker that denotes a timer event.

Figure 2.3: Events

Gateways control the paths taken in a process and can split or merge the paths. The most important gateways are shown in figure 2.4. An exclusive gateway allows a process instance to take only one path after the gateway. This gateway represents a decision in a process. The gateway is mostly annotated with a condition and its outcome (true or false) navigates the process instance to the right path. Inclusive gateways differ from exclusive in the fact that more than one path can be chosen. A parallel gateway splits one path into many paths, where all paths are taken simultaneously. An event-based gateway is similar to the exclusive gateway with the difference being that there is no decision at the gateway itself. The decision of which path has to be taken depends on the first event that follows the gateway (e.g. message arrived). Gateways that allow a process instance to take more than one path must be merged carefully in order to avoid unwanted token multiplication in a process instance.

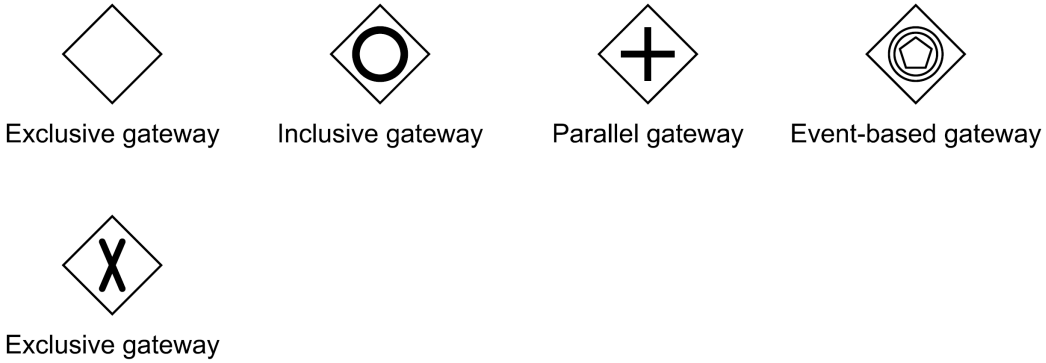


Figure 2.4: Gateways

Artifacts

Artifacts are used to model additional information besides the actual work and its flow through a process. There are three main artifacts: data objects, text annotations, and groups as shown in figure 2.5. Data objects represent the documents that are relevant in a process, e.g. a form that has to be filled out and approved in a vacation approval process. Text annotations are used to provide more detailed information that can not be packed in an activity, gateway, etc. They can be understood as a comment. Groups are used to organize a process into sections of elements that belong together.

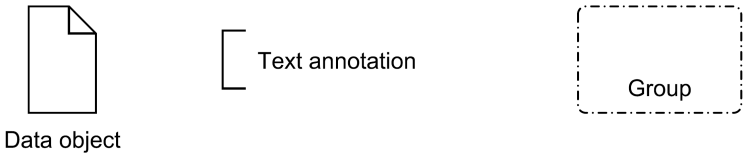


Figure 2.5: Artifacts

Swimlanes

Swimlanes (pools and lanes) are used to organize a process model along collaborative aspects. A pool represents a process participant that collaborates with another participant (pool) in a process. Each participant has its own process that can interact with other participants and their processes. A pool can be divided into lanes as shown in figure 2.6. Lanes can represent organizational units of an organization (e.g. sales, accounting, etc.), roles (e.g. manager), technology, or something else that satisfies the purpose of the model.

Figure 2.6: Swimlanes

Connecting objects

BPMN elements are connected by each other with connecting objects that are shown in figure 2.7. The sequence flow that connects flow objects and determines their order in a process, is represented with an arrow. Message flow defines the interaction and the flow of information between different participants (pools) in a collaboration. It is represented with a dashed arrow. Association is represented with a dotted line or arrow and is used to connect artifacts to the rest of the the process, e.g. to represent the data flow.

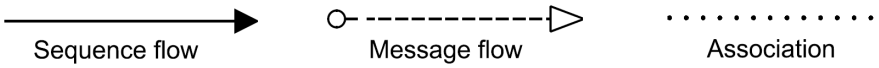


Figure 2.7: Connectors

A very simple example of a BPMN process diagram is shown in figure 2.8. It represents a procurement process from the buyer's perspective. The order has to be approved first and if the outcome is positive, both the order and the shipment are handled in parallel. Finally the order gets reviewed.

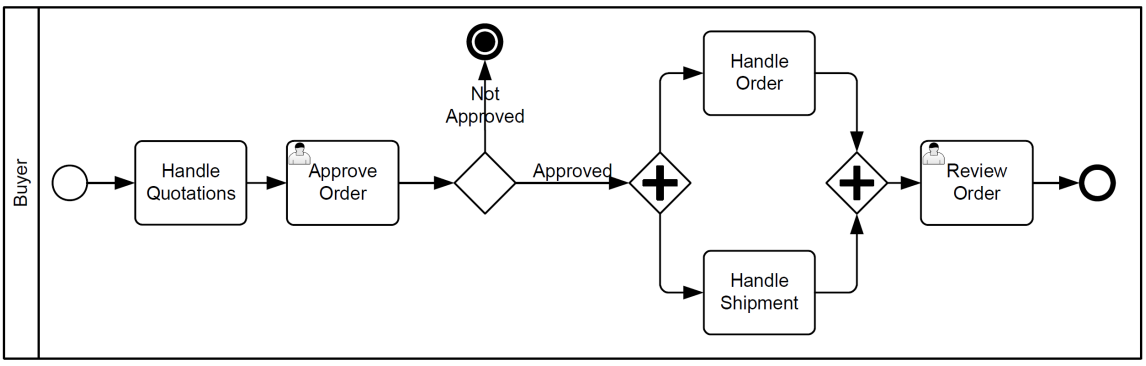


Figure 2.8: Example of a BPMN process model [OMG13]

A business process model is the input for the business process analysis. Business process analysis is described in the following section.

2.2 Business Process Analysis

Business process models, as well as business processes, can be improved with help of business process analysis. There are two classes of analysis: model-based analysis and data-based analysis. Model-based analysis takes place in the first phase of the business process lifecycle - design and analysis - and is briefly described in this chapter. Data-based analysis is based on event logs that are generated by the system during the process execution. Data-based analysis represents the last phase of the business process lifecycle - evaluation - and forms the input for the following cycle iteration. Examples of data-based analyses are business activity monitoring (BAM) and process mining.[vdA13]

Model-based analysis can be categorized into three main approaches: validation, verification, and performance analysis.

Validation

Validation methods are used to validate if a business process model represents what it should and if all possible process instances are covered by the model. A common method to validate a model is a workshop, where different stakeholders discuss a given model. Simulation of process execution, including all relevant information like execution probabilities of alternative paths, can be a useful method to support validation with additional insights.[Wes07]

Verification

Verification techniques are used to check different properties of a business process model. These properties can relate to the process structure, data, temporal aspects, or some other aspects that a model represents. Verification of structural properties, such as liveness and boundedness [Mur89], is typically performed on Petri nets due to their formal foundation. Other process representations can be translated into another, e.g. BPMN to Petri nets [DDO08, DDDGB08, Tsc06, RM06]. Based on Petri nets, van der Aalst introduced workflow nets [vdA96, vdA98]. Workflow nets can be used to verify the soundness property of a process model. A well known tool for verification of workflow nets is Woflan[VvdA00, VBvdA01].

Verification of temporal properties will be addressed in detail in the next chapter.

Performance analysis

Performance analysis aims to improve the effectiveness and efficiency of a process. The performance of a process is usually measured with time-, cost-, or quality-related *Key Performance Indicators* (KPIs). Examples of time-related KPIs are lead time (time from the process start to process end), service time (time it takes to complete a case), waiting time (time it takes to get a free resource), or synchronization time (e.g. waiting for activities from other branches to complete).

In many cases, time can be translated into costs directly. Other process aspects, like resource utilization, are also a basis for cost-related KPIs. There are also different cost models, such as Activity Based Costing (ABC)[Kap87], that are based on processes.

Quality is a little more different to measure, since it is often subjective - e.g. in a provided consulting service. Customer ratings, questionnaires, or number of complaints can reveal a valuable information about the quality. In the first phase of the business process lifecycle, where processes haven't been executed yet, performance analysis can be done in a simulation environment.[vdA16]

This chapter provided an overview of business process management in general. Business process modeling and analysis were described briefly. The following chapter delves deeper into one particular aspect of business process management - time.

Chapter 3

Business Process Time Management

A comprehensive process model includes the control-flow perspective as well as other perspectives like data, resource, function, and time perspective. The time perspective gives an insight into essential temporal information about a process like activity durations, deadlines, time lags between activities, etc.[vdA13]

The investigation of time perspective of business processes started back in the 1990's. Many fields of time management evolved in the last decades, e.g. modeling of temporal information [EP00, BWJ02b, CP03, CKGJ13, CGJ+07, MO99], computation of time plans [EPR99, EEP06, LNCY11], scheduling [EPGN03, AF08, BWJ00, Bus98, CP06, LKR13], time patterns [LWR09, LWR14], exception handling [PWE09, vdARD05], temporal prediction [vdASS11], and others. A short overview of the development of the field time management in business processes is given in [EPR13].

This thesis will focus on design and analysis phase of the business process life cycle, therefore modeling and analysis of the time perspective will be further discussed in this chapter. An overview of widely observed time patterns in business processes is given in section 3.1. Subsets of these patterns can be found in different formal representations of temporal aspects of business processes. We will briefly describe a selection of representations of temporal aspects and their verification in section 3.2.

3.1 Business Process Time Patterns

In analogy to design patterns[GHJV93], van der Aalst *et al.* identified recurring constructs in business processes and developed a pattern framework[1] regarding the control-flow, resource, data and exceptions handling perspective [vdATHKB03, RTHEvdA04, RTHEvdA05, RvdAtH06]. Lanz *et al.* extended the framework with the time perspective. Time patterns that they identified were repeatedly observed in different data sets from the healthcare, automotive, and other domains [LWR09, LWR10, LRW13, LWR14, LRW16]. They categorized the resulting 10 patterns into four pattern classes, which are listed in table 3.1.

Category I: Durations and Time Lags
TP1: Time Lags between two Activities
TP2: Durations
TP3: Time Lags between Arbitrary Events
Category II: Restricting Execution Times
TP4: Fixed Date Elements
TP5: Schedule Restricted Elements
TP6: Time-based Restrictions
TP7: Validity Period
Category III: Variability
TP8: Time-dependent Variability
Category IV: Recurrent Process Elements
TP9: Cyclic Elements
TP10: Periodicity

Table 3.1: Process time pattern catalogue[LWR14]

These patterns have not only been observed in selected data sets, but were described in the literature as well. Lanz *et al.* provide a systematic literature review regarding time patterns in [LWR14]. Table 3.2 shows their consolidated results.

The identified patterns are briefly described in following subsections.

[1] The collection of business process patterns as well as related publications is available at `www.workflowpatterns.com`

Research Group	TP1	TP2	TP3	TP4	TP5	TP6	TP7	TP8	TP9	TP10
Bettini *et al.* (e.g., [BWJ02a, BWJ02b])	X	X	X	X						
Combi *et al.* (e.g., [CGJ+07, CP02])	X	X	X	X	X	X	X		X	X
Eder *et al.* (e.g., [EPR99, EP00])	X	X		X	X					
Li *et al.* (e.g., [LY05][LYC04])	X	X		X						
Mans *et al.* (e.g., [MvdAR+09, MRvdA+10])	X	X		X	X			X		
Marjanovic *et al.* (e.g., [Mar00, MO99])	X	X		X						
Müller *et al.* (e.g., [MR00, MGR04])		X		X						
Sadiq *et al.* (e.g., [SMO00, SO98])	X	X		X					X	
Zhuge *et al.* (e.g., [ZyCkP01, ZPC00])	X	X		X					X	

Table 3.2: Consolidated results of the systematic literature review [LWR14]

3.1.1 Durations and Time Lags

The first time patterns category - durations and time lags - includes definitions of allowed time units that an activity execution may or must take and time units that may or must pass between different activities.

Pattern TP1 encompasses different time lags between two activities. Time lags may differ in their temporal restriction (minimal or maximal value or restricted time interval) and starting and ending points of the lags. The starting point of a time lag can be the start or the end of the first activity, and the ending point the start or the end of the second activity.

An example of a minimal time lag between two activities is a blood test where a patient must not eat anything for at least 8 hours before the blood is taken .

Pattern TP1 is known as the Lower and Upper Bound Constraint in [EPR99], Inter-Task Constraint in [BWJ02b], and as the Relative Constraint in [CGJ+07].

Durations in processes are very similar to time lags and are consolidated in pattern TP2. They can be applied to all kinds of process elements (activities, process models, process instances) and may be restricted to a minimum, maximum, or allowed interval of time units.

An example of constrained activity duration is an exam where students must stop writing after a predefined amount of time, e.g. after 90 minutes.

Time lags between arbitrary events are covered by time pattern TP3. This pattern

is very similar to TP1, except that events don't have a duration, therefore the options start-start, start-end, end-start, and end-end are irrelevant.

Patterns TP1, TP2, and TP3 from Durations and Time Lags pattern class are illustrated in figure 3.1.

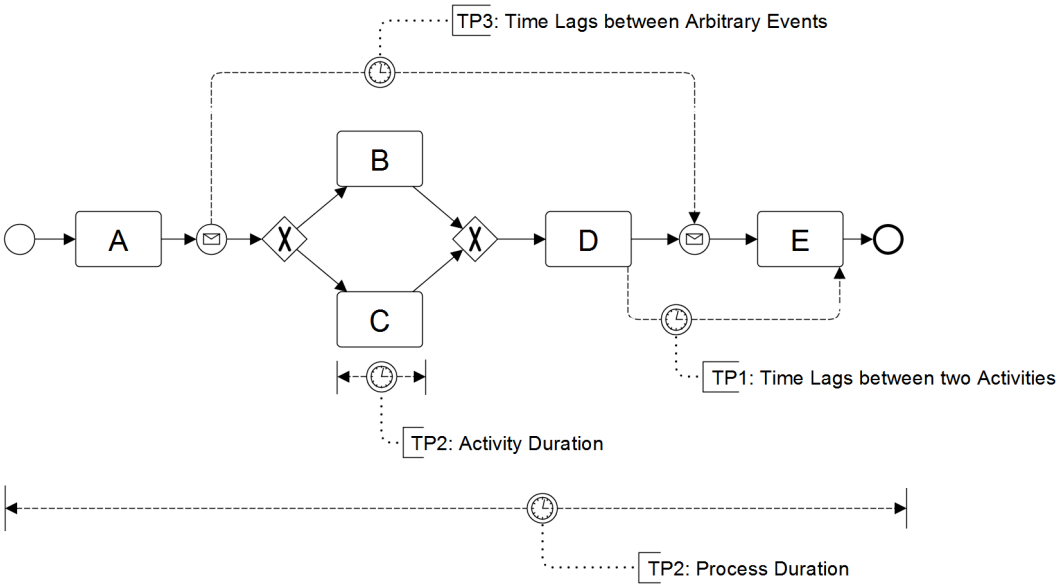


Figure 3.1: Durations and Time Lags

3.1.2 Restricting Execution Times

Patterns TP4 to TP7 in the Restricting Execution Times category regulate the earliest or latest execution or ending points of process elements.

Fixed Date Element in TP4 is used to specify the earliest start, latest start, or a deadline of a process instance or an activity. For example, course assignments must be submitted until Monday each week.

TP5 covers process elements that are restricted to a schedule. Examples of Schedule Restricted Elements are activities or process instances that can only be executed within defined opening hours or scheduled time points, e.g. a train will depart at 7:30.

TP6 represents time-based restrictions. These restrictions define how often within a given time interval an activity or a process instance is allowed to be executed. For

example, in a social media platform, one can change the name only once every 60 days.

TP7 summarizes temporal restrictions of activities or process instances to a particular validity period. Only within the defined period the processes or activities may be executed, otherwise their execution is invalid. Examples for this pattern are starting and expiration dates of laws that are implemented in corresponding processes. The processes/activities may only be executed while the law is valid.

3.1.3 Variability

This pattern class contains only pattern TP8 that represents time-depending process or activity execution. Time dependent variability of control flow means that depending on particular temporal conditions different paths in the process will be taken. For example, one can drive or ride a bike from home to work but if it is not possible to be on time by bike, the only way to come to work is by car. Eder *et al.* investigated this pattern and provided the underlying formalism in [PEC17].

3.1.4 Recurrent Process Elements

The last pattern class deals with recurrent elements. Pattern TP9 describes time lags between activities in a loop, while TP10 deals with periodic occurrences of an activity at fixed times, e.g. every Monday at 10:00. The number of periodic occurrences in TP10 can be unlimited or limited by a predefined number, an end date or by an exit condition. Previous patterns, except TP7, can be combined to mimic TP10.

TP9 is a projection of TP1 to cyclic processes. Time lag can be restricted between different activities or between two occurrences of one activity in arbitrary iterations. For example, a patient has to take a pill every 24 hours.

Recurrent Process Elements patterns are a field of research that still needs to be thoroughly investigated. The research in scope of this thesis delivers an investigation and underlying formalization of the pattern TP9. The starting point of this research are Upper Bound Constraints (pattern TP1) that are being extended to cover cyclic processes in chapter 4. Since pattern TP1 is the basis of the research in this thesis, the underlying representation alternatives will be described in the following section.

3.2 Modeling and Verification of Temporal Aspects

Previous section gave us an overview of time patterns in business processes and this section complements it with selected approaches for specifying and verifying process time perspective. Since the research in this thesis delivers one possible modeling and verification approach for the pattern TP9 (Cyclic Elements), only the main alternative approaches behind TP1 (Time Lags between two Activities) that can be used as a starting point for TP9 are described. An overview of a broader set of modeling and verification approaches can be found in [CKGJ13, CKGJ15].

3.2.1 Timed Workflow Graph

Eder *et al.* adapted project network techniques Program Evaluation and Review Technique (PERT) and Critical Path Method (CPM) as a formalism to represent the time perspective in business processes in [EPL97]. Based on this adaption, Eder *et al.* later introduced Timed Workflow Graph (TWfG) in [EPR99]. TWfG is used as the basis for formal representation for time constraint management in their research.

A **Timed Workflow Graph** is a workflow graph that extends activities with temporal information. The activities in the simplest version of a TWfG contain activity names, activity duration, earliest finishing time, and latest finishing time of an activity. An example of such a TWfG is shown in figure 3.2. This graph does not contain conditional splits, only parallel splits. If we add conditional splits to the underlying process graph, its TWfG gets more complex and contains additional temporal information, like best and worst case scenarios of earliest and latest finishing times [EPPR99, EP00, EGP00]. In figure 3.2, activity names are defined in upper left corner of activities and activity durations (in time units) in the upper right corner. The earliest possible finishing times are placed in the lower left corner of activities and the latest finishing times in the lower right corner. The earliest finishing times are calculated forward by adding duration times, beginning with the starting activity. The latest finishing times are calculated backward by subtracting duration times, beginning with the ending activity.

Together with Timed Workflow Graph, Eder *et al.* introduced a set of time constraints in [EPR99]: Fixed-Date Constraint, Lower Bound Constraint and Upper Bound Constraint.

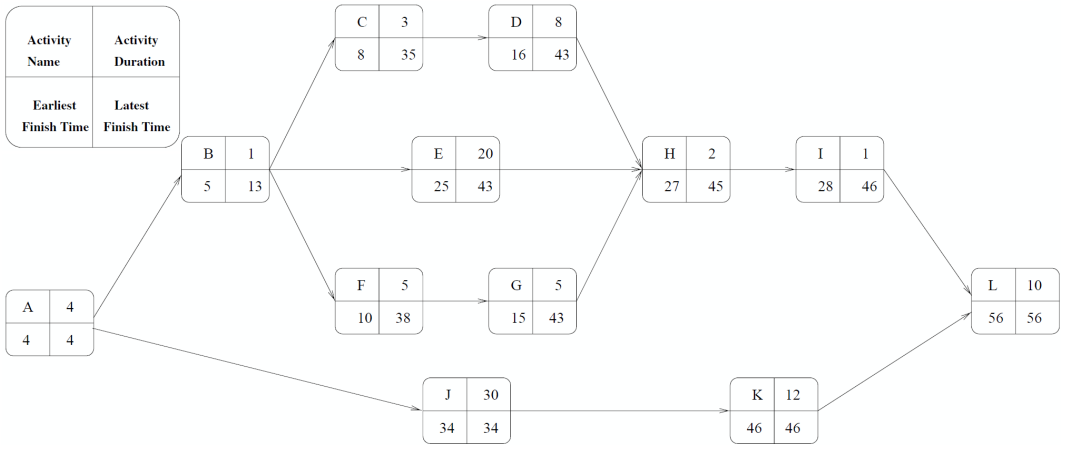


Figure 3.2: Example of a Timed Workflow Graph from [EPR99]

A **Fixed-Date Constraint** represents time pattern TP4 and defines a date on which an activity may get executed. All other dates are invalid.

A **Lower Bound Constraint** $lbc(A, B, \delta)$ represents time pattern TP1 and limits the minimal duration between the ending point of the source activity A and the ending point of destination activity B to δ time units.

An **Upper Bound Constraint** $ubc(A, B, \delta)$ also represents time pattern TP1 and limits the maximal duration between the ending point of the source activity A and the ending point of destination activity B to δ time units.

A set of time constraints is said to be **satisfiable** if there exists a workflow execution that satisfies all time constraints. A Timed Workflow Graph (TWfG) with incorporated time constraints represents all such executions that satisfy all time constraints. Those are the executions where all activities are completed at their earliest finishing times, or latest finishing times, or within that interval. A TWfG satisfies a time constraint if all valid executions that are represented by the TWfG satisfy the time constraint. The satisfiability of a given set of time constraints in a TWfG is therefore verified by checking if the earliest and latest finishing times of each activity in the TWfG are valid regarding the given set of time constraints.

3.2.2 Workflow Constraint Graph

The Workflow Constraint Graph was introduced by Bettini *et al.* in [BWJ02b, BWJ00]. In contrast to a Timed Workflow Graph (TWfG), a Workflow Constraint Graph (WCG) is based on the formalism of temporal constraint networks[DMP91].

In a **Workflow Constraint Graph**, a process activity is not represented only by one node with additional temporal information, but with a pair of nodes that symbolize the starting and the ending instant of an activity. The duration (minimal and maximal) of an activity is represented by the edge that connects the activity starting instant with the corresponding activity ending instant. Generally speaking, in a WCG each edge between nodes X and Y labeled with an interval $[m, n]$ represents a time constraint that defines the minimal (m) and maximal (n) allowed temporal distance between X and Y.

In a special type of time constraint called **Temporal Constraint with Granularity (TCG)**, the interval with the minimal and maximal allowed temporal distance between two nodes is extended with the granularity, e.g. hours, business days (b-days), months, etc. This granularity gives the bounds in the interval additional temporal context.

Figure 3.4 demonstrates a Workflow Constraint Graph that represents a procurement process shown in figure 3.3 with corresponding time constraints. Node names in figure 3.4 consist of initials that are derived from activity names in the procurement process and end with the characters b or e. Character b stands for the beginning of an activity (starting instant) and e for the ending (ending instant). The edges in figure 3.4 represent a set of time constraints, e.g. the edge between Bb and Be with the label $[0, 1]b - day$ constrains the duration of activity *Billing* to minimal 0 and maximal 1 business days. The edge between OPe and LDe with the label $[1, 2]b - day$ corresponds to a Lower and an Upper Bound Constraint in a Timed Workflow Graph. It constrains the local delivery of ordered items to earliest 1 business day after the order has been processed and latest 2 business days after. An edge with the label $<$ is an edge that specifies the chronological order of the activities without any specific time constraints.

A property that determines if it is possible to satisfy all time constraints in a given Workflow Constraint Graph is called inconsistency-freeness. A WCG is said to be **inconsistency-free** if each possible execution thread (with only one path

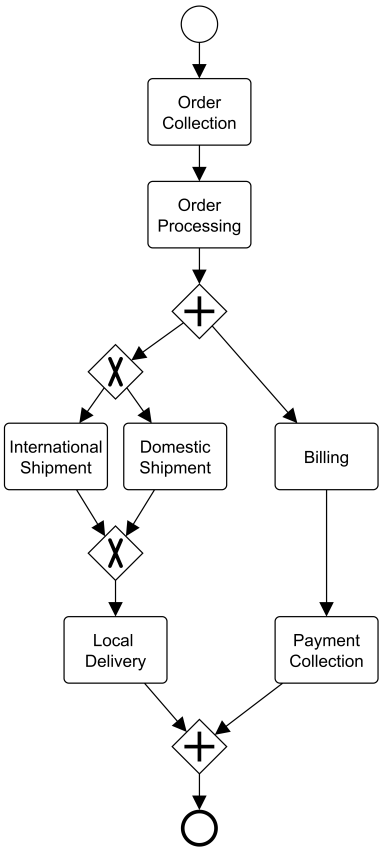


Figure 3.3: Procurement Process

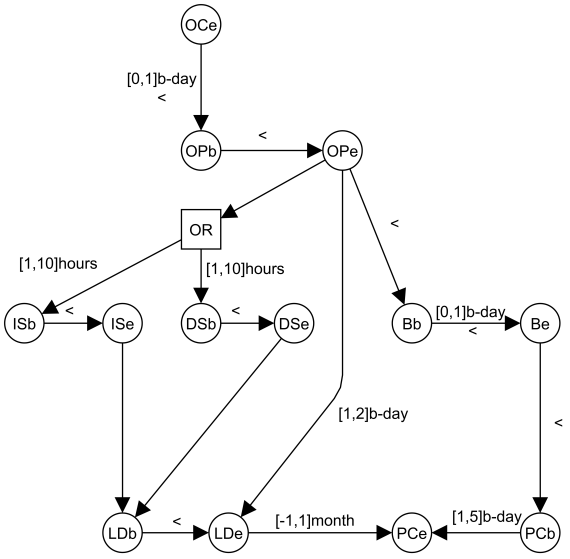


Figure 3.4: Procurement Process WCG (adapted from [BWJ02b])

following an OR-operator) is consistent. The checking of this property is done by 1) decomposing a given WCG to subgraphs called *constraint networks*, where each such network represents a possible execution thread, and 2) checking if each obtained constraint network is consistent. A constraint network is **consistent** if it has a solution in terms of a Simple Temporal Problem (STP)[DMP91] (in workflow terms: if it can be executed without violation of any constraint).

Temporal constraint networks have also been used as the underlying formalism for temporal workflow verification by Combi *et al.*[CGJ+07]. Their approach is described in the next section.

3.2.3 Temporal Workflow

Combi *et al.* use temporal workflows to model a process including relevant temporal information [CP03, CGJ+07]. Their conceptual model is based on the atemporal model introduced by Casati *et al.*[CCPP95]. Graphically, they extend BPMN with additional temporal information as shown in figure 3.5. Temporal information included in their conceptual model is classified in *events*, *durations and delays*, and *temporal constraints*.

Events indicate time instants that may occur during the process execution. **Durations and delays** represent a temporal distance between a starting and ending instant of an activity (duration) or an edge (delay). In contrast to Eder *et al.*, where edges do not have any delay and the durations of activities are fixed, Combi *et al.* allow durations and delays to move flexibly within given minimal and maximal bounds. Furthermore, the time granularity can be explicitly specified like in the model of Bettini *et al.* described in previous section.

Combi *et al.* divide temporal constraints into three classes: relative constraints, absolute constraints, and periodic constraints.[CGJ+07]

A **Relative Constraint** limits the time distance between two activities to a minimum or maximum range and corresponds to Lower and Upper Bound Constraint introduced by Eder *et al.* described earlier in this chapter. In contrast to Eder *et al.*, Combi *et al.* allow the Lower or Upper bound Constraint to be defined between starting or ending instant of the source activity and starting or ending instant of the destination activity. This is necessary since they also allow the duration of an activity to vary within the given minimal and maximal limits. Relative Constraints represent time pattern TP1.

An **Absolute Constraint** defines a time interval during which an activity is allowed (or not allowed) to be executed. The time interval is bounded by two timestamps.

A **Periodic Constraint** also defines a time interval during which an activity is allowed to be executed, however the time interval is periodic, e.g. an activity can be executed between Monday and Friday every week.

Figure 3.5 shows a temporal workflow graph of an *ST-segment Elevation Myocardial Infarction* diagnosis and treatment process with corresponding temporal information. On the bottom of each activity, its allowed duration in a given granularity is defined. First activity T1 (Admission to Emergency Department) can last between 2 and 4 minutes. The delay (noted on the edge) between the first activity T1 and the succeeding activity T2 (Initial patient evaluation) can last between 1 and 5 minutes. Between activity T2 and T3, there are not only minimal and maximal allowed delays that have to be obeyed, but also a relative temporal constraint $E_{T2}[1, 20]E_{T3}\, min$. This constraint requires the finishing of T3 to happen between 1 and 20 minutes after the finishing of T2. Relative Constraint $S_{T4}[-1, 2]E_{T5}\, min$ between parallel activities T4 and T5 includes a negative integer -1 and 2 as the allowed bounds. Those bounds require the patient to take Beta Blocker (T5) at the beginning of the one hour long Reperfusion Fibrinolytic therapy (T4). More precisely, T4 must end somewhere between 1 minutes before or 2 minutes after the start of the therapy T5.[CP09]

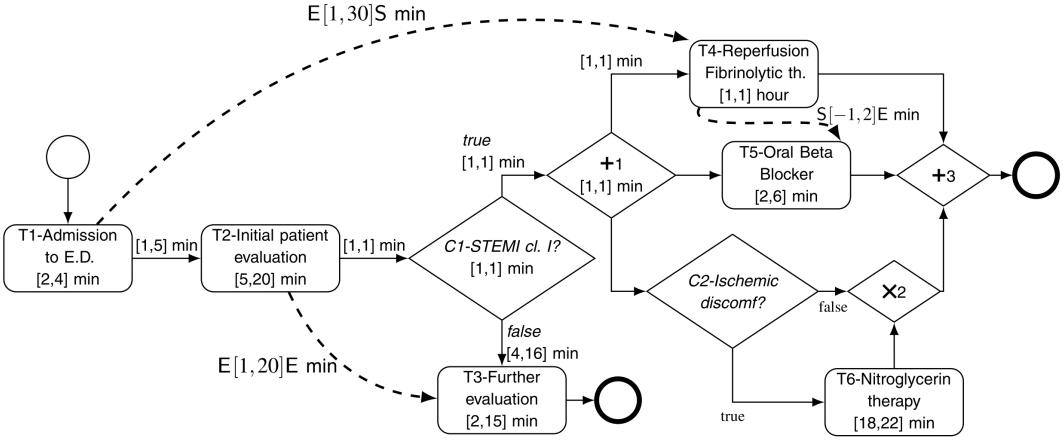


Figure 3.5: Temporal Workflow Graph [CP09])

Combi *et al.* use temporal consistency and controllability checking to verify temporal workflows [CGPP12, CP09].

Consistency of a temporal workflow schema is fulfilled if each workflow path (wf-path), derived from the workflow schema, is consistent. A wf-path is consistent if at least one assignment of start and end instants of activities exist, for which all temporal constraints in this wf-path are satisfiable.[CGPP12]

To check the consistency of a temporal workflow, Combi *et al.* first decompose it into wf-paths. A wf-path represents a possible workflow execution (conditional/alternative split connector has only one successor). In each wf-path the granularities of temporal constraints are transformed into finest possible granularity. In the next step, each wf-path is transformed into Simple Temporal Problem (STP), in which the consistency check is performed as described in [DMP91].

Controllability is a stronger property than consistency and also considers the contingent nature of durations of those tasks that cannot be influenced by the agent (contingent links). In general, in a controllable temporal workflow it is possible to satisfy all temporal constraints for any possible duration of contingent links.[CP09, CP10, CGMP12]

In context of temporal constraint networks, Vidal *et al.* [Vid99, Vid00] distinguish between *strong*, *weak*, and *dynamic* controllability. Strong controllability ensures the existence of one universal solution that fits all possible durations of all contingent links. In contrast to strong controllability, weak controllability ensures the existence of a solution for each possible duration of a contingent link. Such a solution is not universal and does not necessarily fit different durations of contingent links. While strong controllability is suitable if the workflow execution situation is totally unknown, weak controllability is suitable if the execution situation is totally known. In a situation where the execution situation is partially known (e.g. runtime), dynamic controllability becomes very interesting. Dynamic controllability ensures that at any point in time during a workflow execution an agent can make decisions depending on observed execution (and observed durations of contingent links) without preventing any of the possible durations of upcoming contingent links.[Vid99, Vid00]

In [CP09, CP10], Combi *et al.* show how the controllability concept can be used in temporal workflows. For controllability checking, Combi *et al.* translate a temporal workflow into a Simple Temporal Network with Uncertainty (STNU), introduced by Morris *et al.* in [MMV01, MM05], or a Conditional Simple Temporal Network with Uncertainty (CSTNU), introduced by Hunsberger *et al.* in [HPC12]. A CSTNU combines a STNU with the Conditional Simple Temporal Problem and thus makes it suitable as an underlying formalism for workflows with conditional nodes (XORs). Some controllability checking algorithms can be found in [MMV01, MM05, CHP13, LPCR13, CHM$^+$14, CP18].

3.3 Cycle Handling Overview

In the previous section a selection of formal representations of temporal aspects in business processes and their verification was described. All temporal process representations have one shortcoming in common: they don't handle loops comprehensively.

In the literature, loops are mostly not handled at all (e.g. [LSPG06, EGP00]), handled as a complex activity (e.g. [Mar00, BWJ00]), or rolled out into a sequence (e.g. [SKK05]).

Combi *et al.* propose a more advanced approach of loop handling - a translation of loops and related temporal constraints into conditional blocks (XORs) [CGPP12, CGMP12, LPCR13, CGMP14]. Rewriting loops into conditional blocks offers an adequate handling of loops and related temporal constraints, however the authors meet several restrictions to be able to handle the complexity that arises with loops. They limit the maximum number of loop iterations already in the process model as well as the variety of temporal constraints to only such constraints that consider cyclic elements between two directly succeeding iterations [LPCR13] or the same iteration [CGPP12].

In [Pic06], Pichler introduced an advanced loop handling approach that allows the handling of unbounded loops. The author assigns branching probabilities to workflow graphs and uses this information to transform a cyclic workflow graph into an acyclic graph called a *Probabilistic Unfolded Workflow Graph*. To prevent an infinite growth of the graph, graph expansion stops when the probability of missing cases is below a certain threshold.

Pichler *et al.* introduce another interesting approach that considers loops in [PEC17]. They introduce temporal splits and temporal loops that use temporal conditions to decide which branch in a split will be taken or if a loop can be entered. An example of such a temporal condition attached to a temporal loop is $elapsed < 100$. Such loop can only be entered if less than 100 time units passed since the process instance got started.

In the literature, loops are mostly mentioned as the part of the business process that is not the focus of the work. The intention of this thesis is to close this gap and to focus solely on loops. Our contribution is described in following chapters that are the essence of this thesis. They add new knowledge to the field modeling and analysis of time perspective of business processes with loops.

We first introduce Extended Time Constraints for modeling time information in processes with loops in chapter 4. Then we describe a technique for checking the process termination property in chapter 5. This property states that a process with loops can not get stuck in an infinite loop without violating at least one (Extended) Time Constraint related to the process. A prototypical implementation in chapter 6 shows the feasibility of termination property checking.

Chapter 4

Extended Time Constraints for Cyclic Processes

Cyclic processes are very common in the industry, however, there is still a lack of adequate modeling and verification of time constraints in cyclic processes. In this chapter, we introduce *Extended Time Constraints* (ETC) that extend well known *Upper Bound Constraints* and *Lower Bound Constraints*, introduced by Eder *et al.* [EPPR99]. Upper Bound and Lower Bound Constraints can only be defined on acyclic processes, whereas Extended Time Constraints can also be defined on cyclic processes.

With the introduced Extended Time Constraints, we cover typically required time constraints in cyclic processes. An example that requires compliance with such Extended Time Constraints is the energy supplier change process in the deregulated Austrian energy market. This interorganizational process, including all time constraints, is defined in the specification of the energy market communication [Lie18]. This specification represents the implementation of the legal regulation regarding energy supplier switching [Bun14].

Figure 4.1 shows a simplified process of customer switching between various energy suppliers. The process involves the following actors: customer (white activities), current energy supplier (blue activity), new energy supplier (green activities), and distribution network operator (orange activities). All process activities, loops, and time constraints are described below and they are additionally listed in tables 4.1, 4.2, and 4.3 for easier process understanding.

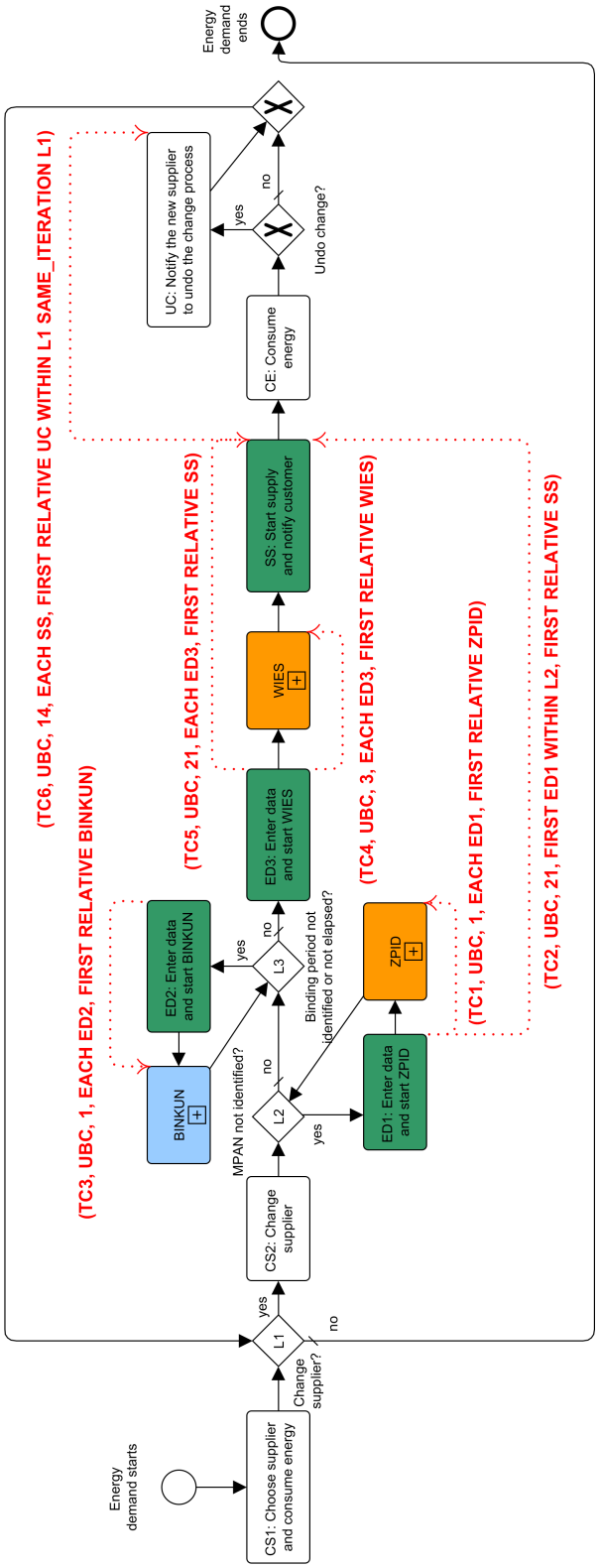


Figure 4.1: Energy supplier switch process in deregulated Austrian energy market (derived from the specification of the energy market communication [Lie18])

Activities		
ID	**Name/Description**	**Actor**
CS1	Choose supplier and consume energy.	Customer
CS2	Change supplier.	Customer
ED1	Enter data and start ZPID.	New energy supplier
ZPID	Identify customer and his/her Meter Point Administration Number (MPAN).	Distribution network operator
ED2	Enter data and start BINKUN.	New energy supplier
BINKUN	Determine customer's binding period and notice period.	Current energy supplier
ED3	Enter data and start WIES.	New energy supplier
WIES	Switch the old energy supplier to the new one and distribute all relevant switch information to all involved market participants.	Distribution network operator
SS	Start supply and notify customer.	New energy supplier
CE	Consume energy.	Customer
UC	Notify the new supplier to undo the change process.	Customer

Table 4.1: Activities in deregulated energy market

Loops	
ID	**Description**
L1	A customer can choose any Austrian energy supplier from the market and change it any time.
L2	If the answer of the distribution network operator does not return the customer's MPAN, the new energy supplier can repeat the MPAN identification steps for any possible variation of customer's data.
L3	ED2 and BINKUN can be repeated until binding and notice period elapsed.

Table 4.2: Loops in deregulated energy market

Extended Time Constraints	
ID	**Definition and Description**
TC1	`(TC1, UBC, 1, EACH ED1, FIRST RELATIVE ZPID)` TC1 constrains the time span between an occurrence of the activity ED1 and first succeeding occurrence of the activity ZPID to a maximum of 1 day (24 hours).
TC2	`(TC2, UBC, 21, FIRST ED1 WITHIN L2, FIRST RELATIVE SS)` Initiation of supplier change officially starts with the first ZPID initiation by the new energy supplier (activity ED1) or with WIES initiation (activity ED3) if there is no need to query the customer's MPAN first.
TC3	`(TC3, UBC, 1, EACH ED2, FIRST RELATIVE BINKUN)` The actual switch of the energy supplier WIES can start after the customer and the corresponding MPAN have been determined, and the current supplier has confirmed that the customer's binding and notice period have elapsed, and thus the customer can switch to another energy supplier. Binding and notice period are delivered via market process BINKUN and may take at most 24 hours (1 day)
TC4	`(TC4, UBC, 3, EACH ED3, FIRST RELATIVE WIES)` The actual supplier switch subprocess (activity WIES) must be completed by the distribution network operator within 72 hours after the new supplier has initiated the processes as modeled by TC4.
TC5	`(TC5, UBC, 21, EACH ED3, FIRST RELATIVE SS)` Initiation of supplier change officially starts with the first ZPID initiation by the new energy supplier (activity ED1) or with WIES initiation (activity ED3) if there is no need to query the customer's MPAN first.
TC6	`(TC6, UBC, 14, EACH SS, FIRST RELATIVE UC WITHIN L1 SAME_ITERATION L1)` However, according to the regulation, the customer can disagree with the contract within the next 14 days and switch back to the old energy supplier without any consequences (activity UC).

Table 4.3: Extended Time Constraints in deregulated energy market

The process starts with customer's energy demand. A customer can choose any Austrian energy supplier from the market and can also change it any time as modeled by the first loop $L1$. If a customer has decided to change the energy supplier and requests a contract from a new supplier, the new supplier needs the customer's Meter Point Administration Number (MPAN) in order to complete the contract. The MPAN identifies a unique point (meter point) to which energy suppliers deliver energy.

According to the market communication specification, customers should be able to switch to a new energy supplier even if they do not know their MPAN. In this case, the new energy supplier must find out the customer's MPAN. This is done by entering customer's data into the system (activity $ED1$) and by initiating the MPAN identification process ($ZPID$) via market communication. If the customer's data doesn't perfectly match with the data of the distribution network operator, the operator must manually search for the corresponding MPAN.

The answer (MPAN or no result) must be sent back to the new energy supplier within 24 hours as required by the market communication specification. This temporal requirement is modeled by Extended Time Constraint `(TC1, UBC, 1, EACH ED1, FIRST RELATIVE ZPID)`. $TC1$ constrains the time span between an occurrence of the activity $ED1$ and first succeeding occurrence of the activity $ZPID$ to maximum 1 day (24 hours).

If the answer of the distribution network operator does not return the customer's MPAN, the new energy supplier can repeat the MPAN identification steps for any possible variation of customer's data, as modeled by loop $L2$.

The actual switch of the energy supplier $WIES$ can start after the customer and the corresponding MPAN have been determined and the current supplier has confirmed that the customer is not violating any binding or notice period and can thus switch to another energy supplier. Binding and notice periods are delivered via market process $BINKUN$ and may take at most 24 hours (1 day), as modeled by the ETC `(TC3, UBC, 1, EACH ED2, FIRST RELATIVE BINKUN)`. $ED2$ and $BINKUN$ can be repeated as often as needed, as modeled by the loop $L3$.

The actual supplier switch subprocess (activity $WIES$) must be completed by the distribution network operator within 72 hours after the new supplier has initiated the processes, as modeled by `(TC4, UBC, 3, EACH ED3, FIRST RELATIVE WIES)`. The

subprocess $WIES$ includes activities such as authority check, process overlapping check, assignment of the new energy supplier to the given MPAN, and several others.

The market communication specification requires that the customer must be able to start consuming the energy from the new supplier in less than three weeks after he/she has initiated the supplier change. Initiation of a supplier change officially starts with the first ZPID initiation by the new energy supplier (activity $ED1$), or with WIES initiation (activity $ED3$) if there is no need to query the customer's MPAN first. This requirement is modeled by ETCs `(TC2, UBC, 21, FIRST ED1 WITHIN L2, FIRST RELATIVE SS)` and `(TC5, UBC, 21, EACH ED3, FIRST RELATIVE SS)`.

Finally, the customer starts consuming energy from the new supplier (activity CE). However, according to the regulation, the customer can disagree with the contract within the next 14 days and switch back to the old energy supplier without any consequences (activity UC), as modeled by the ETC `(TC6, UBC, 14, EACH SS, FIRST RELATIVE UC WITHIN L1 SAME_ITERATION L1)`.

The previous example of an energy supplier switch in the deregulated Austrian energy market demonstrates that there is a need for Extended Time Constraints specification in processes with loops. In this chapter, the underlying basic models will be defined in section 4.1 and a formal specification of Extended Time Constraints will be developed in section 4.2.

4.1 Basic Models and Definitions

In the following sections, we define the basic models that we use as a basis for Extended Time Constraints specification and for time management in processes with loops. The starting point is a cyclic process graph. Based on a process graph, we define a special form of a process graph - a Loop Instance Type (LIT). In an LIT, each cycle from the process graph is resolved in one or many XOR-blocks. Therefore, an LIT is always acyclic. Furthermore, we define an Instance Type, which is a subgraph of an LIT starting with the same start node and ending with the same end node. In an Instance Type, each decision node (XOR-split or LOOP-XOR-split) has only one direct successor instead of two.

