LATVIJAS UNIVERSITATE • UNIVERSITY OF LATVIA

ALGORITHMIC PROBLEMS IN ANALYSIS OF REAL TIME SYSTEM SPECIFICATIONS

Kārlis Čerāns

Rīga, 1992

LATVIJAS UNIVERSITĀTE • UNIVERSITY OF LATVIA

Algorithmic Problems in Analysis of Real Time System Specifications

Kārlis Čerāns

A Thesis for the Dr.sc.comp.
Degree at University of Latvia

Institute of Mathematics and
Computer Science
University of Latvia
Riga, Rainis blvd. 29,
Latvia 226250

University of Latvia
Rīga, 1992

Abstract

The thesis is devoted to the study of analysis automation possibilities (decidability and undecidability of reachability, infinite behaviour possibility and bisimulation equivalence problems) for various kinds of real time system specification formalisms. The investigated formalisms are based on finite state model control structure enriched in various ways to reflect data and time dependencies of the modelled system behaviour.

Decidable proved are, first, the vertex reachability and infinite behaviour possibility (infinite feasible path existence) problems for programs in a simple theoretical programming language, called LTIBA, which is an enrichment of the Finite State Machine model with variables, suitable for modelling real time system behaviour dependencies both on quantitative time constraints (the LTIM system of commands) and external data (the LBASE system of commands). An effective symbolic characteristic of the sets of all feasible program paths in the terms of path set projectivity is also given for LBASE, LTIM and LTIBA programs whenever possible. The *undecidability* of the vertex reachability problem is proved for programs in a language LTIM' which is a slight variation of the considered time constraint specification language LTIM.

In the thesis also the strong and weak (abstracted from system internal actions) bisimulation equivalence problems are proved decidable for the formalism of Parallel Timer Processes (PTP), which are provided with the real time labelled transition system semantics and allow to express in a direct way the quantitative time constraints on the behaviour of concurrent real time systems. Various enrichments of the basic PTP model by additional features (including the processes with the dependencies on external data) are also considered and investigated w.r.t. decidability of the vertex reachability, infinite path feasibility and bisimulation equivalence problems. An undecidability result regarding the considered algorithmic problems is obtained for a class of timed processes supplied with memory cells for moving the timer value information along the time axis.

Finally, in the appendix an example of the reachability and path feasibility analysis of a simple real time system, specified as a process in the C.C.I.T.T. telecommunication system specification language SDL, is presented.

Anotācija

Disertācija veltīta dažādu reālā laika sistēmu apraksta formālismu analīzes automatizācijas iespēju izpētei (pētīta atrisināmība sasniedzamības, bezgalīgas darbības iespējas un bisimulācijas ekvivalences algoritmiskajām masu problēmam). Aplūkoto apraksta formālismu pamatā ņemts galīga automāta modelis, kas bagātināts ar līdzekļiem, kas ļauj aprakstīt modelējamo sistēmu darbības atkarību no laika un datiem.

Algoritmiskā atrisināmība pierādīta, vispirms, programmas virsotnes sasniedzamības un bezgalīga realizējama ceļa eksistences (bezgalīgas programmas darbības iespējas) problēmām programmām vienkāršā teorētiskā valodā LTIBA. Katra programma valodā LTIBA līdz ar galīgu vadības grafu satur mainīgos, kas atļauj aprakstīt modelējamās reāla laika sistēmas atkarību gan no temporāli kvantitatīviem nosacījumiem (LTIM komandu sistēma), gan arī no ārējiem datiem (LBASE komandu sistēma). Situācijās, kurās tas iespējams, aplūkojamo valodu LBASE, LTIM un LTIBA programmu realizējamo ceļu kopām dots efektīvs simbolisks raksturojums projektivitātes terminos. Virsotnes sasniedzamības problēmas algoritmiskā neatrisināmība pierādīta programmām valodā LTIM', kas tikai nedaudz atšķiras no pētītās temporālo atkarību specifikāciju valodas LTIM.

Darbā pierādīta arī stingrās un vājās (no sistēmas darbības iekšējiem notikumiem abstrahētās) bisimulācijas ekvivalences problēmu algoritmiskā atrisināmība paralēlo taimeru procesu (PTP) modelim, kuram definēta reālā laika iezīmēto pāreju sistēmu semantika, un kas atļauj dabiskā veidā aprakstīt paralēlu sistēmu atkarību no temporāli kvantitatīvajiem nosacījumiem. Virsotnes sasniedzamības, bezgalīgas darbības iespējas un bisimulācijas ekvivalences problēmu algoritmiskā atrisināmība pētīta arī dažādiem PTP modeļa paplašinājumiem ar papildus sistēmu apraksta līdzekļiem (t.sk. *procesiem ar ārējo datu atkarības attēlošanas iespējām*). Bisimulācijas ekvivalences un virsotnes sasniedzamības problēmu neatrisināmības rezultāts iegūts ar atmiņas šūnām paplašinātu taimeru procesu klasei.

Darba pielikumā aplūkots piemērs vienkāršas reāla laika sistēmas, kas uzdota kā process C.C.I.T.T. telekomunikāciju specifikāciju valodā SDL, sasniedzamības un ceļu realizējamības analīzei.

Аннотация

Диссертационная работа посвящена исследованию возможностей автоматизации анализа различных моделей систем реального времени (исследованы на разрешимость массовые проблемы достижимости, возможности бесконечной работы и бисимуляционной эквивалентности). Рассмотренные модели базируются на понятие конечного автомата, расширенного здесь различными способами для обеспечения возможностей описания зависимости работы моделируемых систем от временных условиий и от данных.

Алгоритмическая разрешимость доказана сначала для массовых проблем достижимости вершины и возможности бесконечной работи для программ в несложном теоретическом языке LTIBA, являющимся расширением конечно-автоматной модели путем введения переменных и позволяющее выразить зависимость моделируемой системы, как от темпорально количественных ограничений (система команд LTIM), так от внешних данных (система команд LBASE). В ситуациях, когда это возможно, эффективное символическое описание множеств всех реализуемых путей в программах исследуемых языков LBASE, LTIM, LTIBA дано в терминах проективности. Алгоритмическая неразрешимость проблемы достижимости вершины доказана для программ в языке LTIM', которое только понемногу отличается от рассмотренного выше языка описания временных зависимостей LTIM.

В работе доказана также разрешимость строгой и слабой (абстрагированной от внутренних событий системы) бисимуляционной эквивалентности для модели процессов с параленьными таймерами (PTP), имеющим семантику реального времени определенную посредством систем помеченных переходов, и позволяющим прямым образом описать зависимости паралельных систем реального времени от временно количественних условий. Разрешимость проблем достижимости вершины, возможности бесконечной работи и бисимуляционной эквивалентности исследована также для некоторых расширений основной модели PTP (содержащих также средства для описания зависимости моделируемых систем от данных). Результат неразрешимости рассматриваемых алгоритмических проблем получен для класса темпоральных процессов имеющих ячейки памяти для передвижения таймерных значений по временной оси.

Наконец, в приложении рассмотрен пример анализа достижимости и реализуемости путей в спецификации несложной системы реального времени, заданной процессом на языке SDL, разработанного МККТТ для описания телекоммуникационных систем.

Acknowledgements

First of all, I want to thank my thesis advisor Prof. Jānis Bārzdiņš for his suggestion to look at the time constrained real time system analysis, for his really noteworthy advices and continuous encouragement in my work on the thesis over the years.

I am very thankful to all my colleagues at Department of Computer Science, Institute of Mathematics and Computer Science for the provided possibility to work on my thesis, for their understanding, support and encouragement. Among my colleagues I want to thank especially Prof. Audris Kalniņš for a number of interesting discussions we have had on the topic of the real time system analysis and automated test case generation and Lolita Zeltkalne for her help to improve the English in the most important parts of the thesis.

I want to express also my thanks to my colleagues and friends abroad – Uno Holmer, Wang Yi, K.V.S. Prasad, Alan Jeffrey and Prof. Bengt Nordström in Göteborg, as well as Kim G. Larsen and Jens Chr. Godskesen in Aalborg both for numerous interesting discussions on the real time process calculi, and for the provided possibility to have a look on a wide spectrum of the ongoing research in the general area of the program correctness in the world.

My deepest thanks are also to Dr. Agnis Andžāns and Prof. Rūsiņš Freivalds who, though not being in a direct relation with my thesis work, have done very much for my mathematical and scientific education.

Perhaps the most important help I recieved during the work on the thesis was that by my parents Silvija and Henrihs and my sister Kristīne, let me say:

Paldies Jums, mīļie vecāki un māsa!

Rīga, Latvija
June 17, 1992

Kārlis Ćerāns.

Contents

Chapter 1

Introduction

1.1 "Real Time Systems"

The term "real time system" is often used in different situations to denote different categories of objects. In our case let us have as the starting point the convention to put in the real time system category *any object whose behaviour is observed with respect to a "time" dimension*, assumed on its turn to coincide with the usual category of time, shared by most of people (we are not looking at the relativity theory treatment of the time, mathematically the "time" dimension might be identified with the real number axis provided with some additive arithmetics).

A lot of "objects" we encounter in various situations fall under this definition of real time systems. The example list could be started by a simple gas-cooker in a kitchen, by numerous kinds of vending and bank machines, by the electric lamp on my table, or a sensor giving temperature measurements in reaction to the actual temperature fluctuations in some environment. Regarding more "serious" examples of real time systems, one could consider telephone exchanges, computers and their networks, offices, banks, aircrafts and their control systems, etc. Also every human being him- or herself can be viewed as a (very complicated) real time system. The earth's atmosphere is also a real time system, perhaps one of the most complicated (save the human beings themselves) we are to deal with in our everyday life.

The thesis is devoted to the analysis of *models* of real time systems. Generally, a model of one system, S, is some other system S' which has some "important" properties common with S and is "simpler" than S due to not taking into account (due to the abstracting from) some S properties which are considered as "irrelevant". We shall be more precise about the kind of models under consideration a little below.

One can distinguish at least two principal purposes for which the real time system models can be used:

- to describe or to discover the regularities in the behaviour (the rules determining

the behaviour) of some existing r.t.s. (e.g. in order to learn, how to use it for our purposes, or to adopt our behaviour to the system (as in the case of the weather (the atmosphere)));

- to investigate the properties and the behaviour of some system which we want to *build* (most commonly, in order to ensure ourselves that we are actually going to build a system which has sufficiently many properties common with our desire).

The contribution of the thesis is oriented mostly towards the second purpose, however, nobody forbids in principle to apply the below considered models and their analysis algorithms also to the already existing systems, were there a necessity for it. It is also not the primary aim to consider here very complicated r.t.s. as the humans or the atmosphere, it would be nice, if the contributed ideas could help in the analysis of real time systems of "medium complexity" size.

1.2 Symbolic Models of Real Time Systems

Over the last decades of years a lot of models (or, rather, model classes), oriented to the applications in the real time system design and manufacturing have appeared and been widely used. They are ranging from probably the simplest one, the Finite state machines ("FSM", for short), to theoretical and practical real time system specification languages with wide variety of offered system modelling constructions (on the theoretical side, various process calculi like CCS [Mil80, Mil89], CSP [Hoa85], Petri Nets [Pe62] (see also [Re85]), temporal logics [Em91], etc. As to the practical real time specification languages, one could mention, for instance, SDL [CC89], LOTOS [ISO89a] and ESTELLE [ISO89b]). What is common for all these model classes, is their *symbolic* nature which allows to have a computer support in building, modifying and analyzing them.

One can notice that even a computerized model is assumed to reflect all the logical (including timed) behaviour of the entire system, it is considerably *cheeper* to build such a model, if compared with the building of some preliminary version of the system itself from any kind of hardware (or even by some low-level programming language). Observe also that such a preliminary version of a system is almost inevitably being a subject of rather drastic changes afterwards.

So, a possible methodology for the use of models in the real time system manufacturing could be, as follows: first, build a model of the system you want to construct, second, make sure (up to the level of confidence, determined by your requirements and possibilities) that your model is correct (in most of the cases it might be necessary to make some modifications to the model), third, using the final model as a *specification*, implement the real system itself (this implementation also can be, in principle, at least

partially mechanized (automated)). One can use the model (the specification) also for automated test case generation for the system's implementation.

In a stepwise design methodology of real time systems a series of models for a system can be used: at early design stages only some general system properties (e.g. the block structure and signal routing, etc.) are modelled, the model is afterwards step-by-step refined, so making the system specification more and more precise.

If one has obtained a detailed symbolic model of some r.t.s. (say, a program in some executable very high level specification language, such as SDL or LOTOS), it becomes possible to study its behaviour in various environments (to "run" the program on various test examples), so finding out and getting rid of a number of bugs both in the model implementation and, perhaps, also in some principal design decisions (when the model of the system behaves in a way not expected before, and the requirement for the system not to behave so was not realized up to this time). Various program dynamic (runtime) analysis techniques (program annotation, etc.) can be successfully applied at this stage. One can also develop and use various debuggers for error detecting and localization in the model (see, for example, [KE89], [BKA89]). There is an ongoing research [BKM91, Kal91] to obtain methods for SDL program testing automation, intended to capture both program control and data information, also resulting in a tool ultimately oriented towards improving the r.t.s. model quality. One can observe, as in [ABBCK91], that the need in various tools to improve the reliability of r.t.s. is relatively higher than the need of corresponding tools for the most of non-real-time applications since the real time systems have relatively higher *demands* on reliability (consider, e.g., a system controlling an aircraft, or even a nuclear reactor. Nobody will be satisfied also, if some widely used communication protocol shows up some errors time and again).

In the case of very safety-critical systems it might be possible to *prove* some desirable properties of the system, however, this task seems to be very work-consuming and must be done individually for every modification of the system. There are already some proof-assistance systems (in general, not particularly designed for proving programs correct) which may help to make computer-checked proofs, see [BM79, GMW79] (at the considered level of confidence, perhaps, the "manually" carried proofs do not convince strong enough about the model "correctness" for it is possible to make errors also in proofs (for example, by overlooking some possibilities when performing a huge case analysis), as they could have been made in the programs)). It should be noted, however, that the formal property proving nowadays cannot be carried out for the systems of practical size; also in the case it turns possible to prove some correctness properties in principle, one can find this proving too *costly*.

1.3 Automation of Model Analysis

As observed above, the real time system model property proving is a very hard job. One might want to see, if there could be some tools for this *proving* automation, which, say, given a specification of the system by some kind of model (for instance, as a program in SDL) and a property, formalized in some logics, returns either a proof or a disproof of this property possession by the system. Not difficult to see that this plan cannot be fulfilled in its full generality since one is faced very soon with the deciding of undecidable problems (equivalent to, say, the halting node reachability for Turing machines). However, the model analysis automation possibility is so attractive, that it is reasonable to make a (further) abstraction from some features of the model (system) in order to obtain a simpler model (with some more of the modelled system behaviour properties not reflected), which *could* be the subject of the (some kind of) analysis automation.

What is usually abstracted out in order to obtain automatically analyzable models of real time systems, are any kind of system data dependencies. Also *quantitative real time constraints* (stating, for instance, that an event b cannot be performed before 3 seconds have passed after an occurrence of some other event a) are usually a subject of such an abstraction. Various infinite control structures like process signal queues in SDL systems use to be replaced by their bounded "approximations".

So, by abstracting of a number of more or less "unimportant" system work characteristics, one obtains a simple model like a Finite state machine (a finite directed graph with nodes interpreted as "states" and edges interpreted as "transitions" between these states, every transition has ascribed a label, which is interpreted as an "event" observable during the system's behaviour). Provided with such a model one can easily test the *reachability* of a given state, also find out, if the machine satisfies some temporal logic property (see [CES86] for this approach carried out systematically), it is also possible to provide algorithms for automated checking of various equivalences between two systems (most popular kinds of equivalence are (in the order of increasing strength) the *trace* (language) equivalence, the *failures* equivalence [Hoa85] and the *bisimulation* equivalence [Par81, Mil89]). For the FSM models it is easy to check also the presence/absence of deadlocks. The check whether a given system can exhibit an infinite behaviour (or is forced to terminate after a finite number of steps) is also straightforward.

A number of important properties of real time systems still can be checked using the FSM-like models (one can observe that a "parallel composition" of two real time systems, both modelled by FSMs, also can be expressed as a Finite state machine, so the FSM analysis techniques can be used also to check various synchronization problems in concurrent r.t.s.). However, there are also many cases when the system's behaviour is crucially influenced by, say, quantitative time measurements. If two processes, A and B, are working in parallel, it might be very important, whether

a certain signal from A to B is produced by A and consumed by B before a timer in B times out. Also an external observer may wish to distinguish a system which gives a response to some his stimulus in 5 seconds from another one, giving *the same* reaction, but only delayed for 7 or 8 seconds. In such cases the FSM model appears to be *too rough* for adequate modelling of the desired real time behaviour (real time system). A similar problem arises also, if the modelled system appears to be essentially dependent on some other data from an infinite domain of values, as, say, integer or rational numbers (any dependencies on finite-valued data can be, at least in principle, modelled by a FSM with the set of states coinciding with the set of all possible data values. However, if the number of possible data values gets very large, this approach also is to be abandoned due to the practical infeasibility).

The most common solution to this problem consists in designing models (model classes) for real time systems which are, first, *richer* than FSMs in that they are able to reflect, at least partially, the desired system properties (e.g., the dependencies on quantitative timed constraints) and, second, permit automated analysis of one or other kind (it may well be that the reachability problem for the models in such an "enriched" model class is decidable, whereas some equivalence problem is not; even more, for Parallel Timer Processes, considered below, the bisimulation equivalence problem turns out to be decidable, while the trace and failure equivalence problems are not).

The thesis is devoted to the study of possible enrichments of the FSM model by features allowing to specify the system behaviour dependencies on quantitative timed constraints and integer-valued data and the elaboration of suitable algorithms for these enriched model analysis. We generally assume the *density* of the time domain (time axis, going beyond the model system's execution), so any event, which is allowed to appear in some time interval $[t_1, t_2]$ within the model behaviour (t_1 and t_2 are real-valued time constants), may appear at any *real* moment within this interval (actually, for the decidability purposes the domain of allowed action/event firing times is assumed to consist of *rational* numbers).

The main theoretical results of the thesis are

- deciding algorithms for the reachability and infinite behaviour possibility (infinite feasible path existence) problems for programs in a simple language LTIBA, allowing specification of both data and time dependencies of real time systems, and

- deciding algorithms for strong and weak (abstracted from internal actions) bisimulation equivalence problems for Parallel Timer Processes, a model compositional model with the real time semantics, designed for the specification of the quantitative time dependencies of concurrent r.t.s.

We also explain the relations between the considered models and investigate some

variations of them in order both to outline the extendability of the obtained algorithms to more general timing specification formalisms and to show the limits of the real time system analysis automation possibilities (various *undecidability* results obtained).

Before proceeding to a more formal treatment of these problems, we outline the considered models and obtained results in some more detail and give some points to the background and related work by others.

1.4 Languages for Specifying Data and Time Dependencies

1.4.1 Dependencies on Integer Valued Data

First of all we consider a "programming language" LBASE, which in addition to the finite-state control structure permits in every program a finite number of integer-valued internal variables with the following possible commands (here x, y are variable names, c is an integer constant):

- ?x, input of a (new) value from the program input gate into the variable x;
- $x \leftarrow y$ ($x \leftarrow c$), assign the value of the variable y (the constant c) to x;
- $x < y$ ($x < c$, $c < y$), compare the values of the two specified variables (or the variable and the constant), the decision on further program control flow can depend on the result of this comparison;
- NOP, a dummy operator.

No arithmetical transformation of data is permitted in LBASE programs. However, the infinite domain of possible data values (integer numbers) makes the standard FSM analysis techniques based on the "state-space" exploration not directly applicable for LBASE programs.

The language LBASE is a slight dialect (simplification) of the base programming language L_0, considered in [BBK74, BBK77, ABBCK91] (we do not care here about the input tapes for the programs, the program output operators are not considered, as well (the usage of the output operators does not contribute theoretically any new modelling abilities into the language). The STOP operators of L_0 are simply replaced by dead-end nodes here).

It was proved first in [BBK74] (the first English version can be found in [BBK77]) that the statement (node, vertex) reachability problem for L_0 programs is decidable. Actually, these papers consider a more practical problem of automated complete test system generation according to the well-known testing criterion C_1 - covering of all

feasible (executable on some input) program branches, what, of course, includes the deciding which branches in the program are feasible.

To say some more words about the background, in [BKB74, BBK77, Auz84a, Auz84b] the reachability problem was proved decidable also for programs with more complicated data dependencies, including various kinds of counters, stack and direct access files, see [ABBCK91] for a survey. However, as the data structures in programs become more complicated, the node reachability problem becomes undecidable very soon (it is enough to have two 2-way counters provided with the test for 0, or even one 1-way counter provided with the constant assignment to the counter and the comparison of the counter and other variable values [BKB74] (see also Section 8.2)).

All the work, done previously for the automated analysis of L_0-programs and its modifications, was oriented to the non-real-time applications, considering a program as an object which, having started at some time on some finite amount of input data, normally executes a finite number of steps and outputs a result (the case when a program executes forever was considered as an abnormality). So, in the "white-box" approach to the program analysis taken (the program control structure is assumed observable during the program execution) the program behaviour was normally characterized by the set of all *finite* feasible paths in it. As to the real time applications, it is usual to assume (at least theoretically) that the system under specification is going to work endlessly *ad infinitum* by reading more and more input data, as they arrive at the system's input. So a number of important problems concerning the infinite behaviour of the system (and its model) arise, say,

- can a given system, specified by a LBASE program, exhibit an infinite behaviour at all (i.e. is there an infinite feasible path in the given program), or the system is always forced to terminate after finite number of execution steps,
- is a given program statement reachable within such an infinite behaviour, or
- is the system able to visit some designated nodes (for instance, the nodes with input statements attached to them) *infinitely often* within some behaviour (or is it able to avoid some statements in an infinite run), etc.

In Part I of the thesis the reader can find all these problems proved decidable for LBASE programs from program texts, see Section 1.4.3 for a more detailed obtained result outline.

1.4.2 Time Constraint Specification Languages

Concerning the time constraints in real time system specifications, we come up with the following simple language (a slight modification of LBASE), called LTIM. Every program in LTIM is allowed to have, first, a finite number of rational-valued variables

with permitted operations over them just as for LBASE program variables (input, assignment, comparison, NOP). One special variable z, named *real time counter* can also be used in any LTIM program in the following operations:

- $z \stackrel{+}{\leftarrow} x_i$, assigning the value of x_i to z, provided that the value of z does not decrease via this assignment. The decision for further control flow in the program can be done depending upon whether or not the value of z actually was not less than x_i before the assignment.

- $x_i \leftarrow z + c$, assigning the value of z increased by c to x_i (for the sake of simplicity only nonnegative c_s are allowed here).

It is important to note that the real time counter z cannot appear in the program input operators, or as the lefthand side of an assignment operator (so, there is no way to decrease the z value during the program execution).

When using a program in the language LTIM to model the quantitative timed aspects of some r.t.s. behaviour, the program is assumed to read step by step all the sequence of the system transition (event) firing times from its input gate. The "time counting" for LTIM programs is "absolute", and for every moment of the modelled system behaviour the value of the program real time counter z at the corresponding program execution moment is used to represent the numeric value of the "current system clock reading" (the current numeric value of the time moment in the system). Any time a new system event firing time is read by the program, the value of its real time counter is advanced to hold this new time moment by the positive assignment operator. The variable activation operators of the program naturally represent the timer setting in the modelled r.t.s.; the timer timeout in the system can be represented in the program by the z value advancement up to the value of sometimes previously activated variable (and the control flow passing along the appropriate edge in the program graph).

The real time system modelling by means of the programming language LTIM constructs is considered in some detail in Section 9.2.

Having defined two simple languages LBASE and LTIM, we come up with a unified formalism, called the language LTIBA, containing the facilities of the both simple languages.

The usage of the language LTIBA programs in the r.t.s. behaviour modelling allows to describe at least to some extent the dependencies of r.t.s. both on the timing and external data constraints, see Section 10.4 for these modelling possibilities considered in some detail.

Let us note just that according to the above given interpretation the time moments in LTIBA (LTIM) programs can be treated as some kind of *data* - a feature which could be useful in various situations, e.g., when reasoning about real time office information

systems with input and output messages often carrying parameters of the type "time moment".

1.4.3 Results

The main results in Part I, containing a detailed discussion on the languages LBASE, LTIM, LTIBA, can be grouped, as follows:

First, the vertex reachability problem for LTIBA programs is proved decidable. This decidability follows from a more general result stating that the set of all finite feasible paths of an arbitrary LTIBA program P can be given a characteristic of being the set of *projections* of all paths in some FSM, effectively construable from P (let us call the program path sets with this property *projective*). This result implies also that certain kinds of program verification problems (those, concerning finite feasible statement and branch sequences) can be dealt with automatically.

Second, decidable for LTIBA programs is also the problem of infinite feasible path existence (the problem of program ability to exhibit an infinite behaviour). Regarding the characterization of the set of all infinite feasible paths in LTIBA programs, a direct analogue of the path set projectivity does not hold, in general, for there can be infinite *not* feasible paths with *all* finite prefixes feasible both in LTIM and LBASE programs. We propose instead a similar, slightly weaker, property of program infinite path sets, called "F- projectivity" (the specified path set must coincide with the projection set of all *fair* paths in a FSM provided with a special set of designated states; a path in such a machine is fair iff it visits some of the designated states *infinitely often*). It is proved that for all LTIM programs the sets of infinite feasible paths are indeed (effectively) F-projective. F-projective in LTIM programs are also various other interesting infinite path sets, e.g., the set of paths executable with no bound on the "limit value" of the real time counter, etc. As to the LBASE case, a simple example shows that also the F-projectivity criterion is not always satisfied by the set of all infinite feasible paths in a program (so not giving such a hope also in the general case of the whole language LTIBA).

Other infinite path feasibility problems, such as whether there exists an infinite feasible path, containing a given node infinitely often/at least once, are showed for LTIBA programs decidable, as well. We consider also the algorithms effectively generating the infinite program work histories which visit the specified program nodes the desired number of times. These algorithms can be used to generate for *deterministic* LTIBA programs some kind of "infinite tests" which could guide the program behaviour according to, say, some fairness-style correctness conditions.

Third, it is showed that for a slight modification of the language LTIM, where the rational data type for the language program variables is replaced by integers (all the variable handling commands remain the same), the vertex reachability problem becomes *undecidable*. This result gives a new class of programs with counter having

undecidable vertex reachability problem (if a one-way counter can be compared with other program variables, there is no need in order to obtain the undecidability to have a possibility to reset the counter to explicit constants from time to time). The result gives also some evidence about the "modelling power" of LTIM constructs. It is possible to interpret the result also as an instance of comparison of the possibilities to analyze discrete and dense data structures with "the same" constructs over them, where in the dense case "the battle is won" (for the most of models intended to capture the timed behaviour of r.t.s. the situation is quite opposite: in the discrete case the treatment of the time information fully reduces to the analysis of some FSM model, whilst in the corresponding dense case the generated state space is "essentially infinite" and so cannot be dealt with by a straightforward exploration).

We consider also two sublanguages of LTIM, called LTIM_0 and LBASQ (see Section 3.2 for definitions), which are useful in various aspects of the r.t.s. behaviour modelling and admit nicer characteristics of the infinite feasible path sets (the projectivity property) than the whole general language LTIM.

The language LTIM and LTIBA construct interpretation within the real time semantics allows to obtain the analysis automation results for a family of Parallel Timer Process models, discussed below in Section 1.5. We give also an example in Appendix B, how the obtained LTIBA program analysis algorithms can be used in automation of the analysis and testing of programs in the C.C.I.T.T. telecommunication system specification language SDL [CC89]. It is showed, for instance, how to solve a problem of automatic complete test system generation for SDL processes (according to some analogue of the testing completeness criterion C_1).

Most of the results listed in this section are published in Section 9 and Section 10 of [ABBCK91], written by the author (the results concerning the analysis of finite path feasibility and the modelling of the SDL constructs), and [Cer92a], where the infinite path feasibility problems for LTIBA programs and the undecidability result for programs in the integer-valued modification of LTIM are considered.