4.1.1 Process Graph

A process graph is the starting point for Extended Time Constraints and is defined as follows:

Definition 4.1. *(**Process Graph** (P)) A process graph P is a directed graph* $P = (N_P, E_P)$ *that consists of nodes* N_P *and edges* E_P*. Each node* $x \in N_P$ *has a unique label* $x.Label$ *and a node type* $x.Type$*. There are nine types of nodes in a P:*

- *activity node* $(x.Type = ACT)$
- *AND-split* $(x.Type = AS)$
- *AND-join node* $(x.Type = AJ)$
- *XOR-split* $(x.Type = XS)$
- *XOR-join node* $(x.Type = XJ)$
- *LOOP-split* $(x.Type = LS)$
- *LOOP-join node* $(x.Type = LJ)$
- *LOOP-XOR-split node* $(x.Type = LXS)$
- *LOOP-XOR-join node* $(x.Type = LXJ)$.

A directed edge $(x, y) \in E_P$ *connects two nodes* $x \in N_P$ *and* $y \in N_P$*. x is called a direct predecessor of y and y is called a direct successor of x. An edge* $(c, y) \in E_P$ *that follows a conditional node* $c \in N_P$ *with* $c.Type{=}XS$*,* $c.Type{=}LXS$ *or* $c.Type{=}LS$ *is designated as a true-edge* (c, y, T) *if it is triggered in case that the node condition applies or as a false-edge* (c, y, F) *if it is triggered in case that the node condition does not apply.*

Each process graph P starts with one start activity node $s \in N_P$ *with an indegree*[1] $deg^-(s) = 0$ *and ends with one end activity node* $e \in N_P$ *with an outdegree* $deg^+(e) = 0$*. Each activity node* $a \in N_P$ *that is not a process start or end node, has an indegree* $deg^-(a) = 1$ *and an outdegree* $deg^+(a) = 1$*. Each split node* $s \in N_P$ *with* $s.Type{=}AS$*,* $s.Type{=}XS$*, or* $s.Type{=}LXS$ *has an indegree* $deg^-(s) = 1$ *and an outdegree* $deg^+(s) = 2$*. Each join node* $j \in N_P$ *with* $j.Type{=}AJ$*,* $j.Type{=}XJ$*, or*

[1] The indegree $deg^-(x)$ of a node $x \in N_P$ is the number of ingoing edges and the outdegree $deg^+(x)$ of a node $x \in N_P$ is the number of outgoing edges.

$j.Type$=LXJ has an indegree $deg^-(j) = 2$ and an outdegree $deg^+(j) = 1$. Each LOOP-split node $ls \in N_P$ with $ls.Type$=LS has an indegree $deg^-(ls) = 2$ and an outdegree $deg^+(ls) = 2$. Each LOOP-join node $lj \in N_P$ with $lj.Type$=LJ has an indegree $deg^-(lj) = 1$ and an outdegree $deg^+(lj) = 1$. The only outgoing edge of a LOOP-join node connects it with its counterpart LOOP-split node.

Each process graph P adheres to the conformance class full-blocked, where each split-node has a counterpart join-node and each outgoing path of a split node goes through the counterpart join node.

Figure 4.2 shows an example of a (cyclic) process graph. This process graph consists of 8 activity nodes (A, B, C, D, E, F, G, and H), 8 gateway nodes ($XS1$, $XJ1$, $LS1$, $LJ1$, $LS2$, $LJ2$, $AS1$, and $AJ1$), and the connecting edges. Nodes $XS1$, $LS1$, and $LS2$ are conditional nodes (XOR-split and LOOP-split nodes), therefore the outgoing edges have the labels T and F that mark the edges as true- and false-edges. The process graph starts with the start node A and ends with the end node H.

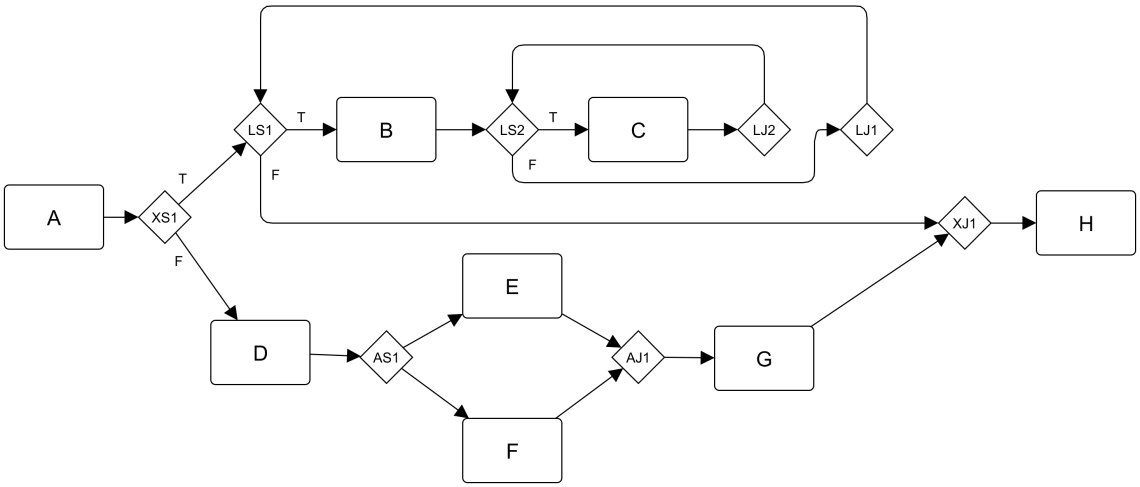


Figure 4.2: Example of a process graph

On a process graph, we define several predicates and functions (predecessor, successor, indegree, outdegree, path, etc.) that we later use to define a Loop Instance Type, an Instance Type, Extended Time Constraints, and Atomic Time Constraints.

Direct predecessor and direct successor

In a process graph P, a node $x \in N_P$ is called a *direct predecessor* of y and y is called a *direct successor* of x, iff there is an edge $(x, y) \in E_P$ between x and y:

$$dpred(x, y, P) := x, y \in N_P \wedge (x, y) \in E_P$$

$$dsucc(x, y, P) := x, y \in N_P \wedge (y, x) \in E_P$$

For example, in figure 4.2, node E is a direct successor of AND-split node $AS1$ and a direct predecessor of AND-join node $AJ1$.

Indegree and outdegree

The number of ingoing edges into a node $x \in N_P$ is called the indegree $deg^-(x)$, and the number of outgoing edges is called the outdegree $deg^+(x)$. Functions $indeg(x, P)$ and $outdeg(x, P)$ return the indegree and outdegree respectively for the node x from the process graph P:

$$indeg(x, P) := |\{x, i \in N_P | dpred(i, x)\}|$$

$$outdeg(x, P) := |\{x, o \in N_P | dsucc(o, x)\}|$$

The indegree of the AND-split node $AS1$ in figure 4.2 is 1 and the outdegree of the same node is 2, since it has two direct successors.

Start and end node

Each process graph P starts with one start node $s \in N_P$ with an indegree $indeg(s, P) = 0$ and ends with one end node e with an outdegree $outdeg(e, P) = 0$ such that the following applies:

$$start(s, P) := s \in N_P \wedge \nexists\, x(x \in N_P \wedge dpred(x, s, P))$$

$$end(e, P) := e \in N_P \wedge \nexists\, x(x \in N_P \wedge dsucc(x, e, P)).$$

In figure 4.2, node A has no ingoing edges (indegree 0) and is therefore the starting node of the process graph. Respectively, node H is the end node of the process, since it has no outgoing edges (outdegree 0).

Path

Two nodes x_i and x_k are connected with a path if the following applies:

$$path(x_i, x_k, P) := x_i, x_k \in N_P \wedge i, k \in \mathbb{N} \wedge i > 0 \wedge k > 1 \wedge$$
$$(dpred(x_i, x_k, P) \vee path(x_i, x_{k-1}, P) \wedge dpred(x_{k-1}, x_k, P))$$

The predicate $path(x_i, x_k, P)$ recursively checks if there is a sequence of edges that connects the first path node x_i with the last path node x_k. Two nodes x_i and x_k are connected with a non-loop path (a path without any edges that connect a LOOP-join with a LOOP-split node) if the following applies:

$$nlpath(x_i, x_k, P) := x_i, x_k \in N_P \wedge i, k \in \mathbb{N} \wedge i > 0 \wedge k > i \wedge$$
$$(dpred(x_i, x_k, P) \wedge \neg(x_i.Type{=}LJ \wedge x_k.Type{=}LS)$$
$$\vee path(x_i, x_{k-1}, P) \wedge dpred(x_{k-1}, x_k, P) \wedge \neg(x_{k-1}.Type{=}LJ \wedge x_k.Type{=}LS))$$

In figure 4.2, there is a path between node A and node B, since there is a sequence of edges ($A{\rightarrow}XS1, XS1{\rightarrow}LS1, LS1{\rightarrow}B$) that connects them. This path is also a non-loop path, since none of the edges that connect A and B is an edge between a LOOP-join node and a LOOP-split node. There is also a path between C and H, e.g.: $C{\rightarrow}LJ2, LJ2{\rightarrow}LS2, LS2{\rightarrow}LJ1, LJ1{\rightarrow}LS1, LS1{\rightarrow}XJ1, XJ1{\rightarrow}H$. However, this path is not a non-loop path, since the sequence of edges in this path consists of two edges between a LOOP-join node and a LOOP-split node ($LJ2{\rightarrow}LS2$ and $LJ1{\rightarrow}LS1$).

Predecessor and Successor

Each node x, for which there exists a path from x to y, is a *predecessor* of the node y and y is a *successor* of the node x:

$$pred(x, y, P) := x, y \in N_P \wedge path(x, y, P)$$

$$succ(y, x, P) := x, y \in N_P \wedge path(x, y, P)$$

Each node x, for which there exists a non-loop path from x to y, is a non-loop predecessor of the node y and y is a non-loop successor of the node x:

$$nlpred(x, y, P) := x, y \in N_P \wedge nlpath(x, y, P)$$

$$nlsucc(y, x, P) := x, y \in N_P \wedge nlpath(x, y, P)$$

In figure 4.2, A is a predecessor of B, but also C is a predecessor of B, since there is a path between C and B: $C \rightarrow LJ2, LJ2 \rightarrow LS2, LS2 \rightarrow LJ1, LJ1 \rightarrow LS1, LS1 \rightarrow B$. Note that C is also a successor of B, as well as H is a successor of B. However, C is not a non-loop predecessor of B, nor is H a non-loop successor of B.

Topological order

We introduce the operator $<$ to describe the topological order of the nodes. The operator $<$ is defined as follows:

$$(x < y) \wedge x, y \in N_P \Leftrightarrow path(x, y, P).$$

The topological order of nodes on a non-loop path is denoted with $<_{nl}$ and defined as follows:

$$(x <_{nl} y) \wedge x, y \in N_P \Leftrightarrow nlPath(x, y, P).$$

In figure 4.2, the following topological order can be observed: $B < C$. However, $C < B$ is also true, whereas $C <_{nl} B$ is not true.

Counterpart

Each process graph P must adhere to the conformance class full-blocked. In a full-blocked process graph, each split-node has a counterpart join-node, and each outgoing path of a split node goes through the corresponding counterpart join node. An AND-block starts with an AND-split node and is closed with its counterpart AND-join node. Respectively, an XOR-block/LOOP-block/LOOP-XOR-block starts with an XOR-split/LOOP-split/LOOP-XOR-split node and ends with its counterpart XOR-join/LOOP-join/LOOP-XOR-join node. The counterpart of an activity is the activity itself.

Node c is the counterpart node of node x in process graph P if the following applies:

$$
\begin{aligned}
&counterpart(c, x, P) := c, x \in N_P \wedge (\\
&(x.Type{=}ACT \wedge c.Type{=}ACT \wedge x = c) \vee \\
&(x.Type{=}LS \wedge c.Type{=}LJ \wedge dpred(c, x, P)) \vee \\
&(x.Type{=}LJ \wedge c.Type{=}LS \wedge dsucc(c, x, P)) \vee \\
&((x.Type{=}LXS \wedge c.Type{=}LXJ \vee x.Type{=}XS \wedge c.Type{=}XJ \vee \\
&x.Type{=}AS \wedge c.Type{=}AJ) \wedge x <_{nl} c \wedge \\
&|\{s \in N_P | s.Type = x.Type \wedge succ(s, x, P) \wedge pred(s, c, P)\}| = \\
&|\{j \in N_P | j.Type = c.Type \wedge succ(j, x, P) \wedge pred(j, c, P)\}|) \vee \\
&((x.Type{=}LXJ \wedge c.Type{=}LXS \vee x.Type{=}XJ \wedge c.Type{=}XS \vee \\
&x.Type{=}AJ \wedge c.Type{=}AS) \wedge c <_{nl} x \wedge \\
&|\{s \in N_P | s.Type = c.Type \wedge pred(s, x, P) \wedge succ(s, c, P)\}| = \\
&|\{j \in N_P | j.Type = x.Type \wedge pred(j, x, P) \wedge succ(j, c, P)\}|)
\end{aligned}
$$

In figure 4.2, the counterpart of $XS1$ is $XJ1$ (and vice versa), the counterpart of $AS1$ is $AJ1$, $LS1$ is the counterpart of $LJ1$, and $LS2$ is the counterpart of $LJ2$. $LJ2$ is not a counterpart of $LS1$. The counterpart of B is B itself.

Loop block

In subsection 4.1.2 we define a Loop Instance Type - an acyclic process graph derived from a given (cyclic) process graph. The evaluation of whether or not a node is located in a given loop helps us to define a Loop Instance Type.

A node $x \in N_P$ is located within a LOOP-block that starts with the LOOP-split node $s \in N_P$ if the following applies:

$$
\begin{aligned}
&inLoop(x, s, P) := x, s, j \in N_P \wedge (s <_{nl} x < j) \\
&\wedge s.Type = LS \wedge j.Type = LJ \wedge counterpart(s, j, P) \\
&\wedge \neg(\exists(s, n, F), \forall(p, q)(n, p, q \in N_P \wedge (s, n, F), (p, q) \in E_P \\
&\qquad \wedge p.Type \neq LS \wedge p < x \wedge n < q))
\end{aligned}
$$

In figure 4.2, C is both in the loop that starts with the split node $LS1$ and in the loop that starts with the split node $LS2$, while B is only in the loop that starts with $LS1$. A, D, E, F, G, and H are neither in loop $LS1$ nor in loop $LS2$.

To determine if node $l \in N_P$ is the split node of the most inner loop a node $x \in N_P$ is placed in, we define the following predicate:

$$closestLoop(l, x, P) := l, x \in N_P \wedge l.Type = LS \wedge l <_{nl} x \wedge \nexists k(k \in N_P \wedge k.Type = LS \wedge (l <_{nl} k <_{nl} x))$$

In figure 4.2, $LS2$ is the closest loop C is in, and $LS1$ is the closest loop B is in. Other activities are not placed in loops at all.

The process graph and related predicates and functions that we introduced in this section are the starting point for Loop Instance Types and Instance Types. Loop Instance Types and Instance Types are derived from a process graph and are needed for Extended Time Constraints definition in our research. Loop Instance Type is introduced in the next section.

4.1.2 Loop Instance Type

A Loop Instance Type (LIT) is an acyclic process graph, derived from a cyclic process graph, where loops (cycles) have been transformed to one or many nested so called LOOP-XOR-blocks (XOR-blocks derived from a loop). We use a Loop Instance Type as a basis to define an Instance Type which we then use to define the semantic of Extended Time Constraints.

A Loop Instance Type L of a process graph P is a directed acyclic process graph such that for each node $x \in N_P$, except LOOP-split and LOOP-join nodes, there is at least one node $x' \in N_L$ such that $x.Label = x'.Label \wedge x.Type = x'.Type$. Each LOOP-block from P is transformed to at least one LOOP-XOR-block in L. The node types LS and LJ from P are transformed into types LXS (LOOP-XOR-split) and LXJ (LOOP-XOR-join) in the derived LIT. We say that $x' \in N_{P'}$ is the equivalent node of $x \in N_P$, where P' is a process graph derived from P (e.g. a Loop Instance Type):

$$equi(x, x', P, P') := x \in N_P \wedge x' \in N_{P'} \wedge x.Label = x'.Label \wedge (x.Type = x'.Type \vee (x.Type{=}LS \wedge x'.Type{=}LXS) \vee (x.Type{=}LJ \wedge x'.Type{=}LXJ))$$

There can be a set of equivalent nodes that were derived from the same node in the underlying process graph. We introduce the **Loop Counter Vector (LCV)**, which makes each of the derived equivalents with the same label in a Loop Instance Type or Instance Type unique.
Each node $n' \in N_{P'}$ in a Loop Instance Type or Instance Type has a Loop Counter Vector $n'.LCV$ that is defined as follows:

Definition 4.2. *(**Loop Counter Vector (LCV)**) A Loop Counter Vector $n'.LCV$ of a node $n' \in N_{P'}$ is a k-tuple $(c^{n'}_{LS_1}, c^{n'}_{LS_2}, \cdots, c^{n'}_{LS_k})$ where each scalar component $c^{n'}_{LS_i} \in \mathbb{N}_0$ denotes an iteration counter of the corresponding LOOP-block, indicated by a LOOP-split node $l \in N_P \wedge l.Type = LS \wedge l.Label = LS_i$ for the node n'.*

A $n'.LCV$ of a node $n' \in N_{P'}$ contains one scalar component $c^{n'}_{LS_i}$ for each LOOP-split node $l \in N_P \wedge l.Type = LS \wedge l.Label = LS_i$, thus the number of dimensions of a $n'.LCV$ is the number of loops in a process graph P: $d = |\{l | l \in N_P \wedge l.Type = LS\}|$. The default value of a $n'.LCV$ of a node $n' \in N_{P'}$ is $(c_1, c_2, \cdots, c_k)$, where $\forall^k_{i=1} i : c_i = 0$ and k is the number of loops in the process graph P.
We define the function $lc(n', LS_i, P')$ that returns the scalar component $c^{n'}_{LS_i}$ (the loop counter of the loop with label LS_i) from the Loop Counter Vector $n'.LCV$ of the node $n' \in N_{P'}$:

$$lc(n', LS_i, P') := c^{n'}_{LS_i}$$

Figure 4.3 shows a Loop Instance Type L that was derived from the process graph P in figure 4.2. The LIT L differs from P only in the upper "true"-part of the XOR-block. In this part, the two LOOP-blocks from P were transformed into three LOOP-XOR-blocks. There is one LOOP-XOR-block in L for the first LOOP-block starting with $LS1$ in P and two LOOP-XOR-blocks for the second LOOP-block starting with $LS2$ in P. Therefore all nodes from P except $LXS2$, C, and $LXJ2$ have one equivalent node in L that was derived from its origin in P. This particular LIT L has two nested LOOP-XOR-blocks starting with $LXS2$ that represent two possible iterations of the LOOP-block starting with $LS2$ in P. As a consequence, nodes with labels $LXS2$, C, and $LXJ2$ appear twice in L. However, these nodes can be uniquely addressed thanks to their Loop Counter Vectors.

Each node in LIT in figure 4.3 has a Loop Counter Vector (LCV) placed in the lower right corner or somewhere near the node. All nodes except the nodes between $LXS1$ and $LXJ1$ have an LCV (0,0), since they do not appear in any LOOP-block, and therefore their iteration counters of surrounding loops can only have the default value 0. A loop iteration is being counted after a loop has been entered, meaning its LOOP-split node has been passed. This can be observed for activity B with the LCV (1,0) that denotes that this B is executed in the first iteration of the first LOOP-block starting with $LS1$. The first occurrence of activity C with the LCV (1,1) is executed in the first iteration of the first LOOP-block starting with $LS1$ and the first iteration of the second LOOP-block starting with $LS2$ that is nested in the first LOOP-block. The second occurrence of activity C with the LCV (1,2) is executed in the first iteration of the first LOOP-block and the second iteration of the second LOOP-block.

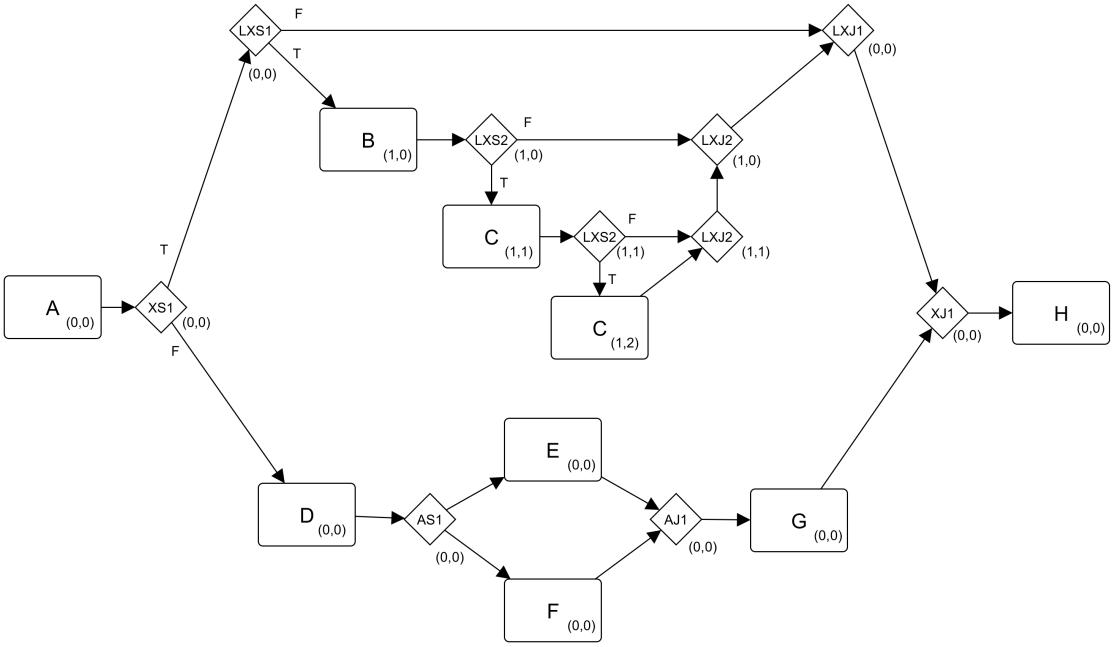


Figure 4.3: Example of a Loop Instance Type

There is an infinite set of Loop Instance Types for each (cyclic) process graph P. Figure 4.3 shows only one possible LIT, derived from P, which has one possible iteration of the first loop and two possible iterations of the second loop. Since we deal with processes with unbounded loops, each possible combination of loop iteration numbers leads to a different LIT.
This leads us to the definition of a Loop Instance Type:

Definition 4.3. *(**Loop Instance Type** (LIT)) A Loop Instance Type L of a process graph P is a directed acyclic process graph $L = (N_L, E_L)$. A process graph L is a Loop Instance Type of a process graph P ($litOf(L, P)$) if and only if it satisfies all following rules:*

Rule 1 - nodes:
For each node x in P, there is at least one derived node x' in the derived L.

$$\forall x \exists x' (x \in N_P \wedge x' \in N_L \wedge equi(x, x', P, L))$$

Rule 2 - start node:
An LIT starts with a start node x' that was derived from the start node x from the corresponding process graph P.

$$\forall x' \exists x ((x \in N_P \wedge x' \in N_L \wedge equi(x, x', P, L) \wedge start(x', L)) \Rightarrow start(x, P))$$

Rule 3 - end node:
An LIT ends with an end node x' that was derived from the end node x from the corresponding process graph P.

$$\forall x' \exists x ((x \in N_P \wedge x' \in N_L \wedge equi(x, x', P, L) \wedge end(x', L)) \Rightarrow end(x, P))$$

Rule 4 - path between nodes:
Each node x' in an LIT is placed on a path between the start node s' and the end node e'.

$$\forall x', s', e'((x', s', e' \in N_L \wedge start(s', L) \wedge end(e', L)) \\ \Rightarrow (pred(s', x', L) \wedge succ(e', x', L)))$$

Rule 5 - LCV (non related loops):
Only valid LCVs must appear in an LIT. In a valid LCV, a scalar component $c^{x'}_{l.Label}$ in the Loop Counter Vector $x'.LCV$ of the node $x' \in N_L$ must equal 0 for each loop l that is not a surrounding loop of the equivalent node of x' in P.

$$\forall x, x', l((x, l \in N_P \wedge x' \in N_L \wedge l.Type = LS \wedge \neg inLoop(x, l, P) \\ \wedge equi(x, x', P, L)) \Rightarrow lc(x', l.Label, L) = 0)$$

Rule 6 - LCV (nested loops):
In a valid LCV, a scalar component for a particular loop can only be positive if the scalar components for all outer loops are greater than 0, since an inner loop can only be entered if the outer loops have been entered before.

$$\forall l, k, x'((l, k \in N_P \wedge x' \in N_L \wedge l.Type = LS \wedge k.Type = LS \\ \wedge l \neq k \wedge inLoop(l, k, P) \wedge lc(x', l.Label, L) > 0) \\ \Rightarrow lc(x', k.Label, L) > 0)$$

Rule 7 - LCV (iteration increment):
A valid LCV of a node is an LCV where each scalar component (loop counter) is a result of an increment of the scalar component of the previous iteration of the corresponding loop. If, for a node $x'' \in N_L$, there is a scalar component (counter) $c^{x''}_{l.Label} = n$ for the loop l in its LCV $x''.LCV$, then there also must exist a node x' with a scalar component $c^{x'}_{l.Label} = n-1$ for the same loop l in its LCV $x'.LCV$, while the other scalar components of both nodes x'' and x' are the same.

$$\forall l, k, x, x''\exists x'((x, l, k \in N_P \wedge x', x'' \in N_L \wedge l.Type = LS \wedge k.Type = LS \\ \wedge l \neq k \wedge equi(x, x', P, L) \wedge equi(x, x'', P, L) \wedge lc(x'', l.Label, L) > 1) \\ \Rightarrow succ(x'', x', L) \wedge lc(x', l.Label, L) = lc(x'', l.Label, L) - 1 \\ \wedge lc(x', k.Label, L) = lc(x'', k.Label, L))$$

Rule 8 - edges:

Each edge $(x, y) \in E_P$ *that does not start or end with a LOOP-split or LOOP-join node in the corresponding* P*, and each edge* $(x, y) \in E_P$ *that ends with a LOOP-split node, but does not start with a LOOP-split node, appears as a derived edge* $(x', y') \in E_L$ *between equivalent nodes* x' *and* y' *that both have the same LCV.*

$$\begin{aligned}
&\forall x, y, x', y'((x, y \in N_P \wedge (x, y) \in E_P \wedge x', y' \in N_L \\
&\wedge ((x.Type \neq LS \wedge x.Type \neq LJ \wedge y.Type \neq LS \wedge y.Type \neq LJ) \\
&\quad \vee (x.Type \neq LS \wedge y.Type = LS)) \\
&\wedge equi(x, x', P, L) \wedge equi(y, y', P, L) \wedge x'.LCV = y'.LCV) \\
&\Rightarrow dsucc(y', x', L))
\end{aligned}$$

Rule 9 - LOOP-split true-edge:

Each true-edge $(x, y, T) \in E_P$ *that follows a LOOP-split node in the corresponding* P *appears as a derived edge* $(x', y', T) \in E_L$ *between the equivalent nodes* x' *and* y'*. Nodes* x' *and* y' *have the same LCV, except the scalar component* $c^{y'}_{x.Label}$ *in the Loop Counter Vector* $y'.LCV$ *of the node* $y' \in N_L$ *is increased by 1 in comparison to the scalar component* $c^{x'}_{x.Label}$ *of node* x'*.*

$$\begin{aligned}
&\forall x, y, x', y', l((x, y, l \in N_P \wedge (x, y, T) \in E_P \wedge x', y' \in N_L \\
&\wedge x.Type = LS \wedge l.Type = LS \wedge x \neq l \\
&\wedge equi(x, x', P, L) \wedge equi(y, y', P, L) \\
&\wedge lc(x', l.Label, L) = lc(y', l.Label, L) \\
&\wedge lc(x', x.Label, L) = lc(y', x.Label, L) - 1) \\
&\Rightarrow dsucc(y', x', L) \wedge (x', y', T) \in E_L)
\end{aligned}$$

Rule 10 - LOOP-split false-edge:

Each false-edge $(x, y, F) \in E_P$ *that starts with a LOOP-split node, but does not end with a LOOP-join node in the corresponding* P*, appears as a derived edge* $(x', y') \in E_L$ *between the counterpart LOOP-join node of the equivalent node* x' *and the equivalent node* y' *that both have the same LCV.*

$$\forall x, y, x', y' j' ((x, y \in N_P \wedge (x, y, F) \in E_P \wedge x', y', j' \in N_L$$
$$\wedge\, x.Type = LS \wedge y.Type \neq LS \wedge equi(x, x', P, L) \wedge equi(y, y', P, L)$$
$$\wedge\, counterpart(j', x', L) \wedge j'.LCV = y'.LCV)$$
$$\Rightarrow dsucc(y', j', L))$$

Rule 11 - LOOP-join outgoing edge:
Each edge $(x, y) \in E_P$ that starts with a LOOP-join and ends with a LOOP-split node in the corresponding P appears as a derived edge $(y', x', F) \in E_L$ between derived nodes y' and x' that both have the same LCV.

$$\forall x, y, x', y' ((x, y \in N_P \wedge (x, y) \in E_P \wedge x', y' \in N_L$$
$$\wedge\, x.Type = LJ \wedge y.Type = LS$$
$$\wedge\, equi(x, x', P, L) \wedge equi(y, y', P, L) \wedge x'.LCV = y'.LCV)$$
$$\Rightarrow (dsucc(x', y', L) \wedge (y', x', F) \in E_L))$$

Rule 12 - LOOP-join ingoing edge (last iteration):
Each edge $(x, y) \in E_P$ that ends with a LOOP-join node in the corresponding P, appears in the last iteration of the corresponding loop in L as a derived edge $(x', y') \in E_L$ between nodes x' and y'. Both nodes, x' and y', have the same LCV, except the scalar component $c^{y'}_{s.Label}$ in the Loop Counter Vector $y'.LCV$ of the node $y' \in N_L$ is decreased by 1 in comparison to the scalar component $c^{x'}_{s.Label}$ of node x'.

$$\forall x, y, l, s, t, x', y' ((x, y, l, s, t \in N_P \wedge (x, y) \in E_P \wedge x', y' \in N_L \wedge l \neq s$$
$$\wedge\, y.Type = LJ \wedge s.Type = LS \wedge counterpart(s, y, P) \wedge equi(y, y', P, L)$$
$$\wedge\, (x.Type \neq LS \wedge equi(x, x', P, L)$$
$$\quad \vee\, x.Type = LS \wedge t.Type = LJ \wedge counterpart(t, x, P) \wedge equi(t, x', P, L))$$
$$\wedge\, lc(x', l.Label, L) = lc(y', l.Label, L)$$
$$\wedge\, lc(x', s.Label, L) = lc(y', s.Label, L) + 1$$
$$\wedge\, \nexists x'' (x'' \in N_L \wedge succ(x'', x', L)$$
$$\quad \wedge\, lc(x', l.Label, L) = lc(y', l.Label, L)$$
$$\quad \wedge\, lc(x'', s.Label, L) > lc(x', s.Label, L))$$
$$\Rightarrow dsucc(y', x', L))$$

Rule 13 - LOOP-join ingoing edge (not last iteration):
Each edge $(x, y) \in E_P$ *that ends with a LOOP-join node in the corresponding* P*, appears in a not-last-iteration of the corresponding loop in* L *as two derived edges* (x', p') *and* $(q', y') \in E_L$*. The first edge* (x', p') *is the edge that connects the last loop-body node occurrence* x' *with the start of the next iteration of the same loop starting with a* $LOOP - XOR - split$ *node. The second edge* (q', y') *connects the end of the loop iteration initiated by the first edge* (x', p') *with the* $LOOP - XOR - join$ *node of the previous iteration of the same loop. Nodes* x'*,* p'*, and* q' *have the same LCV. Node* y' *has the same LCV as node* x'*, except the scalar component* $c^{y'}_{s.Label}$ *in the Loop Counter Vector* $y'.LCV$ *of the node* $y' \in N_L$ *is decreased by 1 in comparison to the scalar component* $c^{x'}_{s.Label}$ *of node* x'*.*

$$
\begin{aligned}
&\forall x, y, l, s, t, x', y', p', q'((x, y, l, s, t \in N_P \wedge (x, y) \in E_P \wedge x', y', p', q' \in N_L \wedge l \neq s \\
&\wedge y.Type = LJ \wedge s.Type = LS \wedge counterpart(s, y, P) \wedge equi(y, y', P, L) \\
&\wedge (x.Type \neq LS \wedge equi(x, x', P, L) \\
&\quad \vee x.Type = LS \wedge t.Type = LJ \wedge counterpart(t, x, P) \wedge equi(t, x', P, L)) \\
&\wedge equi(s, p', P, L) \wedge equi(t, q', P, L) \\
&\wedge lc(x', l.Label, L) = lc(y', l.Label, L) \\
&\wedge lc(x', s.Label, L) = lc(y', s.Label, L) + 1 \\
&\wedge \exists x''(x'' \in N_L \wedge succ(x'', x', L) \\
&\quad \wedge lc(x', l.Label, L) = lc(y', l.Label, L) \\
&\quad \wedge lc(x'', s.Label, L) > lc(x', s.Label, L)) \\
&\Rightarrow dsucc(p', x', L) \wedge dsucc(q', p', L) \wedge dsucc(y', q', LI))
\end{aligned}
$$

The process graph L from the figure 4.3 is a Loop Instance Type of process graph P from the figure 4.2, because it satisfies all 13 rules for a Loop Instance Type. Let's observe, how each rule applies to the LIT L:

Rule 1 - nodes:
For each node in P, there is at least one equivalent node in L. $LS2$, C, and $LJ2$ each have two equivalent nodes, all others have only one each.

Rule 2 - start node:
The start node in L is $A_{(0,0)}$, which is derived from A in P, so A must be the start node in P, which it is.

Rule 3 - end node:
The end node in L is $H_{(0,0)}$, which is derived from H in P, so H must be the end node in P, which it is.

Rule 4 - path between nodes:
Each node in L is a successor of the start node $A_{(0,0)}$ and at the same time a predecessor of the end node $H_{(0,0)}$.

Rule 5 - LCV (non related loops):
Only nodes $LS1$, B, $LS2$, C, $LJ2$, and $LJ1$ are a part of a loop in P, so only their equivalents in L can have scalar components greater than 0, which is the case. The nodes $LS1$, B, and $LJ1$ are only a part of the outer loop starting with $LS1$ in P, but not a part of the inner loop starting with $LS2$. Therefore, the scalar component for the inner loop $LS2$ in the LCVs of their equivalent nodes can only be 0, since the loop $LS2$ is not relevant for these three nodes. This statement is also true. The LIT L would not be valid in respect to this rule if there was a node B with an LCV (2,1). That would mean that this node is the occurrence of B in the second iteration of the outer loop $LS1$ and the first iteration of the inner loop $LS2$. However, B is not placed in the inner loop, and so it can also not occur in an iteration of this loop.

Rule 6 - LCV (nested loops):
Nodes $LS2$, C, and $LJ2$ are embraced with the outer loop $LS1$ in P, so their equivalents in L can have scalar components greater than 0 for both loops $LS1$ and $LS2$. However, the scalar components of the inner loop $LS2$ can only be greater than 0 if the scalar components for the outer loop $LS1$ are also greater than 0. This rule applies to L. The LIT L would not be valid in respect to this rule if there was a node C with an LCV (0,1). This would mean that this node is the occurrence of C in the first iteration of the inner loop $LS2$ but not in an iteration of the outer loop $LS1$. However, this is not possible, since the inner loop $LS2$ can only be entered if the outer loop $LS1$ was entered before.

Rule 7 - LCV (iteration increment):
There is a node C with an LCV (1,2) which means that this node is the occurrence of C in the first iteration of loop $LS1$ and the second iteration of loop $LS2$. If there is a second iteration of loop $LS2$, then there must also be a first iteration of the same loop. This means that there must be a node C with an LCV (1,1), which there is.

Rule 8 - edges:
The edges $A \rightarrow XS1$, $XS1 \rightarrow LS1$, $B \rightarrow LS2$, $XS1 \rightarrow D$, $D \rightarrow AS1$, $AS1 \rightarrow E$, $AS1 \rightarrow F$, $E \rightarrow AJ1$, $F \rightarrow AJ1$, $AJ1 \rightarrow G$, $G \rightarrow XJ1$, and $XJ1 \rightarrow H$ from P appear also as derived edges in L with the same LCV for the edge start and the edge end node: $A_{(0,0)} \rightarrow XS1_{(0,0)}$, $XS1_{(0,0)} \xrightarrow{\mathrm{T}} LS1_{(0,0)}$, $B_{(1,0)} \rightarrow LXS2_{(1,0)}$, $XS1_{(0,0)} \xrightarrow{\mathrm{F}} D_{(0,0)}$, $D_{(0,0)} \rightarrow AS1_{(0,0)}$, $AS1_{(0,0)} \rightarrow E_{(0,0)}$, $AS1_{(0,0)} \rightarrow F_{(0,0)}$, $E_{(0,0)} \rightarrow AJ1_{(0,0)}$, $F_{(0,0)} \rightarrow AJ1_{(0,0)}$, $AJ1_{(0,0)} \rightarrow G_{(0,0)}$, $G_{(0,0)} \rightarrow XJ1_{(0,0)}$, and $XJ1_{(0,0)} \rightarrow H_{(0,0)}$. The remaining edges from P ($LS1 \rightarrow B$, $LS1 \rightarrow XJ1$, $LS2 \rightarrow C$, $LS2 \rightarrow LJ1$, $C \rightarrow LJ2$, $LJ2 \rightarrow LS2$, and $LJ1 \rightarrow LS1$) are regulated by other rules.

Rule 9 - LOOP-split true-edge:
The edges $LS1 \rightarrow B$ and $LS2 \rightarrow C$ from P appear as derived edges $LXS1_{(0,0)} \xrightarrow{\mathrm{T}} B_{(1,0)}$, $LXS2_{(1,0)} \xrightarrow{\mathrm{T}} C_{(1,1)}$, and $LXS2_{(1,1)} \xrightarrow{\mathrm{T}} C_{(1,2)}$ in L, such that the LCV of the edge end node is incremented by 1, since the edge start node marks the start of a new loop iteration.

Rule 10 - LOOP-split false-edge:
The edge $LS1 \rightarrow XJ1$ from P appears as derived edge $LXJ1_{(0,0)} \rightarrow XJ1_{(0,0)}$ in L. The edge $LS2 \rightarrow LJ1$ is covered by rules 12 and 13.

Rule 11 - LOOP-join outgoing edge:
The edges $LJ1 \rightarrow LS1$ and $LJ2 \rightarrow LS2$ from P appear as derived false-edges $LXS1_{(0,0)} \xrightarrow{\mathrm{F}} LXJ1_{(0,0)}$, $LXS2_{(1,0)} \xrightarrow{\mathrm{F}} LXJ2_{(1,0)}$, and $LXS2_{(1,1)} \xrightarrow{\mathrm{F}} LXJ2_{(1,1)}$ in L.

Rule 12 - LOOP-join ingoing edge (last iteration):
The edges $C \rightarrow LJ2$ and $LS2 \rightarrow LJ1$ from P appear as derived edges $C_{(1,2)} \rightarrow LXJ2_{(1,1)}$ and $LXJ2_{(1,0)} \rightarrow LXJ1_{(0,0)}$ in L. Both derived edges close the last iteration of the corresponding loop (the second iteration of the inner loop is the last

iteration of the inner loop, and the first iteration of the outer loop is also the last iteration of the outer loop). This is indicated in the decrease of the loop counter ((1,2) is decreased to (1,1) and (1,0) is decreased to (0,0)).

Rule 13 - LOOP-join ingoing edge (not last iteration):
The edge $C \rightarrow LJ2$ from P appears as two derived edges $C_{(1,1)} \rightarrow LXS2_{(1,1)}$ and $LXJ2_{(1,1)} \rightarrow LXJ2_{(1,0)}$ in L. The first derived edge $C_{(1,1)} \rightarrow LXS2_{(1,1)}$ leads into the second iteration of the inner loop, while the second derived edge $LXJ2_{(1,1)} \rightarrow LXJ2_{(1,0)}$ closes the first iteration of the inner loop.
The outer loop that starts with the node $LS1$ in P has only one iteration in L, therefore rule 13 is irrelevant for this loop and its edge $LS2 \rightarrow LJ1$.

From each Loop Instance Type, we can derive Instance Types. We use Instance Types in the definition of Extended Time Constraints. Instance Type is introduced in the next section.

4.1.3 Instance Type

An Instance Type is a subgraph of a valid Loop Instance Type. An Instance Type starts and ends with the equivalent start and end node as the corresponding LIT. Between the start and end node, an Instance Type contains a subset of nodes and edges of the corresponding LIT, such that they are all on a path between the start and the end node and each decision node has only one direct successor instead of two. We define an Instance Type as follows:

Definition 4.4. *(**Instance Type**) An Instance Type of a process graph P is a directed acyclic graph $I = (N_I, E_I)$, which is a subgraph of a Loop Instance Type $L = (N_L, E_L)$, derived from the same process graph P (litOf(L,P)). A directed acyclic graph I is called an Instance Type of a process graph P (instOf(I,P)) if it satisfies all of the following rules:*

Rule 1 - nodes:
Each node x' in an Instance Type I has one corresponding node x in L from which it is derived.

$$\forall x' \exists x (x \in N_L \wedge x' \in N_I \wedge equi(x, x', L, I) \wedge x.LCV = x'.LCV)$$

Rule 2 - start node:
An Instance Type I starts with a start node x', which is derived from the start node x from the corresponding L.

$$\forall x' \exists x ((x \in N_L \wedge x' \in N_I \wedge equi(x, x', L, I) \wedge x.LCV = x'.LCV) \\ \wedge start(x', I)) \Rightarrow start(x, L))$$

Rule 3 - end node:
An Instance Type I ends with an end node x', which is derived from the end node x from the corresponding L.

$$\forall x' \exists x ((x \in N_L \wedge x' \in N_I \wedge equi(x, x', L, I) \wedge x.LCV = x'.LCV) \\ \wedge end(x', I)) \Rightarrow end(x, L))$$

Rule 4 - path:
Each node x' in I is placed on a path between the start node s' and the end node e'.

$$\forall x', s', e' ((x', s', e' \in N_I \wedge start(s', I) \wedge end(e', I)) \\ \Rightarrow (pred(s', x', I) \wedge succ(e', x', I)))$$

Rule 5 - edges:
Each edge (x', y') in I has one corresponding edge (x, y) in the corresponding LIT L, from which it is derived.

$$\forall x, y, x', y' ((x, y \in N_L \wedge x', y' \in N_I \wedge (x', y') \in E_I \\ \wedge equi(x, x', L, I) \wedge equi(y, y', L, I) \wedge x.LCV = x'.LCV \wedge y.LCV = y'.LCV) \\ \Rightarrow (x, y) \in E_L)$$

Rule 6 - node outdegree:
Each node x' in I, except AND-split nodes, has only one direct successor.

$$\forall x'((x' \in N_I \wedge x'.Type \neq AS) \Rightarrow outdeg(x', I) \leq 1)$$

Figures 4.4, 4.5, and 4.6 each show an example of an Instance Type of the process graph P from figure 4.2. All three Instance Types are a subgraph of a Loop Instance Type of the same process graph P (see figure 4.3), and they all satisfy all the rules for a valid Instance Type:

- they all contain only nodes that also occur in the corresponding LIT from figure 4.3 and have the same LCV;
- they all start with the node $A_{(0,0)}$, which is also the start node in the corresponding LIT;
- they all end with the node $H_{(0,0)}$, which is also the end node in the corresponding LIT;
- they all contain only nodes that are placed on a path between the start node $A_{(0,0)}$ and the end node $H_{(0,0)}$;
- they all contain only edges that also occur in the corresponding LIT between equivalent nodes with the same LCVs as they have in the Instance Types;
- they all contain only nodes with only one outgoing edge or two outgoing edges in case of the AND-split node $AS1_{(0,0)}$ in Instance Type in figure 4.4.

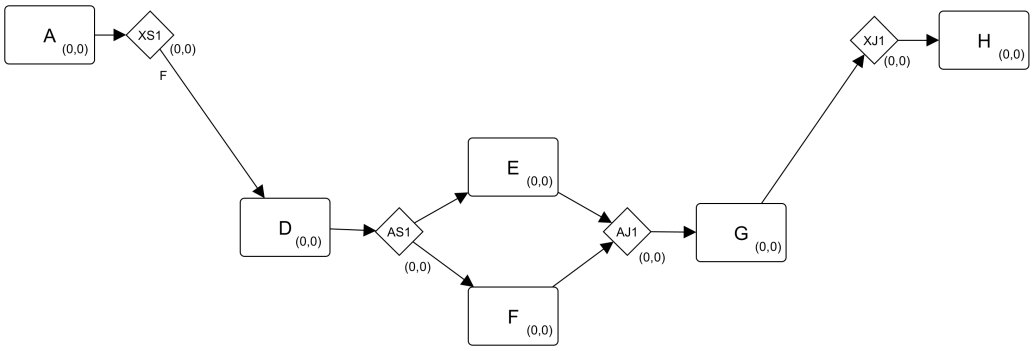


Figure 4.4: Example of a valid Instance Type $I1$

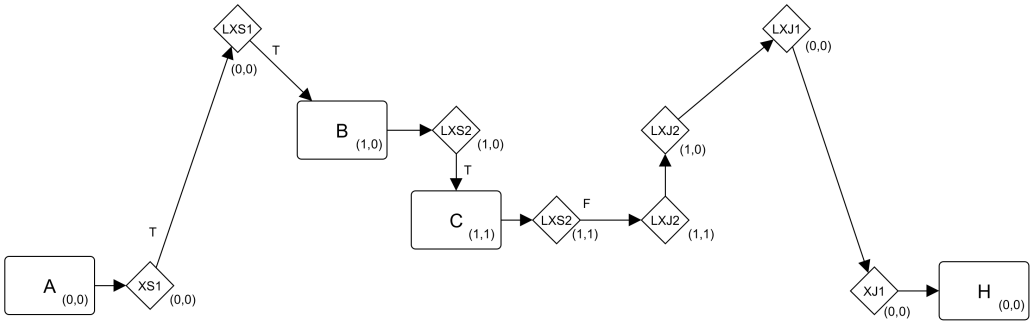


Figure 4.5: Example of a valid Instance Type $I2$

Figure 4.6: Example of a valid Instance Type $I3$

Instance Types are needed for the definition of the semantics of Extended Time Constraints. In the next section we finally introduce Extended Time Constraints which we briefly described in the motivating real world example of energy supplier switching at the beginning of this chapter.