1.5 Parallel Timer Processes

The languages LBASE, LTIM and LTIBA for specifying the dependencies of real time systems on data and/or time constraints allow to specify these dependencies (especially in the case of the time constraint modelling) in an indirect way via the standard programming data structures (variables, counters, etc.).

The direct way of representing the timing information when modelling real time systems consists in the usage of models which also have some notion of *time* (usually, still associated with some mathematical data structure, resembling the important properties of the "real time", (e.g., the set of all nonnegative real numbers provided

with comparison and constant addition operations)) and interpreting the real system timing behaviour using the model timing features. We call such models provided with the notion of time the *models with the real time semantics*.

We consider one such model class, called Parallel Timer Processes ("PTP"s), able to model the r.t.s. behaviour dependencies on the quantitative timed constraints (see Section 9.1 for a formal definition of the model).

For expressing the quantitative timed constraints in a *sequential* system it may seem natural to introduce special "delay" or "waiting" primitives, associated with certain time periods, and insert them into the system's behaviour between the consecutive events the "time distance" between which is of importance. However, the idea of treating these "waiting" actions as primitives does not work, if combining the real time systems in parallel: neither can these "actions" be considered as interleaving, nor it is possible to consider them as indivisible entities for synchronization. Another idea for the time constraint expression, which is implemented also in the PTP model (no claim about the originality, see [MF76], [CC89] and [ACD90] for actually the same idea exploited) is not to introduce the "delays" of the system as its work primitives, but provide the system with some facilities which can "measure" the time interval lengths between various events.

Intuitively, a PTP is a finite state machine provided with a number of "timers" for controlling its timed behaviour. Every timer is an abstract device that can be set up to give its timeout after certain specified period of time. In the PTP model every timer is represented as a variable, which can receive its initial value by its setting, say $t \leftarrow 5$. After the setting the timer value is assumed to decrease synchronously in accordance with some global clock and, having reached the 0 value, the timer is said to give its *timeout*.

Every timer during the process execution is set up when a transition along some edge in the process occurs. On the other hand, every transition along an edge in the process may require some timers to have reached the 0 value, this way the idea that some certain period of time must have been passed between the transitions is implemented. Every PTP also has an explicitly specified initial timer value setting.

The edges in the PTPs are divided into two classes: the first ones (we call them *instantaneous* and colour *red*) have the property that, if a transition along such an edge is enabled (i.e. all the needed timers have reached the 0 value), it *must* be immediately executed. The second class of the process edges, called *possibly waiting* and coloured *black*, can allow the transitions along them to be delayed for some time, e.g. until the process environment decides this transition to be performed.

The semantics of Parallel Timer Processes is given operationally in the terms of *labelled transition systems* ("LTS"). Every edge in a PTP is assumed to have ascribed to it a label from a given finite set L of *action names*, interpreted during the process behaviour as the actions the process participates in (such actions can be either signal input/output, reading a data value from a bus, or even offering a cup of coffee by

some coffee-machine). Following the idea of timed transition systems of [Wan90], we define for every PTP P in the set of its states the relations $P' \xrightarrow{\sigma} P''$, meaning that the process can move from the state P' to P'' by performing the action $\sigma \in L$, and $P' \xrightarrow{\iota(d)} P''$ for $d \geq 0$ (we assume $d \in \mathbf{Q}^{+0}$), representing the process *delay* transitions and meaning that the process state P' can be transferred into P'' just by letting the time to pass for d units.

The PTP model is demonstrated to have a *compositional* semantics (a number of static process algebra combinators, including both the CCS-like and CSP-like parallel composition, are defined for PTPs, see Chapter 14), this allows to model as PTPs wide subsets of algebraic timed specification formalisms Timed CCS of [Wan90] and Timed CSP, as presented operationally in [Sch91].

The PTP model has a number of features common with the already existing models of the Timed Graph family (see [ACD90], [NSY91]), Section 1.6 outlines the common and different points in these models.

We consider also various possible extensions of Parallel Timer Processes (processes with inactive timers, processes with nondeterministic timer settings etc.), either allowing to reduce the size of the system specifications, or introducing some new time-modelling features (e.g. the ability to set a timer nondeterministically to some value in a specified (bounded or unbounded) time interval).

Concerning the specification of real time systems with the dependencies on both data and time constraints, we propose some enrichments of the PTP model with either integer or rational valued external data (see Section 10.4).

1.5.1 Modelling and Reachability

In Section 9.2 we give a method for modelling Parallel Timer Processes by LTIM programs (actually, a subset of LTIM, called LTIM_0, suffices for this modelling purpose). The most important point in this modelling is the precise demonstration how the language LTIM constructs can be used in the real time system modelling. It also allows to obtain an algorithm for deciding the vertex reachability problem for PTPs proved already decidable by other (direct) methods for a related model of Timed Graphs in [ACD90].

Regarding the considered extensions of PTPs, we also show how to model them using the programs in the languages LTIM and LTIBA, so illustrating the need and sufficiency of the considered simple theoretical language constructs to model various real time constructions (e.g., it is showed that it is not possible to model processes with unbounded nondeterministic timer settings by LTIM_0 programs, the whole class of the LTIM programs suffices for this modelling).

Regarding the analysis of the timed specification formalisms with data dependencies, an interesting result shows that really very slight changes in the definition of the

corresponding PTP extensions (called in Section 10.4 PTPBs and $PTPB_0$s) put the obtained specification model class either on the side of models with *decidable* vertex reachability problem, or on that with the vertex reachability problem being *undecidable*. In the case of the decidable model of $PTPB_0$s this decidability is proved via the modelling of the considered data dependent processes by LTIBA programs. The algorithms of this modelling can be also used to decide for $PTPB_0$s other algorithmic problems, discussed in Section 1.4 and considered more thoroughly in Part I (consider, e.g., an infinite feasible fair path existence problem).

We obtain also an undecidability result of the vertex reachability problem for the parallel timer processes with *memory cells* ("PTPM"s). Intuitively, a memory cell can be used to store the value of a timer (when some transition is executed) and to retrieve it afterwards at some later moment of time. A typical example of a timed process with a memory cell is the ice-hockey timer (formalized in the process calculi Timed CCS in [Wan91b]), whose time-counting can be at any moment interrupted and afterwards restarted with the time period value being the one it had at the moment of the interruption.

This undecidability result may look not very surprising since the adding of some extra constructors to any language may well turn some algorithmic problems for it undecidable. The point in the PTPM model consideration can be seen both for the timed behaviour it represents and its seeming "similarities" with the PTPQ model (an extension of PTPs with rational-valued data) considered in Section 10.4 and having the vertex reachability problem decidable.

1.5.2 Bisimulation Equivalences for PTPs

The vertex reachability problem is just the simplest theoretical algorithmic problem in the analysis of models of real time systems. For the sake of real time system *verification* one might often want also to show that his/her implemented system is "equivalent" in some sense to a specification of the system. It would be really nice, if such "equivalence" test could be performed automatically for some class of models.

As to the models for quantitative time constraint specification, a result obtained in [AD90] states the *undecidability* of the timed *trace inclusion* problem for Timed Büchi automata ("TBA"). This result can be easily extended to show also the undecidability of the *trace equivalence* problem for TBAs (see Section 1.6 for some remarks regarding the TBA model). One might naturally want to see, whether there is some decidable equivalence property for concurrent systems with timing.

Since the semantics of Parallel Timer Processes is given by labelled transition systems one can, following [Par81] and [Wan90], define the *bisimulation equivalence* for PTPs. The idea of the bisimulation equivalence is that two processes P_1 and P_2 are equivalent, iff there exists a binary relation $R \subseteq \mathcal{S}^{P_1} \times \mathcal{S}^{P_2}$ ($\mathcal{S}^P$ denotes the set of the process P states) such that

- the pair $(s_0^{P_1}, s_0^{P_2})$ of the process P_1 and P_2 initial states is in R, and
- for every pair $\langle s, r\rangle \in R$, *if* by some transition (real or delay) from s it is possible to move to some state s', *then* there exists *the same* transition (the transition labelled by the same action) from r to such a state $r' \in S^{P_2}$ for which the resulting pair of process states $\langle s', r'\rangle \in R$ (the converse with s and r in changed roles must also hold).

It is proved in the thesis that the problem, whether two given Parallel Timer Processes are bisimilar (bisimulation equivalent), or not, is decidable.

One can notice that the bisimulation equivalence is the *strongest* equivalence for the specification models with the LTS semantics (see, e.g., [Mil89]).

We consider also the so-called *weak* bisimulation equivalences (*observation* equivalences) which declare some actions from the process action repertoire to be *invisible* and for every pair of process states request that every generalized transition (perhaps containing in it an arbitrary number of "invisible" transitions) from the state of one process must be matched by some appropriate generalized transition from the corresponding state of the other process (see Section 11.1 for precise definitions, made in accordance with [Wan90]). It is proved in the thesis that also weak bisimulation equivalence problem for Parallel Timer Processes is decidable.

The decidability of both strong and weak bisimulation equivalences is shown also for a class of processes with nondeterministic timer settings (requiring the nondeterminism in these settings to be *bounded*).

Among the enrichments of PTPs which have decidable bisimulation equivalence problem fall also the processes with rational-valued external data. Regarding the processes with integer-valued external data the bisimulation equivalence problem seems to be much harder (undecidable?), though the reachability problem also for this process class can be easily dealt with using the modelling of the processes by LTIBA programs.

As to the previous and related work on deciding bisimulation equivalences for timed specification formalisms, the decidability results were obtained only regarding rather simple process classes, namely, in [HLW91] the strong bisimulation equivalence was proved decidable for the class of regular (i.e. built without the parallel composition operator) Timed CCS processes (these processes can be, in fact, modelled as PTPs with one timer). In [Che91] a similar decidability result was obtained for recursion-free finite timed processes.

The results regarding the decidability of bisimulation equivalences are published in [Cer92b] (full proofs in a slightly simplified case) and [Cer92c] (a conference paper describing the general results). Both of these papers contain also some discussion on the undecidability of the vertex reachability problem for the timed processes with memory. An early paper on the deciding strong bisimulation equivalence for a variant of PTPs is [Cer91], it also outlines one possible way of interpreting Timed CCS

processes from [Wan90] as PTPs (the way of TCCS modelling as PTPs considered in [Cer91] differs from that considered here in Section 14.2). The results regarding the automated analysis of r.t.s. specification formalisms (the enrichments of PTPs) covering both the system data and time dependencies are unpublished for the moment.

1.6 Comparison with Related Work

The theoretical background of the languages LBASE, LTIM and LTIBA has already been discussed in Section 1.4.1. We consider here the related approaches to the analysis of quantitative-constrained real time system specifications.

The first nontrivial result concerning time-constrained r.t.s. model analysis was the enumerative analysis algorithm in [BM83] for Time Petri Nets ("TPN", for short) [MF76] (in fact, solving the reachability problem for bounded TPNs).

Timing information in TPNs is closely tied up with the net transition firing mechanism, so specializing the possible general results on analyzability of time-dependent systems to a particular model class with time and certain kind of structure information (every transition of the net has associated two real numbers a and b, $a \leq b$, called the *earliest* and *latest* transition firing times, respectively. The transition in the net is allowed to fire when it has been continuously enabled for a time units; when the total transition continuous enabledness time reaches b time units, the transition is *forced* to fire). Regarding the TPN model one can look also [GMMP89], where various transition firing enforcement mechanisms for TPNs are discussed.

As to more recent formalisms for time constraint specification and analysis, the family of Timed Graph models [ACD90], [AD90], [NSY91] has become rather popular (also the PTP model introduced here in Section 1.5 can be considered as some variant of Timed Graphs).

A Timed Graph ("TG"), defined originally in [ACD90], is an automaton provided with a finite number of clocks which are assumed to be the variables whose values are synchronously increasing along the passage of time. It is allowed to reset these variable values to 0 at the automaton transition firing times. The conditions for the automaton's transition enabledness are obtained by observing the "current" values of the clocks (comparing them with a priori given constants).

It is proved in [ACD90] that the model checking for TGs w.r.t. branching-time temporal logic formulae (provided also with some means for expressing timed behavioural properties) is decidable (this result, clearly, means also the decidability of the vertex reachability problem for TGs). Using similar methods (time region graph construction), in [AD90] the *language emptiness* problem for Timed Büchi automata ("TBA") is proved decidable. As mentioned already above, in [AD90] the *language inclusion* problem for TBAs is proved *undecidable* (intuitively, a TBA is a Timed Graph provided with edge labels and Büchi-style acceptance conditions over the timed traces

of the automaton).

Regarding the quantitative time constraint modelling approach by the languages LTIM and LTIBA, it can be viewed, first, as a theoretical clarification of the relations between the typical real time system specification constructs and standard programming techniques, as the program variables and counters are. As observed already in Section 1.4.2, the treatment of the time moment information as *data* for the modelling programs also could be useful for some kind of applications.

The LTIM system of commands is also somewhat more *powerful* time constraint specification formalism, if compared to other existing models with decidable vertex reachability problem (Timed Graphs), see, e.g., the modelling of the PTPNs of Section 10.3 by LTIM programs (one can find some more timed systems which can be described as LTIM programs, but cannot be modelled as TGs). However, the point is to be made about the *indirect* way of the time constraint expression in the language LTIM, what may appear not very convenient for the use of the language by practitioners.

When comparing the language LTIBA to other existing r.t.s. specification formalisms, investigated w.r.t. the analysis automation possibility, an important point is the language LTIBA ability to reflect the system dependencies on both the timing and data constraints. The infinite path feasibility studies in data-dependent formalisms are also novel. Regarding the feasibility of infinite paths in time-dependent formalisms, the TG and TBA models already are given the semantics in the terms of *infinite* traces of the corresponding automata. So, from a practical standpoint, the novelty of the infinite path feasibility studies in LTIM programs is mostly associated with the coping with the generality of the language LTIM constructs (for Timed Graphs and similar timing specification formalisms the infinite path feasibility does not appear as a more serious problem, if compared to the finite path feasibility).

Regarding the Parallel Timer Process model, considered in Part II, its most important difference from TGs is the defined operational semantics (labelled transition system), yielding the possibility to consider the bisimulation equivalences between PTPs (the TG model has the semantics just in the terms of completed traces)[1].

The most "visible" difference between PTPs and TGs is the mechanism of expressing the time constraints: for PTPs this is done by setting a timer initially up to some value and then waiting until the timer decreases downto 0, while the TGs model a particular time interval by initial *resetting* of a continuously increasing clock

[1] If compared to the TG model, as defined in [ACD90], PTPs also do not have the requirement of no more than finite number of transitions, performed in a finite time (it is important to remove this requirement to be able to define the LTS semantics), and they are allowed to have a number of transitions within a single time moment (one could define also the LTS semantics for processes not allowed to have more than one transition associated with one moment of time, however, the algebraic theory of processes becomes much nicer, if the transitions of such kind are allowed).

to 0 and observing the value of the clock in order to find out whether the interval *has been passed*.

It is of some importance to have a formalism with decreasing timers when looking for an interpretation of the language LTIM constructs in the real time. Some extended PTP variations (consider especially the processes with nondeterministic timer settings) also outline some new theoretical situations regarding the time constraint analysis (observe also some "duality" in the time constraint handling by TGs and PTPs (clocks vs. timers)).

There already exists also a variant of TGs (also using increasing clocks, as the original model), defined in [NSY91] and called *Action Timed Graphs* ("ATG"s), which is provided with the LTS semantics (and, so, the notions of bisimulation equivalence)[2].

PTPs differ from ATGs basically by the design decisions regarding the ways how a transition along some edge can be *forced* to execute. In the PTP model the necessity of the transition firing is made explicit by denoting the edge with which the transition is associated as *red*, while in ATGs the transition firing necessity depends on a rather complex statement regarding all possible values of the clocks in the process "waiting future" and the enabledness conditions of other edges. We consider the ATG model thoroughly in Appendix A and show how the PTP analysis algorithms (deciding of the bisimulation equivalences) can be modified to apply to the analysis of ATGs.

The most important novel point in the PTP model presentation is in the provided strong and weak *bisimulation equivalence deciding algorithms* for the processes. The enrichments of PTPs, especially those provided with internal data variables, outline also some possibilities and impossibilities of the reachability analysis in the data dependent formalisms with real time semantics.

As to the deciding bisimulation equivalences for other timed specification formalisms, an interesting interpretation of the results of the thesis can be obtained regarding the decidability and undecidability of the equivalences for various classes of Timed CCS processes defined in [Wan90], [Wan91a] and [Wan91b] (note that the notions of the strong and weak bisimulations are central in the theory of TCCS), see Section 14.2 and Chapter 15 for a discussion on this interpretation topic. Let us note that it is due to the chosen edge firing enforcement mechanism for PTPs that we can translate the nets of regular TCCS processes into the PTP model in a rather *direct* way.

[2]One could notice that the method of defining the semantics for ATGs in [NSY91] does not assign a well defined semantics to the class of *all* ATGs (the so-called "time-stop" processes, which are neither able to perform a real transition, nor can wait for any amount of time, appear in some situations, though, possibly considered as pathological).

1.7 Organization of the Material

Part I contains the investigation of the reachability and various infinite path feasibility problems for programs in the languages LBASE, LTIM and LTIBA (the outline of the part contents can be found in Section 1.4). The presentation of the material of Part I begins with some mathematical background from graph and automata theory (Chapter 2), developing the notation and simple results for future use. Chapter 3 gives the definitions of the languages LBASE, LTIM and LTIBA, as well as the algorithmic problems of program analysis (path feasibility and vertex reachability). In Chapter 4 most of the constructions needed for deciding the statement reachability problem and characterizing finite feasible path sets in LBASE, LTIM and LTIBA programs are given. The last points in these problems are resolved by the path inequality system technique (Chapter 5), useful also afterwards when reasoning about the infinite path feasibility (see especially Section 5.3). The infinite path feasibility analysis in LBASE and LTIM programs is done in Chapter 6 and Chapter 7, respectively. Chapter 7 discusses also the implications of obtained results for LTIBA programs. Finally, Chapter 8 shows the undecidability of the vertex reachability problem for LTIM programs with integer data type (instead of otherwise used rationals).

Part II is mostly devoted to the development and analysis of the Parallel Timer Process model and its enrichments. In Section 9.1 the basic model of PTPs is introduced and a simple example of a real time system, specified as a PTP, is considered. Section 9.2 gives the methodology of the real time construct, used in the definition of PTPs, expressing in the programming language LTIM. In Chapter 10 a number of possible enrichments of the PTP model is considered, the most important being the processes with external data dependencies, described in Section 10.4. Also the vertex reachability problem deciding algorithms for the considered PTP enrichments are outlined there.

The strong and weak bisimulation equivalence problems for PTPs are proved decidable in Chapter 11 and Chapter 12, respectively. In Chapter 13 the bisimulation equivalence deciding algorithms are outlined also for the processes with nondeterministic timer settings and the processes with rational-valued data variables. The compositional properties of the PTP model are considered in Chapter 14, giving also one possible way of Timed CCS process modelling by PTPs. Finally, Chapter 15 shows the undecidability of the vertex reachability and bisimulation equivalence problems for the PTP model enrichment by memory cells.

In Appendix A we show the applicability of the obtained bisimulation deciding algorithms to the Action Timed Graph model.

Appendix B is devoted to the analysis of a simple real time system (a process, controlling a passenger lift), specified in the C.C.I.T.T. specification language SDL; also the complete set of test data for the considered lift process is constructed.

Part I

Languages for Data and Time Dependencies

Chapter 2

Mathematical Preliminaries

In the following chapters we define the programs in the languages LBASE, LTIM, LTIBA formally as *labelled graphs*. We are also going to characterize the program path feasibility in the labelled graph terms. To provide some background for the further studies we, first, refresh some more or less standard graph-theoretic notions (labelled graph, edge, path, path coordinate, etc.) and, second, introduce at the graph-theoretic level some new notions for characterizing finite and infinite path sets in labelled graphs (various kinds of projectivity). Both of these problems are dealt with during this chapter.

2.1 Labelled Graphs

Let us mean by a *labelled graph* a nine-tuple

$$G = (V, E, f, t, L_V, l_V, L_E, l_E, n_0^G), \quad \text{where}$$

- V is a finite set of *vertexes* and E is a finite set of *edges*;
- $f : E \to V$ and $t : E \to V$ give for every edge $e \in E$ its *source* and *target* vertexes, respectively;
- L_V is a set of *vertex labels* and $l_V : V \to L_V$ is a *vertex labelling function*;
- L_E is a set of *edge labels* and $l_E : E \to L_E$ is an *edge labelling function*, finally,
- $n_0^G \in V$ is the graph's *initial vertex*.

We consider also labelled graphs with *acceptance conditions*, the graphs of this kind are defined as tuples

$$G = (V, E, f, t, L_V, l_V, L_E, l_E, n_0^G, S_F, S_I),$$

where, in addition to the graph components, $S_F \subseteq V$ and $S_I \subseteq V$ are called the *finite* and *infinite acceptance sets* for G, respectively.

Definition 2.1 *A finite (resp. infinite) path in the labelled graph G is any sequence of the form $\alpha = n_0e_0, n_1e_1, \ldots, n_k$ (resp. $\alpha = n_0e_0, n_1e_1, \ldots$) where $n_i \in V(G)$ and $e_i \in E(G)$, as well as $f(e_i) = n_i$ and $t(e_i) = n_{i+1}$ for all used edge indices i.*

We introduce for every path α the set $\mathcal{CRd}_\alpha$ of its *coordinates* in a way

- $\mathcal{CRd}_\alpha = \{0, 1, \ldots, k\}$ for a finite path $\alpha = n_0e_0, \ldots, n_k$, and
- $\mathcal{CRd}_\alpha = \mathrm{N} = \{0, 1, 2, \ldots\}$ for an infinite path α.

For a finite path α we call $k = card(\mathcal{CRd}_\alpha) - 1$ the *length* of α (intuitively, the path's length is the number of edges in the path). We use the square brackets $[,]$ to denote the optionality of the last element in an arbitrary path, so the sequence

$$n_0e_0, n_1e_1, \ldots [, n_k]$$

denotes arbitrary finite or infinite path in a specified graph.

Definition 2.2 *A path $\alpha = n_0e_0, n_1e_1, \ldots [, n_k]$ in the graph G is called initial, if it starts from the graph initial vertex (i.e., if $n_0 = n_0^G$).*

We also call a path α starting from the graph G vertex $n \in V$ a n-path in G.

Definition 2.3 *A path α (either initial or not) in the graph with acceptance conditions is called accepting if*

- *for a finite path α its final vertex n_k is in the set S_F,*
- *for an infinite path α the vertexes of the path $n_i \in S_I$ for infinitely many indices i (we call an infinite accepting path also fair).*

For the purpose of the future use we define for two paths α and β in G, such that the path $\alpha = n_0e_0, n_1e_1, \ldots, n_k$ is finite and the path β starts with the last vertex of α (i.e. $\beta = n_ke_k, n_{k+1}e_{k+1}, \ldots [, n_{k+s}]$), the *concatenation* $\alpha + \beta$ in a way

$$\alpha + \beta = n_0e_0, n_1e_1, \ldots, n_ke_k, n_{k+1}e_{k+1}, \ldots [, n_{k+s}].$$

Notation. Given an arbitrary graph G with or without acceptance conditions, we let $V(G)$, $E(G)$, $L_V(G)$, $L_E(G)$ to denote the sets of vertexes, edges, vertex and edge labels of the graph G respectively. The notation of f_G and t_G is further on used for the graph G edge-vertex incidence functions f and g. For G being a graph

with acceptance conditions let $S_F(G)$ and $S_I(G)$ denote the corresponding finite and infinite acceptance sets of G.

Given a labelled graph G, let us denote by $\mathcal{P}_f(G)$ the set of all finite paths of G and by $\mathcal{P}_\omega(G)$ the set of all infinite paths of G. For G being a graph with acceptance conditions, let $\mathcal{P}_{f,a}(G)$ be the set of all finite accepting paths in G and $\mathcal{P}_{\omega,a}(G)$ - the set of all infinite accepting (fair) paths in G. The corresponding sets of initial paths of G (finite, infinite, finite accepting, fair) are denoted by $\mathcal{P}^0_f(G), \mathcal{P}^0_\omega(G), \mathcal{P}^0_{f,a}(G)$ and $\mathcal{P}^0_{\omega,a}(G)$ respectively.

2.2 Projective Path Sets

In this section we introduce some simple notions for characterization of path sets in the labelled graphs, useful later on when speaking about the sets of all (finite, infinite) *feasible* paths in programs.

Definition 2.4 *Given two labelled graphs G and H, such that $L_V(H) = V(G)$ and $L_E(H) = E(G)$, we call a path $\alpha = n_0e_0, n_1e_1, \ldots [, n_k]$ in G a projection of a path $\alpha' = \tilde{n}_0\tilde{e}_0, \tilde{n}_1\tilde{e}_1, \ldots [, \tilde{n}_k]$ in H, written $\alpha = proj(\alpha')$, if the path α coincides with the path α' vertex and edge label sequence, i.e., if*

- $\mathcal{CR}d_\alpha = \mathcal{CR}d_{\alpha'}$ *and*
- $n_i = l_{V(H)}(\tilde{n}_i)$ *and* $e_i = l_{E(H)}(\tilde{e}_i)$ *for all* $i \in \mathcal{CR}d_\alpha$ *($i < k$ for the edge labels in the case of finite α, α').*

We can look at the projection operator *proj* as at a function mapping the set of paths of a given graph H onto the set of sequences of certain kind. In this spirit we allow to write $proj(XX)$ for $XX \subseteq \mathcal{P}_f(H) \cup \mathcal{P}_\omega(H)$ being a subset of the graph H path set to denote the set of the projections of the paths of XX.

In what follows we will be more interested in the characterization of the *initial* path sets, therefore we develop the projectivity theory on the basis of them.

Definition 2.5 *Given a set of finite paths $\mathcal{U} \subseteq \mathcal{P}_f(G)$ of a labelled graph G, we call a graph H a projectivee for the path set $\mathcal{U}$, if $\mathcal{U}$ coincides with the set of projections of all finite initial paths of H, i.e. $proj(\mathcal{P}^0_f(H)) = \mathcal{U}$.*

For $\mathcal{U} \subseteq \mathcal{P}_f(G)$ being a set of finite paths of a graph G we call a graph H with a finite acceptance condition a *T-projectivee* for $\mathcal{U}$, if $\mathcal{U} = proj(\mathcal{P}^0_{f,a}(H))$.
For $\mathcal{U} \subseteq \mathcal{P}_\omega(G)$ being a set of *infinite* paths in G we call the graph H for $\mathcal{U}$

- ω-projectivee, if $\mathcal{U} = proj(\mathcal{P}^0_\omega(H))$,
- F-projectivee, if $\mathcal{U} = proj(\mathcal{P}^0_{\omega,a}(H))$.

Definition 2.6 *A set $\mathcal{U}$ of finite paths in a labelled graph G is called (classically)*

- *projective, if there exists a labelled graph H which is a projectivee for $\mathcal{U}$.*
- *T-projective, if there exists a T-projectivee for $\mathcal{U}$.*

A set $\mathcal{U}$ of infinite paths in G is called classically

- *ω-projective, if there exists an ω-projectivee H for $\mathcal{U}$,*
- *F-projective, if there exists an F-projectivee for $\mathcal{U}$.*

In the case, if this will not lead to confusion, we will use the term "projectivity" also to denote the ω-projectivity of infinite path sets.

In what follows we will be more interested in the *effective* versions of projectivities rather than the classical ones defined above for we want the projectivees for some graph path sets (e.g., the sets of all *feasible* paths in programs) to be built effectively from input of the graphs from some given classes.

Definition 2.7 *Let S be a set of labelled graphs and let for every $G \in S$ $U(G)$ be some finite set, effectively computable from G. Let $\mathcal{P}(G,u) \subseteq \mathcal{P}^0_f(G)$ for every graph $G \in S$ and every $u \in U(G)$ be some set of finite paths in G.*

We call the family of the graph path sets

$$\{P(G,u) \subseteq \mathcal{P}^0_f(G) \mid G \in S \;\&\; u \in U(G)\}$$

effectively projective iff

- *for every $G \in S$ and $u \in U(G)$ the set $\mathcal{P}(G,u)$ is classically projective and*
- *there exists an algorithm which, given the graph $G \in S$ and an element $u \in U(G)$, produces a projectivee for $\mathcal{P}(G)$.*

For the simplicity of formulations we allow ourselves to speak about the effective projectivities of the graph path sets $P(G,u)$ "for every given $G \in S$ and $u \in U(G)$", understanding by this the effective projectivity of the whole graph path set family. We allow also to omit the class S in the definition of the effective projectivity, if it is clear from the context.