4.2 Extended Time Constraints

In a process graph, the time perspective of the process can be considered by specifying time constraints. Two well-known types of time constraints are the Upper Bound Constraint (UBC) and the Lower Bound Constraint (LBC), introduced by Eder *et al.* [EPPR99]. An Upper Bound Constraint $ubc(A, B, \delta)$ limits the maximal duration between the ending point of the source activity A and the ending point of destination activity B to δ time units. In contrast, a Lower Bound Constraint $lbc(A, B, \delta)$ limits the minimal duration between the ending point of the source activity A and the ending point of destination activity B to δ time units as described in section 3.2.1.

The Upper and Lower Bound Constraint, however, work only for acyclic process graphs. If the source or destination activity in a time constraint $ubc(A, B, \delta)$ or $lbc(A, B, \delta)$ appears in a loop, it is not clear which activity occurrence is meant by A or by B, since there can be many of them. In order to specify the exact occurrence of the source and destination activity in a process with loops, we introduce Extended Time Constraints that extend the Upper and the Lower Bound Constraint such that they can be applied to cyclic process graphs.

An Extended Time Constraint (ETC) defines a set of source activity occurrences and a set of destination activity occurrences instead of only one source activity and one destination activity. Depending on the Instance Type on which an ETC is applied, the source and destination occurrence set can be empty, may contain one, or more than one element (nodes). Temporal relations between the single elements of the source and destination set that are defined by an ETC, are derived into Atomic Time Constraints.

An Atomic Time Constraint (ATC) limits the maximal duration (or minimal duration in case of a lower bound constraint type) between the ending point of a particular source activity occurrence and the ending point of a particular destination activity occurrence.

In this section, we define an Extended Time Constraint, including the syntax and the semantic of possible source and destination expressions in an ETC. In the next section, we define an Atomic Time Constraint and how it is derived from an ETC.

Definition 4.5. ***(Extended Time Constraint (ETC))***
An Extended Time Constraint $tc \in TC_P$ in a (cyclic) process graph P is a quintuple (ID, type, δ, source, destination) that constrains the temporal relation between the defined source node set and destination node set to a maximum of δ time units if type of tc is UBC, or to a minimum of δ time units if type of tc is LBC.

The source node set and destination node set in an ETC are specified by the expressions source and destination. Evaluation of the expression source and destination returns the source node set $S \subseteq N_P$ and the destination node set $D \subseteq N_P$.

The syntax of a *source* and *destination* expression is defined in the following subsection 4.2.1. The semantic of each possible expression is defined in subsection 4.2.2

4.2.1 Extended Time Constraints Syntax

The expressions *source* and *destination* in an Extended Time Constraint must comply with following syntax rules:

source := [<quantifier>] *nodeLabel* [<loopRef>]

destination := [<quantifier> [<relation>]] *nodeLabel*
[<loopRef> [<iterationRef>]]

<quantifier> ::= FIRST | LAST | EACH

<relation> ::= RELATIVE | ABSOLUTE

<loopRef> ::= WITHIN *loopLabel*

<iterationRef> ::= <iteration> *loopLabel*

<iteration> ::= SAME_ITERATION | NEXT_ITERATION

Each *source* or *destination* expression must at least contain a *nodeLabel*. There are three different *quantifiers* ($FIRST$, $LAST$, and $EACH$) that can be used in a *source* or *destination* expression. The default *quantifier* is $EACH$, which is the implicit *quantifier* in an expression that contains only *nodeLabel*.

The *relation* in a *destination* expression specifies if the *destination* expression is evaluated without respect to the *source* ($ABSOLUTE$) or with respect to the *source* ($RELATIVE$). The nodes in the result set of an evaluated *destination* expression with *relation* $RELATIVE$ must be successors of at least one node in the *source* result set. The default *relation* is $ABSOLUTE$, therefore *destination* expressions without a *relation* are handled as $ABSOLUTE$.

The loop reference $loopRef$ is optional in *source* and *destination* expression and specifies to which loop the *quantifier* is bound. The loop reference is only relevant if the activity *nodeLabel* we want to constrain is placed in a nested loop. In such case, the *quantifier* alone is not capable of stating the same specification as the *quantifier* together with a $loopRef$. The resulting set of the expression $FIRST\ X$, for example, would contain only one - the first appearing - node with label X. $FIRST\ X\ WITHIN\ LS1$ would result in the exact same set if $LS1$ is the outest loop regarding X. However, if $LS1$ is one of the inner loops, the resulting set of the expression $FIRST\ X\ WITHIN\ LS1$ would contain each first X after each new iteration of each loop that is surrounding the loop $LS1$. The examples in the next section will make the difference easier to understand.

The iteration reference $iterationRef$ makes it possible to include only nodes into the result set of a $RELATIVE$ *destination* that are either in the $SAME\_ITERATION$ of a particular loop with *loopLabel* as the *source*, or in the $NEXT\_ITERATION$ regarding the source node to which they relate.

The ETC syntax rules lead to the following possible absolute source and destination expressions, where X represents an arbitrary activity label and L is the label of an arbitrary LOOP-split node:

- X
- $FIRST\ X$
- $LAST\ X$
- $EACH\ X$
- $FIRST\ X\ WITHIN\ L$
- $LAST\ X\ WITHIN\ L$
- $EACH\ X\ WITHIN\ L$

Furthermore, the syntax rules allow the following relative destination expressions:

- $FIRST\ RELATIVE\ X$
- $LAST\ RELATIVE\ X$
- $EACH\ RELATIVE\ X$
- $FIRST\ RELATIVE\ X\ WITHIN\ L$
- $LAST\ RELATIVE\ X\ WITHIN\ L$
- $EACH\ RELATIVE\ X\ WITHIN\ L$
- $FIRST\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$
- $LAST\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$
- $EACH\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$
- $FIRST\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$
- $LAST\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$
- $EACH\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$

The semantic of each possible expression is defined in the following subsection.

4.2.2 Extended Time Constraints Semantic

Each *source* or *destination* expression *expr* in an Extended Time Constraint *tc* can be evaluated with the function $\xi(expr, tc, I)$ that returns the node result set $R \subseteq N_I$. The node set R contains all source nodes (or destination nodes, respectively) that satisfy the node set specification *expr*. Function $\xi(expr, tc, I)$ is defined as follows:

$$\xi(expr, tc, I) : (expr, tc, I) \mapsto R \subseteq N_I$$

In following, the resulting set $R \subseteq N_I$ of all possible expressions is defined and explained with examples. In these examples, we refer to Instance Types from the previous section ($I1$, $I2$, and $I3$) and three new Instance Types $I4$, $I5$, and $I6$ from figures 4.7, 4.8, and 4.9. All six Instance Types are derived from process graph P, represented in figure 4.2. However, $I1$, $I2$, and $I3$ are a subgraph of the LIT from figure 4.3 and $I4$, $I5$, and $I6$ are a subgraph of some other LIT that can be derived from P.

FIRST X

$$\xi((FIRST\ X), tc, I) := \{n \in N_I | n.Label = X \wedge \nexists m(m \in N_I \wedge m.Label = X \wedge m < n)\}$$

Evaluation of the expression $FIRST\ X$ returns the first node n with label X that represents the first occurrence of activity X. n is the first node with label X if there is no node m with label X that is a predecessor of n. The Loop Counter Vector (LCV) of node n can contain scalar components that are greater than 1, since the first occurrence of a node does not necessarily appear in the first iteration of a loop (consider inner loops and XOR-paths that can be skipped). For an Instance Type where loops that are relevant for activity X are not entered, the result set is empty. This is the case for every expression.

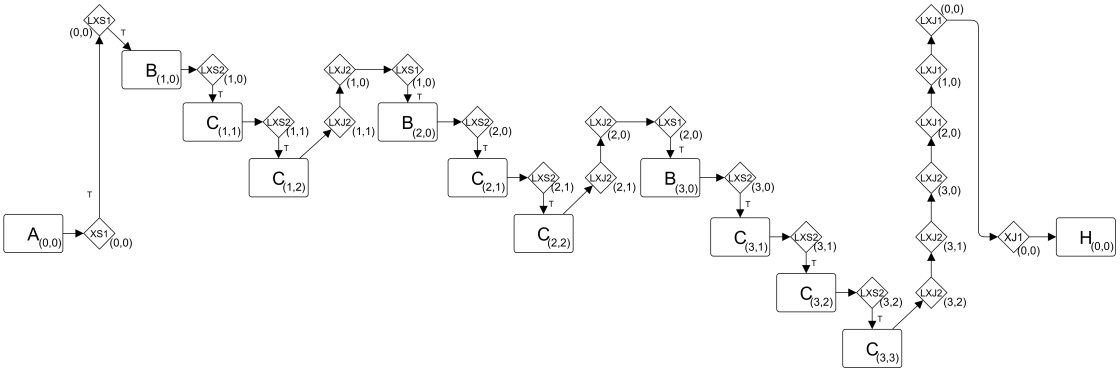


Figure 4.7: Example of a valid Instance Type $I4$

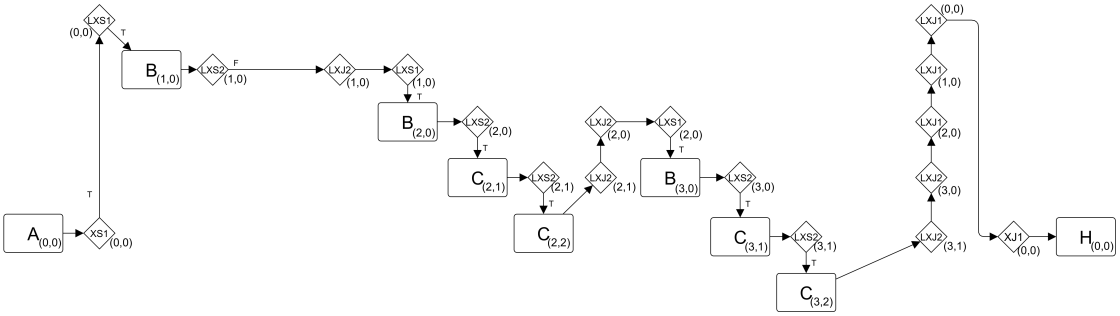


Figure 4.8: Example of a valid Instance Type $I5$

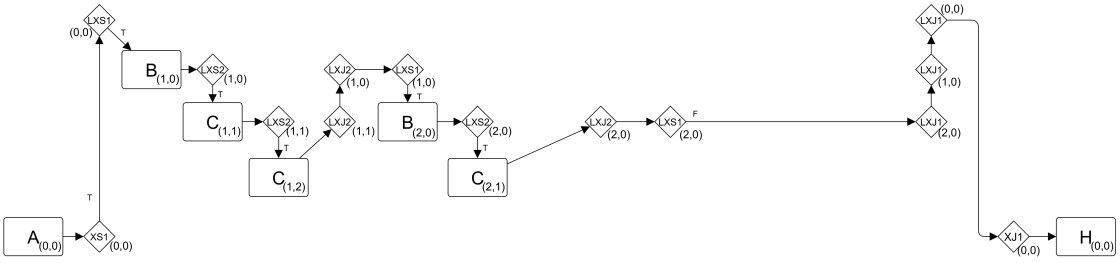


Figure 4.9: Example of a valid Instance Type $I6$

Examples:
ξ((FIRST B), (TC1, UBC, 5, FIRST B, H), I1) = {}
ξ((FIRST B), (TC1, UBC, 5, FIRST B, H), I2) = $\{B_{(1,0)}\}$
ξ((FIRST B), (TC1, UBC, 5, FIRST B, H), I3) = {}
ξ((FIRST B), (TC1, UBC, 5, FIRST B, H), I4) = $\{B_{(1,0)}\}$
ξ((FIRST B), (TC1, UBC, 5, FIRST B, H), I5) = $\{B_{(1,0)}\}$
ξ((FIRST B), (TC1, UBC, 5, FIRST B, H), I6) = $\{B_{(1,0)}\}$

ξ((FIRST C), (TC2, UBC, 5, FIRST C, H), I1) = {}
ξ((FIRST C), (TC2, UBC, 5, FIRST C, H), I2) = $\{C_{(1,1)}\}$
ξ((FIRST C), (TC2, UBC, 5, FIRST C, H), I3) = {}
ξ((FIRST C), (TC2, UBC, 5, FIRST C, H), I4) = $\{C_{(1,1)}\}$
ξ((FIRST C), (TC2, UBC, 5, FIRST C, H), I5) = $\{C_{(2,1)}\}$
ξ((FIRST C), (TC2, UBC, 5, FIRST C, H), I6) = $\{C_{(1,1)}\}$

LAST X

$$\xi((LAST\ X), tc, I) := \{n \in N_I | n.Label = X \wedge \nexists m(m \in N_I \wedge m.Label = X \wedge m > n)\}$$

Evaluation of the expression $LAST\ X$ returns the last node n with label X. n is the last node with label X if there is no node m with label X that is a successor of n.

Examples:
ξ((LAST B), (TC3, UBC, 5, LAST B, H), I1) = {}
ξ((LAST B), (TC3, UBC, 5, LAST B, H), I2) = $\{B_{(1,0)}\}$
ξ((LAST B), (TC3, UBC, 5, LAST B, H), I3) = {}
ξ((LAST B), (TC3, UBC, 5, LAST B, H), I4) = $\{B_{(3,0)}\}$
ξ((LAST B), (TC3, UBC, 5, LAST B, H), I5) = $\{B_{(3,0)}\}$
ξ((LAST B), (TC3, UBC, 5, LAST B, H), I6) = $\{B_{(2,0)}\}$

ξ((LAST C), (TC4, UBC, 5, LAST C, H), I1) = {}
ξ((LAST C), (TC4, UBC, 5, LAST C, H), I2) = $\{C_{(1,1)}\}$
ξ((LAST C), (TC4, UBC, 5, LAST C, H), I3) = {}
ξ((LAST C), (TC4, UBC, 5, LAST C, H), I4) = $\{C_{(3,3)}\}$
ξ((LAST C), (TC4, UBC, 5, LAST C, H), I5) = $\{C_{(3,2)}\}$
ξ((LAST C), (TC4, UBC, 5, LAST C, H), I6) = $\{C_{(2,1)}\}$

EACH X

$$\xi((EACH\ X), tc, I) := \{n \in N_I | n.Label = X\}$$

Evaluation of the expression $EACH\ X$ returns all nodes n with label X.

Examples:
ξ((EACH B), (TC5, UBC, 5, EACH B, H), I1) = {}
ξ((EACH B), (TC5, UBC, 5, EACH B, H), I2) = $\{B_{(1,0)}\}$
ξ((EACH B), (TC5, UBC, 5, EACH B, H), I3) = {}
ξ((EACH B), (TC5, UBC, 5, EACH B, H), I4) = $\{B_{(1,0)}, B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH B), (TC5, UBC, 5, EACH B, H), I5) = $\{B_{(1,0)}, B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH B), (TC5, UBC, 5, EACH B, H), I6) = $\{B_{(1,0)}, B_{(2,0)}\}$

ξ((EACH C), (TC6, UBC, 5, EACH C, H), I1) = {}
ξ((EACH C), (TC6, UBC, 5, EACH C, H), I2) = $\{C_{(1,1)}\}$
ξ((EACH C), (TC6, UBC, 5, EACH C, H), I3) = {}
ξ((EACH C), (TC6, UBC, 5, EACH C, H), I4) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH C), (TC6, UBC, 5, EACH C, H), I5) = $\{C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}\}$
ξ((EACH C), (TC6, UBC, 5, EACH C, H), I6) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}\}$

FIRST X WITHIN L

$$
\begin{aligned}
&\xi((FIRST\ X\ WITHIN\ L), tc, I) := \{n \in N_I | n.Label = X \wedge \\
&\quad (\exists l, k, l^P, k^P, n^P \nexists m (l, k, m \in N_I \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\
&\qquad \wedge l^P.Label = L \wedge l^P.Type = LS \wedge k^P.Type = LS \\
&\qquad \wedge equi(l^P, l, P, I) \wedge equi(k^P, k, P, I) \wedge equi(n^P, n, P, I) \\
&\qquad \wedge closestLoop(k^P, l^P, P) \wedge inLoop(n^P, l^P, P) \\
&\quad \wedge m.Label = X \wedge k < m < n) \\
&\quad \vee \\
&\quad \exists l, l^P, n^P \nexists m, k^P (l, m \in N_I \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\
&\qquad \wedge l^P.Label = L \wedge l^P.Type = LS \wedge k^P.Type = LS \\
&\qquad \wedge equi(l^P, l, P, I) \wedge equi(n^P, n, P, I) \\
&\qquad \wedge closestLoop(k^P, l^P, P) \wedge inLoop(n^P, l^P, P) \\
&\quad \wedge m.Label = X \wedge m < n))\}
\end{aligned}
$$

Evaluation of the expression $FIRST\ X\ WITHIN\ L$ returns the first node n with label X from a series of iterations of the loop L. n is such a node if there is an LXS-node of the nearest outer loop relative to L before the node n and there is no other node with the label X between them. An LXS-node of the nearest outer loop relative to L indicates a series of iterations of the inner loop L. An inner loop can have multiple series of iterations (one for each new iteration of an outer loop) and therefore the result set can have more than one elements. If there is no other loop that is surrounding loop L, the expression $FIRST\ X\ WITHIN\ L$ is equivalent to $FIRST\ X$.

Examples:
ξ((FIRST B WITHIN LS1), (TC7, UBC, 5, FIRST B WITHIN LS1, H), I4) = $\{B_{(1,0)}\}$
ξ((FIRST B WITHIN LS1), (TC7, UBC, 5, FIRST B WITHIN LS1, H), I5) = $\{B_{(1,0)}\}$
ξ((FIRST B WITHIN LS1), (TC7, UBC, 5, FIRST B WITHIN LS1, H), I6) = $\{B_{(1,0)}\}$

ξ((FIRST C WITHIN LS1), (TC8, UBC, 5, FIRST C WITHIN LS1, H), I4) = $\{C_{(1,1)}\}$
ξ((FIRST C WITHIN LS1), (TC8, UBC, 5, FIRST C WITHIN LS1, H), I5) = $\{C_{(2,1)}\}$
ξ((FIRST C WITHIN LS1), (TC8, UBC, 5, FIRST C WITHIN LS1, H), I6) = $\{C_{(1,1)}\}$

ξ((FIRST C WITHIN LS2), (TC9, UBC, 5, FIRST C WITHIN LS2, H), I4) = $\{C_{(1,1)}, C_{(2,1)}, C_{(3,1)}\}$
ξ((FIRST C WITHIN LS2), (TC9, UBC, 5, FIRST C WITHIN LS2, H), I5) = $\{C_{(2,1)}, C_{(3,1)}\}$
ξ((FIRST C WITHIN LS2), (TC9, UBC, 5, FIRST C WITHIN LS2, H), I6) = $\{C_{(1,1)}, C_{(2,1)}\}$

LAST X WITHIN L

$$\begin{aligned}\xi((LAST\ X\ WITHIN\ L), tc, I) := \{n \in N_I | n.Label = X \wedge \\ \exists j, s, j^P, s^P, n^P \nexists m (j, s, m \in N_I \wedge j^P, s^P, n^P \in N_P \wedge instOf(I, P) \\ \wedge s^P.Label = L \wedge s^P.Type = LS \wedge j^P.Type = LJ \\ \wedge equi(s^P, s, P, I) \wedge equi(j^P, j, P, I) \wedge equi(n^P, n, P, I) \\ \wedge counterpart(s^P, j^P, P) \wedge inLoop(n^P, s^P, P) \\ \wedge m.Label = X \wedge n < m < j)\}\end{aligned}$$

Evaluation of the expression $LAST\ X\ WITHIN\ L$ returns the last node n with label X from a series of iterations of the loop L. n is such a node if there is an LXJ-node of loop L after the node n and there is no other node with the label X between them. An LXJ-node of the loop L indicates the end of iteration series of loop L.

Examples:
ξ((LAST B WITHIN LS1), (TC10, UBC, 5, LAST B WITHIN LS1, H), I4) = $\{B_{(3,0)}\}$
ξ((LAST B WITHIN LS1), (TC10, UBC, 5, LAST B WITHIN LS1, H), I5) = $\{B_{(3,0)}\}$
ξ((LAST B WITHIN LS1), (TC10, UBC, 5, LAST B WITHIN LS1, H), I6) = $\{B_{(2,0)}\}$

ξ((LAST C WITHIN LS1), (TC11, UBC, 5, LAST C WITHIN LS1, H), I4) = $\{C_{(3,3)}\}$
ξ((LAST C WITHIN LS1), (TC11, UBC, 5, LAST C WITHIN LS1, H), I5) = $\{C_{(3,2)}\}$
ξ((LAST C WITHIN LS1), (TC11, UBC, 5, LAST C WITHIN LS1, H), I6) = $\{C_{(2,1)}\}$

ξ((LAST C WITHIN LS2), (TC12, UBC, 5, LAST C WITHIN LS2, H), I4) = $\{C_{(1,2)}, C_{(2,2)}, C_{(3,3)}\}$
ξ((LAST C WITHIN LS2), (TC12, UBC, 5, LAST C WITHIN LS2, H), I5) = $\{C_{(2,2)}, C_{(3,2)}\}$
ξ((LAST C WITHIN LS2), (TC12, UBC, 5, LAST C WITHIN LS2, H), I6) = $\{C_{(1,2)}, C_{(2,1)}\}$

EACH X WITHIN L

$$\xi((EACH\ X\ WITHIN\ L), tc, I) := \{n \in N_I | n.Label = X\}$$

The expression $EACH\ X\ WITHIN\ L$ is equivalent to expression $EACH\ X$. Its evaluation returns all nodes n with label X.

Examples:
ξ((EACH B WITHIN LS1), (TC13, UBC, 5, EACH B WITHIN LS1, H), I4) = $\{B_{(1,0)}, B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH B WITHIN LS1), (TC13, UBC, 5, EACH B WITHIN LS1, H), I5) = $\{B_{(1,0)}, B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH B WITHIN LS1), (TC13, UBC, 5, EACH B WITHIN LS1, H), I6) = $\{B_{(1,0)}, B_{(2,0)}\}$

ξ((EACH C WITHIN LS1), (TC14, UBC, 5, EACH C WITHIN LS1, H), I4) =
$\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH C WITHIN LS1), (TC14, UBC, 5, EACH C WITHIN LS1, H), I5) = $\{C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}\}$
ξ((EACH C WITHIN LS1), (TC14, UBC, 5, EACH C WITHIN LS1, H), I6) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}\}$

ξ((EACH C WITHIN LS2), (TC15, UBC, 5, EACH C WITHIN LS2, H), I4) =
$\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH C WITHIN LS2), (TC15, UBC, 5, EACH C WITHIN LS2, H), I5) = $\{C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}\}$
ξ((EACH C WITHIN LS2), (TC15, UBC, 5, EACH C WITHIN LS2, H), I6) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}\}$

All expressions that were described above are absolute and can be used to specify a *source* or a *destination* of an Extended Time Constraint. The following expressions are relative to a given set of *source* nodes and can only be used to specify a *destination* of an Extended Time Constraint.

FIRST RELATIVE X

$$\xi((FIRST\ RELATIVE\ X), tc, I) := \{n \in N_I | n.Label = X \wedge \\ \exists s \nexists m (s, m \in N_I \wedge s \in \xi(tc.source, tc, I) \wedge m.Label = X \wedge s < m < n)\}$$

Evaluation of the (*destination*) expression $FIRST\ RELATIVE\ X$ returns the first node n with label X that appears after a source node s.

Examples:
ξ((FIRST RELATIVE B), (TC16, UBC, 5, A, FIRST RELATIVE B), I4) = $\{B_{(1,0)}\}$
ξ((FIRST RELATIVE C), (TC17, UBC, 5, A, FIRST RELATIVE C), I4) = $\{C_{(1,1)}\}$

ξ((FIRST RELATIVE B), (TC18, UBC, 5, FIRST B, FIRST RELATIVE B), I4) = $\{B_{(2,0)}\}$
ξ((FIRST RELATIVE C), (TC19, UBC, 5, FIRST B, FIRST RELATIVE C), I4) = $\{C_{(1,1)}\}$

ξ((FIRST RELATIVE B), (TC20, UBC, 5, LAST B, FIRST RELATIVE B), I4) = {}
ξ((FIRST RELATIVE C), (TC21, UBC, 5, LAST B, FIRST RELATIVE C), I4) = $\{C_{(3,1)}\}$

ξ((FIRST RELATIVE B), (TC22, UBC, 5, EACH B, FIRST RELATIVE B), I4) = $\{B_{(2,0)}, B_{(3,0)}\}$
ξ((FIRST RELATIVE C), (TC23, UBC, 5, EACH B, FIRST RELATIVE C), I4) = $\{C_{(1,1)}, C_{(2,1)}, C_{(3,1)}\}$

LAST RELATIVE X

$$\xi((LAST\ RELATIVE\ X), tc, I) := \{n \in N_I | n.Label = X \wedge \exists s \nexists m (s, m \in N_I \wedge s \in \xi(tc.source, tc, I) \wedge m.Label = X \wedge s < n < m)\}$$

Evaluation of the (*destination*) expression $LAST\ RELATIVE\ X$ returns the last node n with label X that appears after a source node s.

Examples:
ξ((LAST RELATIVE B), (TC24, UBC, 5, A, LAST RELATIVE B), I4) = $\{B_{(3,0)}\}$
ξ((LAST RELATIVE C), (TC25, UBC, 5, A, LAST RELATIVE C), I4) = $\{C_{(3,3)}\}$

ξ((LAST RELATIVE B), (TC26, UBC, 5, FIRST B, LAST RELATIVE B), I4) = $\{B_{(3,0)}\}$
ξ((LAST RELATIVE C), (TC27, UBC, 5, FIRST B, LAST RELATIVE C), I4) = $\{C_{(3,3)}\}$

ξ((LAST RELATIVE B), (TC28, UBC, 5, LAST B, LAST RELATIVE B), I4) = {}
ξ((LAST RELATIVE C), (TC29, UBC, 5, LAST B, LAST RELATIVE C), I4) = $\{C_{(3,3)}\}$

ξ((LAST RELATIVE B), (TC30, UBC, 5, EACH B, LAST RELATIVE B), I4) = $\{B_{(3,0)}\}$
ξ((LAST RELATIVE C), (TC31, UBC, 5, EACH B, LAST RELATIVE C), I4) = $\{C_{(3,3)}\}$

EACH RELATIVE X

$$\xi((EACH\ RELATIVE\ X), tc, I) := \{n \in N_I | n.Label = X \wedge \exists s (s \in \xi(tc.source, tc, I) \wedge s < n)\}$$

Evaluation of the (*destination*) expression $EACH\ RELATIVE\ X$ returns each node n with label X that appears after a *source* node s.

Examples:
ξ((EACH RELATIVE B), (TC32, UBC, 5, A, EACH RELATIVE B), I4) = $\{B_{(1,0)}, B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH RELATIVE C), (TC33, UBC, 5, A, EACH RELATIVE C), I4) =
$\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE B), (TC34, UBC, 5, FIRST B, EACH RELATIVE B), I4) = $\{B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH RELATIVE C), (TC35, UBC, 5, FIRST B, EACH RELATIVE C), I4) =
$\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE B), (TC36, UBC, 5, LAST B, EACH RELATIVE B), I4) = {}
ξ((EACH RELATIVE C), (TC37, UBC, 5, LAST B, EACH RELATIVE C), I4) = $\{C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE B), (TC38, UBC, 5, EACH B, EACH RELATIVE B), I4) = $\{B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH RELATIVE C), (TC39, UBC, 5, EACH B, EACH RELATIVE C), I4) =
$\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

FIRST RELATIVE X WITHIN L

$$\xi((FIRST\ RELATIVE\ X\ WITHIN\ L), tc, I) := \{n \in \xi((FIRST\ X\ WITHIN\ L), tc, I) \mid \exists s(s \in \xi(tc.source, tc, I) \wedge s < n)\}$$

Evaluation of the (*destination*) expression $FIRST\ RELATIVE\ X\ WITHIN\ L$ returns all nodes n that are returned by the evaluation of the expression $FIRST\ X\ WITHIN\ L$ and that appear after a *source* node s.

Examples:
ξ((FIRST RELATIVE B WITHIN LS1), (TC40, UBC, 5, A, FIRST RELATIVE B WITHIN LS1), I4) =
$\{B_{(1,0)}\}$
ξ((FIRST RELATIVE C WITHIN LS1), (TC41, UBC, 5, A, FIRST RELATIVE C WITHIN LS1), I4) =
$\{C_{(1,1)}\}$
ξ((FIRST RELATIVE C WITHIN LS2), (TC42, UBC, 5, A, FIRST RELATIVE C WITHIN LS2), I4) =
$\{C_{(1,1)}, C_{(2,1)}, C_{(3,1)}\}$

ξ((FIRST RELATIVE B WITHIN LS1), (TC43, UBC, 5, FIRST B, FIRST RELATIVE B WITHIN LS1), I4) =
$\{B_{(2,0)}\}$
ξ((FIRST RELATIVE C WITHIN LS1), (TC44, UBC, 5, FIRST B, FIRST RELATIVE C WITHIN LS1), I4) =
$\{C_{(1,1)}\}$
ξ((FIRST RELATIVE C WITHIN LS2), (TC45, UBC, 5, FIRST B, FIRST RELATIVE C WITHIN LS2), I4) =
$\{C_{(1,1)}, C_{(2,1)}, C_{(3,1)}\}$

ξ((FIRST RELATIVE B WITHIN LS1), (TC46, UBC, 5, LAST B, FIRST RELATIVE B WITHIN LS1), I4) =
{}
ξ((FIRST RELATIVE C WITHIN LS1), (TC47, UBC, 5, LAST B, FIRST RELATIVE C WITHIN LS1), I4) =
$\{C_{(3,1)}\}$
ξ((FIRST RELATIVE C WITHIN LS2), (TC48, UBC, 5, LAST B, FIRST RELATIVE C WITHIN LS2), I4) =
$\{C_{(3,1)}\}$

ξ((FIRST RELATIVE B WITHIN LS1), (TC49, UBC, 5, EACH B, FIRST RELATIVE B WITHIN LS1), I4) =
$\{B_{(2,0)}, B_{(3,0)}\}$
ξ((FIRST RELATIVE C WITHIN LS1), (TC50, UBC, 5, EACH B, FIRST RELATIVE C WITHIN LS1), I4) =
$\{C_{(1,1)}, C_{(2,1)}, C_{(3,1)}\}$
ξ((FIRST RELATIVE C WITHIN LS2), (TC51, UBC, 5, EACH B, FIRST RELATIVE C WITHIN LS2), I4) =
$\{C_{(1,1)}, C_{(2,1)}, C_{(3,1)}\}$

LAST RELATIVE X WITHIN L

$$\xi((LAST\ RELATIVE\ X\ WITHIN\ L), tc, I) := \{n \in \xi((LAST\ X\ WITHIN\ L), tc, I) \mid \exists s(s \in \xi(tc.source, tc, I) \wedge s < n)\}$$

Evaluation of the (*destination*) expression $LAST\ RELATIVE\ X\ WITHIN\ L$ returns all nodes n that are returned by the evaluation of the expression $LAST\ X\ WITHIN\ L$ and that appear after a *source* node s.

Examples:
ξ((LAST RELATIVE B WITHIN LS1), (TC52, UBC, 5, A, LAST RELATIVE B WITHIN LS1), I4) =
$\{B_{(3,0)}\}$
ξ((LAST RELATIVE C WITHIN LS1), (TC53, UBC, 5, A, LAST RELATIVE C WITHIN LS1), I4) =
$\{C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS2), (TC54, UBC, 5, A, LAST RELATIVE C WITHIN LS2), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,3)}\}$

ξ((LAST RELATIVE B WITHIN LS1), (TC55, UBC, 5, FIRST B, LAST RELATIVE B WITHIN LS1), I4) =
$\{B_{(3,0)}\}$
ξ((LAST RELATIVE C WITHIN LS1), (TC56, UBC, 5, FIRST B, LAST RELATIVE C WITHIN LS1), I4) =
$\{C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS2), (TC57, UBC, 5, FIRST B, LAST RELATIVE C WITHIN LS2), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,3)}\}$

ξ((LAST RELATIVE B WITHIN LS1), (TC58, UBC, 5, LAST B, LAST RELATIVE B WITHIN LS1), I4) =
{}
ξ((LAST RELATIVE C WITHIN LS1), (TC59, UBC, 5, LAST B, LAST RELATIVE C WITHIN LS1), I4) =
$\{C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS2), (TC60, UBC, 5, LAST B, LAST RELATIVE C WITHIN LS2), I4) =
$\{C_{(3,3)}\}$

ξ((LAST RELATIVE B WITHIN LS1), (TC61, UBC, 5, EACH B, LAST RELATIVE B WITHIN LS1), I4) = $\{B_{(3,0)}\}$
ξ((LAST RELATIVE C WITHIN LS1), (TC62, UBC, 5, EACH B, LAST RELATIVE C WITHIN LS1), I4) = $\{C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS2), (TC63, UBC, 5, EACH B, LAST RELATIVE C WITHIN LS2), I4) = $\{C_{(1,2)}, C_{(2,2)}, C_{(3,3)}\}$

EACH RELATIVE X WITHIN L

$$\xi((EACH\ RELATIVE\ X\ WITHIN\ L), tc, I) := \{n \in N_I | n.Label = X \wedge \exists s(s \in \xi(tc.source, tc, I) \wedge s < n)\}$$

The expression $EACH\ RELATIVE\ X\ WITHIN\ L$ is equivalent to the expression $EACH\ RELATIVE\ X$. Its evaluation returns all nodes n with label X that are successors of a *source* node s.

Examples:
ξ((EACH RELATIVE B WITHIN LS1), (TC64, UBC, 5, A, EACH RELATIVE B WITHIN LS1), I4) = $\{B_{(1,0)}, B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH RELATIVE C WITHIN LS1), (TC65, UBC, 5, A, EACH RELATIVE C WITHIN LS1), I4) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH RELATIVE C WITHIN LS2), (TC66, UBC, 5, A, EACH RELATIVE C WITHIN LS2), I4) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE B WITHIN LS1), (TC67, UBC, 5, FIRST B, EACH RELATIVE B WITHIN LS1), I4) = $\{B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH RELATIVE C WITHIN LS1), (TC68, UBC, 5, FIRST B, EACH RELATIVE C WITHIN LS1), I4) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH RELATIVE C WITHIN LS2), (TC69, UBC, 5, FIRST B, EACH RELATIVE C WITHIN LS2), I4) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE B WITHIN LS1), (TC70, UBC, 5, LAST B, EACH RELATIVE B WITHIN LS1), I4) = {}
ξ((EACH RELATIVE C WITHIN LS1), (TC71, UBC, 5, LAST B, EACH RELATIVE C WITHIN LS1), I4) = $\{C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH RELATIVE C WITHIN LS2), (TC72, UBC, 5, LAST B, EACH RELATIVE C WITHIN LS2), I4) = $\{C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE B WITHIN LS1), (TC73, UBC, 5, EACH B, EACH RELATIVE B WITHIN LS1), I4) = $\{B_{(2,0)}, B_{(3,0)}\}$
ξ((EACH RELATIVE C WITHIN LS1), (TC74, UBC, 5, EACH B, EACH RELATIVE C WITHIN LS1), I4) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH RELATIVE C WITHIN LS2), (TC75, UBC, 5, EACH B, EACH RELATIVE C WITHIN LS2), I4) = $\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

Relative expressions $FIRST\ RELATIVE\ X\ WITHIN\ L$, $LAST\ RELATIVE\ X\ WITHIN\ L$, and $EACH\ RELATIVE\ X\ WITHIN\ L$ can be extended with an iteration reference $SAME\_ITERATION$ or $NEXT\_ITERATION$. They are specified relative to the related set of the source nodes $\xi(tc.source, tc, I)$. Their semantics are described on the following pages.

FIRST RELATIVE X WITHIN L SAME_ITERATION K

$$\begin{aligned}
&\xi((FIRST\ RELATIVE\ X\ WITHIN\ L\ SAME_ITERATION\ K), tc, I) := \\
&\quad \{n \in \xi((FIRST\ RELATIVE\ X\ WITHIN\ L), tc, I) | \\
&\qquad \exists s, k^P, n^P \forall l^P (s \in \xi(tc.source, tc, I) \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\
&\qquad \wedge\ l^P.Type = LS \wedge k^P.Label = K \wedge k^P.Type = LS \\
&\qquad \wedge\ equi(n^P, n, P, I) \wedge inLoop(n^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
&\qquad\quad lc(n, l^P.Label, I) = lc(s, l^P.Label, I) \wedge \\
&\qquad\quad lc(n, k^P.Label, I) = lc(s, k^P.Label, I))\}
\end{aligned}$$

Evaluation of the (*destination*) expression $FIRST\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$ returns all nodes n that are returned by the evaluation of the expression $FIRST\ RELATIVE\ X\ WITHIN\ L$ and are in the same iteration of the loop with the label K as a *source* node s. Node n and node s are in the same iteration of the loop with label K if they have the same loop counter for the loop with label K and for all other loops in which the loop with label K is nested. The loop counters for all loops that may be nested into the loop with label K are irrelevant and may be unequal.

Examples:

ξ((FIRST RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC76, UBC, 5, FIRST B, FIRST RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}

ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC77, UBC, 5, FIRST B, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,1)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC78, UBC, 5, FIRST C, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC79, UBC, 5, FIRST C, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

ξ((FIRST RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC80, UBC, 5, LAST B, FIRST RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC81, UBC, 5, LAST B, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(3,1)}\}$
ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC82, UBC, 5, LAST C, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC83, UBC, 5, LAST C, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

ξ((FIRST RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC84, UBC, 5, EACH B, FIRST RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC85, UBC, 5, EACH B, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,1)}, C_{(2,1)}, C_{(3,1)}\}$
ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC86, UBC, 5, EACH C, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,2)}\}$
ξ((FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC87, UBC, 5, EACH C, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

LAST RELATIVE X WITHIN L SAME_ITERATION K

$$\begin{aligned}
&\xi((LAST\ RELATIVE\ X\ WITHIN\ L\ SAME_ITERATION\ K), tc, I) := \\
&\quad \{n \in \xi((LAST\ RELATIVE\ X\ WITHIN\ L), tc, I) | \\
&\qquad \exists s, k^P, n^P \forall l^P (s \in \xi(tc.source, tc, I) \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\
&\qquad \wedge l^P.Type = LS \wedge k^P.Label = K \wedge k^P.Type = LS \\
&\qquad \wedge equi(n^P, n, P, I) \wedge inLoop(n^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
&\qquad\quad lc(n, l^P.Label, I) = lc(s, l^P.Label, I) \wedge \\
&\qquad\quad lc(n, k^P.Label, I) = lc(s, k^P.Label, I))\}
\end{aligned}$$

Evaluation of the (*destination*) expression $LAST\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$ returns all nodes n that are returned by the evaluation of the expression $LAST\ RELATIVE\ X\ WITHIN\ L$ and are in the same iteration of the loop with the label K as a *source* node s. Node n and node s are in the same iteration of the loop with label K if they have the same loop counter for the loop with label K and for all other loops in which the loop with label K is nested.

The loop counters for all loops that may be nested into the loop with label K, are irrelevant and may be unequal.

Examples:

ξ((LAST RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC88, UBC, 5, FIRST B, LAST RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC89, UBC, 5, FIRST B, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}\}$
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC90, UBC, 5, FIRST C, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}\}$
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC91, UBC, 5, FIRST C, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

ξ((LAST RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC92, UBC, 5, LAST B, LAST RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC93, UBC, 5, LAST B, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC94, UBC, 5, LAST C, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC95, UBC, 5, LAST C, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

ξ((LAST RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC96, UBC, 5, EACH B, LAST RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC97, UBC, 5, EACH B, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC98, UBC, 5, EACH C, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC99, UBC, 5, EACH C, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

EACH RELATIVE X WITHIN L SAME_ITERATION K

$$\begin{aligned}
&\xi((EACH\ RELATIVE\ X\ WITHIN\ L\ SAME_ITERATION\ K), tc, I) := \\
&\quad \{n \in \xi((EACH\ RELATIVE\ X\ WITHIN\ L), tc, I) | \\
&\qquad \exists s, k^P, n^P \forall l^P (s \in \xi(tc.source, tc, I) \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\
&\qquad \wedge l^P.Type = LS \wedge k^P.Label = K \wedge k^P.Type = LS \\
&\qquad \wedge equi(n^P, n, P, I) \wedge inLoop(n^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
&\qquad\quad lc(n, l^P.Label, I) = lc(s, l^P.Label, I) \wedge \\
&\qquad\quad lc(n, k^P.Label, I) = lc(s, k^P.Label, I))\}
\end{aligned}$$

Evaluation of the (*destination*) expression $EACH\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$ returns all nodes n that are returned by the evaluation of the expression $EACH\ RELATIVE\ X\ WITHIN\ L$ and are in the same iteration of the loop with the label K as a *source* node s. Node n and node s are in the same iteration of the loop with label K if they have the same loop counter for the loop with label K and for all other loops in which the loop with label K is nested. The loop counters for all loops that may be nested into the loop with label K are irrelevant and may be unequal.

Examples:

ξ((EACH RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC100, UBC, 5, FIRST B, EACH RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}

ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC101, UBC, 5, FIRST B, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,1)}, C_{(1,2)}\}$

ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC102, UBC, 5, FIRST C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}\}$

ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC103, UBC, 5, FIRST C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

ξ((EACH RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC104, UBC, 5, LAST B, EACH RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}

ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC105, UBC, 5, LAST B, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC106, UBC, 5, LAST C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{}

ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC107, UBC, 5, LAST C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

ξ((EACH RELATIVE B WITHIN LS1 SAME_ITERATION LS1),
(TC108, UBC, 5, EACH B, EACH RELATIVE B WITHIN LS1 SAME_ITERATION LS1), I4) =
{}
ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC109, UBC, 5, EACH B, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,1)}, C_{(1,2)}, C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1),
(TC110, UBC, 5, EACH C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,2)}, C_{(3,3)}\}$
ξ((EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS2),
(TC111, UBC, 5, EACH C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS2), I4) =
{}

FIRST RELATIVE X WITHIN L NEXT_ITERATION K

$$\begin{aligned}\xi((&FIRST\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K), tc, I) := \\ &\{n \in \xi((FIRST\ RELATIVE\ X\ WITHIN\ L), tc, I) | \\ &\quad \exists s, k^P, n^P \forall l^P (s \in \xi(tc.source, tc, I) \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\ &\quad \wedge l^P.Type = LS \wedge k^P.Label = K \wedge k^P.Type = LS \\ &\quad \wedge equi(n^P, n, P, I) \wedge inLoop(n^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\ &\qquad lc(n, l^P.Label, I) = lc(s, l^P.Label, I) \wedge \\ &\qquad lc(n, k^P.Label, I) = lc(s, k^P.Label, I) + 1)\}\end{aligned}$$

Evaluation of the (*destination*) expression $FIRST\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$ returns all nodes n that are returned by the evaluation of the expression $FIRST\ RELATIVE\ X\ WITHIN\ L$ and are in the succeeding iteration of the loop with the label K as a *source* node s. Node n is in the succeeding iteration of the loop with label K in respect to node s if i) the loop counter of node n for the loop with label K (c_K^n) is greater by 1 than the loop counter of node s for the loop with label K (c_K^s) and ii) the loop counters for all other loops in which the loop with label K is nested are the same. The loop counters for all loops that may be nested into the loop with label K are irrelevant and may be unequal.

Examples:

ξ((FIRST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC112, UBC, 5, FIRST B, FIRST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{B_{(2,0)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC113, UBC, 5, FIRST B, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC114, UBC, 5, FIRST C, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC115, UBC, 5, FIRST C, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
$\{C_{(1,2)}\}$

ξ((FIRST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC116, UBC, 5, LAST B, FIRST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) =
{}

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC117, UBC, 5, LAST B, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
{}

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC118, UBC, 5, LAST C, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
{}

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC119, UBC, 5, LAST C, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
{}

ξ((FIRST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC120, UBC, 5, EACH B, FIRST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{B_{(2,0)}, B_{(3,0)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC121, UBC, 5, EACH B, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}, C_{(3,1)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC122, UBC, 5, EACH C, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}, C_{(3,1)}\}$

ξ((FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC123, UBC, 5, EACH C, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,2)}, C_{(3,3)}\}$

LAST RELATIVE X WITHIN L NEXT_ITERATION K

$$\begin{aligned}
&\xi((LAST\ RELATIVE\ X\ WITHIN\ L\ NEXT_ITERATION\ K), tc, I) := \\
&\quad \{n \in \xi((LAST\ RELATIVE\ X\ WITHIN\ L), tc, I) | \\
&\qquad \exists s, k^P, n^P \forall l^P (s \in \xi(tc.source, tc, I) \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\
&\qquad \wedge\ l^P.Type = LS \wedge k^P.Label = K \wedge k^P.Type = LS \\
&\qquad \wedge\ equi(n^P, n, P, I) \wedge inLoop(n^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
&\qquad\quad lc(n, l^P.Label, I) = lc(s, l^P.Label, I) \wedge \\
&\qquad\quad lc(n, k^P.Label, I) = lc(s, k^P.Label, I) + 1)\}
\end{aligned}$$

Evaluation of the (*destination*) expression $LAST\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$ returns all nodes n that are returned by the evaluation of the expression $LAST\ RELATIVE\ X\ WITHIN\ L$ and are in the succeeding iteration of the loop with the label K as a *source* node s. Node n is in the succeeding iteration of the loop with label K in respect to node s if i) the loop counter of node n for the loop with label K (c_K^n) is greater by 1 than the loop counter of node s for the loop with label K (c_K^s) and ii) the loop counters for all other loops in which the loop with label K is nested are the same. The loop counters for all loops that may be nested into the loop with label K are irrelevant and may be unequal.