In a similar way we define the effective versions of T-projectivity of finite path sets and ω-projectivity and F-projectivity of infinite path sets in graphs from S. Let us further on, unless otherwise stated, mean by the path set projectivities the *effective* versions of them.

The following simple properties of the projective path sets appear useful in various situations of the program path feasibility analysis. Let LL be a class of labelled graphs,

we consider the effective projectivities of the path sets in the graphs $G \in LL$. For the sake of simplicity we do not consider the dependence of the graph path sets on additional parameters $u \in U(G)$, the generalization of the results to the case with the parameters is completely straightforward.

Lemma 2.8 *If for every graph $G \in LL$ finite path sets $A(G) \subseteq \mathcal{P}^0_f(G)$ and $B(G) \subseteq \mathcal{P}^0_f(G)$ are projective (resp. T-projective) then also the sets $C(G) = A(G) \cap B(G)$ and $D(G) = A(G) \cup B(G)$ are projective (resp. T-projective).*

Proof: Let H_A be a projectivee for $A(G)$ and H_B-a projectivee for $B(G)$. We show how to construct the projectivees H_C and H_D for the path sets $C(G)$ and $D(G)$ respectively, so proving the effective projectivity of $C(G)$ and $D(G)$.

Let $V(H_C) = \{\langle n, m\rangle \in V(H_A) \times V(H_B) \mid l_{V(H_A)}(n) = l_{V(H_B)}(m)\}$ with the vertex $\langle n, m\rangle$ labelled by $l_{V(H_C)}(\langle n, m\rangle) = l_{V(H_A)}(n)$. As to the edges in H_C, let us draw an edge $\tilde{e}$ from $\langle n, m\rangle$ to $\langle n', m'\rangle$ labelled by l if and only if there are edges labelled by l both in H_A from n to n' and in H_B from m to m'. It is easy to see that H_C is the desired projectivee for $C(G)$ provided the graph initial vertex $n_0^{H_C} = \langle n_0^{H_A}, n_0^{H_B}\rangle$.

As to the T-projectivity of $C(G)$, it suffices to define for the graph H_C the finite acceptance set $S_F(H_C) = \{\langle n, m\rangle \mid n \in S_F(H_A),\ m \in S_F(H_B)\}$.

For the set $D(G)$ the corresponding projectivees are constructed, by taking a copy of H_A and a copy of H_B, as well as a new initial vertex $n_0^{H_D}$. We label the vertex $n_0^{H_D}$ by n_0^G and define it to have the outgoing edges with targets and labels of both all edges outgoing in H_A from $n_0^{H_A}$ and all edges outgoing in H_B from $n_0^{H_B}$ (in the case of H_A and H_B being the T-projectivees, the finite acceptance set for H_D is defined as $S_F(H_A) \cup S_F(H_B) \cup \Omega$, where $\Omega = \{n_0^{H_D}\}$, if either $n_0^{H_A} \in S_F(H_A)$, or $n_0^{H_B} \in S_F(H_B)$, $\Omega = \emptyset$ otherwise).□

Lemma 2.9 *If for every graph $G \in LL$ infinite path sets $A(G) \subseteq \mathcal{P}^0_\omega(G)$ and $B(G) \subseteq \mathcal{P}^0_\omega(G)$ are ω-projective (resp. F-projective) then also the sets $C(G) = A(G) \cap B(G)$ and $D(G) = A(G) \cup B(G)$ are ω-projective (resp. F-projective).*

Proof: The cases of $D(G)$ projectivities are dealt with by the same method, as in the proof of Lemma 2.8.

As to the set $C(G)$, we construct a graph H_C, as in the proof of Lemma 2.8. Easy to see that H_C is an ω-projectivee of $C(G)$.

In the case of F-projectivity of $A(G)$ and $B(G)$ we construct also first the graph H_C. We construct furthermore a new graph H'_C provided with an infinite acceptance set, which will serve as the F-projectivee for $C(G)$. We define every vertex of H'_C to be a pair $\langle z, a\rangle$ with $z = \langle n, m\rangle \in V(H_C)$ and $a \in \{0, 1, 2\}$. An edge from $\langle z, a\rangle$ to $\langle z', a'\rangle$ with the label l in H'_C is drawn if and only if

- there is an edge from z to z' labelled by l in H_C and

- one of the following holds:
  - $a = a' = 1$ or $a = a' = 2$, or
  - $a = 0$, $a' = 2$ and $z = \langle n, m \rangle$ with $n \in S_I(H_A)$, or
  - $a = 0$, $a' = 1$ and $z = \langle n, m \rangle$ with $m \in S_I(H_B)$, or
  - $a = 1$, $a' = 0$ and $z' = \langle n', m' \rangle$ with $n' \in S_I(H_A)$, or
  - $a = 2$, $a' = 0$ and $z' = \langle n', m' \rangle$ with $m' \in S_I(H_B)$.

We define the graph's H'_C initial vertex to be $\langle \langle n_0^{H_A}, n_0^{H_B} \rangle, 1 \rangle$ and the infinite acceptance set $S_I(H'_C) = \{\langle z, a \rangle \mid a = 0\}$.

It is easy to see that in H'_C accepting are only those paths which infinitely many times visit both the vertexes $\langle \langle n, m \rangle, a \rangle$ with $n \in S_I(H_A)$ and the ones with $m \in S_I(H_B)$ (for a path to be accepting in H'_C, it must for at least one $a \in \{1, 2\}$ both infinitely many times leave the region of vertexes with the second component a and infinitely many times return to it, what is possible only if the path infinitely many times visits both $\langle \langle n, m \rangle, a \rangle$ for $n \in S_I(H_A)$ and for $m \in S_I(H_B)$). So, if a path in H'_C is accepting, its projection belongs to both $A(G)$ and $B(G)$.

On the other hand, for every path $\alpha \in A(G) \cap B(G)$ it is easy to find a path α' in H'_C such that both α' is accepting and $\alpha = proj(\alpha')$, when one builds the path α' inductively, it suffices to choose the target vertex $\langle z, a \rangle$ of the current edge with $a = 0$ whenever possible. □

We show also the following sufficient condition for the path set projectivity which is often in one or other form used in program reachability graph construction (see, for instance [BBK74, ABBCK91]), we will find it later on useful also for our purposes.

Lemma 2.10 *If in every labelled graph $G \in LL$ every finite path $\alpha \in \mathcal{P}_f^0(G)$ has an effective (computable from α) characteristic $S(\alpha)$ such that*

- *for all $\alpha \in \mathcal{P}_f^0(G)$ the characteristic $S(\alpha) \in \mathcal{S}$ for a finite characteristic set $\mathcal{S}$, and*
- *for all $\alpha, \beta \in \mathcal{P}_f^0(G)$ ending with one vertex $n \in V(G)$ for all paths γ starting from n in G $S(\alpha + \gamma) = S(\beta + \gamma)$,*

then for every $\mathcal{S}^0 \subseteq \mathcal{S}$ the set of paths $A(G) = \{\alpha \in \mathcal{P}_f^0(G) \mid S(\alpha) \in \mathcal{S}^0\}$ is (effectively) T-projective.

Proof: We construct a T-projectivee H for the set $A(G)$. Every vertex of H is a pair $\langle n, S \rangle$ for $n \in V(G)$, $S \in \mathcal{S}$, and is labelled by n. We build the graph H inductively by starting with the initial vertex $n_0^H = \langle n_0^G, S(n_0^G) \rangle$ (n_0^G is a path of length 0 in G) and draw an edge from $\langle n, S \rangle$ to $\langle n', S' \rangle$, labelled by $e \in E(G)$, whenever both

- the edge e in G leads from n to n', and
- there exists an initial path α' ending with $\langle n, S\rangle$ in the already drawn part of H, such that for $\alpha = proj(\alpha')$ $S(\alpha + ne, n') = S'$.

It is easy to see both that this edge drawing process eventually stops (since both the sets $V(G)$ and $\mathcal{S}$ are finite) and that for any α' being an initial path in H, which ends with $\langle n, S\rangle$ and has $\alpha = proj(\alpha')$, the characteristic $S(\alpha) = S$. Now it remains to define the finite acceptance set $S_F(H) = V(G) \times \mathcal{S}^0$ for H, so easily obtaining the required T-projectivity.□

For the sake of completeness in studying the notions of the path set projectivity, we investigate also the closeness under the complement of various kinds of projective graph path set classes.

As it is easy to see from simple examples, both in the cases of finite and infinite projective path set classes the complementability property does not hold. As to the T-projectivity and F-projectivity, the following results together with Lemma 2.8 and Lemma 2.9 show that these path set classes are closed under all boolean algebraic operations.

Lemma 2.11 *If for every graph $G \in LL$ a finite path set $A(G) \subseteq \mathcal{P}_f^0(G)$ is T-projective, then also its complement $B(G) = \mathcal{P}_f^0(G) \setminus A(G)$ is T-projective.*

Proof: First of all define for an arbitrary labelled graph H its *characterizing automaton* H^a as a finite automaton with the set of states $V(H)$ and external (input) alphabet $V(H) \times E(H) \times V(H)$. For every edge $e \in E(H)$, leading from $n_1 = f_H(e)$ to $n_2 = t_H(e)$ in H, in H^a a corresponding edge $\tilde{e}$ from n_1 to n_2, labelled by the triple $\langle n_1, e, n_2\rangle$, is drawn. The initial vertex of H^a is defined to be n_0^H and the set of accepting states is $S_F(H)$ (the automaton H^a accepts some words in the alphabet $V(H) \times E(H) \times V(H)$ according to the standard finite automaton word accepting discipline).

In order to show the T-projectivity of $B(G)$, we start with a T-projectivee H_A for $A(G)$. We build a characterizing automaton H_A^a for H_A and afterwards translate each its edge label $\langle f_H(e), e, t_H(e)\rangle$ into $\langle l(f_H(e)), l'(e), l(t_H(e))\rangle$ for $l = l_{V(H)}$ and $l' = l_{E(H)}$ being the H vertex and edge labelling functions, respectively. Let us denote the obtained automaton by H'. It is easy to see that H' accepts those and only those words in the alphabet $V(G) \times E(G) \times V(G)$ which correspond to the paths $\alpha \in A(G)$ of G (corresponding to the path $\alpha = n_0e_0, n_1e_1, \ldots, n_k$ in G is the word

$$\langle n_0, e_0, n_1\rangle, \langle n_1, e_1, n_2\rangle, \ldots, \langle n_{k-1}, e_{k-1}, n_k\rangle).$$

It is easy to see that the language of the words corresponding to G paths $\beta \in B(G)$ can be accepted by some finite automaton with the external alphabet $V(G) \times E(G) \times V(G)$

for it is the intersection of the language of G characterizing automaton G^c with the complement of H' language. The T-projectivity of $B(G)$ follows immediately.□

Lemma 2.12 *If for every graph $G \in LL$ an infinite path set $A(G) \subseteq \mathcal{P}^0_\omega(G)$ is F-projective, then also its complement $B(G) = \mathcal{P}^0_\omega(G) \setminus A(G)$ is F-projective.*

Proof: The proof follows precisely the same lines as the proof of Lemma 2.11, using the infinite path acceptance condition $S_I(H)$ instead of $S_F(H)$. The most sophisticated step of the proof is to show the complementability of Büchi-acceptable infinite word language class, for it the reader is referred to, for instance, [SVW87].□

Chapter 3

Language Definitions

We begin with the definition of the programming language LBASE for dealing with integer-valued data.

3.1 The Language LBASE

Every program P in the language (family of programs) LBASE is a labelled graph

$$P = \langle V, E, f, t, L_V, l_V, L_E, l_E, n_0^P \rangle$$

(see Section 2.1) provided with a finite number of integer-valued internal variables $x_1, \ldots, x_m$ and, intuitively, with an input gate for receiving the variable values from the environment of the program.

Every edge of the program P (i.e. $e \in E$) is labelled either by "+" or by "−" (i.e. $l_E(e) \in L_E = \{"+", "-"\}$ for all $e \in E$). Every P vertex $n \in V(P)$ is labelled by an *operator* of one of the following kind (c denotes here any integer constant, i.e. $c \in \mathbf{Z}$. Here and in the following we use the convention to put variable and constant names in boldface whenever they are used as syntactical parts of operators, otherwise (in mathematical expressions, etc.) they are put in *italics*):

- $?\mathbf{x}_i$ for $1 \leq i \leq m$, the *input* operator, denotes the reading of a (new) value from the program input gate into the variable x_i;
- $\mathbf{x}_i \leftarrow \mathbf{x}_j$ ($\mathbf{x}_i \leftarrow \mathbf{c}$) for $1 \leq i, j \leq m$ and $c \in \mathbf{Z}$, the *assignment* operator, denotes the assigning of the value of the variable x_j (the constant c) to x_i;
- $\mathbf{x}_i < \mathbf{x}_j$ ($\mathbf{x}_i < \mathbf{c}$, $\mathbf{c} < \mathbf{x}_j$) for $1 \leq i, j \leq m$ and $c \in \mathbf{Z}$, the *comparison* operator, denotes the comparison of the two variables (or the variable and the constant), during the execution produces an output flag "+" or "−", depending on whether the actual values of x_i, x_j satisfy the operator's inequality or not (used

to determine the outgoing edges of the current vertex along which the further program control flow can be passed);

- **NOP**, the dummy operator, does not have any effect on the variable values and allows any further control flow.

We consider also LBASE programs with acceptance sets by introducing in the tuple for program definition arbitrary subsets S_F and S_I of the program vertex set V.

Further on let us use the notation $p(n)$ (instead of $l_V(n)$) to denote the operator, attached to a program vertex $n \in V$.

Intuitively, at the beginning of the program execution every variable x_i is assigned the "value" 0. During the program execution every program variable can assume integer values, i.e. at every moment of the program execution the program "variable value vector" is an element of $\mathbf{Z}^m$. Let us give the formal semantics for program operators in the terms of variable value vector transformation and the effects the operators impose on the further program control flow.