Examples:

ξ((LAST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC124, UBC, 5, FIRST B, LAST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) = $\{B_{(2,0)}\}$

ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC125, UBC, 5, FIRST B, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) = $\{C_{(2,2)}\}$

ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC126, UBC, 5, FIRST C, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) = $\{C_{(2,2)}\}$

ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC127, UBC, 5, FIRST C, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) = $\{C_{(1,2)}\}$

ξ((LAST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC128, UBC, 5, LAST B, LAST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) = {}

ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC129, UBC, 5, LAST B, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) = {}

ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC130, UBC, 5, LAST C, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
{}
ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC131, UBC, 5, LAST C, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
{}

ξ((LAST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC132, UBC, 5, EACH B, LAST RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{B_{(2,0)}, B_{(3,0)}\}$
ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC133, UBC, 5, EACH B, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,2)}, C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC134, UBC, 5, EACH C, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,2)}, C_{(3,3)}\}$
ξ((LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC135, UBC, 5, EACH C, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,2)}, C_{(3,3)}\}$

EACH RELATIVE X WITHIN L NEXT_ITERATION K

$$\begin{aligned}
&\xi((EACH\ RELATIVE\ X\ WITHIN\ L\ NEXT_ITERATION\ K), tc, I) := \\
&\quad \{n \in \xi((EACH\ RELATIVE\ X\ WITHIN\ L), tc, I) | \\
&\qquad \exists s, k^P, n^P \forall l^P (s \in \xi(tc.source, tc, I) \wedge l^P, k^P, n^P \in N_P \wedge instOf(I, P) \\
&\qquad \wedge\, l^P.Type = LS \wedge k^P.Label = K \wedge k^P.Type = LS \\
&\qquad \wedge\, equi(n^P, n, P, I) \wedge inLoop(n^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
&\qquad\quad lc(n, l^P.Label, I) = lc(s, l^P.Label, I) \wedge \\
&\qquad\quad lc(n, k^P.Label, I) = lc(s, k^P.Label, I) + 1)\}
\end{aligned}$$

Evaluation of the (*destination*) expression $EACH\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$ returns all nodes n that are returned by the evaluation of the expression $EACH\ RELATIVE\ X\ WITHIN\ L$ and are in the succeeding iteration of the loop with the label K as a *source* node s. Node n is in the succeeding iteration of the loop with label K in respect to node s if i) the loop counter of node n for the loop with label K (c^n_K) is greater by 1 than the loop counter of node s for the loop with label K (c^s_K) and ii) the loop counters for all other loops in which the loop with label K is nested are the same. The loop counters for all loops that may be nested into the loop with label K are irrelevant and may be unequal.

Examples:

ξ((EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC136, UBC, 5, FIRST B, EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{B_{(2,0)}\}$

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC137, UBC, 5, FIRST B, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}, C_{(2,2)}\}$

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC138, UBC, 5, FIRST C, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}, C_{(2,2)}\}$

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC139, UBC, 5, FIRST C, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
$\{C_{(1,2)}\}$

ξ((EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC140, UBC, 5, LAST B, EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) =
{}

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC141, UBC, 5, LAST B, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
{}

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC142, UBC, 5, LAST C, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
{}

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC143, UBC, 5, LAST C, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
{}

ξ((EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1),
(TC144, UBC, 5, EACH B, EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{B_{(2,0)}, B_{(3,0)}\}$

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC145, UBC, 5, EACH B, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1),
(TC146, UBC, 5, EACH C, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
$\{C_{(2,1)}, C_{(2,2)}, C_{(3,1)}, C_{(3,2)}, C_{(3,3)}\}$

ξ((EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS2),
(TC147, UBC, 5, EACH C, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
$\{C_{(1,2)}, C_{(2,2)}, C_{(3,2)}, C_{(3,3)}\}$

In this section, we defined the semantics of *source* and *destination* expressions and which node result sets are returned by evaluation of an expression. In time management, however, we want to be able to define time constraints between two single nodes. We introduce Atomic Time Constraints that constrain two particular nodes in an Instance Type to a given allowed min/max temporal distance. For each Extended Time Constraint, there is a set of derived Atomic Time Constraints. We define and describe Atomic Time Constraints in the next section.

4.3 Atomic Time Constraints

The source and destination specifications in an Extended Time Constraint, and the resulting node sets form the basis for so called Atomic Time Constraints that are relations between the source and destination result sets $\xi(tc.source, tc, I)$ and $\xi(tc.destination, tc, I)$. In an Atomic Time Constraint, the source and the destination are each a particular node in an Instance Type I. The source and the destination specification in an Atomic Time Constraint therefore contains only a *nodeLabel* and an *LCV*. We define an Atomic Time Constraint as follows:

Definition 4.6. ***(Atomic Time Constraint (ATC))***
An Atomic Time Constraint $atc \in ATC_I$ in an Instance Type I is a quintuple (ID, type, δ, source, destination) that constrains the temporal relation between the defined source node and destination node to a maximum of δ time units if type is UBC, or to a minimum of δ time units if type is LBC.

The source node and the destination node in an Atomic Time Constraint are specified by the nodeLabel and the LCV.

Each source-destination pair of nodes in an Atomic Time Constraint $atc \in ATC_I$ that is derived from an Extended Time Constraint $tc \in TC_P$, is an element of a subset of the cartesian product of $\xi(tc.source, tc, I)$ and $\xi(tc.destination, tc, I)$: $ATC_I^{tc} \subseteq \xi(tc.source, tc, I) \times \xi(tc.destination, tc, I)$. Other attributes (ID, type, and δ) of a derived Atomic Time Constraint are the same as in the corresponding Extended Time Constraint tc. The function $atomize(tc, I)$ creates a set of Atomic Time Constraints for the given Extended Time Constraint tc in the given Instance Type I and is defined as follows:

$$
atomize(tc, I) : \begin{cases}
case\,1 : tc.destination.relation = ABSOLUTE^a \\
\quad (tc, I) \mapsto \{(tc.ID,\ tc.type,\ tc.\delta,\ s,\ d) \mid s \in \xi(tc.source, tc, I) \\
\quad \wedge\, d \in \xi(tc.destination, tc, I) \wedge s \neq d\} \\
\\
case\,2 : tc.destination.relation = RELATIVE \\
\wedge tc.destination.quantifier \neq FIRST \\
\wedge tc.destination.iterationRef.iteration = \text{""} \\
\quad (tc, I) \mapsto \{(tc.ID,\ tc.type,\ tc.\delta,\ s,\ d) \mid s \in \xi(tc.source, tc, I) \\
\quad \wedge\, d \in \xi(tc.destination, tc, I) \wedge (s < d)\} \\
\\
case\,3 : tc.destination.relation = RELATIVE \\
\wedge tc.destination.quantifier = FIRST \\
\wedge tc.destination.iterationRef.iteration = \text{""} \\
\quad (tc, I) \mapsto \{(tc.ID,\ tc.type,\ tc.\delta,\ s,\ d) \mid s \in \xi(tc.source, tc, I) \\
\quad \wedge\, d \in \xi(tc.destination, tc, I) \wedge (s < d) \\
\quad \wedge\, \nexists d'(d' \in \xi(tc.destination, tc, I) \wedge s < d' < d)\} \\
\\
case\,4 : tc.destination.relation = RELATIVE \\
\wedge tc.destination.quantifier \neq FIRST \\
\wedge tc.destination.iterationRef.iteration = SAME\_ITERATION \\
\quad (tc, I) \mapsto \{(tc.ID,\ tc.type,\ tc.\delta,\ s,\ d) \mid s \in \xi(tc.source, tc, I) \\
\quad \wedge\, d \in \xi(tc.destination, tc, I) \wedge (s < d) \\
\quad \wedge\, \forall k^P, l^P, s^P, d^P (k^P, l^P, s^P, d^P \in N_P \wedge instOf(I, P) \\
\quad \wedge\, k^P.Type = LS \wedge l^P.Type = LS \\
\quad \wedge\, k^P.Label = tc.destination.iterationRef.loopLabel \\
\quad \wedge\, equi(s^P, s, P, I) \wedge equi(d^P, d, P, I) \\
\quad \wedge\, inLoop(s^P, k^P, P) \wedge inLoop(d^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
\quad \wedge\, lc(s, k^P.Label, I) = lc(d, k^P.Label, I) \\
\quad \wedge\, lc(s, l^P.Label, I) = lc(d, l^P.Label, I)\} \\
\vdots
\end{cases}
$$

[a] Remember that the default *relation* is $ABSOLUTE$, therefore *destination* expressions without a *relation* are handled as $ABSOLUTE$.

$$
atomize(tc, I) : \begin{cases}
\vdots \\
case\,5 : tc.destination.relation = RELATIVE \\
\wedge tc.destination.quantifier = FIRST \\
\wedge tc.destination.iterationRef.iteration = SAME\_ITERATION \\
\quad (tc, I) \mapsto \{(tc.ID,\, tc.type,\, tc.\delta,\, s,\, d) \mid s \in \xi(tc.source, tc, I) \\
\quad \wedge\, d \in \xi(tc.destination, tc, I) \wedge (s < d) \\
\quad \wedge\, \forall k^P, l^P, s^P, d^P (k^P, l^P, s^P, d^P \in N_P \wedge instOf(I, P) \\
\quad \wedge\, k^P.Type = LS \wedge l^P.Type = LS \\
\quad \wedge\, k^P.Label = tc.destination.iterationRef.loopLabel \\
\quad \wedge\, equi(s^P, s, P, I) \wedge equi(d^P, d, P, I) \\
\quad \wedge\, inLoop(s^P, k^P, P) \wedge inLoop(d^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
\quad \wedge\, lc(s, k^P.Label, I) = lc(d, k^P.Label, I) \\
\quad \wedge\, lc(s, l^P.Label, I) = lc(d, l^P.Label, I) \\
\quad \wedge\, \nexists d' (d' \in \xi(tc.destination, tc, I) \wedge s < d' < d)\} \\
\\
case\,6 : tc.destination.relation = RELATIVE \\
\wedge tc.destination.quantifier \neq FIRST \\
\wedge tc.destination.iterationRef.iteration = NEXT\_ITERATION \\
\quad (tc, I) \mapsto \{(tc.ID,\, tc.type,\, tc.\delta,\, s,\, d) \mid s \in \xi(tc.source, tc, I) \\
\quad \wedge\, d \in \xi(tc.destination, tc, I) \wedge (s < d) \\
\quad \wedge\, \forall k^P, l^P, s^P, d^P (k^P, l^P, s^P, d^P \in N_P \wedge instOf(I, P) \\
\quad \wedge\, k^P.Type = LS \wedge l^P.Type = LS \\
\quad \wedge\, k^P.Label = tc.destination.iterationRef.loopLabel \\
\quad \wedge\, equi(s^P, s, P, I) \wedge equi(d^P, d, P, I) \\
\quad \wedge\, inLoop(s^P, k^P, P) \wedge inLoop(d^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
\quad \wedge\, lc(s, k^P.Label, I) = lc(d, k^P.Label, I) - 1 \\
\quad \wedge\, lc(s, l^P.Label, I) = lc(d, l^P.Label, I)\} \\
\vdots
\end{cases}
$$

$$
atomize(tc, I) : \begin{cases}
\vdots \\
case\,7 : tc.destination.relation = RELATIVE \\
\wedge tc.destination.quantifier = FIRST \\
\wedge tc.destination.iterationRef.iteration = NEXT\_ITERATION \\
\quad (tc, I) \mapsto \{(tc.ID,\, tc.type,\, tc.\delta,\, s,\, d) \mid s \in \xi(tc.source, tc, I) \\
\quad \wedge\, d \in \xi(tc.destination, tc, I) \wedge (s < d) \\
\quad \wedge\, \forall k^P, l^P, s^P, d^P (k^P, l^P, s^P, d^P \in N_P \wedge instOf(I, P) \\
\quad \wedge\, k^P.Type = LS \wedge l^P.Type = LS \\
\quad \wedge\, k^P.Label = tc.destination.iterationRef.loopLabel \\
\quad \wedge\, equi(s^P, s, P, I) \wedge equi(d^P, d, P, I) \\
\quad \wedge\, inLoop(s^P, k^P, P) \wedge inLoop(d^P, k^P, P) \wedge inLoop(k^P, l^P, P) \\
\quad \wedge\, lc(s, k^P.Label, I) = lc(d, k^P.Label, I) - 1 \\
\quad \wedge\, lc(s, l^P.Label, I) = lc(d, l^P.Label, I) \\
\quad \wedge\, \nexists d' (d' \in \xi(tc.destination, tc, I) \wedge s < d' < d)\}
\end{cases}
$$

In the previous section, we delivered examples of 147 Extended Time Constraints and their resulting source and destination node sets. Here we deliver examples of results of the atomization function $atomize(tc, I)$ for various Extended Time Constraints tc and Instance Types I. The results of $atomize$((TC23, UBC, 5, EACH B, FIRST RELATIVE C), I4) and $atomize$((TC23, UBC, 5, EACH B, FIRST RELATIVE C), I5) are additionally represented graphically in figures 4.10 and 4.11.

Case 1: tc.destination.relation = ABSOLUTE

$atomize$((TC148, UBC, 5, FIRST B, LAST C), I4)=
{(TC148, UBC, 5, $B_{(1,0)}$, $C_{(3,3)}$)}

$atomize$((TC149, UBC, 5, EACH B, FIRST C), I4)=
{(TC149, UBC, 5, $B_{(1,0)}$, $C_{(1,1)}$), (TC149, UBC, 5, $B_{(2,0)}$, $C_{(1,1)}$), (TC149, UBC, 5, $B_{(3,0)}$, $C_{(1,1)}$)}

$atomize$((TC150, UBC, 5, FIRST C WITHIN LS2, EACH B), I4)=
{(TC150, UBC, 5, $C_{(1,1)}$, $B_{(1,0)}$), (TC150, UBC, 5, $C_{(1,1)}$, $B_{(2,0)}$), (TC150, UBC, 5, $C_{(1,1)}$, $B_{(3,0)}$),
(TC150, UBC, 5, $C_{(2,1)}$, $B_{(1,0)}$), (TC150, UBC, 5, $C_{(2,1)}$, $B_{(2,0)}$), (TC150, UBC, 5, $C_{(2,1)}$, $B_{(3,0)}$),
(TC150, UBC, 5, $C_{(3,1)}$, $B_{(1,0)}$), (TC150, UBC, 5, $C_{(3,1)}$, $B_{(2,0)}$), (TC150, UBC, 5, $C_{(3,1)}$, $B_{(3,0)}$)}

$atomize$((TC151, UBC, 5, FIRST C WITHIN LS1, FIRST C WITHIN LS2), I4)=
{(TC151, UBC, 5, $C_{(1,1)}$, $C_{(2,1)}$), (TC151, UBC, 5, $C_{(1,1)}$, $C_{(3,1)}$)}

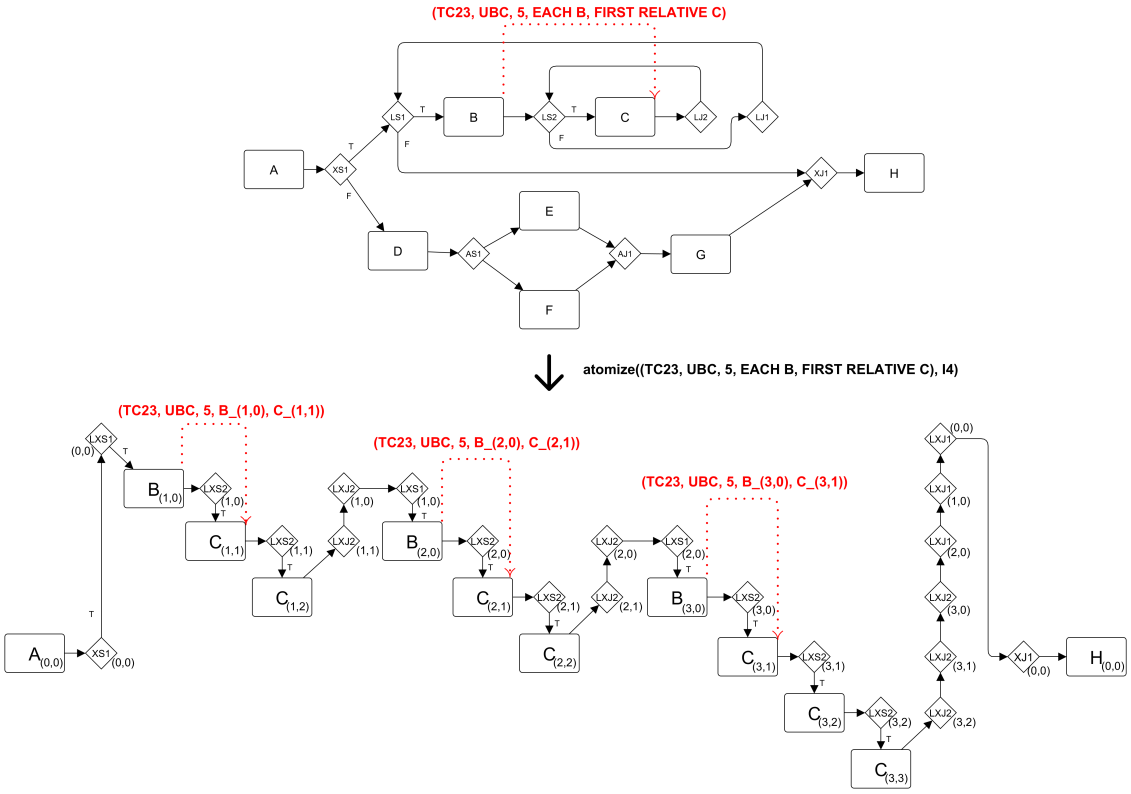


Figure 4.10: Result of the atomization function for $TC23$ on $I4$

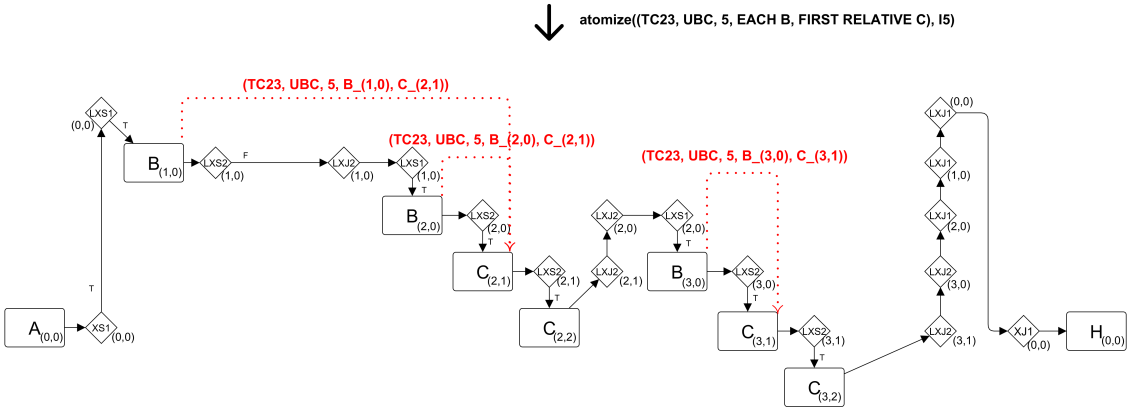


Figure 4.11: Result of the atomization function for $TC23$ on $I5$

Case 2: tc.destination.relation = RELATIVE
∧ tc.destination.quantifier ≠ FIRST
∧ tc.destination.iterationRef.iteration = ""

atomize((TC27, UBC, 5, FIRST B, LAST RELATIVE C), I4)=
{(TC27, UBC, 5, $B_{(1,0)}$, $C_{(3,3)}$)}

atomize((TC152, UBC, 5, FIRST C WITHIN LS2, EACH RELATIVE B), I4)=
{(TC152, UBC, 5, $C_{(1,1)}$, $B_{(2,0)}$), (TC152, UBC, 5, $C_{(1,1)}$, $B_{(3,0)}$), (TC152, UBC, 5, $C_{(2,1)}$, $B_{(3,0)}$)}

Case 3: tc.destination.relation = RELATIVE
∧ tc.destination.quantifier = FIRST
∧ tc.destination.iterationRef.iteration = ""

atomize((TC23, UBC, 5, EACH B, FIRST RELATIVE C), I4)=
{(TC23, UBC, 5, $B_{(1,0)}$, $C_{(1,1)}$), (TC23, UBC, 5, $B_{(2,0)}$, $C_{(2,1)}$), (TC23, UBC, 5, $B_{(3,0)}$, $C_{(3,1)}$)}

atomize((TC23, UBC, 5, EACH B, FIRST RELATIVE C), I5)=
{(TC23, UBC, 5, $B_{(1,0)}$, $C_{(2,1)}$), (TC23, UBC, 5, $B_{(2,0)}$, $C_{(2,1)}$), (TC23, UBC, 5, $B_{(3,0)}$, $C_{(3,1)}$)}

atomize((TC153, UBC, 5, FIRST C WITHIN LS1, FIRST RELATIVE C WITHIN LS2), I4)=
{(TC153, UBC, 5, $C_{(1,1)}$, $C_{(2,1)}$), (TC153, UBC, 5, $C_{(1,1)}$, $C_{(3,1)}$)}

Case 4: tc.destination.relation = RELATIVE
∧ tc.destination.quantifier ≠ FIRST
∧ tc.destination.iterationRef.iteration = SAME_ITERATION

atomize((TC97, UBC, 5, EACH B, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{(TC97, UBC, 5, $B_{(1,0)}$, $C_{(1,2)}$), (TC97, UBC, 5, $B_{(2,0)}$, $C_{(2,2)}$), (TC97, UBC, 5, $B_{(3,0)}$, $C_{(3,3)}$)}

atomize((TC98, UBC, 5, EACH C, LAST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{(TC98, UBC, 5, $C_{(1,1)}$, $C_{(1,2)}$), (TC98, UBC, 5, $C_{(2,1)}$, $C_{(2,2)}$), (TC98, UBC, 5, $C_{(3,1)}$, $C_{(3,3)}$)}

atomize((TC102, UBC, 5, FIRST C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{(TC102, UBC, 5, $C_{(1,1)}$, $C_{(1,2)}$)}

atomize((TC105, UBC, 5, LAST B, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{(TC105, UBC, 5, $B_{(3,0)}$, $C_{(3,1)}$), (TC105, UBC, 5, $B_{(3,0)}$, $C_{(3,2)}$), (TC105, UBC, 5, $B_{(3,0)}$, $C_{(3,3)}$)}

atomize((TC110, UBC, 5, EACH C, EACH RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{(TC110, UBC, 5, $C_{(1,1)}$, $C_{(1,2)}$), (TC110, UBC, 5, $C_{(2,1)}$, $C_{(2,2)}$), (TC110, UBC, 5, $C_{(3,1)}$, $C_{(3,2)}$),
(TC110, UBC, 5, $C_{(3,1)}$, $C_{(3,3)}$)}

Case 5: tc.destination.relation = RELATIVE
∧ tc.destination.quantifier = FIRST
∧ tc.destination.iterationRef.iteration = SAME_ITERATION

atomize((TC85, UBC, 5, EACH B, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{(TC85, UBC, 5, $B_{(1,0)}$, $C_{(1,1)}$), (TC85, UBC, 5, $B_{(2,0)}$, $C_{(2,1)}$), (TC85, UBC, 5, $B_{(3,0)}$, $C_{(3,1)}$)}

atomize((TC86, UBC, 5, EACH C, FIRST RELATIVE C WITHIN LS1 SAME_ITERATION LS1), I4) =
{(TC86, UBC, 5, $C_{(1,1)}$, $C_{(1,2)}$), (TC86, UBC, 5, $C_{(2,1)}$, $C_{(2,2)}$), (TC86, UBC, 5, $C_{(3,1)}$, $C_{(3,2)}$)}

Case 6: tc.destination.relation = RELATIVE
∧ tc.destination.quantifier ≠ FIRST
∧ tc.destination.iterationRef.iteration = NEXT_ITERATION

atomize((TC133, UBC, 5, EACH B, LAST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
{(TC133, UBC, 5, $B_{(1,0)}$, $C_{(2,2)}$), (TC133, UBC, 5, $B_{(2,0)}$, $C_{(3,3)}$) }

atomize((TC147, UBC, 5, EACH C, EACH RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
{(TC147, UBC, 5, $C_{(1,1)}$, $C_{(1,2)}$), (TC147, UBC, 5, $C_{(2,1)}$, $C_{(2,2)}$), (TC147, UBC, 5, $C_{(3,1)}$, $C_{(3,2)}$),
(TC147, UBC, 5, $C_{(3,2)}$, $C_{(3,3)}$)}

Case 7: tc.destination.relation = RELATIVE
∧ tc.destination.quantifier = FIRST
∧ tc.destination.iterationRef.iteration = NEXT_ITERATION

atomize((TC121, UBC, 5, EACH B, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS1), I4) =
{(TC121, UBC, 5, $B_{(1,0)}$, $C_{(2,1)}$), (TC121, UBC, 5, $B_{(2,0)}$, $C_{(3,1)}$) }

atomize((TC123, UBC, 5, EACH C, FIRST RELATIVE C WITHIN LS1 NEXT_ITERATION LS2), I4) =
{(TC123, UBC, 5, $C_{(1,1)}$, $C_{(1,2)}$), (TC123, UBC, 5, $C_{(2,1)}$, $C_{(2,2)}$), (TC123, UBC, 5, $C_{(3,1)}$, $C_{(3,2)}$), (TC123, UBC, 5, $C_{(3,2)}$, $C_{(3,3)}$)}

In this chapter, we described the basic models that we use as a starting point for time management in processes with loops. Furthermore, we introduced Extended Time Constraints (ETCs) that enable the specification of allowed temporal lower and/or upper bounds (time lags) between two activities, where at least one of them appears in a loop. At the end of this chapter, we covered the atomization function that transforms a given ETC into a set of Atomic Time Constraints (ATCs) for a given Instance Type. At runtime, all effects of ETCs are the same as the effects of the execution of an acyclic process with lower and upper bound constraints, since ETCs have to be translated into ATCs at runtime.

With ETCs and their transformation into ATCs, we introduced a novel projection of time pattern *TP1: Time Lags between two Activities*[LWR14] to cyclic processes and with it we extended the time pattern *TP9: Cyclic Elements* and laid its formal foundation. What we do not cover with ETCs are all other time patterns (*TP2: Durations, TP3: Time Lags between Arbitrary Events, TP4: Fixed Date Elements, TP5: Schedule Restricted Elements, TP6: Time-based Restrictions, TP7: Validity Period, TP8: Time-dependent Variability, and TP10: Periodicity*).

During our research, we discovered that - due to the uncertainty of unbounded loops and XORs that are placed in a loop - it is impossible to unfold the loops step by step and at the same time transform the Extended Time Constraints into Atomic Time Constraints. In order to be able to manage time in processes with loops, we developed a pre-step that tests if a process with loops must terminate in order to satisfy all Extended Time Constraints. We call this test the *Termination Check* and describe it in the following chapter.

Termination Check can enable time management in cyclic business processes in various approaches. For example, process time management, as introduced by Eder *et al.* in [EPPR99], can be applied on a cyclic process that passed the Termination Check and was unfolded (transformed) into an acyclic process. Combi *et al.* already do handle loops by transforming loops and related temporal constraints into conditional blocks (XORs) [CGPP12, CGMP12, LPCR13, CGMP14]. However, they limit the maximum number of loop iterations already in the process model. In order to determine a more accurate maximum number of loop iterations for loops that are not actually bounded by an explicit condition, Termination Check could help.

The Termination Check divides cyclic processes into two groups: 1) cyclic processes that can iterate infinitely but still satisfy all temporal constraints and 2) cyclic processes that have to terminate in order not to violate any of the temporal constraints. This characteristic helps us to sort out the cyclic processes that fall into first group, since they do not have a limit of the number of loop iterations that can be derived from temporal constraints. Cyclic processes that fall into second group do have a maximum number of loop iterations, constrained by its temporal constraints. In such a cyclic process, the maximum number of loop iterations can be calculated. After determining the number of maximum iterations for each loop, the loops can be transformed into conditional blocks as usual and if transformed into CSTNUs, dynamic controllability can be checked as well, as introduced by Hunsberger *et al.* in [HPC12].

Chapter 5

Termination Check for Cyclic Processes

In this chapter, we introduce the *Termination Check* which tests whether a given cyclic process must terminate in order to satisfy all specified Extended Time Constraints or not. Termination Check is a useful test to sort out the processes that can run forever before further process time management steps are taken. Process time management (e.g. as described in [EPPR99]) can only be applied to processes that have tested positive in the Termination Check. After the Termination Check, there are still a few steps to go before existing time management algorithms can be applied. E.g., a process must first be unfolded into an acyclic process, where loops are transformed into nested XORs. The unfolding process itself must cope with the problem of which loop gets unfolded and how often. This problem can be solved by an exploration of a search space with unfolded (acyclic) processes with different numbers of loop iterations for each loop. However, we postpone this challenge as future work and focus on the Termination Check in this thesis.

To understand the Termination Check, let us first observe a few examples. Figure 5.1 shows a process with one loop and an Extended Time Constraint (ETC) that is placed between the first and the last activity over the loop. The figure further shows three Instance Types derived from the given process. Now let us assume that activities A and C in the process have a duration of 10 days and activity B 30 days. In the first Instance Type, activity C ends 40 days after A has ended, in the second Instance Type 70 days after, and in the third Instance Type 100 days after.

The given ETC ($TC158$, UBC, 90, A, C) is satisfied for the first and the second Instance Types. However, in an Instance Type with three or more loop iterations, the ETC can not be satisfied. This means that the loop (and therefore the process) must terminate in order to satisfy the given ETC. The Termination Check for this process is positive.

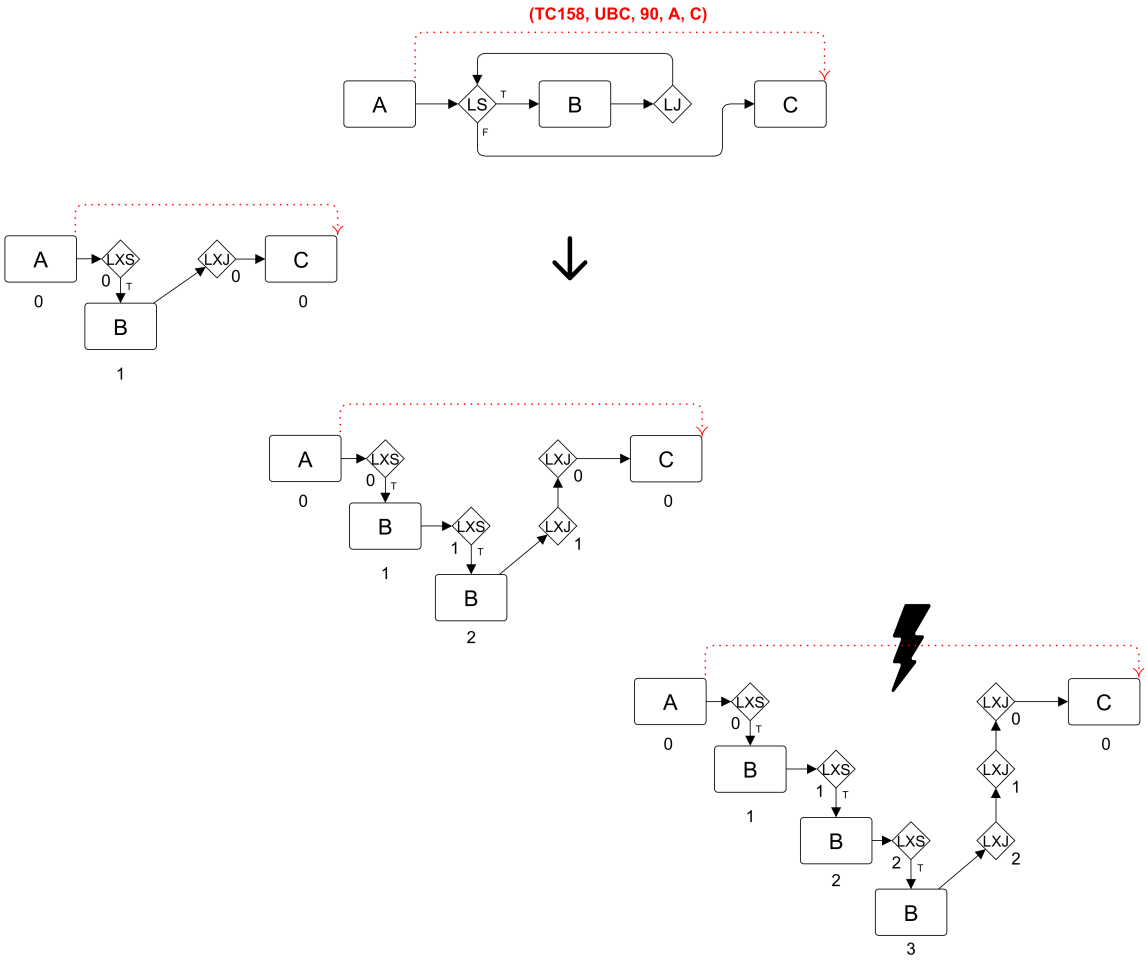


Figure 5.1: Example 1 – positive Termination Check

In the process in figure 5.2, the ETC ($TC154$, UBC, 90, $FIRST\,B$, C) represents the upper bound between the end of the first appearance of activity B and the end of activity C. This ETC behaves like the ETC from figure 5.1 and binds the number of loop iterations to a maximum of 3. In an Instance Type with 4 appearances of activity B, C would end 100 days after the first appearance of activity B and the ETC would have been violated. Since the number of loop iterations is bounded by the ETC, the Termination Check for this process is positive.

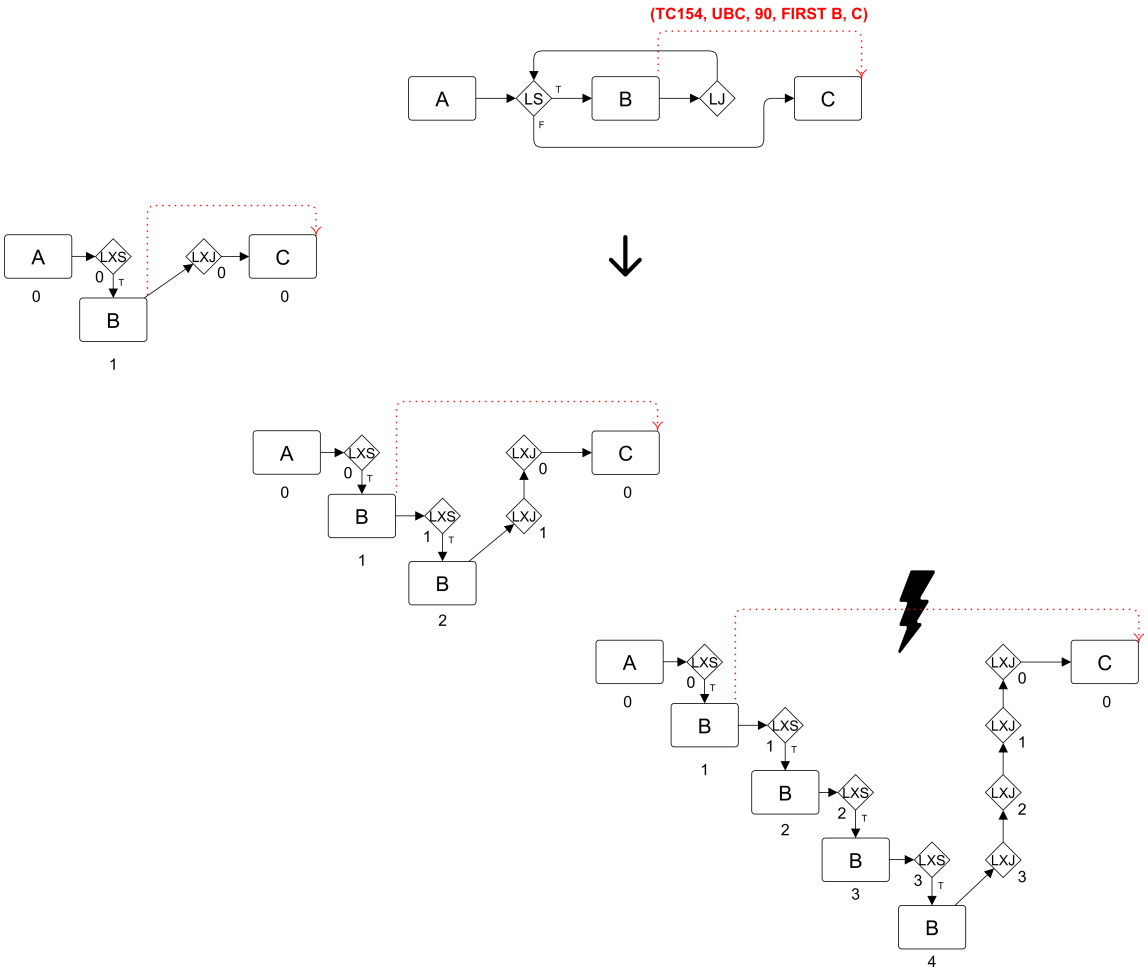


Figure 5.2: Example 2 – positive Termination Check

The ETC ($TC159$, UBC, 90, A, $LAST\ B$) from figure 5.3 represents the upper bound between activity A and the last appearance of activity B. Also this ETC behaves like the ETC from figure 5.1 and binds the number of loop iterations to a maximum of 3. In an Instance Type with 4 appearances of activity B, the last B would end 120 days after the activity A and the ETC would have been violated. Also here, the number of loop iterations is bounded by the ETC and therefore the Termination Check for this process is positive.

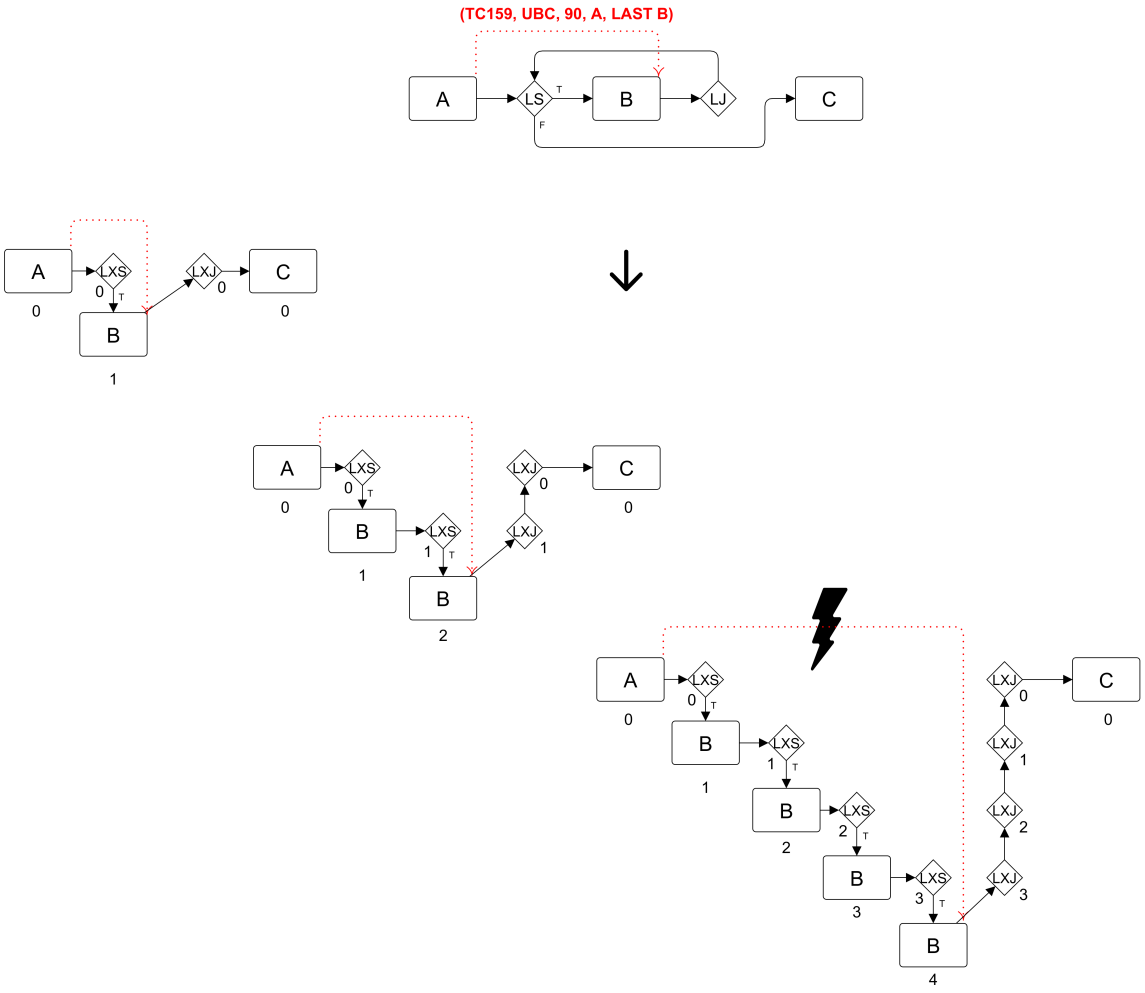


Figure 5.3: Example 3 – positive Termination Check

The process in figure 5.4 has to satisfy the ETC ($TC160$, UBC, 90, A, $FIRST\ B$) between activity A and the first appearance of activity B. After the first appearance of activity B, there can be infinitely more appearances of B and the ETC would still be satisfied. This means that the process does not have to terminate in order to satisfy the given ETC. The Termination Check for this process is negative.

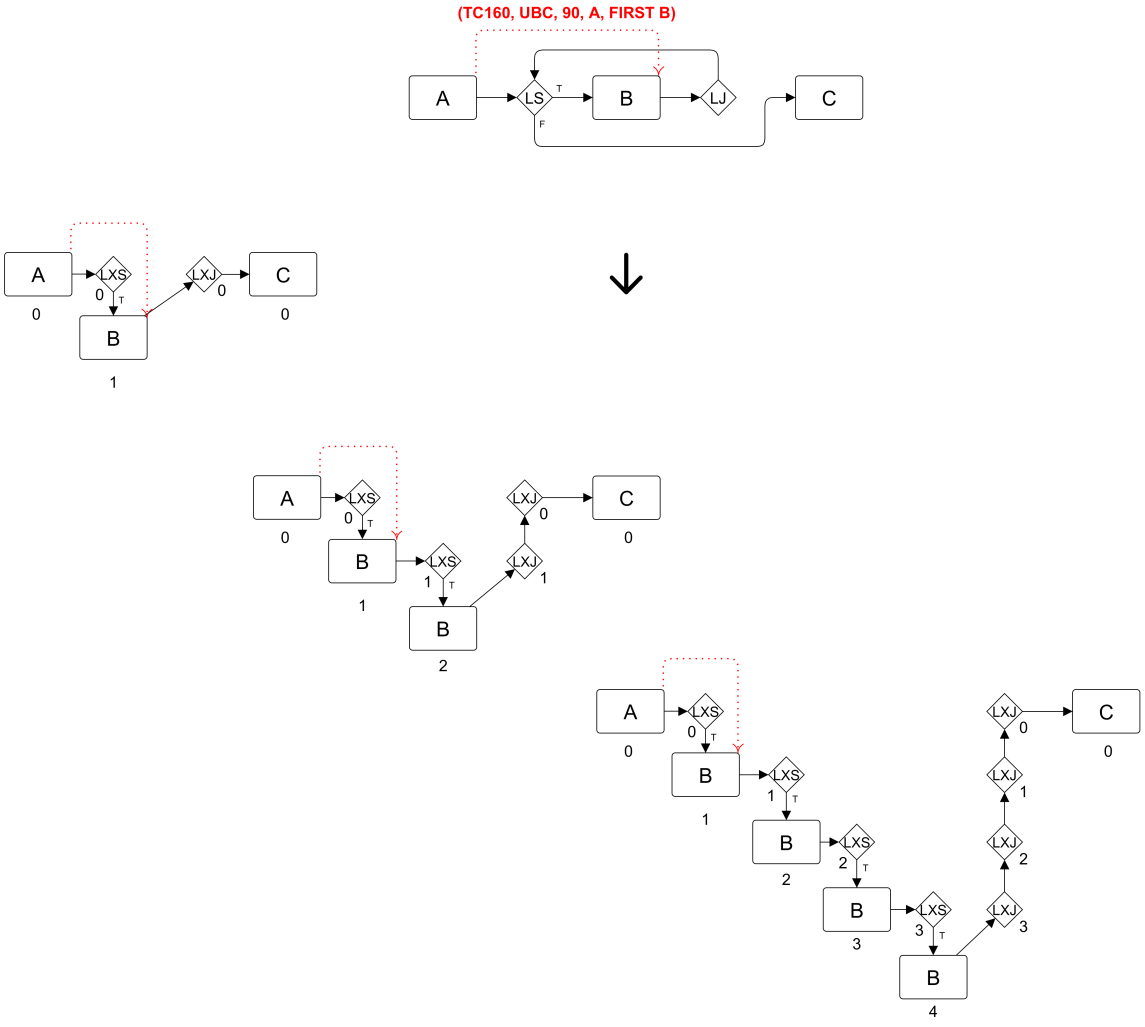


Figure 5.4: Example 4 – negative Termination Check

Now that we know, what the Termination Check aims to test, let us move on to the Termination Check itself.

The Termination Check is a process that consists of three steps:

1. Process transformation
2. Time constraints inference
3. Termination check

In the first step, we transform a cyclic process into an acyclic process with a sequence of three loop iterations instead of a loop and instantiated Atomic Time Constraints (ATCs). We call this transformed process a 3-iterated Process Graph (3-iPG). In the second step, we infer all possible time constraints from the instantiated ATCs. These two steps form the basis for the third step – termination check in a narrower sense. In this last step, we check for each loop if there is an inferred time constraints that binds the loop. We describe each step in detail in the following sections.

5.1 Process Transformation

In the first step of the Termination Check, we simulate three iterations of each loop in a cyclic process. We do that by transforming each loop into a sequence of three loop iterations. We call the transformed process graph a *3-iterated Process Graph*.

The idea behind this step, inspired by the Pumping lemma for regular languages [RS59, BHPS61], is that each loop execution has a first iteration, a last iteration, and an arbitrary number of iterations in between. Each of those three iteration blocks can also be "empty", meaning that the number of iterations of a loop is smaller than 3. Each of the three loop blocks (first, last, and everything in between) is represented by one iteration in the transformed process.

Figure 5.5 shows the process transformation of a minimal process. Note that after the last iteration of the loop, there is another LS-node. This is obviously the case, because after each last iteration of a loop, the LS-node must be passed one more time before continuing the process. Extended Time Constraints are transformed to Atomic Time Constraints in the transformed process mostly according to the same rules that we use for transforming ETCs to ATCs in Instance Types in chapter 4.

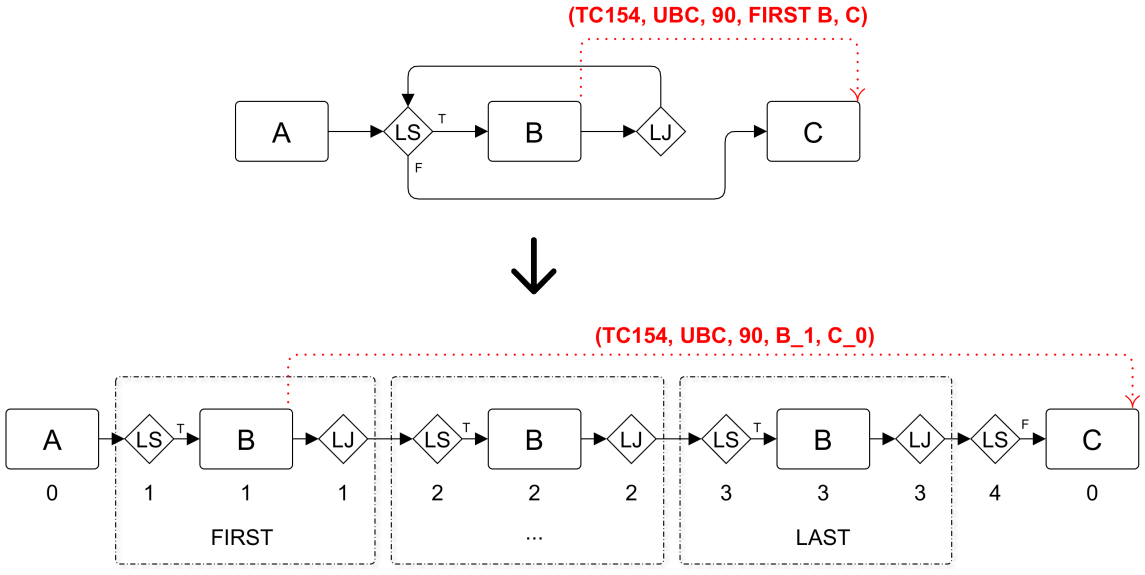


Figure 5.5: Minimal example of a process transformation

We define and describe the transformation of a process graph into a 3-iterated Process Graph in subsection 5.1.1, and the transformation of Extended Time Constraints in subsection 5.1.2.

5.1.1 Process Graph Transformation

A 3-iterated Process Graph (3-iPG), which we define in this subsection, is a process graph with 3 iterations of each loop. This iterations are unfolded into sequences and not nested LOOP-XOR-blocks as in a Loop Instance Type. Therefore, there are no LOOP-XOR-split nodes and no LOOP-XOR-join nodes in a 3-iPG. For this reason we adapt the predicate $equi(x, x', P, P')$ from chapter 4 to a stricter version as follows:

$$sEqui(x, x', P, P') := x \in N_P \wedge x' \in N_{P'} \wedge x.Label = x'.Label \wedge x.Type = x'.Type$$

We define a 3-iterated Process Graph that we use for the Termination Check, as follows:

Definition 5.1. ***(3-iterated Process Graph (3-iPG))*** *A 3-iterated Process Graph (3-iPG) of a process graph P is a directed acyclic process graph $I = (N_I, E_I)$ that is derived from the process graph P. Each node $n \in N_I$ has a label $n.Label$, a type $n.Type$, and a positive integer counter $n.C$. A directed acyclic graph I is called a valid 3-iterated Process Graph of a process graph P (3iPGOf(I,P)) if it satisfies all following rules:*

Rule 1 - non-loop nodes:
For each node n in P that does not appear in a loop, and is not a LOOP-split or a LOOP-join node, there is a derived node n' with the counter $n'.C = 0$.

$$\forall n, s \exists n' (n, s \in N_P \wedge n' \in N_I \wedge n.Type \neq LS \wedge n.Type \neq LJ \wedge s.Type = LS \wedge \neg inLoop(n, s, P) \wedge sEqui(n, n', P, I) \wedge n'.C = 0$$

Rule 2 - LOOP-split nodes:
For each node p' in I that is equivalent to p in P that is a direct predecessor of a LOOP-split node ls in P, and not a LOOP-join node, there are 4 derived nodes ls' with the counter $ls'.C = p'.C * 10 + x$, where x is an integer between 1 and 4.

$$\forall ls, p, p', x \exists ls'(ls, p \in N_P \wedge ls', p' \in N_I \wedge x \in \{1,2,3,4\}$$
$$\wedge\, ls.Type = LS \wedge p.Type \neq LJ \wedge sEqui(ls, ls', P, I) \wedge sEqui(p, p', P, I)$$
$$\wedge\, dpred(p, ls, P) \wedge ls'.C = p'.C * 10 + x$$

Rule 3 - LOOP-join nodes:
For each LOOP-split node ls' in I with a counter $ls'.C$ with a remainder 1, 2, or 3 of the division by 10, there is a counterpart LOOP-join node lj' in I with the same counter.

$$\forall ls, lj, ls', x \exists lj'(ls, lj \in N_P \wedge ls', lj' \in N_I \wedge x \in \{1,2,3\}$$
$$\wedge\, ls.Type = LS \wedge lj.Type = LJ \wedge counterpart(lj, ls, P)$$
$$\wedge\, sEqui(ls, ls', P, I) \wedge sEqui(lj, lj', P, I) \wedge ls'.C\%10 = x \wedge lj'.C = ls'.C$$

Rule 4 - nodes within a loop (except LOOP-split nodes, LOOP-join nodes, and direct successors of LOOP-split nodes):
For each node p' in I that is equivalent to p in P, which is a direct predecessor of a node n in P, there is a node n' in I with the same counter as p' with a remainder 1, 2, or 3 of the division by 10. Node n is placed in a loop ls and is neither a LOOP-split node nor a LOOP-join node. Neither is p a LOOP-split node.

$$\forall n, p, p', x \exists ls, n'(n, p, ls \in N_P \wedge n', p' \in N_I \wedge x \in \{1,2,3\}$$
$$\wedge\, n.Type \neq LS \wedge n.Type \neq LJ \wedge p.Type \neq LS \wedge ls.Type = LS$$
$$\wedge\, dpred(p, n, P) \wedge inLoop(n, ls, P) \wedge sEqui(n, n', P, I) \wedge sEqui(p, p', P, I)$$
$$\wedge\, p'.C\%10 = x \wedge n'.C = p'.C$$

Rule 5 - true-direct-successors of a LOOP-split node (except LOOP-split and LOOP-join nodes):
For each node p' in I that is equivalent to p in P, which is a direct predecessor of a node n in P, there is a node n' in I with the same counter as p' with a remainder 1, 2, or 3 of the division by 10. Node n is not a LOOP-split node and not a LOOP-join node, and p is a LOOP-split node. Node n is a true-direct-successor of p.

$$\forall n, p, p', x \exists (p, n, T), n' (n, p \in N_P \wedge (p, n, T) \in E_P \wedge n', p' \in N_I \wedge x \in \{1, 2, 3\}$$
$$\wedge\, n.Type \neq LS \wedge n.Type \neq LJ \wedge p.Type = LS$$
$$\wedge\, dpred(p, n, P) \wedge sEqui(n, n', P, I) \wedge sEqui(p, p', P, I)$$
$$\wedge\, p'.C\%10 = x \wedge n'.C = p'.C$$

Rule 6 - false-direct-successors of a LOOP-split node within a loop (except LOOP-split and LOOP-join nodes):
For each node pp' in I that is equivalent to pp in P, there is a node n' in I with the same counter as pp'. Node n is neither a LOOP-split node nor a LOOP-join node. Node n is placed in a loop, and is a false-direct-successor of p. Node p is a LOOP-split node and pp is not a LOOP-join node. Node p is a direct successor of pp.

$$\forall n, p, pp, p', pp' \exists ls, (p, n, F), n' (n, p, pp, ls \in N_P \wedge (p, n, F) \in E_P$$
$$\wedge\, n', p', pp' \in N_I \wedge ls \neq p \wedge dpred(p, n, P) \wedge dpred(pp, p, P) \wedge inLoop(n, ls, P)$$
$$\wedge\, n.Type \neq LS \wedge n.Type \neq LJ \wedge p.Type = LS \wedge ls.Type = LS \wedge pp.Type \neq LJ$$
$$\wedge\, sEqui(n, n', P, I) \wedge sEqui(p, p', P, I) \wedge sEqui(pp, pp', P, I) \wedge n'.C = pp'.C$$

Rule 7 - edges between nodes with the same counter:
For all nodes s' and e' in I that are derived from s and e in P and have the same counter, there is an edge (s', e') in I between them if s' is not a LOOP-join node, and e' is not a LOOP-split node, and if there is an edge between s and e in P.

$$\forall s, e, s', e' \exists (s', e') (s, e \in N_P \wedge s', e' \in N_I \wedge (s', e') \in E_I$$
$$\wedge\, \neg (s.Type = LJ \wedge e.Type = LS) \wedge sEqui(s, s', P, I) \wedge sEqui(e, e', P, I)$$
$$\wedge\, dpred(s, e, P) \wedge s'.C = e'.C$$

Rule 8 - edges between LOOP-join and LOOP-split nodes:
For all LOOP-join nodes s' and LOOP-split nodes e' in I that are derived from s and e in P, there is an edge (s', e') in I between them if there is an edge between s and e in P and the counter of e' is the counter of s' increased by 1.

$$\forall s, e, s', e' \exists (s', e')(s, e \in N_P \wedge s', e' \in N_I \wedge (s', e') \in E_I$$
$$\wedge\, s.Type = LJ \wedge e.Type = LS \wedge sEqui(s, s', P, I) \wedge sEqui(e, e', P, I)$$
$$\wedge\, dpred(s, e, P) \wedge s'.C + 1 = e'.C$$

Rule 9 - edges between LOOP-split predecessors and LOOP-split nodes:
For all LOOP-split predecessor nodes s' and LOOP-split nodes e' in I that are derived from s and e in P, there is an edge (s', e') in I between them if there is an edge between s and e in P and the counter of e' is the counter of s' multiplied by 10 and increased by 1.

$$\forall s, e, s', e' \exists (s', e')(s, e \in N_P \wedge s', e' \in N_I \wedge (s', e') \in E_I$$
$$\wedge\, e.Type = LS \wedge sEqui(s, s', P, I) \wedge sEqui(e, e', P, I)$$
$$\wedge\, dpred(s, e, P) \wedge s'.C * 10 + 1 = e'.C$$

Rule 10 - edges between LOOP-split and false-direct-successor:
For all LOOP-split nodes s' and nodes e' in I that are derived from s and e in P, there is an edge (s', e') in I between them if there is a false-edge between s and e in P and the counter of s' returns a remainder 4 for the division by 10, and the counter of e' is the counter of s' divided by 10 (remember that the node counter is an integer).

$$\forall s, e, s', e' \exists (s, e, F), (s', e')(s, e \in N_P \wedge (s, e, F) \in E_P \wedge s', e' \in N_I \wedge (s', e') \in E_I$$
$$\wedge\, s.Type = LS \wedge sEqui(s, s', P, I) \wedge sEqui(e, e', P, I)$$
$$\wedge\, s'.C\%10 = 4 \wedge e'.C = s'.C/10$$

Figure 5.6 shows a more complex example of the process transformation of the process graph from figure 4.2 with the Extended Time Constraint (TC23, UBC, 5, EACH B, FIRST RELATIVE C).

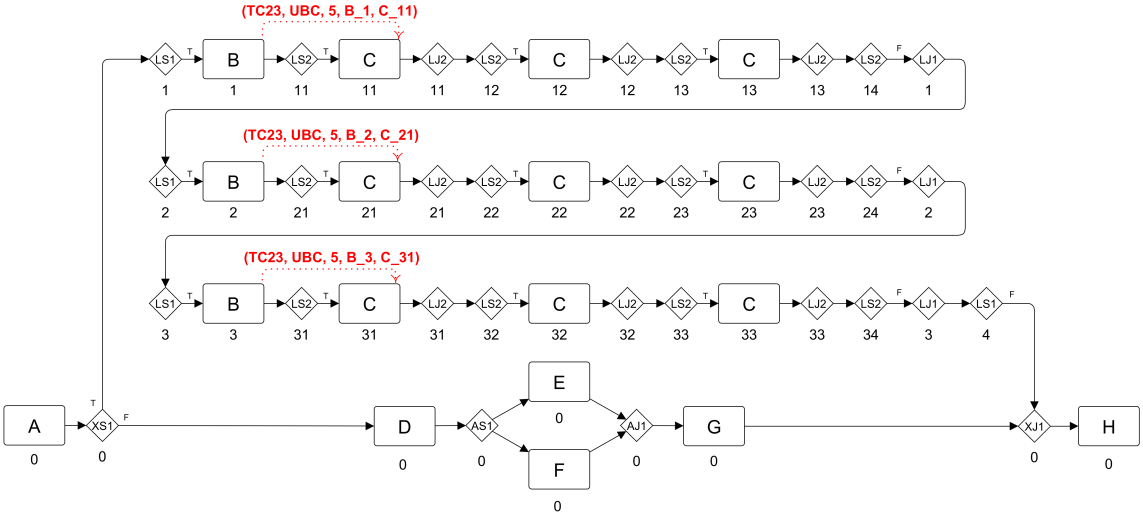


Figure 5.6: 3-iterated Process Graph of the process graph from figure 4.2

Note that the Extended Time Constraint in this example can be transformed to a set of Atomic Time Constraints in the resulting 3-iterated Process Graph according to the rules in chapter 4. Unfortunately, this is not the case for all Extended Time Constraints due to the different structure of a 3-iterated Process Graph and an Instance Type. The details are discussed in the next subsection.

5.1.2 Extended Time Constraints Transformation

Extended Time Constraints, that contain one of the following source expressions:

- *X*
- *FIRST X*
- *LAST X*
- *EACH X*
- *FIRST X WITHIN L*
- *EACH X WITHIN L*

and one of the following destination expressions:

- X
- $FIRST\ X$
- $LAST\ X$
- $EACH\ X$
- $EACH\ X\ WITHIN\ L$
- $FIRST\ RELATIVE\ X$
- $LAST\ RELATIVE\ X$
- $EACH\ RELATIVE\ X$
- $FIRST\ RELATIVE\ X\ WITHIN\ L$
- $LAST\ RELATIVE\ X\ WITHIN\ L$
- $EACH\ RELATIVE\ X\ WITHIN\ L$

are transformed into Atomic Time Constraints in a 3-iterated Process Graph with the same rules as they are in an Instance Type (see chapter 4).

Extended Time Constraints that contain the source expression $FIRST\ X\ WITHIN\ L$, or $LAST\ X\ WITHIN\ L$, and/or one of the following destination expressions:

- $FIRST\ X\ WITHIN\ L$
- $LAST\ X\ WITHIN\ L$
- $FIRST\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$
- $LAST\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$
- $EACH\ RELATIVE\ X\ WITHIN\ L\ SAME\_ITERATION\ K$
- $FIRST\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$
- $LAST\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$
- $EACH\ RELATIVE\ X\ WITHIN\ L\ NEXT\_ITERATION\ K$

cannot be transformed into Atomic Time Constraints according to the rules presented in chapter 4 due to the different structure of a 3-iterated Process Graph and an Instance Type.

In this subsection, we adapt the source/destination expression evaluation function $\xi(expr, tc, I)$ from chapter 4 for expressions listed above, such that it can be used to reach equal semantic results in a 3-iterated Process Graph as in an Instance Type.

FIRST X WITHIN L

$$
\begin{aligned}
&\xi((FIRST\ X\ WITHIN\ L), tc, I) := \{n \in N_I | n.Label = X \wedge \\
&\quad (\exists l, k, l^P, k^P, n^P \nexists m (l, k, m \in N_I \wedge l^P, k^P, n^P \in N_P \wedge 3iPGOf(I, P) \\
&\qquad \wedge l^P.Label = L \wedge l^P.Type = LS \wedge k^P.Type = LS \\
&\qquad \wedge equi(l^P, l, P, I) \wedge equi(k^P, k, P, I) \wedge equi(n^P, n, P, I) \\
&\qquad \wedge closestLoop(k^P, l^P, P) \wedge inLoop(n^P, l^P, P) \\
&\quad \wedge m.Label = X \wedge k < m < n) \\
&\quad \vee \\
&\quad \exists l, l^P, n^P \nexists m, k^P (l, m \in N_I \wedge l^P, k^P, n^P \in N_P \wedge 3iPGOf(I, P) \\
&\qquad \wedge l^P.Label = L \wedge l^P.Type = LS \wedge k^P.Type = LS \\
&\qquad \wedge equi(l^P, l, P, I) \wedge equi(n^P, n, P, I) \\
&\qquad \wedge closestLoop(k^P, l^P, P) \wedge inLoop(n^P, l^P, P) \\
&\quad \wedge m.Label = X \wedge m < n))\}
\end{aligned}
$$

Evaluation of the expression $FIRST\ X\ WITHIN\ L$ on a 3-iPG differs from the evaluation on an Instance Type in chapter 4 only in the predicate $instOf(I, P)$ that is substituted by the predicate $3iPGOf(I, P)$.

LAST X WITHIN L

$$
\begin{aligned}
&\xi((LAST\ X\ WITHIN\ L), tc, I) := \{n \in N_I | n.Label = X \wedge \\
&\quad (\exists l, k, kj, l^P, k^P, n^P \nexists m (l, k, kj, m \in N_I \wedge l^P, k^P, n^P \in N_P \\
&\qquad \wedge 3iPGOf(I, P) \wedge counterpart(kj, k, P) \\
&\qquad \wedge l^P.Label = L \wedge l^P.Type = LS \wedge k^P.Type = LS \\
&\qquad \wedge equi(l^P, l, P, I) \wedge equi(k^P, k, P, I) \wedge equi(n^P, n, P, I) \\
&\qquad \wedge closestLoop(k^P, l^P, P) \wedge inLoop(n^P, l^P, P) \\
&\quad \wedge m.Label = X \wedge n < m < kj) \\
&\quad \vee \\
&\quad \exists l, l^P, n^P \nexists m, k^P (l, m \in N_I \wedge l^P, k^P, n^P \in N_P \wedge 3iPGOf(I, P) \\
&\qquad \wedge l^P.Label = L \wedge l^P.Type = LS \wedge k^P.Type = LS \\
&\qquad \wedge equi(l^P, l, P, I) \wedge equi(n^P, n, P, I) \\
&\qquad \wedge closestLoop(k^P, l^P, P) \wedge inLoop(n^P, l^P, P) \\
&\quad \wedge m.Label = X \wedge n < m))\}
\end{aligned}
$$

In contrast to the evaluation of the expression $LAST\ X\ WITHIN\ L$ on an Instance Type, the evaluation on a 3-iPG is based on the same idea as the evaluation of expression $FIRST\ X\ WITHIN\ L$. Instead of the constraint that there is no twin node m of n between the LOOP-split node k and n, there is a constraint that there is no twin node m of n between n and the LOOP-join node kj.

Evaluation of the destination expressions that contain iteration references $SAME\_ITERATION$ or $NEXT\_ITERATION$ on a 3-iPG differs from the evaluation on an Instance Type in chapter 4 only in the predicate $instOf(I, P)$ that is substituted by the predicate $3iPGOf(I, P)$, as well as an adapted definition of the function $lc(n', LS, P')$ that returns the loop (iteration) counter of node $n' \in N_{P'}$ in a 3-iPG P' for the loop with the LOOP-split node with the label LS. We (re-)define the function $lc(n', LS, P')$ that is applied to a 3-iPG as follows:

$$
\begin{aligned}
&lc(n', LS, P') := ls'.C\%10 \,|\, ls', n' \in N_{P'} \wedge ls, n \in N_P \wedge 3iPGOf(P', P) \\
&\wedge ls'.Type = LS \wedge ls'.Label = LS \\
&\wedge sEqui(ls, ls', P, P') \wedge sEqui(n, n', P, P') \wedge inLoop(n, ls, P) \\
&\wedge \nexists ks'(ks' \in N_{P'} \wedge ks'.Type = LS \wedge ks'.Label = LS \wedge ls' < ks' < n')
\end{aligned}
$$

Transformation of Extended Time Constraints into a set of Atomic Time Constraints in a 3-iPG is executed with the atomization function $atomize(tc, I)$ from chapter 4, where the predicate $instOf(I, P)$ is substituted by the predicate $3iPGOf(I, P)$ and the loop iteration counter function $lc(n', LS, P')$ is overridden by the function that we defined in the previous paragraph.

An example of a transformation of a process with several Extended Time Constraints is shown in figure 5.7.

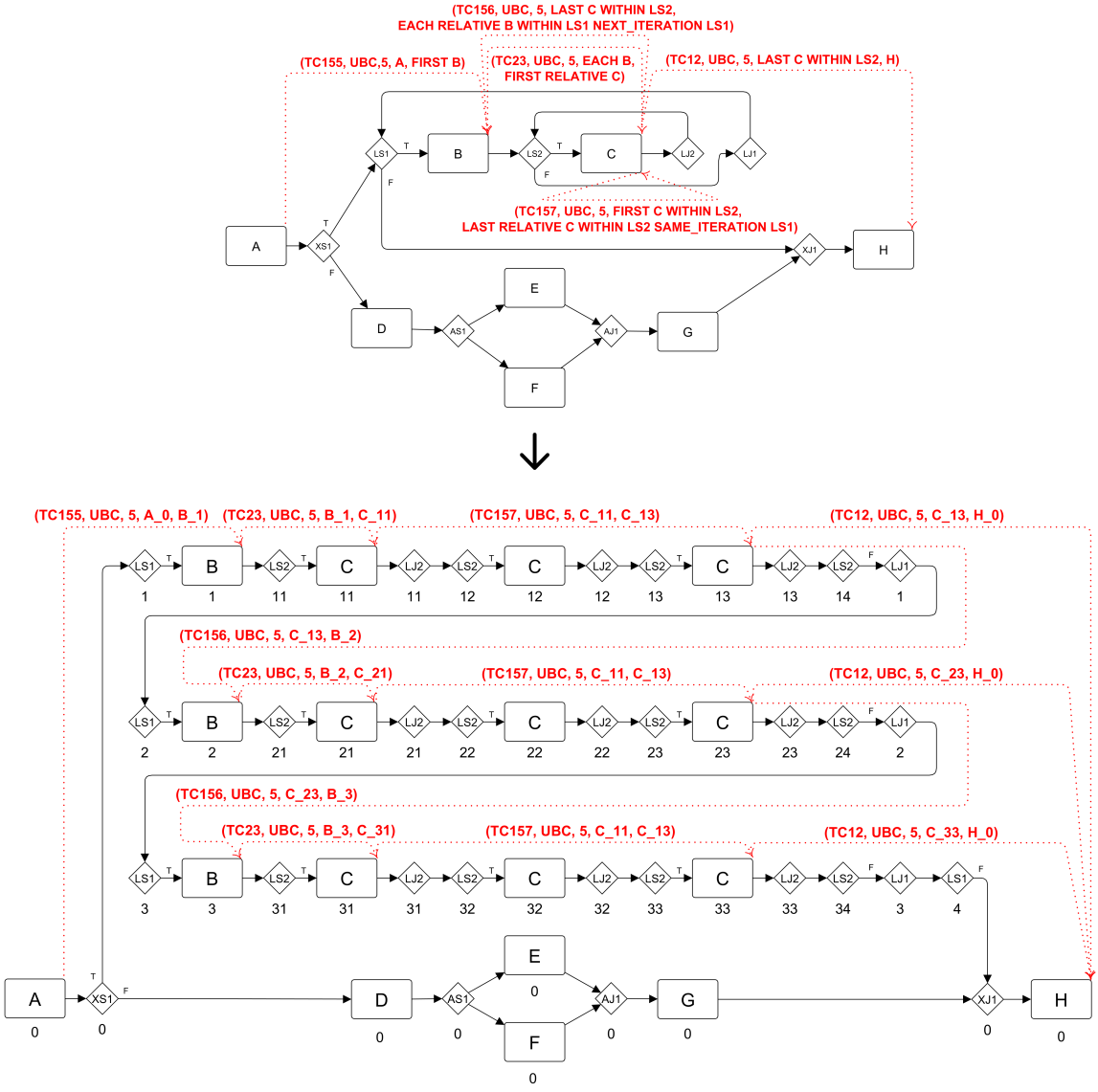


Figure 5.7: 3-iterated Process Graph of the process graph from figure 4.2 with a set of ETCs

In this section, we described how we transform a process graph into a 3-iterated Process Graph, and how we transform Extended Time Constraints into Atomic Time Constraints in the obtained 3-iPG. In the next section, we move to the next step of the Termination Check process: the Inference of Time Constraints.

5.2 Time Constraints Inference

Atomic Time Constraints with type Lower Bound Constraint (LBC) require at least δ time units to pass between the source and destination. In contrast to ATCs with type LBC, ATCs with type Upper Bound Constraint (UBC) require at most δ time units to pass between the source and destination. If a process contains a sequence of activities A, B, and C and there is an ATC with type UBC and δ=5 days between A and C, then we can infer that also between A and B, or between B and C no more than 5 days are allowed to pass. To be precise, no more than 5 days minus duration of C are allowed to pass between A and B, and no more than 5 days minus duration of B are allowed to pass between B and C.

We use this property for the Termination Check where we analyze if a process with unbounded loops must terminate in order to satisfy all Extended Time Constraints with the type UBC. In other words, we investigate if unbounded loops in a process can be bounded with Extended Time Constraints. In order to do that, we first infer all possible (inferred) ATCs from ATCs with type UBC that we derived from the given ETCs. In inferred ATCs, we leave δ unchanged, since it is irrelevant for the investigation if ETCs with type UBC can bind unbounded loops. In general, we infer new ATCs by moving the source/destination node of an inferred ATC in two directions:

Source from the left to the right We infer an ATC between the source node y and the destination node z from an (inferred) ATC between the source node x and destination node z, such that y is a direct successor of x.

Destination from the right to the left We infer an ATC between the source node x and the destination node y from an (inferred) ATC between the source node x and destination node z, such that y is a direct predecessor of z.

Depending on the structure of a 3-iPG, it is not always possible to infer new inferred ATCs until the source node of an inferred ATC is the same node as the destination node. The function $atcClosure(atc, I)$ defines a set of inferred ATCs that can be inferred from the given ATC atc in respect to all possible structure cases of a given 3-iPG I.

In case that in an ATC the destination node is a predecessor of the source node, the source is handled as a destination and the destination as a source, since an ATC simply represents a temporal interval (without any direction) between two nodes. In function $atcClosure(atc, I)$, this is handled by $CASE\ 0a$.

We define the closure of a given ATC atc for a given 3-iPG I as follows:

$$
\begin{aligned}
&atcClosure(atc, I) := \{(atc.ID, atc.type, atc.\delta, s, d) \mid atc \in ATC_I \wedge s, d \in N_I \\
&\qquad \wedge atc.type = UBC \wedge s \neq d \wedge (\\
&(path(d, s, I) \wedge \\
&\triangleright \text{ CASE 0a: the given ATC atc (with destination} < \text{source)} \\
&\triangleright \text{ is an element of its closure} \\
&\qquad (s = atc.destination \wedge d = atc.source) \\
&) \\
&\vee \\
&(path(s, d, I) \wedge (\\
&\triangleright \text{ CASE 0b: the given ATC atc (with source} < \text{destination)} \\
&\triangleright \text{ is an element of its closure} \\
&\qquad (s = atc.source \wedge d = atc.destination) \\
&\vee \\
&\triangleright \text{ CASE 1: sequence and AND-block - source to the right} \\
&\qquad (s.Type \neq XJ \wedge s.Type \neq LS \\
&\qquad \wedge \exists iatc, x(iatc \in atcClosure(atc, I) \wedge x \in N_I \\
&\qquad \wedge iatc.source = x \wedge iatc.destination = d \wedge dpred(x, s))) \\
&\vee \\
&\vdots
\end{aligned}
$$

$$
\begin{aligned}
&\vdots \\
&\triangleright \text{ CASE 2: sequence and AND-block - destination to the left} \\
&\quad (d.Type \neq XS \wedge d.Type \neq LS \\
&\quad \wedge \exists iatc, x(iatc \in atcClosure(atc, I) \wedge x \in N_I \\
&\quad \wedge iatc.source = s \wedge iatc.destination = x \wedge dpred(d, x))) \\
&\vee \\
&\triangleright \text{ CASE 3: XOR-block - source to the right} \\
&\quad (s.Type = XJ \\
&\quad \wedge \exists iatc1, iatc2, x, y(iatc1, iatc2 \in atcClosure(atc, I) \wedge x, y \in N_I \wedge x \neq y \\
&\quad \wedge iatc1.source = x \wedge iatc1.destination = d \wedge dpred(x, s) \\
&\quad \wedge iatc2.source = y \wedge iatc2.destination = d \wedge dpred(y, s))) \\
&\vee \\
&\triangleright \text{ CASE 4: XOR-block - destination to the left} \\
&\quad (s.Type = XS \\
&\quad \wedge \exists iatc1, iatc2, x, y(iatc1, iatc2 \in atcClosure(atc, I) \wedge x, y \in N_I \wedge x \neq y \\
&\quad \wedge iatc1.source = s \wedge iatc1.destination = x \wedge dpred(d, x) \\
&\quad \wedge iatc2.source = s \wedge iatc2.destination = y \wedge dpred(d, y))) \\
&\vee \\
&\triangleright \text{ CASE 5: LOOP-block (counter } < 4) \text{ - source to the right} \\
&\quad (s.Type = LS \wedge s.C\%10 \neq 4 \\
&\quad \wedge \exists iatc, x(iatc \in atcClosure(atc, I) \wedge x \in N_I \\
&\quad \wedge iatc.source = x \wedge iatc.destination = d \wedge dpred(x, s))) \\
&\vee \\
&\triangleright \text{ CASE 6: LOOP-block (counter } > 1) \text{ - destination to the left} \\
&\quad (d.Type = LS \wedge d.C\%10 \neq 1 \\
&\quad \wedge \exists iatc, x(iatc \in atcClosure(atc, I) \wedge x \in N_I \\
&\quad \wedge iatc.source = s \wedge iatc.destination = x \wedge dpred(d, x))) \\
&\vee \\
&\vdots
\end{aligned}
$$

$$
\begin{aligned}
&\vdots \\
&\triangleright \text{ CASE 7: LOOP-block (counter} = 4) \text{ - source to the right} \\
&\quad (s.Type = LS \wedge s.C\%10 = 4 \\
&\quad \wedge \exists iatc1, iatc2, x, y(iatc1, iatc2 \in atcClosure(atc, I) \wedge x, y \in N_I \\
&\quad \wedge x \neq y \wedge y.Type = LS \wedge y.C = s.C - 3 \wedge dpred(x, s) \\
&\quad \wedge iatc1.source = x \wedge iatc1.destination = d \\
&\quad \wedge iatc2.source = y \wedge iatc2.destination = d)) \\
&\vee \\
&\triangleright \text{ CASE 8: LOOP-block (counter} = 1) \text{ - destination to the left} \\
&\quad (d.Type = LS \wedge d.C\%10 = 1 \\
&\quad \wedge \exists iatc1, iatc2, x, y(iatc1, iatc2 \in atcClosure(atc, I) \wedge x, y \in N_I \\
&\quad \wedge x \neq y \wedge y.Type = LS \wedge y.C = s.C + 3 \wedge dpred(d, x) \\
&\quad \wedge iatc1.source = s \wedge iatc1.destination = x \\
&\quad \wedge iatc2.source = s \wedge iatc2.destination = y)) \\
&)\}
\end{aligned}
$$

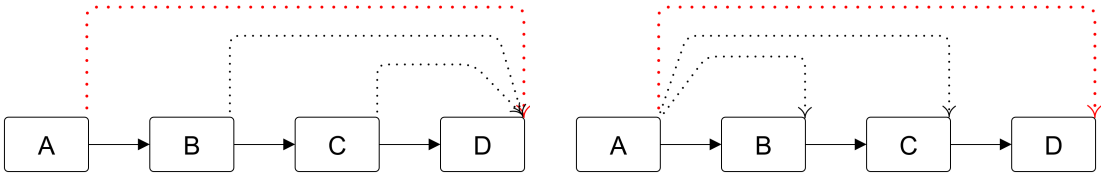
Now let us take a look at the differences in inference of ATCs depending on various structures of a given 3-iPG on minimal examples.

Inference of ATCs in a sequence

In a sequence, new ATCs can be inferred from a given ATC, or an inferred ATC, by moving the source node to the right and the destination node to the left until the source node is a direct predecessor of the destination node.

Figure 5.8 shows a sequence of 4 activities and a given ATC between activity A and activity D. In subfigure 5.8a, there are two inferred ATCs (ATC between B and D and ATC between C and D), which we infer by moving the source from node A to the right to node B and node C.

Subfigure 5.8b shows how we infer ATCs by moving the destination from node D to the left. There are two inferred ATCs (ATC between A and C and ATC between A and B) that we infer from the given ATC between A and D.



(a) Source to the right (b) Destination to the left

Figure 5.8: Inference of ATCs in a sequence

If we repeatedly infer new ATCs in both directions from other inferred ATCs, we obtain an ATC closure. The ATC closure for the given ATC between the source node A and destination node D is represented in figure 5.9. The ATC closure of the ATC between A and D contains (inferred) ATCs between following source and destination nodes: A-D, A-C, A-B, B-D, C-D, and B-C.

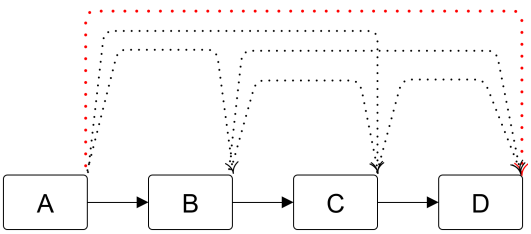
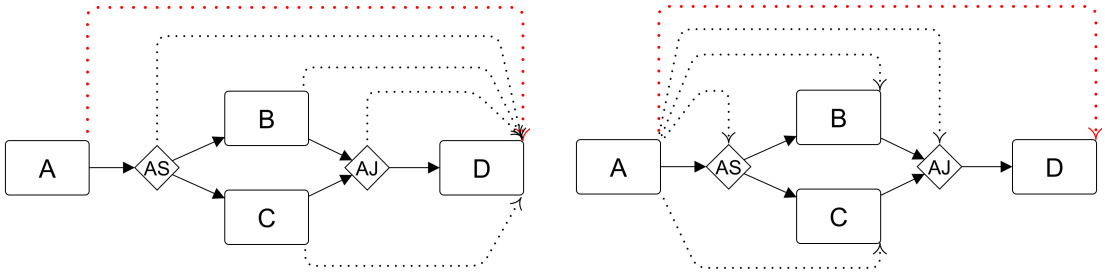


Figure 5.9: ATC closure

Inference of ATCs in an AND-block

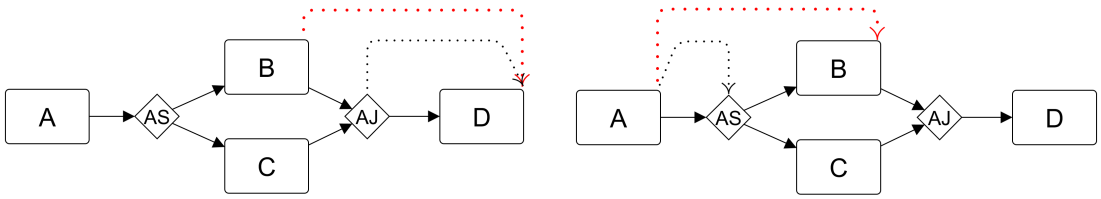
In an AND-block, new ATCs are inferred the same way as in a sequence.
Figure 5.10 shows how ATCs are inferred from a given ATC between the source node A and destination node D. The source and destination node are both outside the AND-block.

In figure 5.11a, the source of the given ATC between B and D is placed in the AND-block, and in figure 5.11b, the source of the given ATC between A and B is placed in the AND-block. In all cases, ATCs can be inferred as they are in a sequence, since both branches are executed in parallel in an AND-block.



(a) Source to the right (b) Destination to the left

Figure 5.10: Inference of ATCs in an AND-block (1)



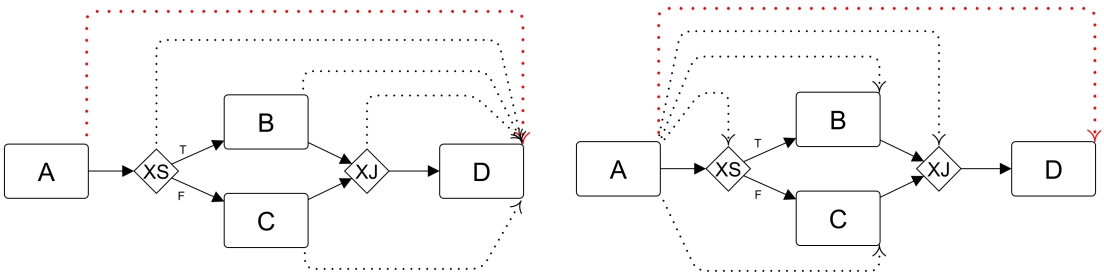
(a) Source to the right (b) Destination to the left

Figure 5.11: Inference of ATCs in an AND-block (2)

Inference of ATCs in an XOR-block

In contrast to an AND-block, in an XOR-block only one of two branches gets executed, and at design time we do not know which one it will be. Because of this behavior of an XOR-block, we have to adapt the inference rules to handle the uncertainty properly.

If the source node of a given ATC is placed before the XOR-block, and the destination node after the XOR-block, new ATCs can be inferred like they are in a sequence or an AND-block. This can be done because the ATC, which spans over the whole XOR-block, constrains the allowed execution time in the whole XOR-block and not only one branch. Figure 5.12 shows the inference of ATCs in such case.

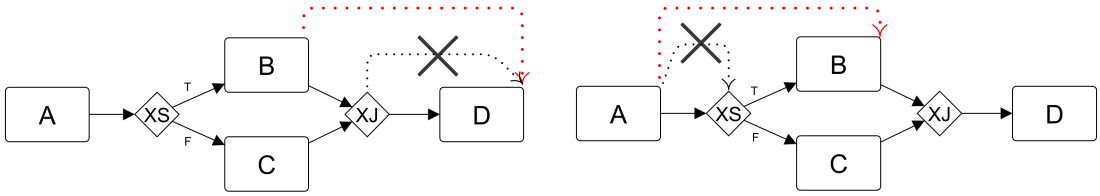


(a) Source to the right (b) Destination to the left

Figure 5.12: Inference of ATCs in an XOR-block (1)

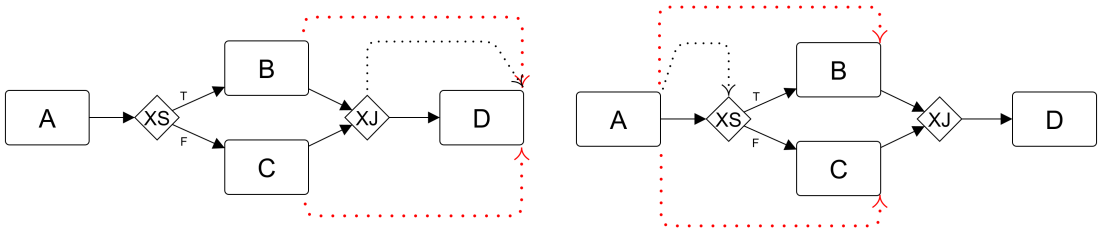
However, if the source node of a given ATC is placed in an XOR-block and the destination node somewhere after the XOR-block, we can only move the source to the right until we reach the XOR-join node, as illustrated in figure 5.13a. We cannot move the source to the XOR-join node and further to the right because of the existing uncertainty. The other branch in the XOR-block that does not involve any ATC could be triggered at runtime. However, if there are two source nodes of two different ATCs in an XOR-block - each in a different XOR-branch - then the source node(s) can be moved to the XOR-join node and further, since both branches are constrained with an ATC. This second scenario is illustrated in 5.14a.

The other way around, if the destination node of a given ATC is placed in an XOR-block and the source node somewhere before the XOR-block, we can only move the destination to the left until we reach the XOR-split node as illustrated in figure 5.13b. We cannot move the destination to the XOR-split node and further to the left because of the existing uncertainty. The other branch in the XOR-block that does not involve any ATC could be triggered at runtime. However, if there are two destination nodes of two different ATCs in an XOR-block - each in a different XOR-branch - then the destination node(s) can be moved to the XOR-split node and further, since both branches are constrained with an ATC. This second scenario is illustrated in 5.14b.



(a) Source to the right (b) Destination to the left

Figure 5.13: Inference of ATCs in an XOR-block (2)



(a) Source to the right (b) Destination to the left

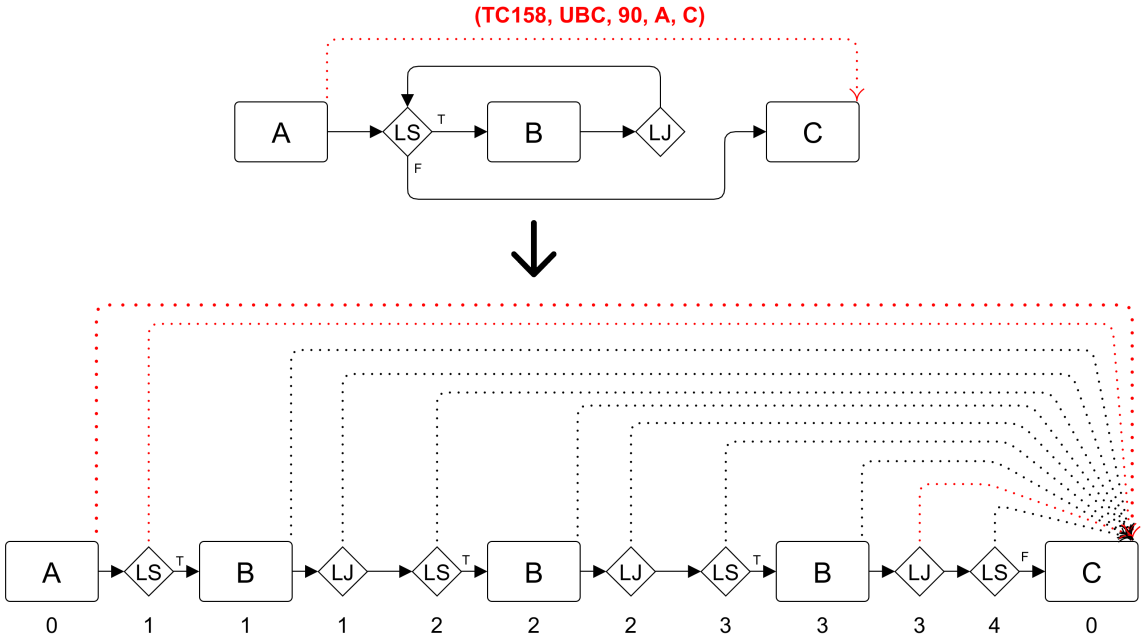
Figure 5.14: Inference of ATCs in an XOR-block (3)

Inference of ATCs in a LOOP-block

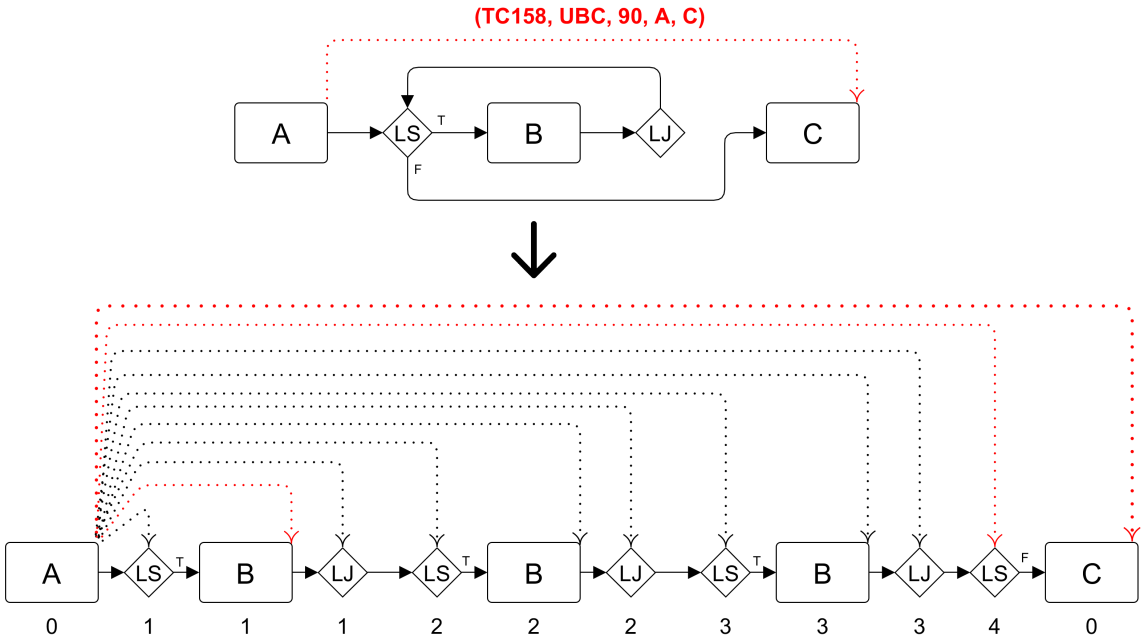
A LOOP-block and an XOR-block have in common that their execution is not certain at design time. Therefore, the ATC inference rules in both structures have some similarities.

If the source node of a given ATC is placed before the LOOP-block, and the destination node after the LOOP-block, new ATCs can be inferred like they are in a sequence. This can be done because the ATC, which spans over the whole LOOP-block, constrains the allowed execution time of the entire LOOP-block (and thus limits the number of loop iterations). Figure 5.15 shows the inference of ATCs in such a case.

However, if the source node of a given ATC is placed in a LOOP-block and the destination node somewhere after the LOOP-block, we can only move the source to the right until we reach the LOOP-split node with a counter that ends with a digit 4 as illustrated in figure 5.16a. We cannot move the source to that LOOP-split node and further to the right because of the existing uncertainty. The loop could not be entered at all at runtime and thus there would be no ATC in the resulting Instance Type.