For arbitrary LBASE program P we define its set of *variable value vectors*, denoted by $\mathcal{V}_P$, to be $\mathbf{Z}^m$, where m is the program P variable set cardinality. We call the element $\vec{v}_0^P = (0, \ldots, 0) \in \mathcal{V}_P$ the initial P variable value vector. We denote the components of any m-dimensional vector $\vec{v} \in \mathbf{Z}^m$ as $\vec{v}.x_1, \ldots, \vec{v}.x_m$ (so stressing the relation between the variable value vector components and the program variables (the value of the variable x_i in the vector $\vec{v} \in \mathbf{Z}^m$ is $\vec{v}.x_i$)).

Let $\vec{v}[x_i \leftarrow c]$ for any vector $\vec{v} \in \mathbf{Z}^m$ denotes another vector $\vec{v}'$ with all but the ith component coinciding with corresponding components of $\vec{v}$ (i.e. for $j \neq i$ $\vec{v}'.x_j = \vec{v}.x_j$), the ith component of $\vec{v}'$ is taken to be c ($\vec{v}'.x_i = c$, it is required that $c \in \mathbf{Z}$ for this construction being admissible).

The semantics for every language operator p is given as two following functions:

- ϕ_p, the variable value transformation function, for p being an assignment, comparison or dummy operator $\phi_p : \mathcal{V}_P \rightarrow \mathcal{V}_P$, for an input operator p $\phi_p : \mathcal{V}_P \times \mathbf{Z} \rightarrow \mathbf{Z}^m$ (the extra $\mathbf{Z}$ intuitively standing for the domain of values, appearing at the program input gate) and
- $\psi_p : \mathcal{V}_P \rightarrow 2^{\{"+","-"\}}$ ($\psi_p : \mathcal{V}_P \times \mathbf{Z} \rightarrow 2^{\{"+","-"\}}$), the set-valued function of admitted operator's output flags.

Assume $\vec{v} \in \mathcal{V}_P = \mathbf{Z}^m, c \in \mathbf{Z}, 1 \leq i, j \leq m$, the definitions of functions ϕ_p and ψ_p for a program operator p are, as follows:

- for p being an input operator $?\mathrm{x_i}$ $\phi_p(\vec{v}, c) = \vec{v}[x_i \leftarrow c]$, $\psi_p(\vec{v}, c) = \{" + "\}$;
- for p being an assignment operator $\mathrm{x_i} \leftarrow \mathrm{x_j}$ $\phi_p(\vec{v}) = \vec{v}[x_i \leftarrow \vec{v}.x_j]$, for p being the operator $\mathrm{x_i} \leftarrow \mathrm{c}$ $\phi_p(\vec{v}) = \vec{v}[x_i \leftarrow c]$, in both cases $\psi_p(\vec{v}) = \{" + "\}$;

- for a comparison operator p always $\phi_p(\vec{v}) = \vec{v}$, the output flag function is defined, as follows:
  - if p is an operator $\mathbf{x_i} < \mathbf{x_j}$ then $\psi_p(\vec{v}) = \{"+"\}$, if $\vec{v}.x_i < \vec{v}.x_j$, otherwise $\psi_p(\vec{v}) = \{"-"\}$,
  - if p is an operator $\mathbf{c} < \mathbf{x_j}$ then $\psi_p(\vec{v}) = \{"+"\}$, if $c < \vec{v}.x_j$, otherwise $\psi_p(\vec{v}) = \{"-"\}$,
  - if p is an operator $\mathbf{x_i} < \mathbf{c}$ then $\psi_p(\vec{v}) = \{"+"\}$, if $\vec{v}.x_i < c$, otherwise $\psi_p(\vec{v}) = \{"-"\}$;
- for p being **NOP** always $\phi_p(\vec{v}) = \vec{v}$ and $\psi_p(\vec{v}) = \{"+","-"\}$.

The semantics of a whole LBASE program is defined by its *state transition system*:

$$\langle V \times \mathcal{V}_P, E, \longrightarrow, \langle n_0^P, \vec{v}_0^P \rangle \rangle \quad \text{where}$$

- $V \times \mathcal{V}_P$ is the set of system's *states* (i.e. every state of the system is a pair $\langle n, \vec{v} \rangle$ where $n \in V$ is a program vertex and $\vec{v} \in \mathcal{V}_P$ is a variable value vector);
- $s_0^P = \langle n_0^P, \vec{v}_0^P \rangle$ is the system's *initial state* (intuitively, the program execution begins from its initial vertex n_0^P with every variable initialized with 0 value);
- $\longrightarrow \subseteq (V \times \mathcal{V}_P) \times E \times (V \times \mathcal{V}_P)$ is the transition relation between states (every transition has its start state, end state and is labelled by some program edge $e \in E$), defined the following way (we use the more comprehensible notation $s_1 \xrightarrow{e} s_2$ to denote $(s_1, e, s_2) \in \longrightarrow$):

  $$\langle n_1, \vec{v}_1 \rangle \xrightarrow{e} \langle n_2, \vec{v}_2 \rangle$$

  iff the edge e is leading from n_1 to n_2, the edge label $l_E(e) \in \psi_{p(n_1)}(\vec{v}_1)$ and the new variable value vector $\vec{v}_2 = \phi_{p(n_1)}(\vec{v}_1)$ (for $p(n_1)$ being an input operator $l_E(e) \in \psi_{p(n_1)}(\vec{v}_1, a)$ and $\vec{v}_2 = \phi_{p(n_1)}(\vec{v}_1, a)$ for some integer $a \in \mathbf{Z}$. For the purpose of the path feasibility studies it is not necessary to consider the inputs of different values as different transitions, however later on we will sometimes use the notation $s_1 \xrightarrow{e:a} s_2$ to denote that the transition from s_1 to s_2 is done along the edge e by reading the value a from the environment).

Without loosing generality we can assume that for every vertex $n \in V(P)$ for a LBASE program P with either an input or assignment operator $p(n)$ attached all outgoing edges are labelled by "+" (according to the defined semantics (state transition system) the "−"-labelled edges from these vertexes cannot affect the program behaviour in any way).

One more observation to be made is that the programs in the language are, in general, highly nondeterministic for we do not impose any requirements on the ways of the program connectivity (the number of, say, "+"-labelled edges, outgoing from one vertex in the program is not a priori bounded).

3.2 Languages LTIM and LTIBA

Every program in LTIM also, as the LBASE programs are, is a graph provided with a finite number of internal variables and (intuitively) with an input gate for receiving variable values from the environment. The first difference of LTIM programs from LBASE programs is that every program variable may assume *rational* values (instead of integer ones in LBASE), rational are also the values the program receives through its input gate.

The programs in LTIM has as the permitted vertex labels (operators ascribed to its vertexes) first of all the analogues of LBASE operators (here c denotes a rational constant):

- ?x, the input operator;
- $x_i \leftarrow x_j$ ($x_i \leftarrow c$), the assignment operator;
- $x_i < x_j$ ($x_i < c$, $c < x_j$), the comparison operator;
- **NOP**, dummy operator.

Furthermore, every LTIM program has one special internal variable z, named the *real time counter*, with the following operators at the program vertexes permitted:

- $z \stackrel{+}{\leftarrow} x_i$, the *positive assignment* operator, assigns the value of x_i to z, provided that the value of z does not decrease via this assignment. The operator produces the output flag "+", if the assignment was successful, otherwise (i.e., if the value of x_i is less than that of z) the "−" flag is produced (the value of z is not changed in this case);
- $x_i \leftarrow z + c$, the *variable activation* operator, assigns the value of z increased by c to x_i (for the sake of simplicity only nonnegative c's are allowed here).

As in the LBASE case, we require every LTIM program P edge $e \in E(P)$ to have a label $l_{E(P)}(e) \in \{"+","-"\}$. We consider also the LTIM programs with acceptance conditions by introducing in the program (labelled graph) defining tuple arbitrary subsets S_F and S_I of the program vertex set V, as well as use the notation $p(n)$ (instead of $l_V(n)$) to denote the operator, attached to a program vertex $n \in V$.

We give the formal semantics of LTIM programs in a way similar to the defining semantics for LBASE programs in Section 3.1.

Since every program variable now may assume rational values, we define for arbitrary LTIM program P its set of *variable value vectors*, denoted by $\mathcal{V}_P$, to be $\mathbf{Q}^{+0} \times \mathbf{Q}^m$, where m is the number of ordinary P variables (not including the real time counter), by $\mathbf{Q}^{+0}$ we denote here and further on the set of nonnegative rational

numbers (the usage of $\mathbf{Q}$ for the set of all rationals is standard). We define the element $\vec{v}_0^P = (0,\ldots,0) \in \mathcal{V}_P$ to be the initial P variable value vector. Let us denote the components of any $m+1$-dimensional vector $\vec{v} \in \mathbf{Q}^{+0} \times \mathbf{Q}^m$ as $\vec{v}.z, \vec{v}.x_1, \ldots, \vec{v}.x_m$ (with the intention to use such vectors $\vec{v}$ as program the program P variable value vectors)

As in the case of LBASE programs, we introduce the notation $\vec{v}[x_i \leftarrow c]$ for $\vec{v} \in \mathbf{Q}^{+0} \times \mathbf{Q}^m$, $c \in \mathbf{Q}^{+0}$ and $i = 0,1,\ldots,m$ (assume $x_0 \equiv z$, meaning by $\equiv$ the syntactical identity) in a way

- $\vec{v}[x_i \leftarrow c].x_i = c$ and
- $\vec{v}[x_i \leftarrow c].x_j = \vec{v}.x_j$ for $x_j \not\equiv x_i$.

The semantics for every LTIM operator p is given as two functions:

- ϕ_p, the variable value transformation function, for p being an assignment, comparison or dummy operator $\phi_p : \mathcal{V}_P \rightarrow \mathcal{V}_P$, for an input operator p $\phi_p : \mathcal{V}_P \times \mathbf{Q} \rightarrow \mathcal{V}_P$ and
- $\psi_p : \mathcal{V}_P \rightarrow 2^{\{"+","-"\}}$ ($\psi_p : \mathcal{V}_P \times \mathbf{Q} \rightarrow 2^{\{"+","-"\}}$ for an input operator p), the set-valued function of admitted operator's output flags.

Assume $\vec{v} \in \mathcal{V}_P, c \in \mathbf{Q}$, $1 \le i,j \le m$, the definitions of functions ϕ_p and ψ_p for input, assignment and comparison and dummy (NOP) operator p in the program P are precisely the same, as for corresponding LBASE operators (see Section 3.1). For p being a specific LTIM operator, its semantics is defined, as follows:

- if p is a positive assignment operator $\mathrm{z} \overset{+}{\leftarrow} \mathrm{x_j}$ then
  - if $\vec{v}.z \le \vec{v}.x_j$ then $\phi_p(\vec{v}) = \vec{v}[z \leftarrow \vec{v}.x_j]$ and $\psi_p(\vec{v}) = \{" + "\}$,
  - otherwise (i.e., if $\vec{v}.z > \vec{v}.x_j$) $\phi_p(\vec{v}) = \vec{v}$ and $\psi_p(\vec{v}) = \{" - "\}$;
- for p being a variable activation operator $\mathrm{x_i} \leftarrow \mathrm{z} + \mathrm{c}$ $\phi_p(\vec{v}) = \vec{v}[x_i \leftarrow \vec{v}.z + c]$ and $\psi_p(\vec{v}) = \{" + "\}$.

The semantics of a whole LTIM program is defined by its *state transition system* just as for LBASE programs (see Section 3.1) with the only exception being $a \in \mathbf{Q}$ (instead of $a \in \mathbf{Z}$) taken as the second argument in the semantic functions ϕ_p and ψ_p for an input operator p.

As in the case of LBASE programs we assume that for every vertex $n \in V(P)$ for a LTIM program P with either an input or assignment, or variable activation operator $p(n)$ attached all outgoing edges are labelled by " + ".

It is important to note that the LTIM program real time counter z cannot appear in the input operators, as well as the lefthand side of any assignment operator. So the

variable z is not subject for arbitrary value assignment, one can find out that there is no way to decrease the z value during the program execution. Easy to see that without this or some other discipline limiting the use of the counter variable (i.e., if we allow it to receive arbitrary values from the input and retain all other kinds of operators in the programs), we could not obtain any decidability results for algorithmic problems in the considered program analysis.

The language LTIM appears to be able to express rather general time-dependent behaviour of real time systems (see Section 9.2 and Section 10 below for the detailed discussion on the real time system modelling by LTIM programs). We consider also some sublanguages of LTIM the programs in which appear later on to be the subject of easier automated analysis.

First of such languages, we call it LBASQ, is the language consisting of LTIM programs *without* real time counter. To put it otherwise, in the language LBASQ programs are permitted just the language LBASE operators, except that all constants used in them can be arbitrary rationals, as well as all program variables are rational-valued. For the analysis of LBASQ programs it is the best to consult Section 4.4 where the absence of the real time counter in the programs is exploited in order to obtain more general results than those possible for the general case of LTIM programs.

Yet another sublanguage of LTIM, useful when modelling temporal quantitative constraints on r.t.s. is LTIM_0 which constrains the programs of using the LTIM operators by dividing the ordinary program variables into two categories (along with the ordinary variables every LTIM_0 program can have also a real time counter):

- simple variables $x_1, x_2, \ldots, x_s$, allowed to appear just in the comparison, variable activation and both sides of the assignment operators;
- a single special input variable x which is the *only* program variable which can be used in the input operators and the right-hand side of the positive assignment operators. It is required also that no assignment of the form $x_s \leftarrow x$ is used in the program (i.e. there is no way, how to get an arbitrary value from the input into a simple variable except than moving this value through z).

An important language considered in the theses is LTIBA with every program P in it allowed to have both a finite number of integer valued variables $x_1^B, x_2^B, \ldots, x_m^B$ with allowed LBASE operators over them and a finite number of rational valued variables $z, x_1^T, x_2^T, \ldots, x_n^T$ with allowed LTIM operators (z is the real time counter). Every vertex in a LTIBA program P has either a LTIM or LBASE operator, ascribed to it (type mismatch in LTIBA programs is forbidden).