(a) Source to the right



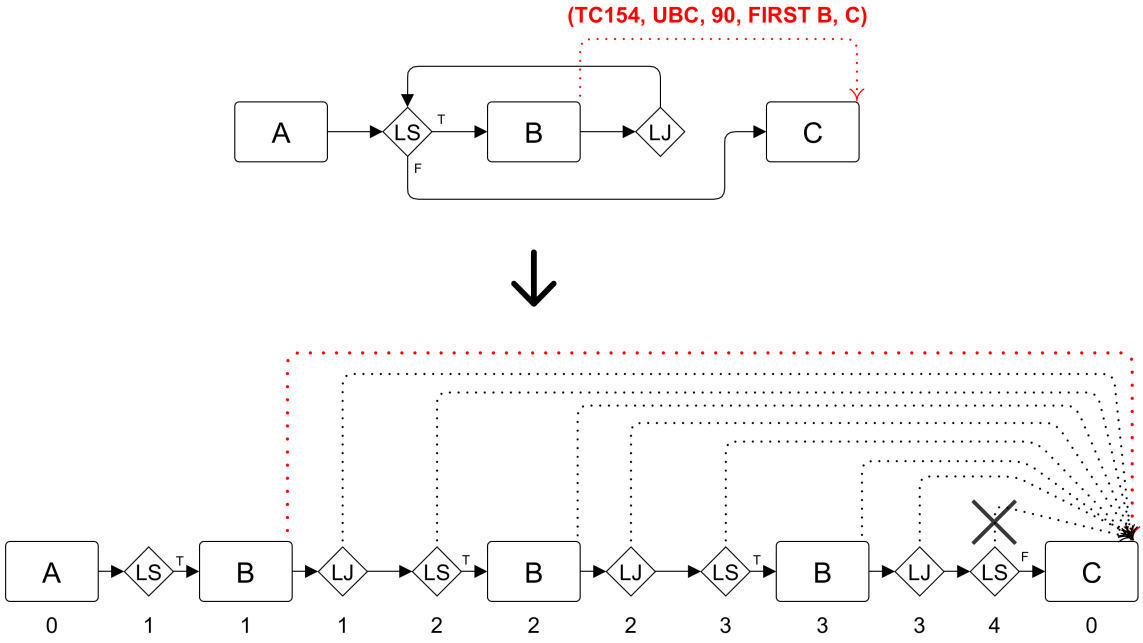
(b) Destination to the left

Figure 5.15: Inference of ATCs in a LOOP-block (1)

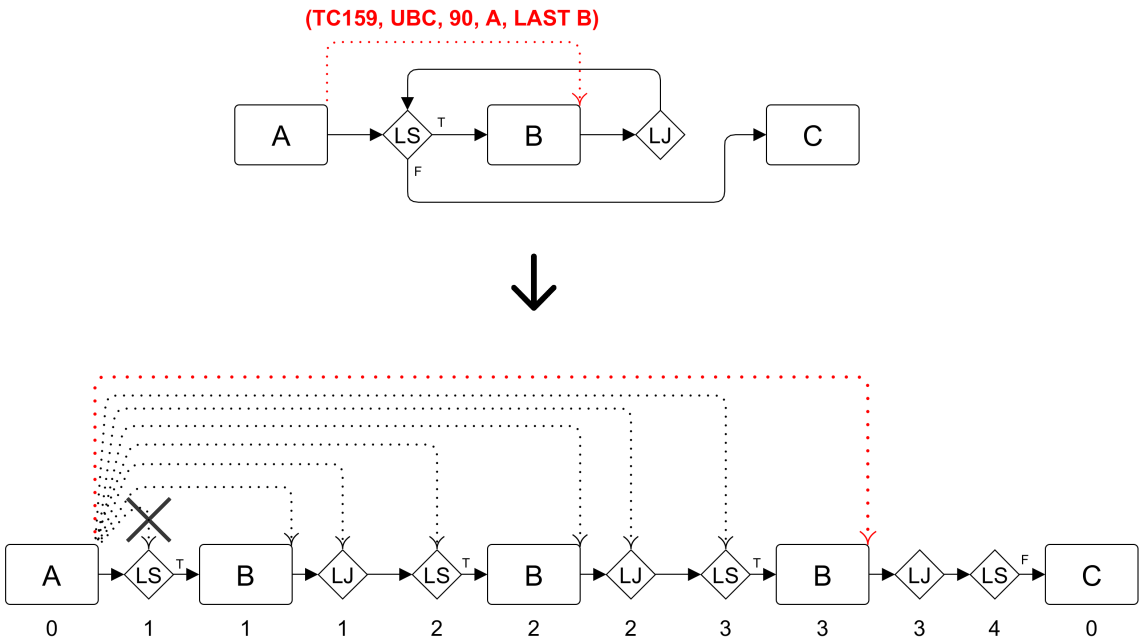
The other way around, if the destination node of a given ATC is placed in a LOOP-block and the source node somewhere before the LOOP-block, we can only move the destination to the left until we reach the direct successor of the LOOP-split node with a counter that ends with a digit 1 as illustrated in figure 5.16b. We cannot move the source to that LOOP-split node and further to the left because of the existing uncertainty. The loop could not be entered at all at runtime and thus there would be no ATC in the resulting Instance Type.

In this section, we specified the rules for ATC inference and defined the ATC closure. We defined the function $atcClosure(atc, I)$, which returns a set of all inferred ATCs that can be inferred from a given ATC atc in a given 3-iPG I.

Based on the ATC closure, we specify the predicate $terminationCheck(I)$ in the next section. This predicate checks if a given process must terminate in order to satisfy all Extended Time Constraints. We additionally illustrate, with several examples, how the introduced mechanism works as well as how and why it is able to check if a process must terminate in order to satisfy all Extended Time Constraints.



(a) Source to the right



(b) Destination to the left

Figure 5.16: Inference of ATCs in a LOOP-block (2)

5.3 Termination Check

In section 5.1, we introduced the 3-iterated Process Graph (3-iPG). The idea behind the 3-iPG is that each loop has (at most) one first iteration, (at most) one last iteration, and an arbitrary number of iterations between the first and the last iteration. The second iteration of a loop in a 3-iPG symbolizes all iterations between the first and the last iteration.

If the execution time of loops in a process is limited with Extended Time Constraints (ETCs), then only a limited number of iterations can be "pumped" between the first and the last iterations of the loops. This means that the second iteration of the loops in a 3-iPG is limited.

If there are loops in a process that are not limited with ETCs, then the number of iterations that can be "pumped" between the first and the last iterations of those loops is not limited. In this case, the second iteration of those loops in a 3-iPG is not limited. This means that the whole process is (structurally and temporally) unlimited.

We use this characteristic in the termination check. If each loop in a process P is temporally bounded by an ETC (or a set of ETCs), then the whole process is bounded and must terminate in order to satisfy all ETCs. To check this termination property, we check if there is an inferred ATC between the LOOP-split node and the LOOP-join node of the second iteration of each loop in the 3-iPG I of the process P. If this is the case, then only a limited number of iterations can be "pumped" in each loop in P and therefore the whole process is limited and must terminate in order to satisfy all ETCs.

We check the termination property of a process P on its corresponding 3-iPG I with the predicate $terminationCheck(I)$ that we define as follows:

$$
\begin{aligned}
&terminationCheck(I) := \forall s\, \exists j, atc, iatc \mid s, j \in N_I \\
&\wedge\, atc \in ATC_I \wedge iatc \in atcClosure(atc, I) \\
&\wedge\, s \neq j \wedge s.Type = LS \wedge j.Type = LJ \wedge s.C = j.C \wedge s.C\%10 = 2 \\
&\wedge\, iatc.source = s \wedge iatc.destination = j
\end{aligned}
$$

A proof of correctness and completeness of the Termination Check turns out to be very challenging. In order to demonstrate the plausibility of the termination check, we investigate several process examples on the following pages. The examples represent all relevant process structures and nestings in combination with different Extended Time Constraints. For each example, we describe how the Termination Check works and discuss its result. Furthermore, we describe in terms of Instance Types, why a process must terminate in order to be in compliance with all ETCs, or why it does not have to and can still be in compliance with all ETCs.

In following figures we present several termination check examples. Each figure shows two processes: 1) a given process with ETCs that has to be termination-checked, and 2) its transformation in a 3-iPG with inferred ATCs. In the resulting 3-iPGs, red ATCs represent transformed ETC(s), black ATCs represent inferred ATCs, and the green ATCs represent inferred ATCs that are required for a positive termination check. Green ATCs that are marked with a cross are required for a positive termination check but cannot be inferred from ATCs in the resulting 3-iPGs (are not in ATC closure). In some examples, the ATC closure(s) contain so many ATCs that the example figure contains only a subset of the ATC closure. An overview of all following examples is given in table 5.1

Example No.	ETC	ETC Pattern Description	Termination Check
1	(TC158, UBC, 90, A, C)	ETC over loop	positive
2	(TC154, UBC, 90, FIRST B, C)	ETC from loop	positive
3	(TC159, UBC, 90, A, LAST B)	ETC into loop	positive
4	(TC160, UBC, 90, A, FIRST B)	ETC into loop	negative
5	(TC161, UBC, 90, LAST B, C)	ETC from loop	negative
6	(TC162, UBC, 90, FIRST B, LAST B)	ETC in loop	positive
7	(TC163, UBC, 90, FIRST B, D)	ETC from nested AND	positive
8	(TC164, UBC, 90, FIRST B, D)	ETC from nested XOR	negative
9	(TC165, UBC, 90, A, FIRST B) (TC164, UBC, 90, FIRST B, D)	ETC into nested XOR ETC from nested XOR	negative
10	(TC164, UBC, 90, FIRST B, D) (TC166, UBC, 90, FIRST C, D)	ETC from nested XOR ETC from nested XOR	positive
11	(TC164, UBC, 90, FIRST B, D) (TC167, UBC, 90, LAST C, D)	ETC from nested XOR ETC from nested XOR	negative
12	(TC168, UBC, 90, FIRST C, D)	ETC from nested XOR	negative
13	(TC169, UBC, 90, A, FIRST C) (TC168, UBC, 90, FIRST C, D)	ETC into nested loop ETC from nested loop	negative
14	(TC155, UBC, 5, A, FIRST B) (TC156, UBC, 5, LAST C WITHIN LS2, EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1) (TC23, UBC, 5, EACH B, FIRST RELATIVE C) (TC12, UBC, 5, LAST C WITHIN LS2, H) (TC157, UBC, 5, FIRST C WITHIN LS2, LAST RELATIVE C WITHIN LS2 SAME_ITERATION LS1)	ETC into loop ETC from nested loop ETC into nested loop ETC from nested loop ETC in nested loop	negative

Table 5.1: Termination check examples

Example 1: (TC158, UBC, 90, A, C) - positive termination check

Figure 5.17 shows the transformation of a process with one loop and the ETC $TC158$ that spans over the loop. In the resulting 3-iPG, we can see a part of the ATC closure of the resulting ATC ($TC158$, UBC, 90, A_0, C_0). The ATC between LS_2 and LJ_2 is required for a positive termination check. This ATC can be inferred from the ATC between A_0 and C_0, therefore the given process must terminate in order to be in compliance with ETC $TC158$.

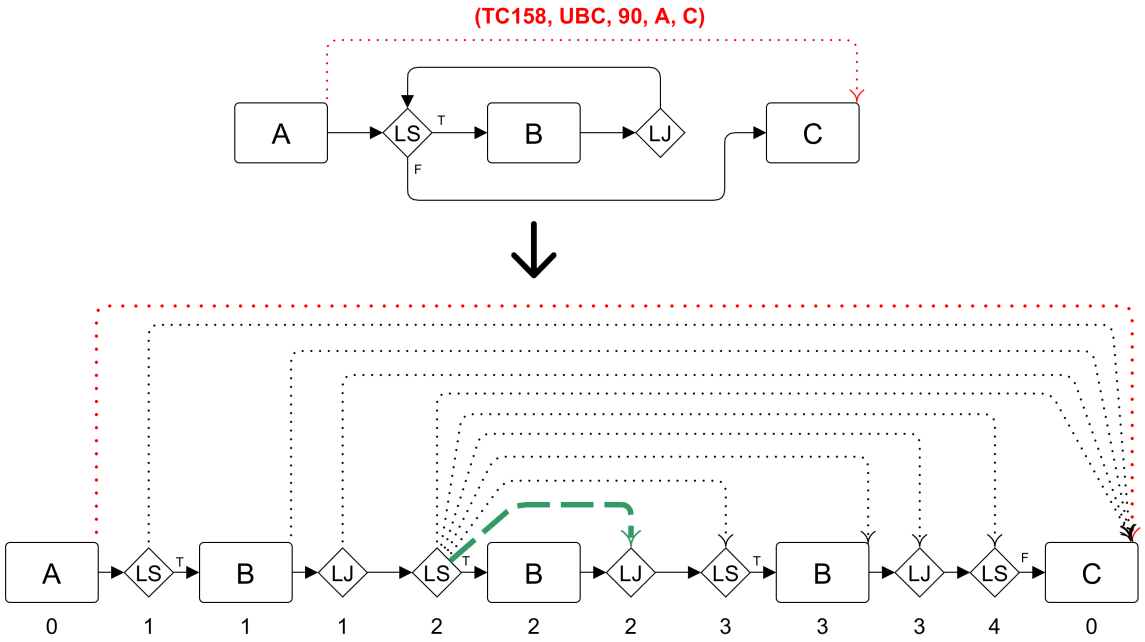


Figure 5.17: Example 1 - positive termination check

There is only a finite number of loop iterations such that the process satisfies the ETC $TC158$. Therefore, there is also a finite number of Instance Types of the given process that satisfy the ETC. A finite number of Instance Types makes further steps in Time Management (e.g. forward an backward time calculation) possible, while an infinite number of Instance Types does not.

Example 2: (TC154, UBC, 90, FIRST B, C) - positive termination check

The process from figure 5.18 also has to terminate in order to satisfy the ETC $TC154$, even though the ETC does not span over the loop. In the resulting 3-iPG, we can see a part of the ATC closure of the resulting ATC ($TC154$, UBC, 90, B_1, C_0). The ATC between LS_2 and LJ_2 is required for a positive termination check. This ATC can be inferred from the ATC between B_1 and C_0, therefore the given process must terminate in order to be in compliance with ETC $TC154$.

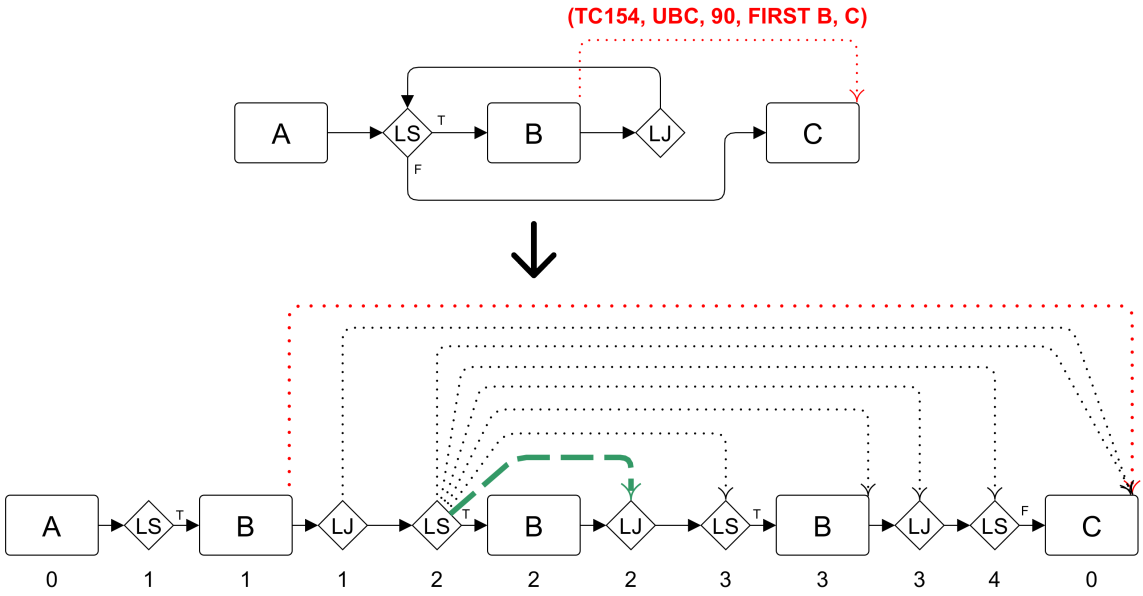


Figure 5.18: Example 2 - positive termination check

There is only a finite number of loop iterations such that the process satisfies the ETC $TC154$. Therefore, there is also a finite number of Instance Types of the given process that satisfy the ETC. This is the case because each Instance Type, in which the loop is entered, has an occurrence of B in the first loop iteration. This one binds the loop, since after that the execution time is limited.

Example 3: (TC159, UBC, 90, A, LAST B) - positive termination check

The process from figure 5.19 has to terminate as well in order to satisfy the ETC $TC159$. In the resulting 3-iPG, we can see a part of the ATC closure of the resulting ATC ($TC159$, UBC, 90, A_0, B_3). The ATC between LS_2 and LJ_2 can be inferred from the ATC between A_0 and B_3, therefore the given process must terminate in order to be in compliance with the ETC $TC159$.

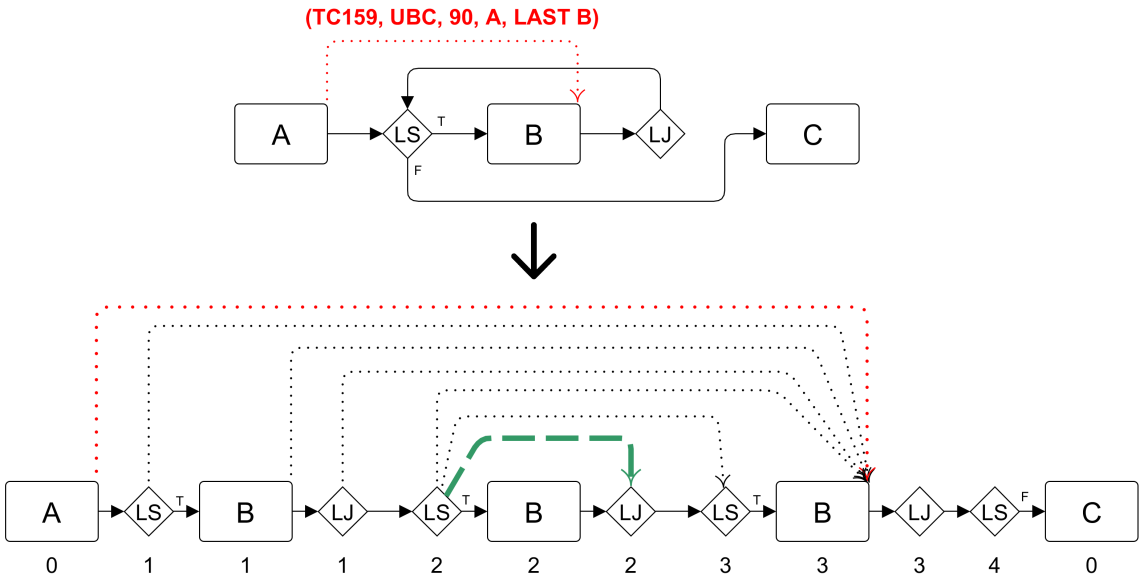


Figure 5.19: Example 3 - positive termination check

There is only a finite number of loop iterations such that the process satisfies the ETC $TC159$. Therefore, there is also a finite number of Instance Types of the given process that satisfy the ETC. This is the case because each Instance Type, in which the loop is entered, has an occurrence of B in each loop iteration. There is one occurrence of B where the ETC is still satisfied and in each occurrence of B after that, the ETC is not satisfied anymore. This particular occurrence of B determines the last possible loop iteration and therefore binds the loop and the number of allowed Instance Types.

Example 4: (TC160, UBC, 90, A, FIRST B) - negative termination check

Figure 5.20 shows an example of a process that satisfies the ETC $TC160$, even if it does not terminate. In the resulting 3-iPG, we can see the ATC closure of the resulting ATC ($TC160$, UBC, 90, A_0, B_1). The ATC between LS_2 and LJ_2 cannot be inferred from the ATC between A_0 and B_1, therefore the given process does not have to terminate in order to be in compliance with the ETC $TC160$.

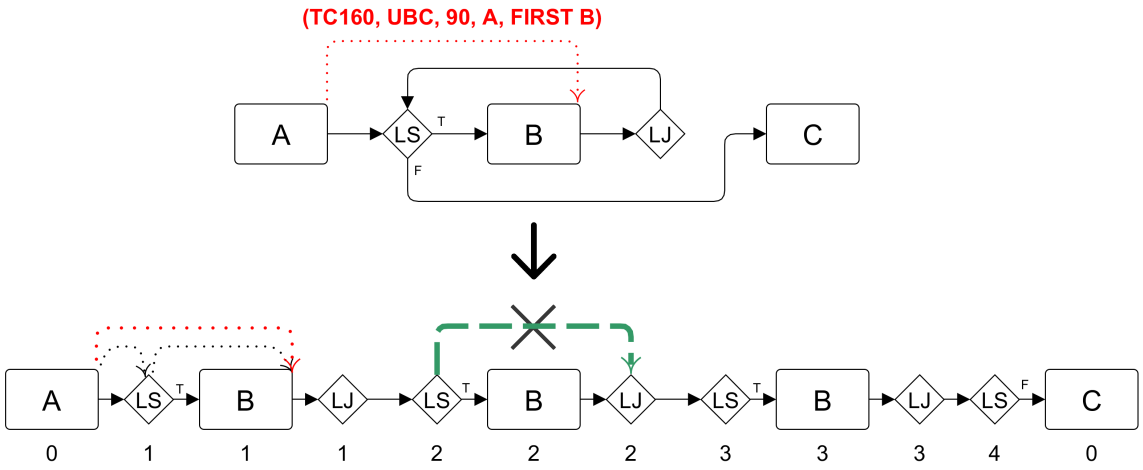


Figure 5.20: Example 4 - negative termination check

The process satisfies the ETC $TC160$ even if the number of loop iterations is infinite. Therefore, there is also an infinite number of Instance Types of the given process that satisfy the ETC. This is the case because each Instance Type, in which the loop is entered, has the first occurrence of B in the first loop iteration. This occurrence of B must be in compliance with the ETC $TC160$. All further occurrences of B are not constrained with the ETC and therefore neither is the number of loop iterations.

Example 5: (TC161, UBC, 90, LAST B, C) - negative termination check

Figure 5.21 shows a similar example of a process that satisfies the ETC $TC161$, even if it does not terminate. In the resulting 3-iPG, we can see the ATC closure of the resulting ATC ($TC161$, UBC, 90, B_3, C_0). The ATC between LS_2 and LJ_2 cannot be inferred from the ATC between B_3 and C_0, therefore the given process does not have to terminate in order to be in compliance with the ETC $TC161$.

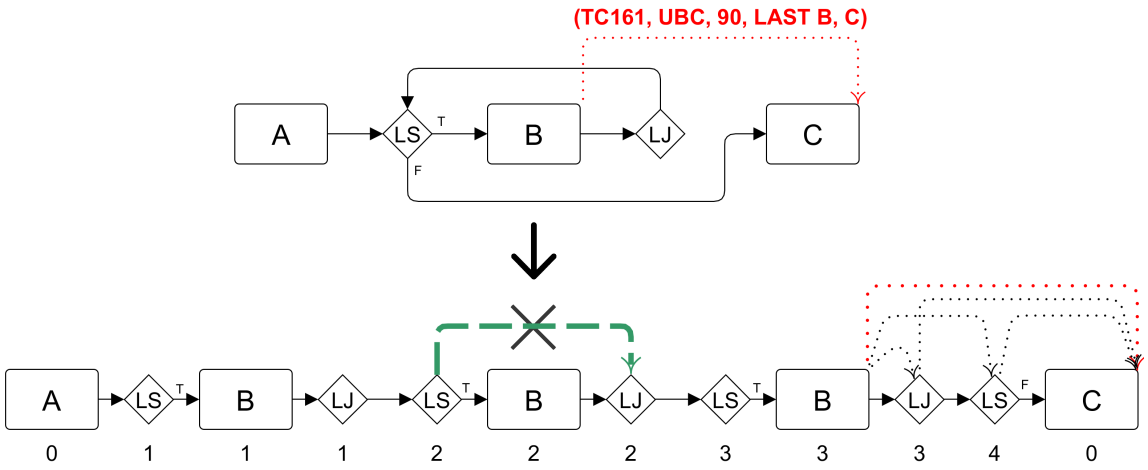


Figure 5.21: Example 5 - negative termination check

The process satisfies the ETC $TC161$ even if the number of loop iterations is infinite. Therefore, there is also an infinite number of Instance Types of the given process that satisfy the ETC. This is the case because each Instance Type can have an infinite number of loop iterations with occurrences of B. The ETC $TC160$ will be satisfied in each Instance Type, since it constrains only the last occurrence of B with the following C - no matter how many occurrences of B there are before the last one.

Example 6: (TC162, UBC, 90, FIRST B, LAST B) - positive termination check

Figure 5.22 shows an example of a process that can satisfy the ETC $TC162$ only if it terminates. In the resulting 3-iPG, we can see a part of the ATC closure of the resulting ATC ($TC162$, UBC, 90, B_1, B_3). The ATC between LS_2 and LJ_2 can be inferred from the ATC between B_1 and B_3, therefore the given process must terminate in order to be in compliance with the ETC $TC162$.

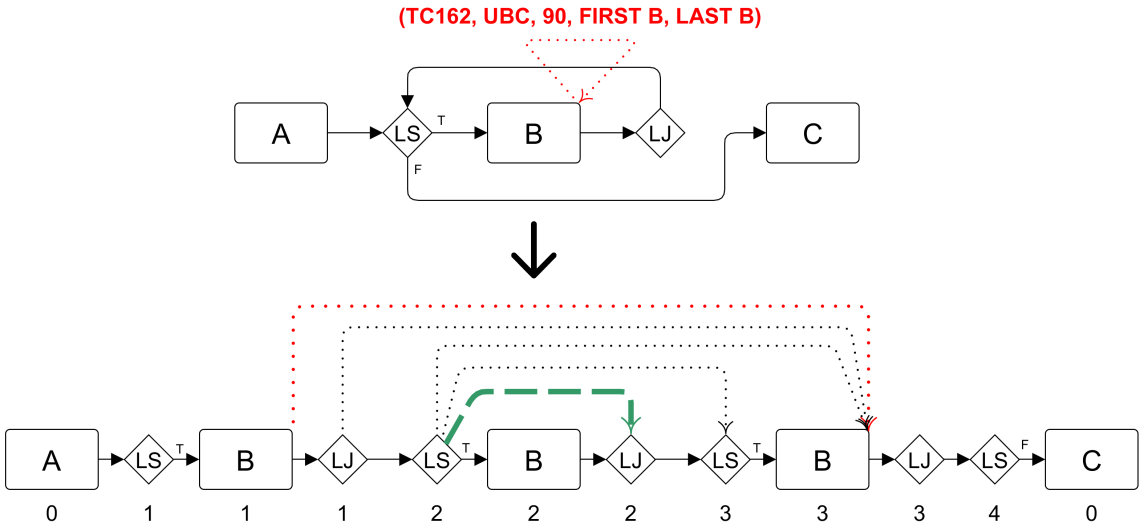


Figure 5.22: Example 6 - positive termination check

The process satisfies the ETC $TC162$ only if the number of loop iterations is finite. Therefore, there is also a finite number of Instance Types of the given process that satisfy the ETC. This is the case because each Instance Type, in which the loop is entered, has an occurrence of B in each loop iteration. The first B occurs in the first loop iteration. After it, there is one particular occurrence of B where the ETC is still satisfied and in each occurrence of B after that, the ETC is not satisfied anymore. This particular occurrence of B determines the last possible loop iteration and therefore binds the loop and the number of allowed Instance Types.

Example 7: (TC163, UBC, 90, FIRST B, D) - positive termination check

The process from figure 5.23 has to terminate in order to satisfy the ETC $TC163$. In the resulting 3-iPG, we can see a part of the ATC closure of the resulting ATC ($TC163$, UBC, 90, B_1, D_0). The ATC between LS_2 and LJ_2 can be inferred from the ATC between B_1 and D_0, therefore the given process must terminate in order to be in compliance with ETC $TC163$.

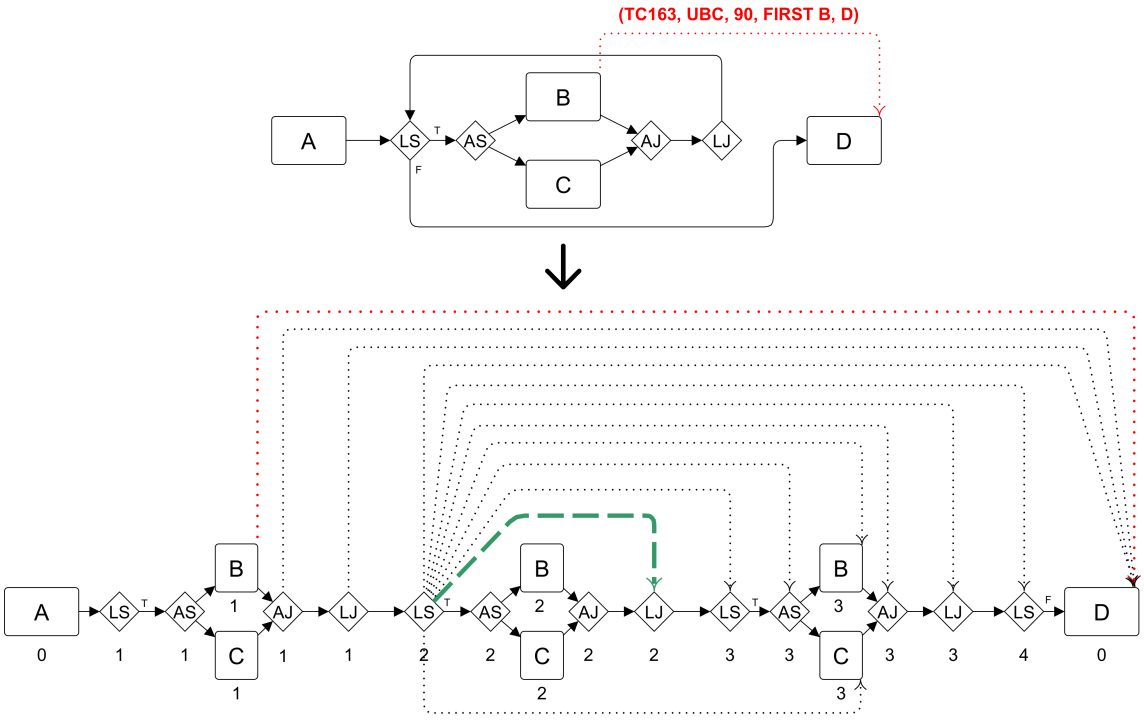


Figure 5.23: Example 7 - positive termination check

There is only a finite number of loop iterations such that the process satisfies the ETC $TC163$. Therefore, there is also a finite number of Instance Types of the given process that satisfy the ETC. This is the case because each Instance Type, in which the loop is entered, has an occurrence of B (and C) in the first loop iteration. This one binds the loop, since after that the execution time is limited.

Example 8: (TC164, UBC, 90, FIRST B, D) - negative termination check

The process from figure 5.24 does not have to terminate in order to satisfy the ETC $TC164$. In the resulting 3-iPG, we can see the complete closure of the resulting ATC ($TC164$, UBC, 90, B_1, D_0). In the closure, we can observe that all inferred ATCs result from moving of the destination D_0 to the left. The source B_1 cannot be moved to the right because there is no ATC source in the false-branch of the XOR-block (see $CASE\,3$ of the function $atcClosure(atc, I)$). The ATC between LS_2 and LJ_2 cannot be inferred from the ATC between B_1 and D_0, therefore the loop in the given process can iterate infinitely and the process is still in compliance with ETC $TC164$.

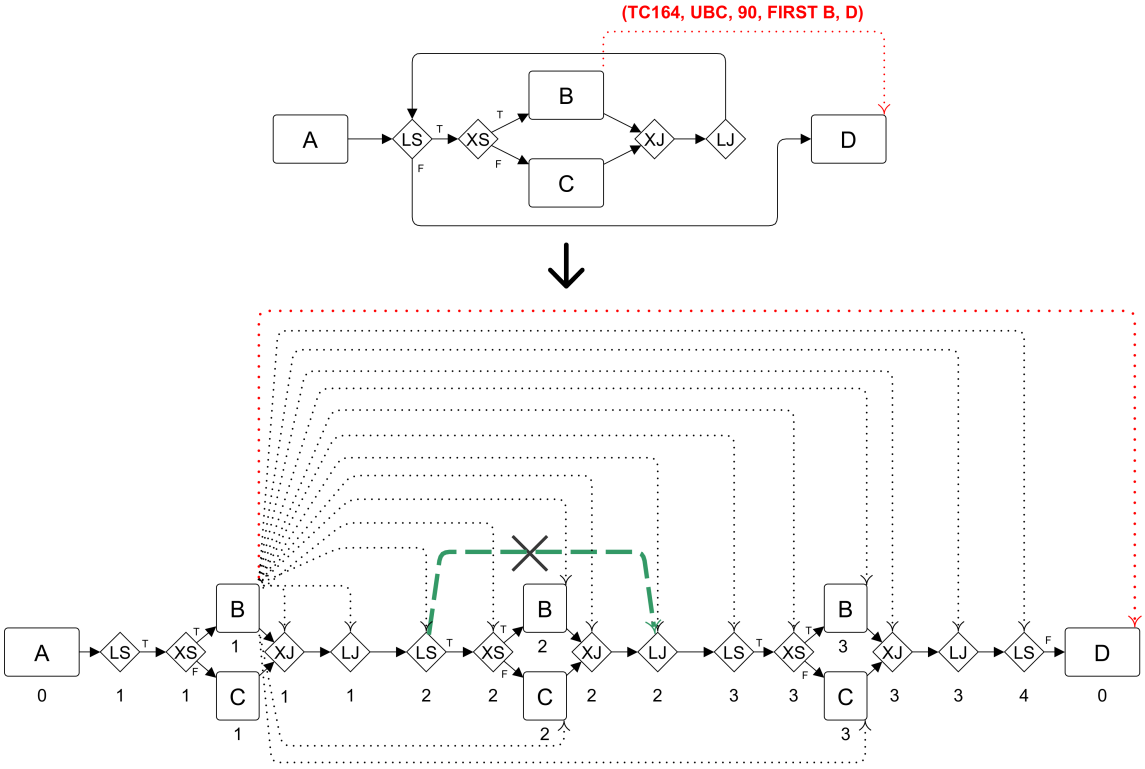


Figure 5.24: Example 8 - negative termination check

The process satisfies the ETC $TC164$ even if the number of loop iterations is infinite. Therefore, there is also an infinite number of Instance Types of the given process that satisfy the ETC. This is the case because an Instance Type can have an infinite number of loop iterations without any occurrence of B (in each iteration, the false-branch (C) can be executed, instead of the true-branch (B)). In this case, there is not even a derived ATC in an Instance Type. An Instance Type can also have an infinite number of iterations with occurrence of C and in the $\infty + 1$ iteration, the first B occurs. In this case, the ETC $TC164$ will be satisfied, although the number of iterations is $\infty + 1$.

Example 9: (TC165, UBC, 90, A, FIRST B) and (TC164, UBC, 90, FIRST B, D) - negative termination check

The process from figure 5.25 does not have to terminate in order to satisfy the ETCs $TC164$ and $TC165$. In the resulting 3-iPG, we can see the complete closure of the resulting ATCs ($TC164$, UBC, 90, B_1, D_0) and ($TC165$, UBC, 90, A_0, B_1). In the closure of ATC $TC164$, we can observe that all inferred ATCs result from moving of the destination D_0 to the left. The source B_1 cannot be moved to the right because there is no ATC source in the false-branch of the XOR-block (see $CASE\,3$ of the function $atcClosure(atc, I)$). In the closure of ATC $TC165$, all inferred ATCs result from moving of the source A_0 to the right. The destination B_1 cannot be moved to the left because there is no ATC destination in the false-branch of the XOR-block (see $CASE\,4$ of the function $atcClosure(atc, I)$). The ATC between LS_2 and LJ_2 cannot be inferred from the ATC between B_1 and D_0, nor from the ATC between A_0 and B_1, therefore the loop in the given process can iterate infinitely and the process is still in compliance with both ETCs.

There is an infinite number of Instance Types of the given process that satisfy the ETC. This is the case because an Instance Type can have an infinite number of loop iterations without any occurrence of B (in each iteration, the false-branch (C) can be executed, instead of the true-branch (B)). In this case, there are no derived ATCs in the Instance Types, therefore no ATCs have to satisfied at all.

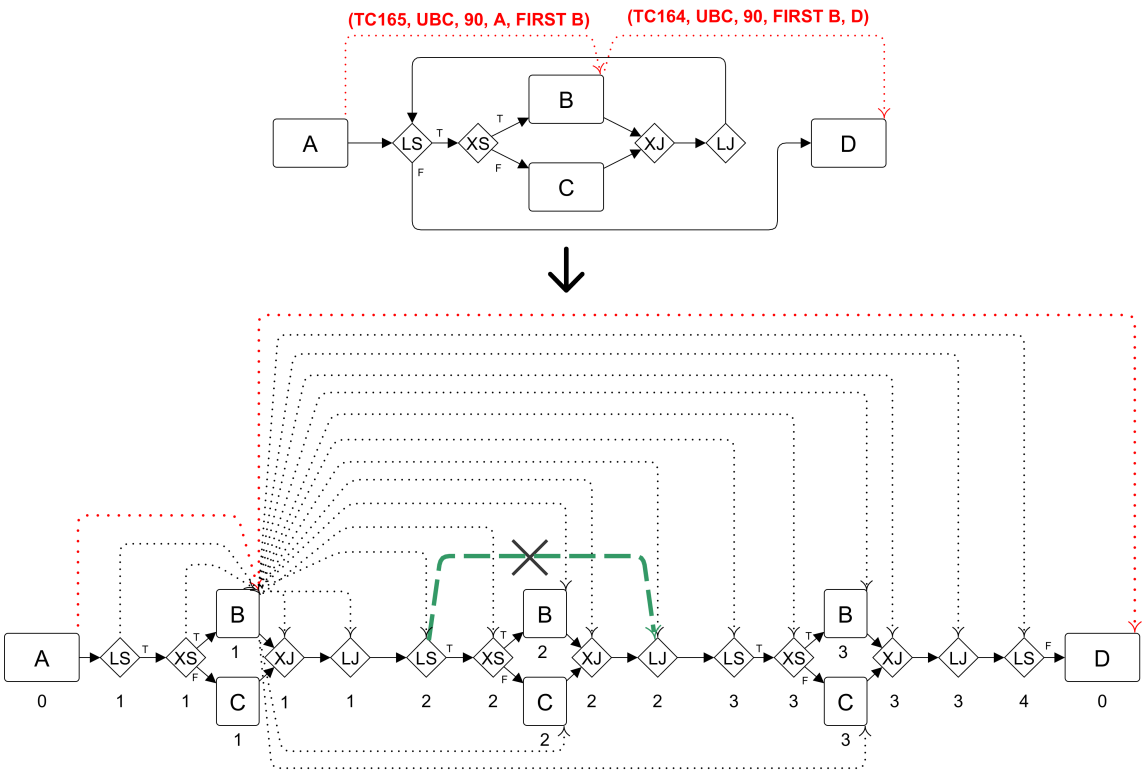


Figure 5.25: Example 9 - negative termination check

Example 10: (TC164, UBC, 90, FIRST B, D) and (TC166, UBC, 90, FIRST C, D) - positive termination check

In contrast to the previous example, the process from figure 5.26 has to terminate in order to satisfy the ETCs $TC164$ and $TC166$. In the resulting 3-iPG, we can see a part of the closure of the resulting ATCs ($TC164$, UBC, 90, B_1, D_0) and ($TC166$, UBC, 90, C_1, D_0). In contrast to the previous example, we can not obtain the inferred ATCs only by moving the destination (D_0) to the left, but also by moving the source (B_1 or C_1) to the right. This is enabled by $CASE\,3$ of the function $atcClosure(atc, I)$). The ATC between LS_2 and LJ_2 can be inferred from the ATC between B_1 and D_0, and the ATC between C_1 and D_0, therefore the loop in the given process cannot iterate infinitely without violating the given ETCs.

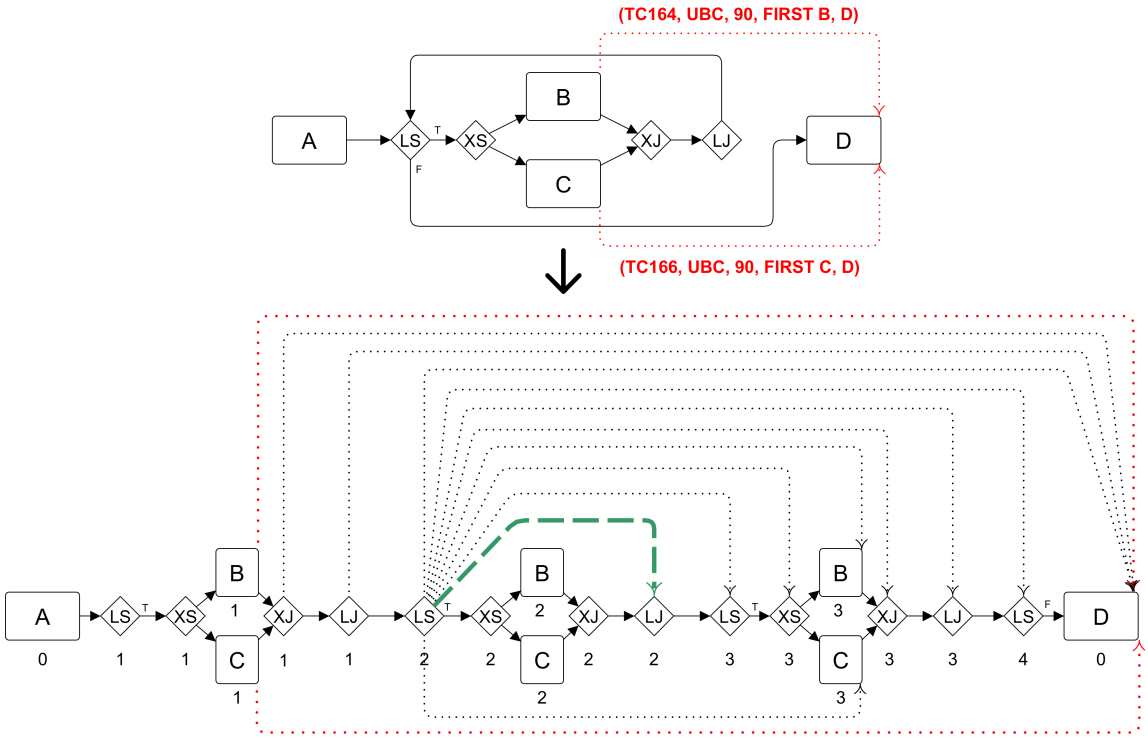


Figure 5.26: Example 10 - positive termination check

There is only a finite number of loop iterations such that the process satisfies the ETCs $TC164$ and $TC166$. Therefore, there is also a finite number of Instance Types of the given process that satisfy both ETCs. This is the case because each Instance Type, in which the loop is entered, has either an occurrence of B (true-branch of the XOR-block), or an occurrence of C (false-branch of the XOR-block) in the first loop iteration. Since both possible cases, $FIRST\,B$ and $FIRST\,C$, are constrained with an ETC, the first loop iteration (no matter which XOR-branch is executed) binds the loop, since after that the execution time is limited by the ETCs.

Example 11: (TC164, UBC, 90, FIRST B, D) and (TC167, UBC, 90, LAST C, D) - negative termination check

The process from figure 5.27 does not have to terminate in order to satisfy the ETCs $TC164$ and $TC167$. In the resulting 3-iPG, we can see the complete closure of the resulting ATCs ($TC164$, UBC, 90, B_1, D_0) and ($TC167$, UBC, 90, C_3, D_0). In the closure of ATC $TC164$, we can observe that all inferred ATCs result from moving of the destination D_0 to the left. The source B_1 cannot be moved to the right because there is no ATC source in the false-branch of the first XOR-block (see $CASE\,3$ of the function $atcClosure(atc, I)$). The same applies for the ATC $TC167$: there is no ATC source in the true-branch of the last XOR-block (see $CASE\,3$ of the function $atcClosure(atc, I)$). The ATC between LS_2 and LJ_2 cannot be inferred from the ATC between B_1 and D_0, nor from the ATC between C_3 and D_0, therefore the loop in the given process can iterate infinitely and the process is still in compliance with both ETCs.

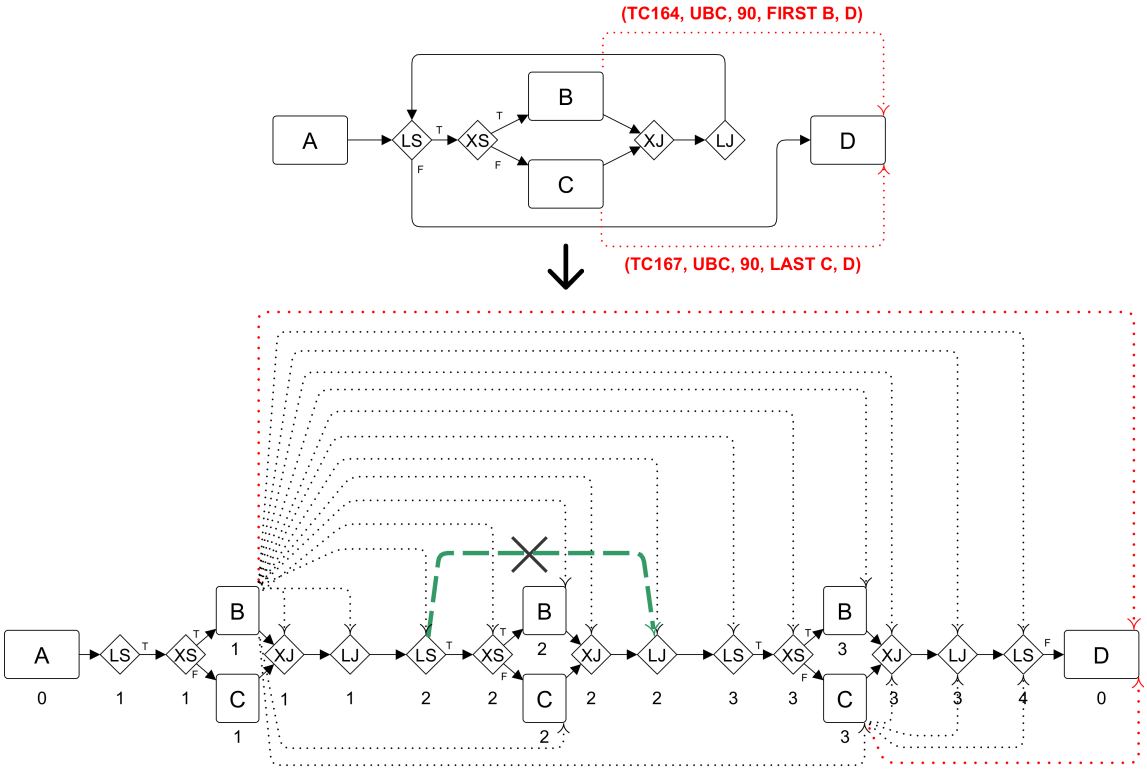


Figure 5.27: Example 11 - negative termination check

The process satisfies both ETCs, even if the number of loop iterations is infinite. Therefore, there is also an infinite number of Instance Types of the given process that satisfy the ETCs. This is the case because an Instance Type can have an infinite number of loop iterations without any occurrence of B (in each iteration, the false-branch (C) can be executed, instead of the true-branch (B)). In this case, the ETC $TC167$ between the last C and D will always be satisfied - no matter how many loop iterations there are - and the ETC $TC164$ would not mirror in any ATC at all.