As in the LBASE and LTIM programs, every edge of P has either a label $\{" + "\}$ or $\{" - "\}$, one of the program vertexes, n_0^P, is marked as initial. We follow also the conventions not to ascribe $\{" - "\}$ labels to the edges outgoing from the vertexes with input, assignment or variable activation operators.

Given a LTIBA program P we define for it the variable value sets $\mathcal{V}_P^B = \mathbf{Z}^m$ and $\mathcal{V}_P^T = \mathbf{Q}^{+0} \times \mathbf{Q}^n$, let $\mathcal{V}_P = \mathcal{V}_P^B \times \mathcal{V}_P^T$.

For the sake of simplicity in defining the semantics of the program P we associate with it a LBASE program P^B which is obtained from P by replacing all LTIM operators, used in P by a LBASE operator NOP (clearly, P_B is a LBASE program, moreover $V(P^B) = V(P)$ and $E(P^B) = E(P)$, changed are just the program vertex labels) and a LTIM program P^T, obtained from P by replacing all LBASE operators by a LTIM operator NOP.

Clearly, $\mathcal{V}_{PB} = \mathcal{V}_P^B$ and $\mathcal{V}_{PT} = \mathcal{V}_P^T$. Let the program P initial variable value vector $\vec{v}_0^P = \langle \vec{v}_0^{P^B}, \vec{v}_0^{P^T} \rangle$ for $\vec{v}_0^{P^B}$ being the program P^B initial variable value vector and $\vec{v}_0^{P^T}$ being the initial variable value vector for the program P^T.

The semantics of P is defined as in the case of LBASE and LTIM programs by the state transition system

$$\langle V(P) \times \mathcal{V}_P, E(P), \longrightarrow, \langle n_0^P, \vec{v}_0^P \rangle \rangle \quad \text{where}$$

- $V \times \mathcal{V}_P$ is the set of system's *states*
- $s_0^P = \langle n_0^P, \vec{v}_0^P \rangle$ is the system's *initial state*
- $\longrightarrow \subseteq (V(P) \times \mathcal{V}_P) \times E(P) \times (V(P) \times \mathcal{V}_P)$ is the transition relation between states defined in a way (using the notation $s_1 \xrightarrow{e} s_2$ to denote $\langle s_1, e, s_2 \rangle \in \longrightarrow$):

$$\langle n_1, \langle \vec{v}_1^B, \vec{v}_1^T \rangle \rangle \xrightarrow{e} \langle n_2, \langle \vec{v}_2^B, \vec{v}_2^T \rangle \rangle$$

iff both

- $\langle n_1, \vec{v}_1^B \rangle \xrightarrow{e} \langle n_2, \vec{v}_2^B \rangle$ in P^B and
- $\langle n_1, \vec{v}_1^T \rangle \xrightarrow{e} \langle n_2, \vec{v}_2^T \rangle$ in P^T.

3.3 Feasibility and Reachability

Let LL denote any of the three programming languages LBASE, LTIM and LTIBA.

We define for any finite or infinite path $\alpha = \tilde{n}_0\tilde{e}_0, \tilde{n}_1\tilde{e}_1, \ldots [, \tilde{n}_k]$ in a LL-program P the selectors for the ith vertex of α to be $n_i(\alpha) = \tilde{n}_i$ and the ith edge by $e_i(\alpha) = \tilde{e}_i$. For a finite path α the last vertex of α is denoted by $n_*(\alpha)$ (i.e. $n_*(\alpha) = \tilde{n}_k$).

A *history* of a LL-program P is any sequence

$$\beta = s_0e_0, s_1e_1, \ldots [, s_k]$$

with $s_i = \langle n_i, \vec{v}_i \rangle$ for every used index i, provided $s_i \xrightarrow{e_i} s_{i+1}$ for every i; $i \leq k-1$ in the case of the finite sequence β.

We define also the selectors $s_i(\beta) = s_i$, $e_i(\beta) = e_i$, as well as $n_i(\beta) = n_i$ and $\vec{v}_i(\beta) = \vec{v}_i$ to denote the ith state, edge, graph vertex and variable value vector of β, respectively. For a finite history β we define also $s_*(\beta) = s_k$, $n_*(\beta) = n_k$ and $\vec{v}_*(\beta) = \vec{v}_k$ to denote the last state, vertex and variable value vector of the history β.

We call a history β of a program *initial* if the history's first state $s_0(\beta)$ is the initial state s_0^P of the program's state transition system.

It is according to the definition of the history that for a every history β the sequence $\alpha(\beta) = n_0(\beta)e_0(\beta), n_1(\beta)e_1(\beta), \ldots [, n_*(\beta)]$ is a path in the program, let us say that the history β goes *along* the path $\alpha(\beta)$ in the program. Let for every history β along the path α in the program the set of coordinates $\mathcal{CR}d_\beta = \mathcal{CR}d_\alpha$.

Definition 3.1 *A path α in a program P is called weakly feasible iff there is a history along it. An initial path α in a program is called feasible iff it has an initial program history along it.*

It is easy to see that there can be initial paths in LL programs, which are weakly feasible, but not feasible, e.g. in the case of LBASE program one can consider a one-edge path

$$\mathsf{x > 0;\ "+";\ ?y}$$

provided the vertex to which the operator "x > 0" is ascribed to is the program P initial vertex n_0^P (it is for the sake of readability that we show in the examples here and further on only the path vertex and edge *labels*).

Definition 3.2 *Let P be a LL-program with acceptance sets $S_F(P)$ for finite paths and $S_I(P)$ for infinite paths. We call an initial path α in P correct iff it is accepting. A vertex $n \in V(P)$ is called*

- *reachable iff it is contained in some feasible path α (i.e. $n = n_i(\alpha)$ for some i);*
- *correctly reachable iff it is contained in some feasible correct path of P;*
- *f-correctly reachable iff it is contained in some finite feasible correct path of P;*
- *ω-correctly reachable iff it is contained in some infinite correct path of P.*

It is easy to see that both f-correct reachability and ω- correct reachability as well as ordinary reachability problems for every LL-program P reduce to the correct reachability by choosing appropriate acceptance sets $S'_F(P), S'_I(P)$:

- reachability: $S'_F(P) = V(P)$, $S'_I(P) = V(P)$ (actually, $S'_I(P)$ is irrelevant);
- f-correct reachability: $S'_F(P) = S_F(P)$, $S'_I(P) = \emptyset$;
- ω-correct reachability: $S'_F(P) = \emptyset$, $S'_I(P) = S_I(P)$.

Solving the vertex reachability problem for some class of programs, it is sometimes useful to demonstrate in the case of the positive answer on the reachability, *how* the given vertex is reached. In the case of the ordinary reachability this problem can be simply solved by exhibiting a history along some path, leading to this vertex. If, however, some kind of correct vertex reachability is considered, it seems useful that one could exhibit such a history reaching the given vertex, which could be *continued* along some accepting path. For the case of f-correct reachability the problem can be solved by exhibiting a history along the whole accepting path, containing the given vertex (this path is *finite*). As to the ω-correct reachability we have, in general, no ways, how to define the whole history along some fair path in the program in a finite time. To overcome this problem we define the following notion of the history computing (generating) by an algorithm.

Definition **3.3** *We say that an algorithm $\mathcal{M}$ computes (generates) an infinite history ν along a path α in a LL program P, if on the input of the path α coordinate $k \in \mathcal{CR}d_\alpha = \mathbf{N}$ and the program P variable x_i, the algorithm $\mathcal{M}$ in a finite time outputs the value $\vec{v}_k(\nu).x_i$ of the variable x_i in the history ν at the path α coordinate k.*

One could allow such an algorithm to use various possibly infinite input data structures containing the description of the path α (or some other extra input, specified for each concrete history computing problem separately). However, since the algorithm is required to work in finite time (= finite number of steps), it can use for determining the value $\vec{v}_k(\nu).x_i$ for fixed i and k only the information from some *finite* path α fragment (finite region of the input data in general).

In what follows, we proceed to deciding the considered reachability problems, as well as investigate the possibilities to characterize (in the terms of projectivity, see Section 2.2) the sets of feasible (correct) paths in LBASE, LTIM and LTIBA programs.

Chapter 4

Finite Path Feasibility

The aim of this chapter is to give the most of the proof of the following theorem:

Theorem 4.1 *The set of all finite feasible paths of every LTIBA program is effectively projective.*

We base the proof of the finite feasible path set projectivity in given LTIBA program P on the explicit construction of the graph containing as projections of paths all feasible paths in P. The proofs of the finite feasible path set projectivity are given for LBASE and LTIM programs independently, i.e. we prove

Theorem 4.2 *There exists an algorithm which, given a LBASE program P, constructs for it a graph $G(P)$ such that the set of all finite (initial) feasible paths in P coincides with the set of the projections of all finite initial paths in $G(P)$.*

Theorem 4.3 *There exists an algorithm which, given a LTIM program P, constructs for it a graph $G(P)$ such that the set of all finite (initial) feasible paths in P coincides with the set of the projections of all finite initial paths in $G(P)$.*

After having proved Theorem 4.2 and Theorem 4.3, the overall result of Theorem 4.1 for LTIBA programs can be obtained according to Lemma 2.8 the following way.

Given a LTIBA program P we construct for it the LBASE program P^B by replacing all LTIM operators in P by a LBASE operator NOP and the program P^T by replacing all LBASE operators of P by NOP. Easy to see that a sequence

$$\beta = \langle n_0, \langle v_0^B, v_0^T \rangle \rangle e_0, \langle n_1, \langle v_1^B, v_1^T \rangle \rangle e_1, \ldots [, \langle n_k, \langle v_k^B, v_k^T \rangle \rangle]$$

is a history of P along a path α if and only if both

- $\beta^B = \langle n_0, v_0^B \rangle e_0, \langle n_1, v_1^B \rangle e_1, \ldots [, \langle n_k, v_k^B \rangle]$ is a history of P^B along α and

- $\beta^T = (n_0, v_0^T)e_0, (n_1, v_1^T)e_1, \ldots [, (n_k, v_k^T)]$ is a history of P^T along α.

So, a path α in P is feasible if and only if it is feasible both in P^B and P^T, the applicability of Lemma 2.8 is straightforward.

Before we start the description of the projectivees, called also the path feasibility graphs, for LBASE and LTIM programs, let us note that the constructions (including the graphs themselves) made during the proof are to be used afterwards (in the infinite path feasibility analysis) also without direct relation to the theorem.

4.1 Variable Vector Value Set Partitionings

We begin with the description of the path feasibility graph vertexes for LBASE and LTIM programs. For this purpose we define for a given LBASE or LTIM program P a finite partitioning C_P of the set $\mathcal{V}_P$ of all program P variable value vectors the following way.

Let c_{min} be the minimal and c_{max} the maximal among the constants, used in the operators of P, and the initial variable value 0. Let $Cons(P) = \{c_{min}, c_{min} + 1, \ldots, c_{max}\}$ be the set of P constants (recall that we have agreed to have all constants for both LBASE and LTIM programs *integers*), $Vars(P) = \{x_1, x_2, \ldots, x_m\}$ - the set of P variables, let $\mathcal{A}_P = Cons(P) \cup Vars(P)$.

Let for a variable value vector $\vec{v} \in \mathcal{V}_P$ and a constant $c \in Cons(P)$ always $\vec{v}.c = c \in \mathbb{Z}$ (we already have the notation on $\vec{v}.x$ to denote the value of the variable x in the vector $\vec{v}$).

We consider first the case of a LBASE program P.

Definition 4.4 *Two variable value vectors $\vec{v}_1 \in \mathcal{V}_P$ and $\vec{v}_2 \in \mathcal{V}_P$ of a LBASE program P are called equivalent, written $\vec{v}_1 \cong \vec{v}_2$, and belong to one C_P-equivalence class iff for any $a, b \in \mathcal{A}_P$ the relation $\vec{v}_1.a \leq \vec{v}_1.b$ holds if and only if the relation $\vec{v}_2.a \leq \vec{v}_2.b$ does.*

Intuitively, two variable value vectors for a LBASE program P are equivalent, if they coincide on P variable and constant value ordering.

Example 4.5 *If the program P has constants -1 and 5 and variables $x_1, \ldots, x_4$, then the variable value vectors*

$$\vec{v}_1 = \langle 0,\ 6,\ 16,\ -7 \rangle \ \textit{and}\ \ \vec{v}_2 = \langle 0,\ 8,\ 1991,\ -2 \rangle$$

are equivalent, but

$$\vec{v}_3 = \langle 0,\ 6,\ 16,\ -7 \rangle \ \textit{and}\ \ \vec{v}_4 \langle 1,\ 6,\ 16,\ -7 \rangle$$

are not, since $\vec{v}_3.x_1 = 0 \in \{c_{min}, \ldots, c_{max}\}$, while $\vec{v}_4.x_1 = 1 \neq 0$.

It is easy to see that every element of C_P of a LBASE program P can be characterized by a simple inequality system, which determines the ordering of all program variable and constant values (representing all the constants $\{c_{min}, \ldots, c_{max}\}$) on the number line.

For $\vec{v} \in \mathbf{Q}^{+0} \times \mathbf{Q}^m$ being a variable value vector for a LTIM program P we define for any additive expression $expr$ over P variables and constants its value $\vec{v}.expr$ w.r.t. $\vec{v}$ in a way that $\vec{v}.c = c$ and

$$\vec{v}.expr_1 + \vec{v}.expr_2 = \vec{v}.(expr_1 + expr_2), \quad \vec{v}.expr_1 - \vec{v}.expr_2 = \vec{v}.(expr_1 - expr_2)$$

(for $x \in Vars(P)$ $\vec{v}.x$ is already defined as the value of x). We define the set $\mathcal{B}_P(\vec{v})$ of *base expressions* (called further also *base points*), associated with $\vec{v}$, to consist of

- all P variables and constants $a \in \mathcal{A}_P$,
- the expressions $x + c$ for a variable $x \in Vars(P)$ and $c \in \mathbf{Z}$, such that both $\vec{v}.z \leq \vec{v}.x \leq \vec{v}.z + c_{max}$ and $\vec{v}.z \leq \vec{v}.x + c \leq \vec{v}.z + c_{max}$ (notice that c can be also negative).