Example 12: (TC168, UBC, 90, FIRST C, D) - negative termination check

The process from figure 5.28 is very similar to the process in figure 5.24. Instead of an XOR-block in the loop, there is an inner loop. In both processes, there is another block in the loop that involves uncertainty. Just as the process from figure 5.24, this process does not have to terminate in order to satisfy the ETC $TC168$.

In the resulting 3-iPG, we can see a part of the closure of the resulting ATC ($TC168$, UBC, 90, C_{11}, D_0). In the closure, we can observe that the source C_{11} can only be moved to the right until $LJ2_{13}$. It cannot be moved further, because there is no ATC with the source $LS2_{11}$ and a destination $LJ1_1$ or some other node after $LJ1_1$ (see $CASE\,7$ of the function $atcClosure(atc, I)$). The destination D_0 can be moved all the way to the left until $LJ2_{11}$ has been reached (see $CASE\,6$ and $CASE\,8$ of the function $atcClosure(atc, I)$).

Because the source can only be moved to the right until $LJ2_{13}$, only the ATC between $LS2_{12}$ and $LJ2_{12}$ can be inferred from the ATC between C_{11} and D_0. The ATC between $LS2_{22}$ and $LJ2_{22}$, the ATC between $LS2_{32}$ and $LJ2_{32}$, and the ATC between $LS1_2$ and $LJ1_2$ cannot be inferred. Therefore, the loop in the given process can iterate infinitely and the process is still in compliance with the ETC $TC168$.

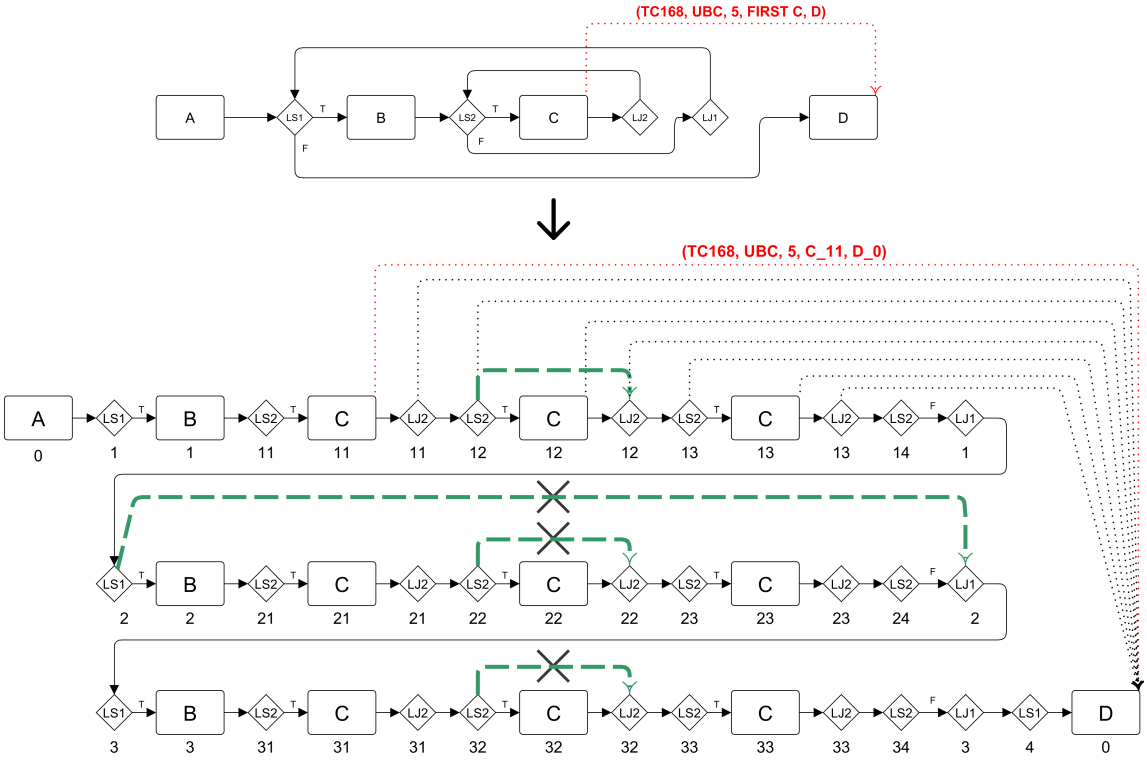


Figure 5.28: Example 12 - negative termination check

The process satisfies the ETC $TC168$ even if the number of loop iterations is infinite. Therefore, there is also an infinite number of Instance Types of the given process that satisfy the ETC. This is the case because an Instance Type can have an infinite number of loop iterations without any occurrence of C (in each iteration of the outer loop, the inner loop (C) can be skipped). In this case, there is not even a derived ATC in an Instance Type. An Instance Type can also have an infinite number of iterations of the outer loop without any occurrence of C and in the $\infty + 1$ iteration of the outer loop, for the first (and last) time the inner loop with activity C is executed. In this case, the ETC $TC168$ will be satisfied, although the number of iterations is $\infty + 1$.

Example 13: (TC169, UBC, 90, A, FIRST C) and (TC168, UBC, 90, FIRST C, D) - negative termination check

The process from figure 5.29 is very similar to the process in figure 5.25. Instead of an XOR-block in the loop, there is an inner loop. In both processes, there is another block in the loop that involves uncertainty. Just as the process from figure 5.25, this process does not have to terminate in order to satisfy the ETCs $TC168$ and $TC169$.

In the resulting 3-iPG, we can see a part of the closure of the resulting ATC ($TC168$, UBC, 90, C_{11}, D_0) and the complete closure of the ATC ($TC169$, UBC, 90, A_0, C_{11}). In the closure of ATC $TC169$, we can observe that only the source A_0 can be moved to the right. The destination C_{11} cannot be moved to the left because there is no (inferred) ATC between a predecessor of C_{11} and LS_{14} (see $CASE\,8$ of the function $atcClosure(atc, I)$).

In the closure of ATC $TC168$, we can observe that the source C_{11} can only be moved to the right until $LJ2_{13}$. It cannot be moved further, because there is no ATC with the source $LS2_{11}$ and a destination $LJ1_1$ or some other node after $LJ1_1$ (see $CASE\,7$ of the function $atcClosure(atc, I)$). The destination D_0 can be moved all the way to the left until $LJ2_{11}$ has been reached (see $CASE\,6$ and $CASE\,8$ of the function $atcClosure(atc, I)$).

Because the source of ATC $TC168$ can only be moved to the right until $LJ2_{13}$, only the ATC between $LS2_{12}$ and $LJ2_{12}$ can be inferred from the ATC between C_{11} and D_0. The ATC between $LS2_{22}$ and $LJ2_{22}$, the ATC between $LS2_{32}$ and $LJ2_{32}$, and the ATC between $LS1_2$ and $LJ1_2$ cannot be inferred. Therefore, the loop in the given process can iterate infinitely and the process is still in compliance with the ETC $TC168$.

The process is in compliance with both ETCs, even if the number of loop iterations is infinite. Therefore, there is also an infinite number of Instance Types of the given process that are in compliance with the ETCs. This is the case because an Instance Type can have an infinite number of loop iterations without any occurrence of C (in each iteration of the outer loop, the inner loop (C) can be skipped). In this case, there are no derived ATCs in the Instance Types, therefore no ATCs have to be satisfied at all.

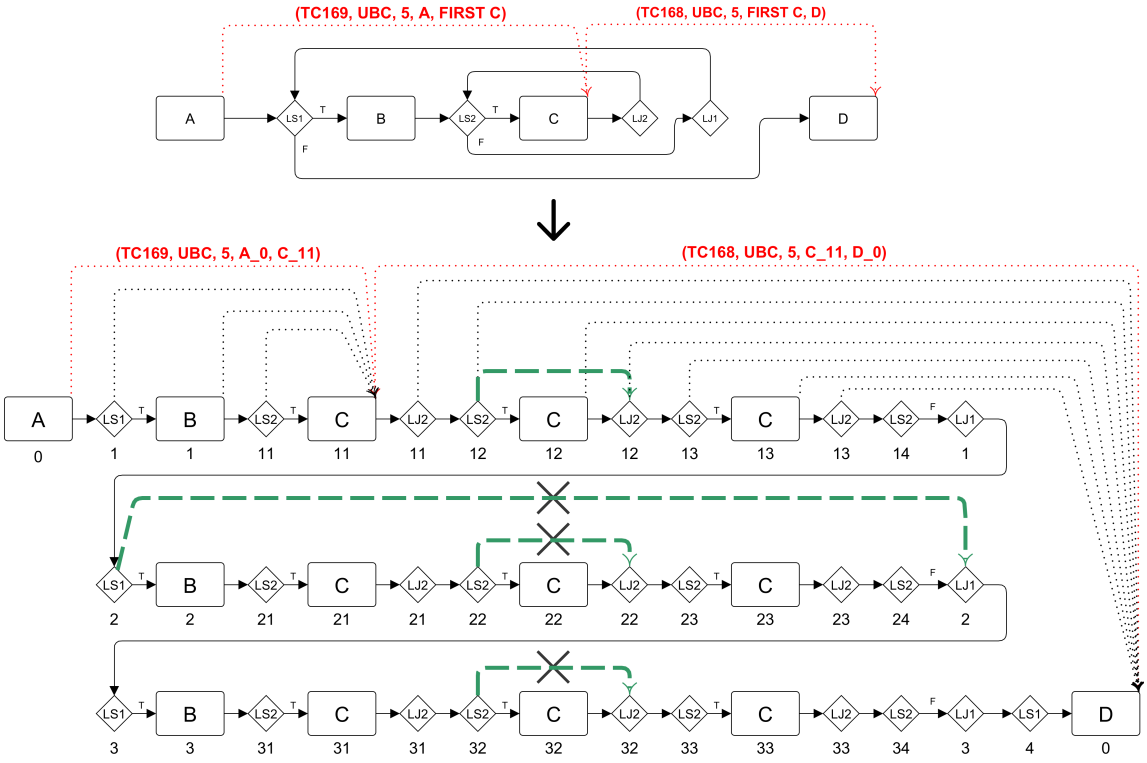


Figure 5.29: Example 13 - negative termination check

Example 14: a set of ETCs - negative termination check

In this final example, we check the termination property of the process from figure 5.7. In figure 5.30, inferred ATCs that are required for a positive termination check are added to the figure 5.7. Other inferred ATCs that are in closure of the resulting ATCs in the 3-iPG are not included in the figure 5.30 due to a better readability. Now let us check each ETC, its resulting ATCs and the closure of the resulting ATCs.

(TC155, UBC, 5, A, FIRST B)

The ETC $TC155$ is transformed into the following ATC: ($TC155$, UBC, 5, A_0, B_1). The closure of this ATC contains only one inferred ATC between $LS1_1$ and B_1. The ATC between A_0 and $LS1_1$ cannot be inferred, because there is no (inferred) ATC between A_0 and $LS1_4$ in any ATC closure (see $CASE\,8$ of $atcClosure(atc, I)$). However, there are no LS- and LJ-nodes between the source A_0 and the destination B_1 of the ATC $TC155$, so the ATC cannot bind any loop anyway.

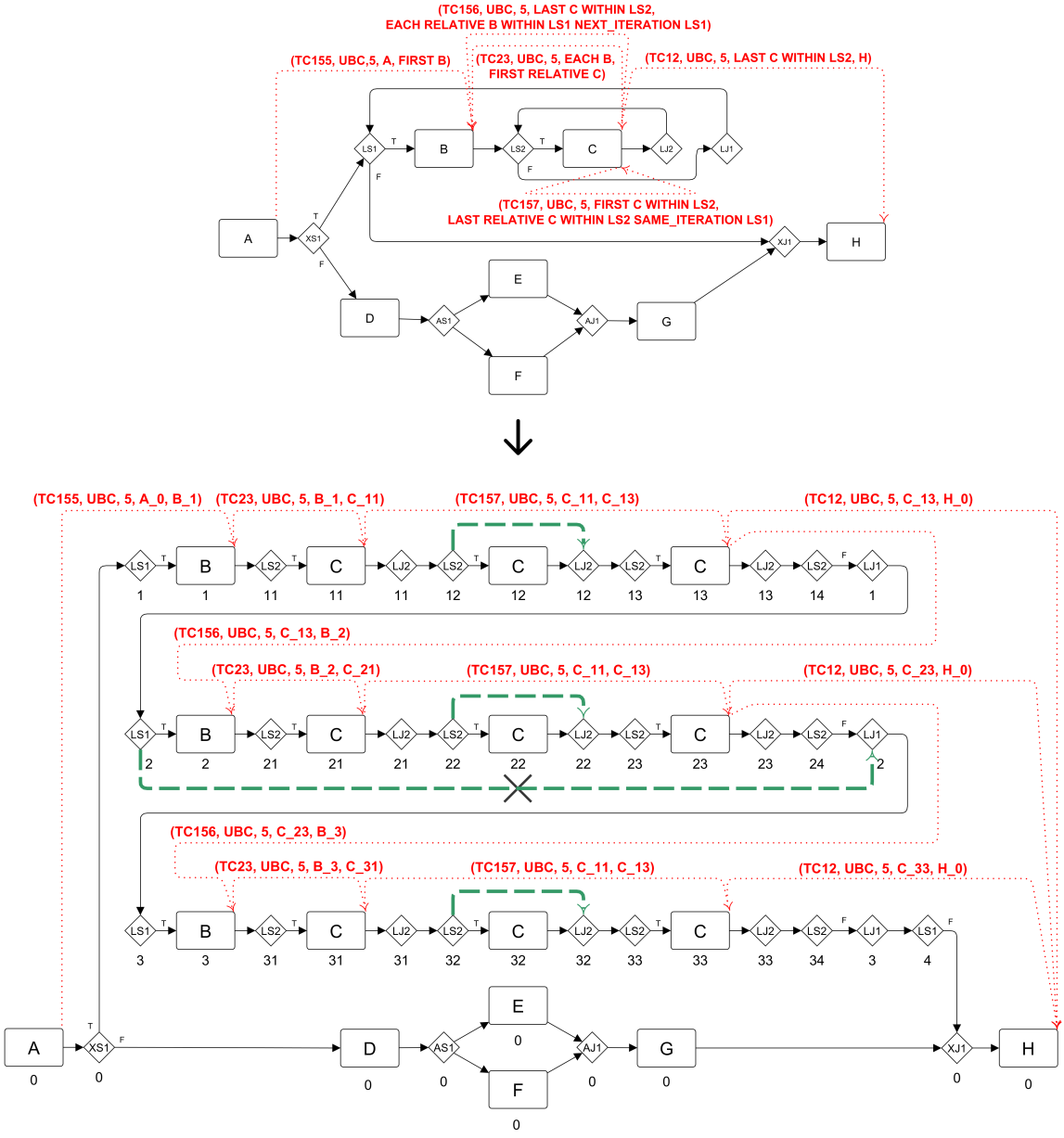


Figure 5.30: Example 14 - negative termination check

(TC156, UBC, 5, LAST C WITHIN LS2, EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1)

The ETC $TC156$ is transformed into the following ATCs: ($TC156$, UBC, 5, C_{13}, B_2), and ($TC156$, UBC, 5, C_{23}, B_3). Both ATCs do not embrace any LS- and LJ-node-pairs that are required for a positive termination check, so the ATCs cannot bind any loop.

(TC23, UBC, 5, EACH B, FIRST RELATIVE C)

The ETC $TC23$ is transformed into the following ATCs: ($TC23$, UBC, 5, B_1, C_{11}), ($TC23$, UBC, 5, B_2, C_{21}), and ($TC23$, UBC, 5, B_3, C_{31}). The resulting ATCs are not placed around any LS- and LJ-node-pairs that are required for a positive termination check, so the ATCs cannot bind any loop.

(TC12, UBC, 5, LAST C WITHIN LS2, H)

The ETC $TC12$ is transformed into the following ATCs: ($TC12$, UBC, 5, C_{13}, H_0), ($TC12$, UBC, 5, C_{23}, H_0), and ($TC12$, UBC, 5, C_{33}, H_0). In this case, the resulting ATCs are placed around three LS- and LJ-node-pairs ($LS1_2$ - $LJ1_2$, $LS2_{22}$ - $LJ2_{22}$, and $LS2_{32}$ - $LJ2_{332}$) that are required for a positive termination check. However, these required ATCs between LS- and LJ-nodes cannot be inferred from the resulting ATCs $TC12$. The destination of all three ATCs can indeed be moved to the left until direct successor of the source of the respective ATC is reached. However, the source can only be moved to the right one time to the $LJ2_{13}$ (and respectively $LJ2_{23}$ and $LJ2_{33}$), but not further because there is no (inferred) ATC with the source $LS2_{11}$ (and respectively $LS2_{21}$ and $LS2_{31}$) and a destination that is a successor of $LS2_{14}$ (and respectively $LS2_{24}$ and $LS2_{34}$) - see $CASE\,7$ of $atcClosure(atc, I)$.

Because the source is stuck and cannot be moved to the right often enough to obtain the required ATCs between LS- and LJ-nodes, the ATCs $TC12$ do not bind any loop.

(TC157, UBC, 5, FIRST C WITHIN LS2, LAST RELATIVE C WITHIN LS2 SAME_ITERATION LS1)
The ETC $TC157$ is transformed into the following ATCs: ($TC157$, UBC, 5, C_{11}, C_{13}), ($TC157$, UBC, 5, C_{21}, C_{23}), and ($TC157$, UBC, 5, C_{31}, C_{33}). Also in this case, the resulting ATCs are placed each around an LS- and LJ-node-pair ($LS2_{12}$ - $LJ2_{12}$, $LS2_{22}$ - $LJ2_{22}$, and $LS2_{32}$ - $LJ2_{32}$) that are required for a positive termination check. Here, the required ATCs between LS- and LJ-nodes can be inferred from the resulting ATCs $TC157$. The source can be moved all the way to the right (see $CASE\,1$ and $CASE\,5$ of $atcClosure(atc, I)$) and the destination can be moved all the way to the left (see $CASE\,2$ and $CASE\,6$ of $atcClosure(atc, I)$. Therefore, 3 of 4 required ATCs between LS- and LJ-nodes can be inferred. Thereby, the inner loop is bounded.

Even though at first sight it looks like this process must terminate in order to satisfy all those ETCs, the process can actually be in compliance with the ETCs, even if it does not terminate. The ETC $TC157$ indeed binds the inner loop, however none of the ETCs binds the outer loop (ATC between $LS1_2$ and $LJ1_2$ cannot be inferred from any ATC). The outer loop can iterate forever, if the inner loop is skipped in each iteration. Therefore, there is an infinite number of Instance Types that all satisfy the ETC $TC155$ and all other ETCs do not have to be satisfied at all, since the inner loop is skipped in all those Instance Types.

If there was one single ETC ($TC169$, UBC, 5, $FIRST\,B$, H) instead of the given set of ETCs, the process would have to terminate in order to satisfy it. As soon as the loop would be entered, activity B would be executed and so the process execution time (and thus the number of iterations of both loops) would be limited.

In this chapter, we introduced the Termination Check, which is a novel approach to test if a cyclic process must terminate in order to satisfy all Extended Time Constraints (ETCs). We constructed several examples that cover different ETC patterns as listed in table 5.1. These examples underline that, in general, a cyclic process must terminate in order to satisfy all ETCs if each loop is bounded by at least one ETC. Such bounding ETC could be a process deadline (translated into an upper bound constraint (UBC) between the start and the end node), an UBC between an activity before the loop and an activity after the loop, and a few more that interfere with the loop itself. Temporal constraints between activities out of a loop (that do not span over the loop) are not able to bind the loop. However, lower bound constraints and task durations do have an impact on the number of maximal loop iterations, in case the loop is bounded by other ETCs.

As a pre-step, Termination Check enables further time management steps, since it can help to sort out the potentially infinite processes. It consists of three steps: process transformation, time constraints inference, and termination check. We described each of these steps in this chapter and delivered several examples of processes with different structures and different Extended Time Constraints.

After the Termination Check, there are still a few steps to go before existing time management algorithms (e.g. Controllability Check by Combi *et al.* in [CHP13]) can be applied. E.g., a cyclic process must first be unfolded into an acyclic process, where loops are transformed into nested XORs. The unfolding process itself must consider all ETCs that bind the loops, as well as all other time constraints and task durations in the process. At the same time, the unfolding process must cope with the problem of which loop gets unfolded and how often. The problem seems to be very complex and could be solved by an exploration of a search space that consists of unfolded (acyclic) processes with different numbers of loop iterations for each loop. The solution to this problem would add the last missing puzzle piece on the way to use existing time management algorithms on cyclic processes and is the main focus of the future work.

In the next chapter, we deliver a proof of concept for all Termination Check steps that we described in this chapter, as well as all necessary predicates that we introduced in chapter 4.

Chapter 6

Prototypical Implementation

In this chapter, we deliver a proof of concept for the Termination Check that we introduced in the previous chapter, as well as all necessary predicates and functions that we introduced in chapter 4. For the construction of the prototype, we used Answer Set Programming (ASP) that we briefly describe in section 6.1. The entire code can be found in Appendices A, B, C, D, and E. The code overview is given in section 6.2 and the evaluation of the prototype is summed up in section 6.3.

6.1 Answer Set Programming

Answer Set Programming (ASP) is a form of declarative programming that is tailor-made for problem solving in the field of Knowledge Representation and Reasoning. In contrast to imperative programming (procedural programming, object-oriented programming), where the path to the problem solution (control flow) is described, a declarative program expresses the problem itself. In ASP, the problem is formally represented in the syntax of (first-order) logic programs that consist of facts (in our case the input process) and rules (in our case the rules from chapters 4 and 5). In the solving process, the represented problem is first grounded (translated into propositional logic program) by the grounder. Afterwards, the solver calculates stable models (answer sets) from which the problem solution is extracted.[GKKS12, Lif08]

For the grounding and solving of our Termination Check prototype, we used version 5 of the tool `clingo`[GKK+16], which integrates the grounder `gringo` and solver `clasp`. `clingo` can be downloaded as a part of the Potassco[GKK+11, oP] suite of ASP systems at `potassco.org`.

6.2 Prototype Overview

The prototype is subdivided into the following 5 files: `facts.dl`, `rulesBasic.dl`, `rulesProcessTransformation.dl`, `rulesTCInference.dl`, and `rulesTermination-Check.dl`. Each file is represented in an appendix. We briefly describe each file in following paragraphs.

facts.dl
The file `facts.dl` contains all facts (rules without a body) and represents the input process that shall be checked if it terminates or not. The input process (facts) consists of a set of nodes, a set of edges, and a set of Extended Time Constraints (ETCs). In the facts file in appendix, we use the last example from chapter 5 (figure 5.30) as the input process (facts). This particular input process does not have to terminate in order to satisfy all given ETCs.

rulesBasic.dl
The file `rulesBasic.dl` contains all basic predicates that are required for the Termination Check. Most of them are directly adapted from chapter 4, e.g. direct predecessor $dpred(X, Y)$ or $counterpart(LS, LJ)$. Some of them cannot be adapted directly due to the restrictions in ASP, however they are semantically equivalent.

rulesProcessTransformation.dl
The file `rulesProcessTransformation.dl` contains all required rules for the transformation of an input process into a 3-iPG as described in section 5.1. It contains 3 sections: transformation of nodes, transformation of edges, and transformation of time constraints.

The function $\xi(expr, tc, I)$ and the atomization function $atomize(tc, I)$ from chapter 4 are implemented in this file. Due to the restrictions in ASP, the functions are combined together and translated into predicates, however, they are semantically equivalent to those in chapter 4.

rulesTCInference.dl
The file `rulesTCInference.dl` contains all time constraints inference rules as introduced in section 5.2.

rulesTerminationCheck.dl
Finally, the file `rulesTerminationCheck.dl` contains the termination check rules as defined by the predicate $terminationCheck(I)$ in section 5.3. It checks if there is an inferred Atomic Time Constraint (ATC) between the LOOP-split node and its counterpart LOOP-join node of each second loop iteration. If this is the case, the program determines that the process must terminate in order to satisfy all ETCs ($process\_terminates$). Otherwise, it determines that it does not have to terminate and can still satisfy all ETCs ($process\_not\_terminates$).

6.3 Prototype Evaluation

We tested the prototype with all examples that we introduced in chapter 5 as inputs. The tests were performed on a computer with an Intel(R) Core(TM) i7 CPU and 16 GB RAM. The code was executed in Sublime Text 3 Editor. Each input was tested 3 times with the `--quiet` option and 3 times with the `--text` option. If `--text` option is set, the ground program is displayed in text format in the console and if `--quiet` option is set, the ground program is not displayed in the console, which speeds up the execution and is useful for benchmarking.

We deliver an overview an overview of all test inputs and their median execution times with the `--quiet` option as well as with the `--text` option in table 6.1. We can observe that the execution time grows with the growing process complexity, the number of nested loops, and the number of ETCs. However, for our examples, the execution time never exceeded 1 second, therefore we conclude that the approach is feasible for the majority of cyclic processes.

Example No.	ETC	ETC Pattern Description	Termination Check	Median Execution Time (`--quiet`)	Median Execution Time (`--text`)
1	(TC158, UBC, 90, A, C)	ETC over loop	positive	0.05 s	0.2 s
2	(TC154, UBC, 90, FIRST B, C)	ETC from loop	positive	0.05 s	0.2 s
3	(TC159, UBC, 90, A, LAST B)	ETC into loop	positive	0.05 s	0.2 s
4	(TC160, UBC, 90, A, FIRST B)	ETC into loop	negative	0.04 s	0.2 s
5	(TC161, UBC, 90, LAST B, C)	ETC from loop	negative	0.05 s	0.2 s
6	(TC162, UBC, 90, FIRST B, LAST B)	ETC in loop	positive	0.05 s	0.2 s
7	(TC163, UBC, 90, FIRST B, D)	ETC from nested AND	positive	0.08 s	0.2 s
8	(TC164, UBC, 90, FIRST B, D)	ETC from nested XOR	negative	0.07 s	0.3 s
9	(TC165, UBC, 90, A, FIRST B) (TC164, UBC, 90, FIRST B, D)	ETC into nested XOR ETC from nested XOR	negative	0.06 s	0.3 s
10	(TC164, UBC, 90, FIRST B, D) (TC166, UBC, 90, FIRST C, D)	ETC from nested XOR ETC from nested XOR	positive	0.08 s	0.3 s
11	(TC164, UBC, 90, FIRST B, D) (TC167, UBC, 90, LAST C, D)	ETC from nested XOR ETC from nested XOR	negative	0.06 s	0.2 s
12	(TC168, UBC, 90, FIRST C, D)	ETC from nested XOR	negative	0.20 s	0.4 s
13	(TC169, UBC, 90, A, FIRST C) (TC168, UBC, 90, FIRST C, D)	ETC into nested loop ETC from nested loop	negative	0.21 s	0.3 s
14	(TC155, UBC, 5, A, FIRST B) (TC156, UBC, 5, LAST C WITHIN LS2, EACH RELATIVE B WITHIN LS1 NEXT_ITERATION LS1) (TC23, UBC, 5, EACH B, FIRST RELATIVE C) (TC12, UBC, 5, LAST C WITHIN LS2, H) (TC157, UBC, 5, FIRST C WITHIN LS2, LAST RELATIVE C WITHIN LS2 SAME_ITERATION LS1)	ETC into loop ETC from nested loop ETC into nested loop ETC from nested loop ETC in nested loop	negative	0.34 s	0.5 s

Table 6.1: Prototype execution time evaluation

Chapter 7

Conclusions and Future Work

Process Time Management has been researched for decades, however, processes with loops never gained a lot of attention. In our research, we focused entirely on processes with loops and dealt with the following two research question:

RQ1: How can we define time constraints in a cyclic process?

RQ2: How can we check the controllability of a cyclic process?

The answer to the RQ1 are Extended Time Constraints (ETCs) that we introduced in chapter 4. In this part of our research, we first elaborated a narrowed down set of the most reasonable cases of time constraints that have an impact on activities that are placed in a loop. We introduced Extended Time Constraints that can represent those cases and defined a general syntax of an ETC and the syntax of the expressions that we use to define the set of source nodes as well as the set of destination nodes in an ETC. Furthermore, we provided the semantics of the source and destination expressions that specify the source and destination node sets in Instance Types. Finally the atomization function for the instantiation of ETCs into Atomic Time Constraints (ATCs) was introduced at the end of the same chapter.

The RQ2 turned out to be much more complicated than we expected. Our original approach was to unfold the loops step by step, as long as the unfolded process graph with the given time constraints is controllable[1]. The unfolding would have stopped as soon as a time constraint would have been violated. Each unfold step should have consisted of transforming of the loop into a nested XOR-block, calculating the finishing times, and instantiating of ATCs from the given ETCs. The satisfiability of instantiated ATCs and the controllability of the unfolded process graph should have been proven continuously in each unfold step. However, the nature of some ETCs, and the uncertainties of XOR-blocks and loops (nested loops behave similarly to XOR-blocks) make this approach useless, since the unfolding process could have run forever. The extension of the loop unfold step with a simultaneous XOR-unfold step would not have changed the underlying problem. In order to make the unfolding and time calculation work, we had to find out a way to check beforehand if the specified ETCs temporally bind all loops in a given process or not. We defined a sub-research question RQ2a:

RQ2a: How can we check if a cyclic process must terminate in order to satisfy all time constraints?

As we investigated the RQ2a, we came up with the idea of the Termination Check, inspired by the pumping lemma. In the Termination Check that we introduced in chapter 5, we transform a given process into a so-called 3-iterated Process Graph (3-iPG), where each loop is transformed into a sequence of 3 loop iterations. We instantiate ATCs of the given ETCs in a 3-iPG according to the atomization function and ETC semantics from chapter 4. To check if a process must terminate in order to satisfy all ATCs, we determine an ATC closure (that contains inferred ATCs) for each ATC. Finally, we check if the loop split and loop join nodes of each 2. iteration of each loop are constrained with an inferred ATC. If this is the case, the whole process must terminate in order to satisfy all ATCs/ETCs.

[1] In general, in a controllable temporal process it is possible to satisfy all temporal constraints for any possible duration of tasks that cannot be influenced by the agent (contingent links).[CP09][CP10][CGMP12]

With the Termination Check, we laid an important milestone for time management in processes with loops. The Termination Check brings us one step closer to be able to calculate the earliest and the latest finishing times of a process with loops and to check its controllability without running into problems of endlessness.

As a proof of concept, we implemented an ASP prototype that is able to interpret ETCs, transform them into ATCs, and to perform the Termination Check on any cyclic process with Extended Time Constraints. The output of this prototype can be used to preselect the processes for further time management steps. We tested the prototype with a set of different process inputs and observed that the Termination Check was performed within one second for all tested inputs.

Further time management steps in processes with loops are the matter of future work. The calculation of earliest and latest finishing times and the controllability check need to cope with interesting challenges, e.g. generation and exploration of a search space of unfolded acyclic processes that originate from a given cyclic process, and specification of suitable strategies for balancing the number of iterations between the loops in a process. With our contribution, the topic got even more interesting than it was before.

Appendix A

facts.dl

```
%============================================================================
%============================ process facts =================================
%============================================================================

%================================== nodes ===================================
%node(label,type,duration): represents a node with a label, type, and duration.
%There are the following node types: act=activity, as=AND-split, aj=AND-join,
%xs=XOR-split, xj=XOR-join, ls=LOOP-split, lj=LOOP-join.
node(a,act,10).
node(xs1,xs,0).
node(d,act,10).
node(as1,as,0).
node(e,act,10).
node(f,act,10).
node(aj1,aj,0).
node(g,act,10).
node(ls1,ls,0).
node(b,act,10).
node(ls2,ls,0).
node(c,act,10).
node(lj2,lj,0).
node(lj1,lj,0).
node(xj1,xj,0).
node(h,act,10).


```

```
%================================= edges ==================================
%edge(x,y,condition): represents a directed edge from node x to node y.
%The edges with condition null are always taken, whereas the edges with
%condition true are taken if the condition of the node x was evaluated true
%and the edges with condition false are taken if the condition of the node x
%was evaluated false.
edge(a,xs1,null).
edge(xs1,ls1,true).
edge(xs1,d,false).
edge(d,as1,null).
edge(as1,e,null).
edge(as1,f,null).
edge(e,aj1,null).
edge(f,aj1,null).
edge(aj1,g,null).
edge(g,xj1,null).
edge(ls1,b,true).
edge(ls1,xj1,false).
edge(b,ls2,null).
edge(ls2,c,true).
edge(ls2,lj1,false).
edge(c,lj2,null).
edge(lj2,ls2,null).
edge(lj1,ls1,null).
edge(xj1,h,null).


%======================= extended time constraints =========================
%etc(id, type, delta, source, destination): represents an Extended Time Constraint.
%source(quantifier, node_label, loop_reference): represents a source in an ETC.
%destination(quantifier, relation, node_label, loop_reference, iteration_reference,
%loop_label): represents a destination in an ETC.
etc(12,ubc,5,source(last, c, ls2),destination(null,null,h,null,null,null)).
etc(23,ubc,5,source(each, b, null),destination(first,relative,c,null,null,null)).
etc(155,ubc,5,source(null, a, null),destination(first,absolute,b,null,null,null)).
etc(156,ubc,5,source(last, c, ls2),
        destination(each,relative,b,ls1,next_iteration,ls1)).
etc(157,ubc,5,source(first, c, ls2),
        destination(last,relative,c,ls2,same_iteration,ls1)).
```

Appendix B

rulesBasic.dl

```
#include "facts.dl".

%============================================================================
%============================ basic process rules ============================
%============================================================================

%================================ node types ================================
act(X) :- node(X,act,_).    %type of node X is activity
xs(X) :- node(X,xs,_).      %type of node X is XOR-split
xj(X) :- node(X,xj,_).      %type of node X is XOR-join
as(X) :- node(X,as,_).      %type of node X is AND-split
aj(X) :- node(X,aj,_).      %type of node X is AND-join
ls(X) :- node(X,ls,_).      %type of node X is LOOP-split
lj(X) :- node(X,lj,_).      %type of node X is LOOP-join

act(node(X,act,XD)) :- node(X,act,XD).  %type of node X is activity
xs(node(X,xs,XD)) :- node(X,xs,XD).     %type of node X is XOR-split
xj(node(X,xj,XD)) :- node(X,xj,XD).     %type of node X is XOR-join
as(node(X,as,XD)) :- node(X,as,XD).     %type of node X is AND-split
aj(node(X,aj,XD)) :- node(X,aj,XD).     %type of node X is AND-join
ls(node(X,ls,XD)) :- node(X,ls,XD).     %type of node X is LOOP-split
lj(node(X,lj,XD)) :- node(X,lj,XD).     %type of node X is LOOP-join

tact(X) :- tnode(X,act,_,_).    %type of node X is activity
txs(X) :- tnode(X,xs,_,_).      %type of node X is XOR-split
txj(X) :- tnode(X,xj,_,_).      %type of node X is XOR-join
```

```
27 tas(X) :- tnode(X,as,_,_).      %type of node X is AND-split
28 taj(X) :- tnode(X,aj,_,_).      %type of node X is AND-join
29 tls(X) :- tnode(X,ls,_,_).      %type of node X is LOOP-split
30 tlj(X) :- tnode(X,lj,_,_).      %type of node X is LOOP-join
31
32 tact(tnode(X,act,XD,XC)) :- tnode(X,act,XD,XC). %type of node X is activity
33 txs(tnode(X,xs,XD,XC)) :- tnode(X,xs,XD,XC).    %type of node X is XOR-split
34 txj(tnode(X,xj,XD,XC)) :- tnode(X,xj,XD,XC).    %type of node X is XOR-join
35 tas(tnode(X,as,XD,XC)) :- tnode(X,as,XD,XC).    %type of node X is AND-split
36 taj(tnode(X,aj,XD,XC)) :- tnode(X,aj,XD,XC).    %type of node X is AND-join
37 tls(tnode(X,ls,XD,XC)) :- tnode(X,ls,XD,XC).    %type of node X is LOOP-split
38 tlj(tnode(X,lj,XD,XC)) :- tnode(X,lj,XD,XC).    %type of node X is LOOP-join
39
40
41 %================================== edges ===================================
42 %loop edge
43 %ledge(X,Y): represents a loop edge if the source X is a LOOP-JOIN-node
44 %and the destination a LOOP-SPLIT-node.
45 ledge(X,Y,null) :- node(X,lj,_), node(Y,ls,_), edge(X,Y,null).
46
47
48 %======================== predecessors & successors ========================
49 %direct predecessor
50 %dpred(X,Y): X is a direct predecessor of Y if there is an edge from X to Y.
51 dpred(X,Y) :- edge(X,Y,_).
52 %=====for transformed graph=====
53 tdpred(X,Y) :- tedge(X,Y,_).
54
55 %direct successor
56 %dsucc(X,Y): X is a direct successor of Y if there is an edge from Y to X.
57 dsucc(X,Y) :- edge(Y,X,_).
58 %=====for transformed graph=====
59 tdsucc(X,Y) :- tedge(Y,X,_).
60
61 %predecessor
62 %pred(X,Y): X is a predecessor of Y if X is a direct predecessor of Y or
63 %if there is a sequence of direct predecessors leading from X to Y.
64 pred(X,Y) :- dpred(X,Y).
65 pred(X,Y) :- dpred(X,Z), pred(Z,Y).
66 %=====for transformed graph=====
```

```
67 tpred(X,Y) :- tdpred(X,Y).
68 tpred(X,Y) :- tdpred(X,Z), tpred(Z,Y).
69
70 %non-loop-predecessor
71 %nlpred(X,Y): X is a non-loop-predecessor of Y if it is a predecessor
72 %in the graph without loop-edges.
73 nlpred(X,Y) :- dpred(X,Y), not ledge(X,Y,null).
74 nlpred(X,Y) :- dpred(X,Z), not ledge(X,Z,null), nlpred(Z,Y).
75
76 %successor
77 %succ(X,Y): X is a successor of Y if X is a direct successor of Y or if
78 %there is a sequence of direct successors leading from X to Y.
79 succ(X,Y) :- dsucc(X,Y).
80 succ(X,Y) :- dsucc(X,Z), succ(Z,Y).
81 %=====for transformed graph=====
82 tsucc(X,Y) :- tdsucc(X,Y).
83 tsucc(X,Y) :- tdsucc(X,Z), tsucc(Z,Y).
84
85 %non-loop-successor
86 %nlsucc(X,Y): X is a non-loop-successor of Y if it is a successor
87 %in the graph without loop-edges.
88 nlsucc(X,Y) :- dsucc(X,Y), not ledge(Y,X,null).
89 nlsucc(X,Y) :- dsucc(X,Z), not ledge(Z,X,null), nlsucc(Z,Y).
90
91
92 %================================== path ==================================
93 %path
94 %path(X,Y): there is a path from X to Y if X is a predecessor of Y
95 %or if Y is a successor of X.
96 path(X,Y) :- pred(X,Y).
97 path(X,Y) :- succ(Y,X).
98 %=====for transformed graph=====
99 tpath(X,Y) :- tpred(X,Y).
100 tpath(X,Y) :- tsucc(Y,X).
101
102 %non-loop-path
103 %nlpath(X,Y): there is a non-loop-path from X to Y if X is a non-loop-predecessor
104 %of Y or if Y is a non-loop-successor of X.
105 nlpath(X,Y) :- nlpred(X,Y).
106 nlpath(X,Y) :- nlsucc(Y,X).
```

```
107
108 %non-conditional path
109 %ncpath(X,Y): there is a non-conditional path from X to Y if all edges
110 %that lead from X to Y do not have a condition true or false.
111 ncpath(X,Y) :- edge(X,Y,null).
112 ncpath(X,Y) :- edge(X,Z,null), ncpath(Z,Y).
113
114
115 %============================== indegree & outdegree ========================
116 %indegree
117 %indeg(X,I): node X has an indegree I (I is the number of edges and conditional
118 %edges pointing to the node X).
119 indeg(X,I) :- node(X,_,_), #count{E: edge(E,X,_)}=I.
120 %=====for transformed graph=====
121 tindeg(X,I) :- tnode(X,_,_,_), #count{E: tedge(E,X,_)}=I.
122
123 %outdegree
124 %outdeg(X,O): node X has an outdegree O (O is the number of edges and conditional
125 %edges going out from node X).
126 outdeg(X,O) :- node(X,_,_), #count{E: edge(X,E,_)}=O.
127 %=====for transformed graph=====
128 toutdeg(X,O) :- tnode(X,_,_,_), #count{E: tedge(X,E,_)}=O.
129
130
131 %============================== start & end nodes ============================
132 %start node
133 %start(X): X is a start node if its indegree is 0
134 %(no edges point to the start node).
135 start(X) :- indeg(X,0).
136 %=====for transformed graph=====
137 tstart(X) :- tindeg(X,0).
138
139 %end node
140 %end(X): X is an end node if its outdegree is 0
141 %(no edges are going out from the end node).
142 end(X) :- outdeg(X,0).
143 %=====for transformed graph=====
144 tend(X) :- toutdeg(X,0).
145
146
```

```
147 %========================== counterpart nodes ==============================
148 %counterparts:
149 %In block structured processes each split node has a corresponding join node.
150 %The corresponding split and join nodes are counterpart nodes.
151 %LS, LJ
152 counterpart(LS,LJ) :- node(LS,ls,_), node(LJ,lj,_), edge(LJ,LS,_).
153 counterpart(LJ,LS) :- node(LS,ls,_), node(LJ,lj,_), edge(LJ,LS,_).
154
155 %XS, XJ
156 counterpart(XS,XJ) :- node(XS,xs,_), node(XJ,xj,_),
157                           #count{X: node(X,xs,_), succ(X,XS), pred(X,XJ)}=N,
158                           #count{Y: node(Y,xj,_), succ(Y,XS), pred(Y,XJ)}=N.
159 counterpart(XJ,XS) :- node(XS,xs,_), node(XJ,xj,_),
160                           #count{X: node(X,xs,_), succ(X,XS), pred(X,XJ)}=N,
161                           #count{Y: node(Y,xj,_), succ(Y,XS), pred(Y,XJ)}=N.
162
163 %AS, AJ
164 counterpart(AS,AJ) :- node(AS,as,_), node(AJ,aj,_),
165                           #count{X: node(X,as,_), succ(X,AS), pred(X,AJ)}=N,
166                           #count{Y: node(Y,aj,_), succ(Y,AS), pred(Y,AJ)}=N.
167 counterpart(AJ,AS) :- node(AS,as,_), node(AJ,aj,_),
168                           #count{X: node(X,as,_), succ(X,AS), pred(X,AJ)}=N,
169                           #count{Y: node(Y,aj,_), succ(Y,AS), pred(Y,AJ)}=N.
170
171 %=========== counterpart tnodes in a transformed process graph ==============
172 tcounterpart(tnode(SL,ST,SD,SC),tnode(JL,JT,JD,JC)) :-
173             tnode(SL,ST,SD,SC), tnode(JL,JT,JD,JC),
174             node(SL,ST,SD), node(JL,JT,JD), counterpart(SL,JL),
175             SC=JC.
176
177
178 %========================== nodes in a loop ================================
179 %inloop(X,LS): X is placed within a loop that starts with the node LS.
180 inloop(X,LS) :- node(X,_,_), node(LS,ls,_), node(LJ,lj,_),
181             nlpath(LS,X), path(X,LJ), counterpart(LS,LJ), X!=LS,
182             not counterpart(X,LS), not split_false_ncpath(LS,X).
183
184 %insomeloop(X): X is placed within at least one loop.
185 insomeloop(X) :- node(X,_,_), node(LS,ls,_), inloop(X,LS).
186
```

```
187 %split_false_ncpath(LS,X): There is a path between LS and X that starts with
188 %a false-edge, followed by non-conditional edges.
189 split_false_ncpath(LS,X) :- node(LS,ls,_), node(X,_,_), edge(LS,X,false).
190
191 split_false_ncpath(LS,X) :- node(LS,ls,_), node(X,_,_), node(Z,_,_),
192             edge(LS,Z,false), ncpath(Z,X).
```

Appendix C

rulesProcessTransformation.dl

```
#include "rulesBasic.dl".

%============================================================================
%==================== process transformation rules ==========================
%============================================================================

%====================== transformed nodes - tnodes ===========================

%tnode(label,type,duration,counter): represents a transformed node
%with a label, type, duration, and counter.

%Rule 1 - non-loop nodes:
%For each node node(L,T,D) in P that does not appear in a loop, and is not
%a LOOP-split or a LOOP-join node, there is a derived node tnode(L,T,D,C)
%in 3-iPG with the counter C = 0.
tnode(L,T,D,C) :-   node(L,T,D), not ls(L), not lj(L),
                    node(LS,ls,_), not insomeloop(L), C=0.

%Rule 2 - LOOP-split nodes:
%For each node tnode(P,PT,PD,PC) in 3-iPG that is equivalent to node(P,PT,PD)
%in P that is a direct predecessor of a LOOP-split node node(LS,ls,LSD) in P,
%and not a LOOP-join node, there are 4 derived nodes tnode(L,ls,D,C)
%with the counter C=PC*10+X, where X is an integer between 1 and 4.
tnode(L,ls,D,C) :-  node(L,ls,D),
                    node(P,PT,PD), dpred(P,L), not lj(P),
                    tnode(P,PT,PD,PC), C=TMP+X, TMP=PC*10, X=1..4.
```

```
27
28 %Rule 3 - LOOP-join nodes:
29 %For each LOOP-split node tnode(LS,ls,LSD,C) in 3-iPG with a counter C with
30 %a remainder 1, 2, or 3 of the division by 10, there is a counterpart
31 %LOOP-join node tnode(L,lj,D,C) in 3-iPG with the same counter.
32 tnode(L,lj,D,C) :-  node(L,lj,D),
33                     node(LS,ls,LSD), counterpart(L,LS),
34                     tnode(LS,ls,LSD,C), C\10=X, X=1..3.
35
36 %Rule 4 - nodes within a loop (except LOOP-split nodes, LOOP-join nodes,
37 %and direct successors of LOOP-split nodes):
38 %For each node tnode(P,PT,PD,C) in 3-iPG that is equivalent to node(P,PT,PD)
39 %in P, which is a direct predecessor of a node node(L,T,D) in P, there is a node
40 %tnode(L,T,D,C) in 3-iPG with the same counter as tnode(P,PT,PD,C)
41 %with a remainder 1, 2, or 3 of the division by 10.
42 %Node node(L,T,D) is placed in a loop node(LS,ls,_) and is neither a LOOP-split
43 %node nor a LOOP-join node. Neither is node(P,PT,PD) a LOOP-split node.
44 tnode(L,T,D,C) :-   node(L,T,D), not ls(L), not lj(L),
45                     node(LS,ls,_), inloop(L,LS),
46                     node(P,PT,PD), not ls(P), edge(P,L,_),
47                     tnode(P,PT,PD,C), C\10=X, X=1..3.
48
49 %Rule 5 - true-direct-successors of a LOOP-split node (except LOOP-split
50 %and LOOP-join nodes):
51 %For each node tnode(P,PT,PD,C) in 3-iPG that is equivalent to node(P,PT,PD)
52 %in P, which is a direct predecessor of a node node(L,T,D) in P, there is a node
53 %tnode(L,T,D,C) in 3-iPG with the same counter as tnode(P,PT,PD,C) with a
54 %remainder 1, 2, or 3 of the division by 10. Node node(L,T,D) is not a LOOP-split
55 %node and not a LOOP-join node, and node(P,PT,PD) is a LOOP-split node.
56 %Node node(L,T,D) is a true-direct-successor of node(P,PT,PD).
57 tnode(L,T,D,C) :-   node(L,T,D), not ls(L), not lj(L),
58                     node(LS,ls,_), inloop(L,LS),
59                     node(P,PT,PD), ls(P), edge(P,L,true),
60                     tnode(P,PT,PD,C), C\10=X, X=1..3.
61
62 %Rule 6 - false-direct-successors of a LOOP-split node within a loop
63 %(except LOOP-split and LOOP-join nodes):
64 %For each node tnode(PP,PPT,PPD,C) in 3-iPG that is equivalent to
65 %node(PP,PPT,PPD) in P, there is a node tnode(L,T,D,C) in 3-iPG with
66 %the same counter as tnode(PP,PPT,PPD,C). Node node(L,T,D) is neither
```

```
%a LOOP-split node nor a LOOP-join node. Node node(L,T,D) is placed
%in a loop, and is a false-direct-successor of node(P,ls,_).
%Node node(P,ls,_) is a LOOP-split node and node(PP,PPT,PPD) is
%not a LOOP-join node. Node node(P,ls,_) is a direct successor of
%node(PP,PPT,PPD).
tnode(L,T,D,C) :-   node(L,T,D), not ls(L), not lj(L),
                    node(LS,ls,_), inloop(L,LS),
                    node(P,ls,_), edge(P,L,false),
                    node(PP,PPT,PPD), not lj(PP), edge(PP,P,_),
                    tnode(PP,PPT,PPD,C).



%====================== transformed edges - tedges ==========================

%tedge(start,end,condition): there is a directed edge from the node start
%to the node end with a truth value condition (true or false).

%Rule 7 - edges between nodes with the same counter:
%For all nodes tnode(S,ST,SD,SC) and tnode(E,ET,ED,EC) in 3-iPG
%that are derived from node(S,ST,SD) and node(E,ET,ED) in P and have the same
%counter SC=EC, there is an edge tedge(tnode(S,ST,SD,SC),tnode(E,ET,ED,EC),C)
%in 3-iPG between them if tnode(S,ST,SD,SC) is not a LOOP-join node,
%and tnode(E,ET,ED,EC) is not a LOOP-split node, and if there is an edge between
%node(S,ST,SD) and node(E,ET,ED) in P.
tedge(tnode(S,ST,SD,SC),tnode(E,ET,ED,EC),C) :-
            tnode(S,ST,SD,SC), tnode(E,ET,ED,EC),
            SC=EC, edge(S,E,C), not ledge(S,E,C).

%Rule 8 - edges between LOOP-join and LOOP-split nodes:
%For all LOOP-join nodes tnode(S,lj,SD,SC) and LOOP-split nodes tnode(E,ls,ED,EC)
%in 3-iPG that are derived from node(S,lj,SD) and node(E,ls,ED) in P, there is
%an edge tedge(tnode(S,lj,SD,SC),tnode(E,ls,ED,EC),C) in 3-iPG between them if
%there is an edge between node(S,lj,SD) and node(E,ls,ED) in P and the counter of
%tnode(E,ls,ED,EC) is the counter of tnode(S,lj,SD,SC) increased by 1.
tedge(tnode(S,lj,SD,SC),tnode(E,ls,ED,EC),C) :-
            tnode(S,lj,SD,SC), tnode(E,ls,ED,EC),
            EC=SC+1, edge(S,E,C), ledge(S,E,C).



```

```
107 %Rule 9 - edges between LOOP-split predecessors and LOOP-split nodes:
108 %For all LOOP-split predecessor nodes tnode(S,ST,SD,SC) and LOOP-split nodes
109 %tnode(E,ls,ED,EC) in 3-iPG that are derived from node(S,ST,SD) and
110 %node(E,ls,_) in P, there is an edge tedge(tnode(S,ST,SD,SC),tnode(E,ls,ED,EC),C)
111 %in 3-iPG between them if there is an edge between node(S,ST,SD) and node(E,ls,_)
112 %in P and the counter of tnode(E,ls,ED,EC) is the counter of tnode(S,ST,SD,SC)
113 %multiplied by 10 and increased by 1.
114 tedge(tnode(S,ST,SD,SC),tnode(E,ls,ED,EC),C) :-
115             tnode(S,ST,SD,SC), tnode(E,ls,ED,EC),
116             EC=TMP+1, TMP=SC*10, node(E,ls,_), edge(S,E,C).
117
118 %Rule 10 - edges between LOOP-split and false-direct-successor:
119 %For all LOOP-split nodes tnode(S,ls,SD,SC) and nodes tnode(E,ET,ED,EC) in 3-iPG
120 %that are derived from node(S,ls,_) and node(E,ET,ED) in P, there is an edge
121 %tedge(tnode(S,ls,SD,SC),tnode(E,ET,ED,EC),null) in 3-iPG between them if there
122 %is a false-edge between node(S,ls,_) and node(E,ET,ED) in P and the counter of
123 %tnode(S,ls,SD,SC) returns a remainder 4 for the division by 10, and the counter
124 %of tnode(E,ET,ED,EC) is the counter of tnode(S,ls,SD,SC) divided by 10
125 %(remember that the node counter is an integer).
126 tedge(tnode(S,ls,SD,SC),tnode(E,ET,ED,EC),null) :-
127             tnode(S,ls,SD,SC), tnode(E,ET,ED,EC),
128             SC\10=4, EC=SC/10, node(S,ls,_), edge(S,E,false).
129
130
131
132 %====================== transformed time constraints ========================
133
134 %========================== atomizatioin function ===========================
135
136 %Extended Time Constraint
137 %etc(id, type, delta, source, destination):
138 %source: source(quantifier, node_label, loop_reference_label)
139 %destination: destination(quantifier, relation, node_label, loop_reference_label,
140 %iteration_reference, iteration_reference_loop_label)
141 %
142 %  atomized into
143 %
144 %Atomic Time Constraint
145 %atc_<absolute|relative>(id,type,delta,source_node,destination_node)
146
```

```
147 %An atc is an Atomic Time Constraint (ATC) of an etc if the atc source node
148 %tnode(SL,ST,SD,SC) is an element of the source node set source(SQ,SL,SLRL)
149 %of the etc and the atc destination node tnode(DL,DT,DD,DC) is an element of
150 %the destination node set destination(DQ,_,DL,DLRL,DIR) of the etc.
151
152 %ATC with destination outside all loops
153 atc_absolute(ID,T,D,tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)) :-
154         etc(ID,T,D,source(SQ,SL,SLRL),destination(null,null,DL,null,null,null)),
155         source_node(tnode(SL,ST,SD,SC),source(SQ,SL,SLRL)), tnode(DL,DT,DD,DC),
156         not nodes_equal(tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)).
157
158 %ATC with absolute destination
159 atc_absolute(ID,T,D,tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)) :-
160         etc(ID,T,D,source(SQ,SL,SLRL),destination(DQ,absolute,DL,DLRL,DIR,DIRLL)),
161         source_node(tnode(SL,ST,SD,SC),source(SQ,SL,SLRL)),
162         destination_node_absolute(tnode(DL,DT,DD,DC),
163                 destination(DQ,absolute,DL,DLRL,DIR,DIRLL)),
164         not nodes_equal(tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)).
165
166 %ATC with relative destination
167 atc_relative(ID,T,D,tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)) :-
168         etc(ID,T,D,source(SQ,SL,SLRL),destination(DQ,relative,DL,DLRL,DIR,DIRLL)),
169         source_node(tnode(SL,ST,SD,SC),source(SQ,SL,SLRL)),
170         destination_node_relative(tnode(DL,DT,DD,DC),tnode(SL,ST,SD,SC),
171                 destination(DQ,relative,DL,DLRL,DIR,DIRLL)),
172         not nodes_equal(tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)),
173         tsucc(tnode(DL,DT,DD,DC),tnode(SL,ST,SD,SC)).
174
175
176 %============================== source nodes ==============================
177
178 %A tnode is an element of the source node set if the criteria defined below holds
179 %L
180 source_node(tnode(L,T,D,C), source(null,SL,null)) :-
181             tnode(L,T,D,C), node(SL,ST,SD), SL=L.
182
183 %FIRST L
184 source_node(tnode(L,T,D,C), source(first,SL,null)) :-
185             tnode(L,T,D,C), node(SL,ST,SD), SL=L,
186             not not_first_tnode(tnode(L,T,D,C)).
```

```
187
188 %LAST L
189 source_node(tnode(L,T,D,C), source(last,SL,null)) :-
190             tnode(L,T,D,C), node(SL,ST,SD), SL=L,
191             not not_last_tnode(tnode(L,T,D,C)).
192
193 %EACH L
194 source_node(tnode(L,T,D,C), source(each,SL,null)) :-
195             tnode(L,T,D,C), node(SL,ST,SD), SL=L.
196
197 %=============================================================================
198
199 %FIRST L WITHIN SLR
200 source_node(tnode(L,T,D,C), source(first,SL,SLRL)) :-
201             tnode(L,T,D,C), node(SL,ST,SD), SL=L,
202             tnode(LL,LT,LD,LC), node(SLRL,SLRT,SLRD), SLRL=LL, LC\10=1,
203             tpred(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
204             not twin_2_between(tnode(LL,LT,LD,LC),tnode(L,T,D,C)).
205
206 %LAST L WITHIN SLR
207 source_node(tnode(L,T,D,C), source(last,SL,SLRL)) :-
208             tnode(L,T,D,C), node(SL,ST,SD), SL=L,
209             tnode(LL,LT,LD,LC), node(SLRL,SLRT,SLRD), SLRL=LL, LC\10=4,
210             tpred(tnode(L,T,D,C),tnode(LL,LT,LD,LC)),
211             not twin_1_between(tnode(L,T,D,C),tnode(LL,LT,LD,LC)).
212
213 %EACH L WITHIN SLR
214 source_node(tnode(L,T,D,C), source(each,SL,SLRL)) :-
215             tnode(L,T,D,C), node(SL,ST,SD), SL=L, node(SLRL,_,_).
216
217
218
219 %============================ destination nodes ============================
220
221 %================================ ABSOLUTE ================================
222 %A tnode is an element of the destination node set if the criteria defined below holds
223 %L
224 destination_node_absolute(tnode(L,T,D,C),
225 destination(null,absolute,DL,null,null,null)) :-
226             tnode(L,T,D,C), node(DL,DT,DD), DL=L.
```

```
227
228 %FIRST ABSOLUTE L
229 destination_node_absolute(tnode(L,T,D,C),
230 destination(first,absolute,DL,null,null,null)) :-
231             tnode(L,T,D,C), node(DL,DT,DD), DL=L,
232             not not_first_tnode(tnode(L,T,D,C)).
233
234 %LAST ABSOLUTE L
235 destination_node_absolute(tnode(L,T,D,C),
236 destination(last,absolute,DL,null,null,null)) :-
237             tnode(L,T,D,C), node(DL,DT,DD), DL=L,
238             not not_last_tnode(tnode(L,T,D,C)).
239
240 %EACH ABSOLUTE L
241 destination_node_absolute(tnode(L,T,D,C),
242 destination(each,absolute,DL,null,null,null)) :-
243             tnode(L,T,D,C), node(DL,DT,DD), DL=L.
244
245 %=============================================================================
246
247 %FIRST ABSOLUTE L WITHIN SLR
248 destination_node_absolute(tnode(L,T,D,C),
249 destination(first,absolute,DL,DLRL,null,null)) :-
250             tnode(L,T,D,C), node(DL,DT,DD), DL=L,
251             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=1,
252             tpred(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
253             not twin_2_between(tnode(LL,LT,LD,LC),tnode(L,T,D,C)).
254
255 %LAST ABSOLUTE L WITHIN SLR
256 destination_node_absolute(tnode(L,T,D,C),
257 destination(last,absolute,DL,DLRL,null,null)) :-
258             tnode(L,T,D,C), node(DL,DT,DD), DL=L,
259             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=4,
260             tpred(tnode(L,T,D,C),tnode(LL,LT,LD,LC)),
261             not twin_1_between(tnode(L,T,D,C),tnode(LL,LT,LD,LC)).
262
263 %EACH ABSOLUTE L WITHIN SLR
264 destination_node_absolute(tnode(L,T,D,C),
265 destination(each,absolute,DL,DLRL,null,null)) :-
266             tnode(L,T,D,C), node(DL,DT,DD), DL=L, node(DLRL,_,_).
```

```
267
268
269 %================================= RELATIVE ================================
270 %destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
271 %destination(each,relative,DL,null,null,null)):
272 %tnode(L,T,D,C) is an element of destination(each,relative,DL,null,null,null)
273 %for the source node tnode(SL,ST,SD,SC).
274
275 %FIRST RELATIVE L
276 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
277 destination(first,relative,DL,null,null,null)) :-
278             tnode(L,T,D,C), tnode(SL,ST,SD,SC), node(DL,DT,DD), DL=L,
279             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
280             not twin_2_between(tnode(SL,ST,SD,SC),tnode(L,T,D,C)).
281
282 %LAST RELATIVE L
283 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
284 destination(last,relative,DL,null,null,null)) :-
285             tnode(L,T,D,C), tnode(SL,ST,SD,SC), node(DL,DT,DD), DL=L,
286             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
287             not not_last_tnode(tnode(L,T,D,C)).
288
289 %EACH RELATIVE L
290 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
291 destination(each,relative,DL,null,null,null)) :-
292             tnode(L,T,D,C), tnode(SL,ST,SD,SC), node(DL,DT,DD), DL=L,
293             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)).
294
295 %===========================================================================
296
297 %FIRST RELATIVE L WITHIN DLR
298 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
299 destination(first,relative,DL,DLRL,null,null)) :-
300             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
301             node(DL,DT,DD), node(DLRL,_,_), DL=L,
302             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
303             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=1,
304             tpred(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
305             not twin_2_between(tnode(LL,LT,LD,LC),tnode(L,T,D,C)).
306
```

```
307 %LAST RELATIVE L WITHIN DLR
308 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
309 destination(last,relative,DL,DLRL,null,null)) :-
310             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
311             node(DL,DT,DD), node(DLRL,_,_), DL=L,
312             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
313             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=4,
314             tsucc(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
315             not twin_1_between(tnode(L,T,D,C),tnode(LL,LT,LD,LC)).
316
317 %EACH RELATIVE L WITHIN DLR
318 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
319 destination(each,relative,DL,DLRL,null,null)) :-
320             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
321             node(DL,DT,DD), node(DLRL,_,_), DL=L,
322             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)).
323
324 %============================================================================
325
326 %FIRST RELATIVE L WITHIN DLR SAME_ITERATION DKR
327 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
328 destination(first,relative,DL,DLRL,same_iteration,DKRL)) :-
329             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
330             node(DL,DT,DD), node(DLRL,_,_), DL=L,
331             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
332             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=1,
333             tpred(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
334             not twin_2_between(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
335             same_iteration(tnode(L,T,D,C),tnode(SL,ST,SD,SC),DKRL).
336
337 %LAST RELATIVE L WITHIN DLR SAME_ITERATION DKR
338 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
339 destination(last,relative,DL,DLRL,same_iteration,DKRL)) :-
340             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
341             node(DL,DT,DD), node(DLRL,_,_), DL=L,
342             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
343             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=4,
344             tsucc(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
345             not twin_1_between(tnode(L,T,D,C),tnode(LL,LT,LD,LC)),
346             same_iteration(tnode(L,T,D,C),tnode(SL,ST,SD,SC),DKRL).
```

```
347
348 %EACH RELATIVE L WITHIN DLR SAME_ITERATION DKR
349 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
350 destination(each,relative,DL,DLRL,same_iteration,DKRL)) :-
351             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
352             node(DL,DT,DD), node(DLRL,_,_), DL=L,
353             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
354             same_iteration(tnode(L,T,D,C),tnode(SL,ST,SD,SC),DKRL).
355
356 %=============================================================================
357
358 %FIRST RELATIVE L WITHIN DLR NEXT_ITERATION DKR
359 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
360 destination(first,relative,DL,DLRL,next_iteration,DKRL)) :-
361             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
362             node(DL,DT,DD), node(DLRL,_,_), DL=L,
363             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
364             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=1,
365             tpred(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
366             not twin_2_between(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
367             next_iteration(tnode(L,T,D,C),tnode(SL,ST,SD,SC),DKRL).
368
369 %LAST RELATIVE L WITHIN DLR NEXT_ITERATION DKR
370 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
371 destination(last,relative,DL,DLRL,next_iteration,DKRL)) :-
372             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
373             node(DL,DT,DD), node(DLRL,_,_), DL=L,
374             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
375             tnode(LL,LT,LD,LC), node(DLRL,DLRT,DLRD), DLRL=LL, LC\10=4,
376             tsucc(tnode(LL,LT,LD,LC),tnode(L,T,D,C)),
377             not twin_1_between(tnode(L,T,D,C),tnode(LL,LT,LD,LC)),
378             next_iteration(tnode(L,T,D,C),tnode(SL,ST,SD,SC),DKRL).
379
380 %EACH RELATIVE L WITHIN DLR NEXT_ITERATION DKR
381 destination_node_relative(tnode(L,T,D,C), tnode(SL,ST,SD,SC),
382 destination(each,relative,DL,DLRL,next_iteration,DKRL)) :-
383             tnode(L,T,D,C), tnode(SL,ST,SD,SC),
384             node(DL,DT,DD), node(DLRL,_,_), DL=L,
385             tsucc(tnode(L,T,D,C),tnode(SL,ST,SD,SC)),
386             next_iteration(tnode(L,T,D,C),tnode(SL,ST,SD,SC),DKRL).
```

```
387 %============================ HELPER RULES ================================
388
389 %There is another node with the label L that is a predecessor of the given tnode.
390 not_first_tnode(tnode(L,T,D,C)) :-
391             tnode(L,T,D,C), tnode(L2,_,_,C2),
392             L=L2, C2<C.
393
394 %There is another node with the label L that is a successor of the given tnode.
395 not_last_tnode(tnode(L,T,D,C)) :-
396             tnode(L,T,D,C), tnode(L2,_,_,C2),
397             L=L2, C<C2.
398
399 %There is another node with the label TL2 that is a predecessor of the given
400 %tnode(L2,T2,D2,C2) and successor of tnode(L1,T1,D1,C1).
401 twin_2_between(tnode(L1,T1,D1,C1),tnode(L2,T2,D2,C2)) :-
402             tnode(L1,T1,D1,C1), tnode(L2,T2,D2,C2), tnode(TL2,TT2,TD2,TC2),
403             L2=TL2, TC2<C2,
404             tpath(tnode(L1,T1,D1,C1),tnode(TL2,TT2,TD2,TC2)),
405             tpath(tnode(TL2,TT2,TD2,TC2),tnode(L2,T2,D2,C2)).
406
407 %There is another node with the label TL1 that is a successor of the given
408 %tnode(L1,T1,D1,C1) and predecessor of tnode(L2,T2,D2,C2).
409 twin_1_between(tnode(L1,T1,D1,C1),tnode(L2,T2,D2,C2)) :-
410             tnode(L1,T1,D1,C1), tnode(L2,T2,D2,C2), tnode(TL1,TT1,TD1,TC1),
411             L1=TL1, TC1>C1,
412             tpath(tnode(L1,T1,D1,C1),tnode(TL1,TT1,TD1,TC1)),
413             tpath(tnode(TL1,TT1,TD1,TC1),tnode(L2,T2,D2,C2)).
414
415 %Node tnode(SL,ST,SD,SC) and node tnode(DL,DT,DD,DC)) are not equal.
416 nodes_equal(tnode(L1,T1,D1,C1),tnode(L2,T2,D2,C2)) :-
417             tnode(L1,T1,D1,C1), tnode(L2,T2,D2,C2),
418             L1=L2, C1=C2.
419
420 %Between split node tnode(LSL,ls,LSD,LSC) and node tnode(L,T,D,C),
421 %there is no other split node with the label LSL.
422 ls_between(tnode(LSL,ls,LSD,LSC),tnode(L,T,D,C)) :- tnode(L,T,D,C),
423             tnode(LSL,ls,LSD,LSC), tnode(KSL,ls,KSD,KSC), LSL=KSL,
424             tpred(tnode(LSL,ls,LSD,LSC),tnode(KSL,ls,KSD,KSC)),
425             tpred(tnode(KSL,ls,KSD,KSC),tnode(L,T,D,C)).
426
```

```
427 %Node tnode(DL,DT,DD,DC) is placed in the same iteration of the loop
428 %IPL as node tnode(SL,ST,SD,SC) if both nodes are placed in the same
429 %loop in the given process and there is no split node with the label
430 %IPL and an arbitrary counter between the node DL and node SL.
431 same_iteration(tnode(DL,DT,DD,DC),tnode(SL,ST,SD,SC),IPL) :-
432             tnode(DL,DT,DD,DC), tnode(SL,ST,SD,SC), tnode(IL,ls,ID,IC),
433             tnode(DL,DT,DD,DC)!=tnode(SL,ST,SD,SC),
434             tnode(SL,ST,SD,SC)!=tnode(IL,ls,ID,IC),
435             tnode(DL,DT,DD,DC)!=tnode(IL,ls,ID,IC),
436             node(DPL,DPT,DPD), node(SPL,SPT,SPD), node(IPL,ls,IPD),
437             DL=DPL, SL=SPL, IL=IPL,
438             inloop(DPL,IPL), inloop(SPL,IPL),
439             tpred(tnode(IL,ls,ID,IC),tnode(DL,DT,DD,DC)),
440             tpred(tnode(IL,ls,ID,IC),tnode(SL,ST,SD,SC)),
441             not ls_between(tnode(IL,ls,ID,IC),tnode(DL,DT,DD,DC)),
442             not ls_between(tnode(IL,ls,ID,IC),tnode(SL,ST,SD,SC)).
443
444 %Node tnode(DL,DT,DD,DC) is placed in the next iteration of
445 %the loop IPL as node tnode(SL,ST,SD,SC) if both nodes are placed
446 %in the same loop in the given process and
447 %there is exactly one split node with the label IPL and an arbitrary
448 %counter between the node DL and node SL.
449 next_iteration(tnode(DL,DT,DD,DC),tnode(SL,ST,SD,SC),IPL) :-
450             tnode(DL,DT,DD,DC), tnode(SL,ST,SD,SC),
451             tnode(IL1,ls,ID1,IC1), tnode(IL2,ls,ID2,IC2),
452             tnode(DL,DT,DD,DC)!=tnode(SL,ST,SD,SC),
453             tnode(SL,ST,SD,SC)!=tnode(IL1,ls,ID1,IC1),
454             tnode(DL,DT,DD,DC)!=tnode(IL1,ls,ID1,IC1),
455             tnode(SL,ST,SD,SC)!=tnode(IL2,ls,ID2,IC2),
456             tnode(DL,DT,DD,DC)!=tnode(IL2,ls,ID2,IC2),
457             node(DPL,DPT,DPD), node(SPL,SPT,SPD), node(IPL,ls,IPD),
458             DL=DPL, SL=SPL, IL1=IPL, IL2=IPL,
459             inloop(DPL,IPL), inloop(SPL,IPL),
460             tpred(tnode(IL1,ls,ID1,IC1),tnode(DL,DT,DD,DC)),
461             tpred(tnode(IL1,ls,ID1,IC1),tnode(SL,ST,SD,SC)),
462             not ls_between(tnode(IL1,ls,ID1,IC1),tnode(SL,ST,SD,SC)),
463             ls_between(tnode(IL1,ls,ID1,IC1),tnode(DL,DT,DD,DC)),
464             not ls_between(tnode(IL1,ls,ID1,IC1),tnode(IL2,ls,ID2,IC2)),
465             not ls_between(tnode(IL2,ls,ID2,IC2),tnode(DL,DT,DD,DC)).
```

Appendix D

rulesTCInference.dl

```
#include "rulesProcessTransformation.dl".

%============================================================================
%================== time constraint inference rules =========================
%============================================================================

%================================ given ATC ==================================

%CASE 0:
%the given ATC atc is also a derived atc
derived_tc(tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)) :-
                atc_absolute(_,_,_,tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)).

derived_tc(tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)) :-
                atc_relative(_,_,_,tnode(SL,ST,SD,SC),tnode(DL,DT,DD,DC)).



%======================== S--> source to the right ==========================

%CASE 1: Sequence and AND
%(source to right except over XJ or LJ)
derived_tc(S,D) :-  tedge(X,S,_), derived_tc(X,D),
                    S!=D, tpath(S,D),
                    not txj(S), not tls(S).

```

```
27 %CASE 3: XOR
28 %(source to right over XJ only if there are derived ATCs in both XOR-branches)
29 derived_tc(S,D) :-  tedge(X,S,_), derived_tc(X,D),
30                     tedge(Y,S,_), derived_tc(Y,D),
31                     S!=D, X!=Y, tpath(S,D),
32                     txj(S).
33
34 %CASE 5: LOOP - all but last LS
35 %(source to right over a LS with counter%10!=4)
36 derived_tc(tnode(SL,ls,SD,SC),D) :-
37                     tedge(X,tnode(SL,ls,SD,SC),_), derived_tc(X,D),
38                     tpath(tnode(SL,ls,SD,SC),D),
39                     SC\10!=4.
40
41 %CASE 7: LOOP - last LS
42 %(source to right over LS with CLS%10=4 only if there is
43 %another derived ATC from another LS1 with the counter CLS-3)
44 derived_tc(tnode(SL,ls,SD,SC),D) :-
45                     tedge(X,tnode(SL,ls,SD,SC),_), SC\10=4, derived_tc(X,D),
46                     tnode(YL,ls,YD,YC), YC=SC-3, derived_tc(tnode(YL,ls,YD,YC),D),
47                     X!=tnode(YL,ls,YD,YC), tpath(tnode(SL,ls,SD,SC),D),
48                     tls(tnode(SL,ls,SD,SC)).
49
50
51
52 %========================= <--D destination to the left ====================
53
54 %CASE 2: Sequence and AND
55 %(destination to left except over XS or LS)
56 derived_tc(S,D) :-  tedge(D,X,_), derived_tc(S,X),
57                     S!=D, tpath(S,D),
58                     not txs(D), not tls(D).
59
60 %CASE 4: XOR
61 %(destination to left over XS only if there are derived ATCs in both XOR-branches)
62 derived_tc(S,D) :-  tedge(D,X,_), derived_tc(S,X),
63                     tedge(D,Y,_), derived_tc(S,Y),
64                     S!=D, X!=Y, tpath(S,D),
65                     txs(D).
66
```

```
67 %CASE 6: LOOP - all but first and last LS
68 %(destination to left over a LS with counter%10!=1)
69 derived_tc(S,tnode(DL,ls,DD,DC)) :-
70                     tedge(tnode(DL,ls,DD,DC),X,_), derived_tc(S,X),
71                     tpath(S,tnode(DL,ls,DD,DC)),
72                     DC\10!=1.
73
74 %CASE 8: LOOP - first LS
75 %(destination to left over LS with CLS%10=1 only if there is
76 %another derived ATC from the source to another LS4 with the counter CLS+3)
77 derived_tc(S,tnode(DL,ls,DD,DC)) :-
78                     tedge(tnode(DL,ls,DD,DC),X,_), DC\10=1, derived_tc(S,X),
79                     tnode(YL,ls,YD,YC), YC=DC+3, derived_tc(S,tnode(YL,ls,YD,YC)),
80                     X!=tnode(YL,ls,YD,YC), tpath(S,tnode(DL,ls,DD,DC)),
81                     tls(tnode(DL,ls,DD,DC)).
```

Appendix E

rulesTerminationCheck.dl

```
#include "rulesTCInference.dl".

%============================================================================
%==================== process termination check ============================
%============================================================================

%A process terminates if it has no loops.
process_terminates :-   #count{LS: node(LS,ls,_)}=0.

%A process with loops must terminate in order to satisfy all ETCs
%if each loop is time constrained (=there is a derived ATC between
%each LS with counter%10=2 and its counterpart).
process_terminates :-
        derived_tc(tnode(LSL,ls,LSD,LSC),tnode(LJL,lj,LJD,LJC)),
        tcounterpart(tnode(LSL,ls,LSD,LSC),tnode(LJL,lj,LJD,LJC)),
        LSC\10=2, LJC\10=2, not process_not_terminates.

%A process does not have to terminate in order to satisfy all ETCs
%if there are counterpart LS- and LJ-nodes with the same counter%10=2,
%but there is no derived ATC between them.
process_not_terminates :-
        not derived_tc(tnode(LSL,ls,LSD,LSC),tnode(LJL,lj,LJD,LJC)),
        tcounterpart(tnode(LSL,ls,LSD,LSC),tnode(LJL,lj,LJD,LJC)),
        LSC\10=2, LJC\10=2.
```

Bibliography

[AF08] Artin Avanes and Johann-Christoph Freytag. Adaptive workflow scheduling under resource allocation constraints and network dynamics. *Proc. VLDB Endow.*, 1(2):1631–1637, August 2008.

[BHPS61] Yehoshua Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961. Reprinted in Y. Bar-Hillel. (1964). *Language and Information: Selected Essays on their Theory and Application*, pages 116–150, 1964.

[Bun14] Bundeskanzleramt. Verordnung der E-Control über den Wechsel, die Anmeldung, die Abmeldung und den Widerspruch (Wechselverordnung 2014, WVO 2014), 2014. BGBl. II Nr. 167/2014.

[Bus98] Christoph Bussler. Workflow instance scheduling with project management tools. In *DEXA Workshop*, pages 753–758, 1998.

[BWJ00] Claudio Bettini, Xiaoyang Sean Wang, and Sushil Jajodia. Free schedules for free agents in workflow systems. In *TIME*, pages 31–38, 2000.

[BWJ02a] Claudio Bettini, Xiaoyang Sean Wang, and Sushil Jajodia. Solving multi-granularity temporal constraint networks. *Artificial Intelligence*, 140(1-2):107–152, 2002.

[BWJ02b] Claudio Bettini, Xiaoyang Sean Wang, and Sushil Jajodia. Temporal reasoning in workflow systems. *Distributed and Parallel Databases*, 11(3):269–306, 2002.

[CBS04] Jorge Cardoso, Robert P Bostrom, and Amit Sheth. Workflow management systems and erp systems: Differences, commonalities, and applications. *Information Technology and Management*, 5(3-4):319–338, 2004.

[CCPP95] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Conceptual modeling of workflows. In Michael P. Papazoglou, editor, *OOER '95: Object-Oriented and Entity-Relationship Modeling*, pages 341–354, 1995.

[CGJ+07] Carlo Combi, Matteo Gozzi, José M. Juárez, Barbara Oliboni, and Giuseppe Pozzi. Conceptual modeling of temporal clinical workflows. In *14th International Symposium on Temporal Representation and Reasoning (TIME 2007), 28-30 June 2007, Alicante, Spain*, pages 70–81, 2007.

[CGMP12] Carlo Combi, Mauro Gambini, Sara Migliorini, and Roberto Posenato. Modelling temporal, data-centric medical processes. In *Proceedings of the 2Nd ACM SIGHIT International Health Informatics Symposium*, IHI '12, pages 141–150, 2012.

[CGMP14] Carlo Combi, Mauro Gambini, Sara Migliorini, and Roberto Posenato. Representing business processes through a temporal data-centric workflow modeling language: An application to the management of clinical pathways. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, 44(9):1182–1203, 2014.

[CGPP12] Carlo Combi, Matteo Gozzi, Roberto Posenato, and Giuseppe Pozzi. Conceptual modeling of flexible temporal workflows. *TAAS*, 7(2):19, 2012.

[CHM+14] A. Cimatti, L. Hunsberger, A. Micheli, R. Posenato, and M. Roveri. Sound and complete algorithms for checking the dynamic controlla-

bility of temporal networks with uncertainty, disjunction and observation. In *Temporal Representation and Reasoning (TIME), 2014 21st International Symposium on*, pages 27–36, 2014.

[CHP13] Carlo Combi, Luke Hunsberger, and Roberto Posenato. An algorithm for checking the dynamic controllability of a conditional simple temporal network with uncertainty. In *ICAART 2013 - Proceedings of the 5th International Conference on Agents and Artificial Intelligence*, pages 144–156, 2013.

[CKGJ13] Saoussen Cheikhrouhou, Slim Kallel, Nawal Guermouche, and Mohamed Jmaiel. A survey on time-aware business process modeling. In *International Conference on Enterprise Information Systems (ICEIS)*, page 10p., 2013.

[CKGJ15] Saoussen Cheikhrouhou, Slim Kallel, Nawal Guermouche, and Mohamed Jmaiel. The temporal perspective in business process modeling: A survey and research challenges. *Serv. Oriented Comput. Appl.*, 9(1):75–85, 2015.

[CP02] Carlo Combi and Giuseppe Pozzi. Towards temporal information in workflow systems. In *International Conference on Conceptual Modeling*, pages 13–25, 2002.

[CP03] Carlo Combi and Giuseppe Pozzi. Temporal conceptual modelling of workflows. In *International Conference on Conceptual Modeling*, pages 59–76, 2003.

[CP06] Carlo Combi and Giuseppe Pozzi. Task scheduling for a temporal workflow management system. In *Temporal Representation and Reasoning, 2006. TIME 2006. Thirteenth International Symposium on*, pages 61–68, 2006.

[CP09] Carlo Combi and Roberto Posenato. Controllability in temporal conceptual workflow schemata. In *Business Process Management, 7th International Conference*, pages 64–79, 2009.

[CP10] Carlo Combi and Roberto Posenato. Towards temporal controllabilities for workflow schemata. In *2010 17th International Symposium on Temporal Representation and Reasoning*, pages 129–136, 2010.

[CP18] Carlo Combi and Roberto Posenato. Extending Conditional Simple Temporal Networks with Partially Shrinkable Uncertainty. In *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*, volume 120 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:16, 2018.

[DDDGB08] Gero Decker, Remco Dijkman, Marlon Dumas, and Luciano García-Bañuelos. Transforming bpmn diagrams into yawl nets. In *International Conference on Business Process Management*, pages 386–389, 2008.

[DDO08] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software technology*, 50(12):1281–1294, 2008.

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1):61 – 95, 1991.

[DRMR13] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer Berlin Heidelberg, 2013.

[EEP06] Johann Eder, Hannes Eichner, and Horst Pichler. A probabilistic approach to reduce the number of deadline violations and the tardiness of workflows. In *On the Move to Meaningful Internet Systems 2006*, pages 5–7, 2006.

[EGP00] Johann Eder, Wolfgang Gruber, and Euthimios Panagos. Temporal modeling of workflows with conditional execution paths. In *Database and Expert Systems Applications, 11th International Conference*, pages 243–253, 2000.

[EP00] Johann Eder and Euthimios Panagos. Managing time in workflow systems. In *Workflow Handbook 2001*, pages 109–132. October 2000.

[EPGN03] Johann Eder, Horst Pichler, Wolfgang Gruber, and Michael Ninaus. Personal schedules for workflow systems. In *International Conference on Business Process Management (BPM 2003)*, pages 216–231, 2003.

[EPL97] Johann Eder, Heinz Pozewaunig, and Walter Liebhart. epert: Extending pert for workflow management systems. 1997.

[EPPR99] Johann Eder, Euthimios Panagos, Heinz Pozewaunig, and Michael Rabinovich. Time management in workflow systems. In *3rd International Conference on Business Information Systems (BIS 1999)*, pages 265–280, 1999.

[EPR99] Johann Eder, Euthimios Panagos, and Michael Rabinovich. Time constraints in workflow systems. In *Advanced Information Systems Engineering, 11th International Conference CAiSE'99, Heidelberg, Germany, June 14-18, 1999, Proceedings*, pages 286–300, 1999.

[EPR13] Johann Eder, Euthimios Panagos, and Michael Rabinovich. Workflow time management revisited. In Janis Bubenko, John Krogstie, Oscar Pastor, Barbara Pernici, Colette Rolland, and Arne Sølvberg, editors, *Seminal Contributions to Information Systems Engineering*, pages 207–213. 2013.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431, 1993.

[GKK+11] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[GKK+16] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASICS*, pages 2:1–2:15, 2016.

[GKKS12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[HPC12] Luke Hunsberger, Roberto Posenato, and Carlo Combi. The dynamic controllability of conditional stns with uncertainty. *CoRR*, abs/1212.2005, 2012.

[Kap87] Robert S Kaplan. *Accounting and Management: Field Study Perspectives: Proceedings of a Colloquium Held June 16-18, 1986, at the Harvard Business School*. Harvard Business School Press, 1987.

[KIG+15] Felix Kossak, Christa Illibauer, Verena Geist, Jan Kubovy, Christine Natschläger, Thomas Ziebermayr, Theodorich Kopetzky, Bernhard Freudenthaler, and Klaus-Dieter Schewe. *A Rigorous Semantics for BPMN 2.0 Process Diagrams*. Springer International Publishing, 2015.

[Lie18] Lieferanten und Netzbetreiber Strom and Versorger und Netzbetreiber Gas and oee and FGW and ECA and APCS and AGCS und A&B etc. Spezifikation zur Umsetzung der Wechselverordnung gemäß Wechselverordnung 2014 und des elektronischen Kündigungsprozesses 2014. `https://www.energylink.at/energylink/techn.-dokumentation/aenderungen%20ab%2001.10.2018/Spezifikation%20zur%20Umsetzung%20der%20Wechselverordnung%20V4.2.pdf`, 2018.

[Lif08] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1594–1597, 2008.

[LKR13] Andreas Lanz, Jens Kolb, and Manfred Reichert. Enabling personalized process schedules with time-aware process views. In *CAiSE 2013 Workshops, 2nd Int'l Workshop on Human-Centric Information Systems (HCIS 2013)*, number 148 in Lecture Notes in Business Information Processing (LNBIP), pages 205–216, 2013.

[LNCY11] Xiao Liu, Zhiwei Ni, Jinjun Chen, and Yun Yang. A probabilistic strategy for temporal constraint management in scientific workflow systems. *Concurrency and Computation: Practice and Experience*, 23(16):1893–1919, 2011.

[LPCR13] Andreas Lanz, Roberto Posenato, Carlo Combi, and Manfred Reichert. Controllability of time-aware processes at run time. In *On the Move to Meaningful Internet Systems*, pages 39–56, 2013.

[LRW13] Andreas Lanz, Manfred Reichert, and Barbara Weber. A formal semantics of time patterns for process-aware information systems. Technical Report UIB-2013-02, University of Ulm, January 2013.

[LRW16] Andreas Lanz, Manfred Reichert, and Barbara Weber. Process time patterns: A formal foundation. *Information Systems*, 57:38 – 68, 2016.

[LSPG06] Ruopeng Lu, Shazia Wasim Sadiq, Vineet Padmanabhan, and Guido Governatori. Using a temporal constraint network for business process execution. In *ADC*, volume 49 of *CRPIT*, pages 157–166, 2006.

[LWR09] Andreas Lanz, Barbara Weber, and Manfred Reichert. Time patterns for process-aware information systems: A pattern-based analysis - revised version. Technical report, University of Ulm, Ulm, December 2009.

[LWR10] Andreas Lanz, Barbara Weber, and Manfred Reichert. Workflow time patterns for process-aware information systems. In *Enterprise, Business-Process and Information Systems Modeling - 11th International Workshop, BPMDS 2010, and 15th International Conference, EMMSAD 2010, held at CAiSE 2010*, pages 94–107, 2010.

[LWR14] Andreas Lanz, Barbara Weber, and Manfred Reichert. Time patterns for process-aware information systems. *Requir. Eng.*, 19(2):113–141, 2014.

[LY05] Hongchen Li and Yun Yang. Dynamic checking of temporal constraints for concurrent workflows. *Electronic Commerce Research and Applications*, 4(2):124 – 142, 2005.

[LYC04] Hongchen Li, Yun Yang, and TY Chen. Resource constraints analysis of workflow specifications. *Journal of Systems and Software*, 73(2):271–285, 2004.

[Mar00] Olivera Marjanovic. Dynamic verification of temporal constraints in production workflows. In *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*, pages 74–81, 2000.

[MGR04] Robert Müller, Ulrike Greiner, and Erhard Rahm. Agentwork: a workflow system supporting rule-based workflow adaptation. *Data & Knowledge Engineering*, 51(2):223–256, 2004.

[MM05] Paul H Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *Aaai*, pages 1193–1198, 2005.

[MMV01] Paul H Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. 2001.

[MO99] Olivera Marjanovic and Maria E Orlowska. On modeling and verification of temporal constraints in production workflows. *Knowledge and Information Systems*, 1(2):157–192, 1999.

[MR00] Robert Müller and Erhard Rahm. Dealing with logical failures for collaborating workflows. In *International Conference on Cooperative Information Systems*, pages 210–223, 2000.

[MRvdA+10] Ronny S Mans, Nick C Russell, Wil MP van der Aalst, Arnold J Moleman, and Piet JM Bakker. Schedule-aware workflow management systems. In *Transactions on Petri nets and other models of concurrency IV*, pages 121–143. 2010.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[MvdAR+09] Ronny S Mans, Wil MP van der Aalst, Nick C Russell, Piet JM Bakker, and Arnold J Moleman. Process-aware information system development for the healthcare domain-consistency, reliability, and effectiveness. In *International Conference on Business Process Management*, pages 635–646, 2009.

[OMG13] OMG. Business process model and notation (bpmn), version 2.0.2, December 2013.

[oP] University of Potsdam. Potsdam answer set solving collection. `https://potassco.org/`. Accessed: 2019-12-16.

[PEC17] Horst Pichler, Johann Eder, and Margareta Ciglic. Modelling processes with time-dependent control structures. In *International Conference on Conceptual Modeling*, pages 50–58. Springer, 2017.

[Pic06] Horst Pichler. *Time management for workflow systems. A probabilistic approach for basic and advanced control flow structures.* PhD thesis, Alpen-Adria-Universitaet Klagenfurt, 2006.

[PWE09] Horst Pichler, Michaela Wenger, and Johann Eder. Composing time-aware web service orchestrations. In *Advanced Information Systems Engineering*, pages 349–363, 2009.

[RM06] Jan C Recker and Jan Mendling. On the translation between bpmn and bpel: Conceptual mismatch between process modeling languages. In *The 18th International Conference on Advanced Information Systems Engineering*, pages 521–532, 2006.

[RS59] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959.

[RTHEvdA04] Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow resource patterns. Technical report, BETA

Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004.

[RTHEvdA05] Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow data patterns: Identification, representation and tool support. In *International Conference on Conceptual Modeling*, pages 353–368, 2005.

[RvdAtH06] Nick Russell, Wil MP van der Aalst, and Arthur ter Hofstede. Workflow exception patterns. In *International Conference on Advanced Information Systems Engineering*, pages 288–302, 2006.

[SKK05] Jin Hyun Son, Jung Sun Kim, and Myoung Ho Kim. Extracting the workflow critical path from the extended well-formed workflow schema. *J. Comput. Syst. Sci.*, 70(1):86–106, 2005.

[SMO00] Shazia W Sadiq, Olivera Marjanovic, and Maria E Orlowska. Managing change and time in dynamic workflow processes. *International Journal of Cooperative Information Systems*, 09(01n02):93–116, 2000.

[SO98] Shazia W Sadiq and Maria E Orlowska. Dynamic modification of workflows. Technical report, University of Queensland, Department of Computer Science and Electrical Engineering, 1998.

[Tsc06] Willi Tscheschner. Transformation from epc to bpmn. *Business Process Technology*, 1(3):7–21, 2006.

[VBvdA01] Henricus MW Verbeek, Twan Basten, and Wil MP van der Aalst. Diagnosing workflow processes using woflan. *The computer journal*, 44(4):246–279, 2001.

[vdA96] Wil MP van der Aalst. Structural characterizations of sound workflow nets. *Computing Science Reports*, 96(23):18–22, 1996.

[vdA98] Wil MP van der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998.

[vdA13] Wil MP van der Aalst. Business process management: a comprehensive survey. *ISRN Software Engineering*, 2013, 2013.

[vdA16] Wil MP van der Aalst. *Process Mining: Data Science in Action*. Springer Berlin Heidelberg, 2016.

[vdALRS16] Wil MP van der Aalst, Marcello La Rosa, and Flávia Maria Santoro. Business process management. *Business & Information Systems Engineering*, 58(1):1–6, 2016.

[vdARD05] Wil MP van der Aalst, Michael Rosemann, and Marlon Dumas. Deadline-based escalation in process-aware information systems. Technical report, BPMcenter.org, 2005.

[vdASS11] Wil MP van der Aalst, Helen Schonenberg, and Minseok Song. Time prediction based on process mining. *Inf. Syst.*, 36(2):450–475, 2011.

[vdATHKB03] Wil MP van der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[Vid99] Thierry Vidal. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999.

[Vid00] Thierry Vidal. Controllability characterization and checking in contingent temporal constraint networks. In *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning*, KR00, pages 559–570, 2000.

[VvdA00] Eric Verbeek and Wil MP van der Aalst. Woflan 2.0 a petri-net-based workflow diagnosis tool. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000*, pages 475–484, 2000.

[Wes07] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Berlin Heidelberg, 2007.

[WM08] Stephen A White and Derek Miers. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Business Process Management Process Modeling. Future Strategies Incorporated, 2008.

[ZMI10] Michael Zur Muehlen and Marta Indulska. Modeling languages for business processes and business rules: A representational analysis. *Information systems*, 35(4):379–390, 2010.

[ZMR13] Michael Zur Muehlen and Jan Recker. How much language is enough? theoretical and practical use of the business process modeling notation. In *Seminal Contributions to Information Systems Engineering*, pages 429–443. 2013.

[ZPC00] Hai Zhuge, Hung Keng Pung, and To-Yat Cheung. Timed workflow: Concept, model, and method. In *Web Information Systems Engineering, 2000. Proceedings of the First International Conference on*, volume 1, pages 183–189, 2000.

[ZyCkP01] Hai Zhuge, To yat Cheung, and Hung keng Pung. A timed workflow process model. *Journal of Systems and Software*, 55(3):231 – 243, 2001.