Example 4.6 *Let a program P have constants $0, 1$ and 3 and variables z, t_1, t_2, t_3 let for the value vector $\vec{v}$ $\vec{v}.x = 15.43$, $\vec{v}.t_1 = 16.2$, $\vec{v}.t_2 = 17.43$, $\vec{v}.t_3 = 19$. Then the set $\mathcal{B}_P(\vec{v})$ contains the following base points:*

- 0, 1, 3, z, t_1, t_2 *and* t_3,
- $z + 1$, $z + 2$, $z + 3$ *(we treat the expressions like z and $z + 0$ as identical) and*
- $t_1 + 1$, $t_1 + 2$, $t_2 - 2$, $t_2 - 1$, *and* $t_2 + 1$ *(there is no base point of the form $t_3 + c$ for $c \in \mathbf{Z}$, because $\vec{v}.t_3 \leq \vec{v}.z + 3$ does not hold).*

Definition 4.7 *The partitioning C_P for a LTIM program P is defined by the equivalence relation $\cong$, such that for $\vec{v}_1, \vec{v}_2 \in \mathcal{V}_P$ $\vec{v}_1 \cong \vec{v}_2$ iff*

- $\mathcal{B}_P(\vec{v}_1) = \mathcal{B}_P(\vec{v}_2) = \mathcal{B}_P$ *and*
- *for all $expr, expr' \in \mathcal{B}_P$ $\vec{v}_1.expr \leq \vec{v}_1.expr'$ if and only if $\vec{v}_2.expr \leq \vec{v}_2.expr'$.*

Intuitively, one C_P-equivalence class for a LTIM program P consists of all variable vector values which coincide on

- variable and constant value ordering,
- integral parts and ordering of fractional parts for all expression $e_i = x_i - z$ values laying between 0 and c_{max}.

Example 4.8 *Let a program P and its variable value vector $\vec{v}$ be those described in Example 4.6. Example 4.6 already gives us the set of base points $\mathcal{B}_P(\vec{v}) = \mathcal{B}_P(C)$ for $\vec{v} \in C$. The equivalence class C consists of those and only those vectors $\vec{v}$ with $\mathcal{B}_P(\vec{v}) = \mathcal{B}_P(C)$ for which the values $\vec{v}.b$ for $b \in BP(C)$ turns into evidence the following inequality system:*

$$0 < 1 < 3 < z = t_2 - 2 < t_1 < z + 1 = t_2 - 1 < t_1 + 1,$$
$$t_1 + 1 < z + 2 = t_2 < t_1 + 2 < z + 3 = t_2 + 1 < t_3.$$

We call any element $C \in \mathcal{C}_P$ for either LBASE or LTIM program P a *(variable) configuration* of P. For P being a LTIM program let for any configuration C $\mathcal{B}_P(C) \stackrel{\text{def}}{=} \mathcal{B}_P(\vec{v})$ for some (=for every) $\vec{v} \in C$. For the sake of uniformity we define $\mathcal{B}_P(C) = \mathcal{A}_P$ for every $C \in \mathcal{C}_P$ for every LBASE program P.

Proposition 4.9 *The set $\mathcal{C}_P$ of all configurations for any program P either in LBASE or LTIM is finite.*

Proof: In the LBASE case it is obvious that the finite number of all program variables and constants can be ordered on the number line only in finitely many different ways.

For P being a LTIM program, notice that there are only finitely many different base point sets $\mathcal{B}_P(\vec{v})$ for $\vec{v} \in \mathcal{V}_P$ (for any variable or constant $x \in \mathcal{A}_P$ no more than $c_{max}+1$ expressions of the form $x + c$ for $c \in \mathbb{Z}$ may satisfy $\vec{v}.z \leq \vec{v}.x, \vec{v}.x + c \leq \vec{v}.z + c_{max}$, as well as the expression $x + c$ may belong to $\mathcal{B}_P(\vec{v})$ only if $-c_{max} \leq c \leq c_{max}$). For every base point set $\mathcal{B}_P = \mathcal{B}_P(\vec{v})$ for some $\vec{v}$ the points of $\mathcal{B}_P$ can be ordered on the number line only in finite number of different ways (actually, these orderings are far from being all feasible, however, this is not important for proving the finiteness).□

We define for every configuration $C \in \mathcal{C}_P$ of either LBASE or LTIM program P the relations $<_C$, $\leq_C$ and $=_C$ for $a, b \in \mathcal{B}_P(C)$ (for $a, b \in \mathcal{A}_P$, if P is a LBASE program) in a way that

- $a \leq_C b$ iff $\vec{v}.a \leq \vec{v}.b$ for some (= for every) $\vec{v} \in C$;
- $a =_C b$ iff $a \leq_C b$ and $b \leq_C a$;
- $a <_C b$ iff $a \leq_C b$ and not $b \leq_C a$ (not $a =_C b$).

We use also $a \geq_C b$ as a synonym to $b \leq_C a$ and $a >_C b$ as a synonym to $b <_C a$.

For a variable value vector $\vec{v} \in \mathcal{V}_P$ let $[\![\vec{v}]\!]$ denote the configuration $C \in \mathcal{C}_P$ to which the value vector $\vec{v}$ belongs to: $\vec{v} \in C$.

It is easy to see that for P being either LBASE or LTIM program, the $\mathcal{C}_P$-equivalence class $[\![\vec{v}_0^P]\!]$ containing the initial variable value vector $(\langle 0, 0, \ldots, 0\rangle)$ is singleton (i.e. $[\![\vec{v}_0^P]\!] = \{\vec{v}_0^P\}$).

4.2 Path Feasibility Graphs

In this section we define the path feasibility graphs $BG(P)$ based on the introduced variable value set partitionings $\mathcal{C}_P$ (Section 4.1) for both LBASE and LTIM programs P (actually the constructions of this section can be performed provided that $\mathcal{C}_P$ is an arbitrary finite program P variable value set partitioning with the singleton class $[\![\vec{v}_0^P]\!]$ for program initial variable value vector).

Let us associate with the program P and its variable value set partitioning $\mathcal{C}_P$ a family of $\mathcal{C}_P$-graphs. A graph G is said to be a $\mathcal{C}_P$-graph, iff

- the set of its vertexes $V(G) \subseteq V(P) \times \mathcal{C}_P$ (i.e. every vertex in G is a pair $\langle n, C\rangle$ for $n \in V(P)$, $C \in \mathcal{C}_P$);
- $l_V(\langle n, C\rangle) = n \in V(P)$ for every $\langle n, C\rangle \in V(G)$;
- $E(G) \subseteq E(P) \times \mathcal{C}_P \times \mathcal{C}_P$, every edge $\tilde{e} = \langle e, C_1, C_2\rangle \in E(G)$ is leading from $f_G(\tilde{e}) = \langle f_P(e), C_1\rangle$ to $t_G(\tilde{e}) = \langle t_P(e), C_2\rangle$ and is labelled by $l_E(\tilde{e}) = e \in E(P)$;
- the initial vertex of G is $n_0^G = \langle n_0^P, [\![\vec{v}_0^P]\!]\rangle$.

A maximal $\mathcal{C}_P$-graph, denoted $MG(P)$, is defined as a $\mathcal{C}_P$-graph with the set of vertexes $V(G) = V(P) \times \mathcal{C}_P$ and the set of edges $E(G) = E(P) \times \mathcal{C}_P \times \mathcal{C}_P$.

Intuitively, the maximal $\mathcal{C}_P$-graph is a graph which is obtained by taking $card(\mathcal{C}_P)$ copies of the program P, one copy for each configuration $C \in \mathcal{C}_P$ and afterwards drawing all edges from $\langle n_i, C_j\rangle$ to $\langle n_i', C_j'\rangle$ whenever an edge from n_i to n_i' is drawn in P. The whole family of $\mathcal{C}_P$-graphs is obtained by erasing in various ways some edges of $MG(P)$.

Definition 4.10 *We call a $\mathcal{C}_P$-graph G for a LBASE or LTIM program P the* basic *graph, denoted $BG(P)$ provided G has an edge $\tilde{e} = \langle e, C_1, C_2\rangle$ leading from $\langle n_1, C_1\rangle$ to $\langle n_2, C_2\rangle$ if and only if there exist variable value vectors $\vec{v}_1 \in C_1$ and $\vec{v}_2 \in C_2$ such that $\langle n_1, \vec{v}_1\rangle \xrightarrow{e} \langle n_2, \vec{v}_2\rangle$ in the P state transition system.*

The remainder of the proof of Theorem 4.2 and Theorem 4.3 is devoted to

- the proof that the graph $BG(P)$ is indeed a path feasibility graph for every LBASE or LTIM program P, and
- the demonstration of the effectivity of the graph $BG(P)$ construction from the input of the given program P.

In fact, we study the properties of the graphs $BG(P)$ in slightly more detail than needed for the proofs of the theorems because of these graph further usability in the program infinite path feasibility analysis.

Given a path

$$\alpha = \langle n_0, C_0\rangle \tilde{e}_0, \langle n_1, C_1\rangle \tilde{e}_1, \ldots [, \langle n_k, C_k\rangle]$$

in an arbitrary C_P-graph G we define the selectors $n_i(\alpha)$, $C_i(\alpha)$ and $e_i(\alpha)$ to denote the ith program vertex, C_P-equivalence class and program edge on the path α, respectively (i.e. $n_i(\alpha) = n_i$, $C_i(\alpha) = C_i$ and $e_i(\alpha) = l_{E(G)}(\tilde{e}_i)$). The similar notation of $n_*(\alpha)$, $C_*(\alpha)$ is used to denote the last program vertex and C_P-equivalence class of a finite path α. Let us define also for every G edge $\tilde{e} = \langle e, C_1, C_2\rangle$ the corresponding program edge label to be $ll(\tilde{e}) = l_{E(P)}(l_{E(G)}(\tilde{e})) = l_{E(P)}(e) \in \{" + ", " - "\}$.

Now we define for every C_P-graph also the set of its *histories*, so giving some semantics to the syntactical C_P-graph construction done so far.

Let $\beta = \langle n_0, \vec{v}_0\rangle e_0, \ldots [, \langle n_k, \vec{v}_k\rangle]$ be a history of the program P, then for an arbitrary C_P-graph G we call β a history along a path $\alpha = \langle n_0, C_0\rangle \tilde{e}_0, \ldots [, \langle n_k, C_k\rangle]$ in G iff $\mathcal{CR}d_\beta = \mathcal{CR}d_\alpha$ and for all $i \in \mathcal{CR}d_\alpha$ both $\vec{v}_i \in C_i$ and $\tilde{e}_i$ is labelled by e_i (equivalently, $\tilde{e}_i = \langle e_i, C_i(\alpha), C_{i+1}(\alpha)\rangle$).

Intuitively, a C_P-graph is a "refinement" of the program P in a sense it admits along any path α only those P histories in which after every step the program variable value vector satisfies the configuration (belongs to the C_P-equivalence class), ascribed to the current vertex of the path.

Definition 4.11 *A path α (either initial or not) in a C_P-graph G is called feasible iff there exists a history of G along α.*

Observe that in the case of the C_P-graphs we do not distinguish between the feasibility and weak feasibility of the graph paths, as it was done for the programs.

Fact 4.12 *Every history ν of any (not necessarily basic) C_P-graph G along a finite or infinite path α is also a history of P along $proj(\alpha)$. For α being an initial path in G the history ν is initial along $proj(\alpha)$.*

Proof: Follows from definitions, observing in the case of the initiality the property of $[\![\vec{v}_0^P]\!]$ being singleton: $[\![\vec{v}_0^P]\!] = \{\vec{v}_0^P\}$.□

Fact 4.13 *Every history β along a path α in P is also a history along a path α', such that $\alpha = proj(\alpha')$, in the C_P- basic graph $BG(P)$. Moreover, if β is initial, the path α' also can be chosen initial in $BG(P)$.*

Proof: Given a history

$$\beta = \langle n_0, \vec{v}_0\rangle e_0, \langle n_1, \vec{v}_1\rangle e_1, \ldots [, \langle n_k, \vec{v}_k\rangle]$$

the corresponding path in $BG(P)$ is of the form

$$\langle n_0, [\![\vec{v}_0]\!]\rangle \tilde{e}_0, \langle n_1, [\![\vec{v}_1]\!]\rangle \tilde{e}_1, \ldots [, \langle n_k, [\![\vec{v}_k]\!]\rangle],$$

the existence of the appropriate $\dot{e}_i$s is due to the definition of $BG(P)$.

The initiality of α' is also straightforward from the definitions. □

As one can see, in the basic graph $BG(P)$ every path with the length 1 (containing only one edge in it) is feasible, no matter what partitioning $\mathcal{C}_P$ of the program variable vector value set one chooses (actually, for this purpose also the trivial partitioning with 1 element containing all the set $\mathcal{V}_P$ would do, though it is not admitted here for in this case the set $[\![\vec{v}_0^P]\!]$ is not singleton).

Definition 4.14 *A basic graph $BG(P)$ is called perfect, if every finite path in $BG(P)$ is feasible.*

Proposition 4.15 *For a perfect graph $BG(P)$ the set of all finite feasible paths in the program P coincides with the set of projections of all finite initial paths in $BG(P)$.*

Proof: Straightforward from Fact 4.12, Fact 4.13 and the definitions of path feasibility in programs and $\mathcal{C}_P$-graphs.□

Corollary 4.16 *If there exists an algorithm which, given a program P, constructs a perfect graph $BG(P)$, then the set of all finite feasible paths of P is projective.*

It is easy to see that similar results can also be obtained for the T-projectivity of the set of all finite feasible *correct* (accepting) paths of the program P (one simply defines the finite acceptance condition for the graph $BG(P)$ in a way $\langle n, C\rangle \in S_F(BG(P))$ iff $n \in S_F(P)$).

In order to complete the proof of Theorem 4.1 we show the perfectness of the basic graphs $BG(P)$ for both LBASE and LTIM programs w.r.t. the partitionings $\mathcal{C}_P$ defined in Section 4.1 (Lemma 4.17 and Lemma 4.29), as well as the effectivity of the graph $BG(P)$ construction from given program P (Proposition 5.5 and Proposition 5.10).

4.3 LBASE: Perfectness of BG(P)

Lemma 4.17 *Every finite path in the basic $\mathcal{C}_P$-graph $BG(P)$ for given LBASE program P is feasible (i.e. for every LBASE program P the graph $BG(P)$ is perfect).*

Proof: We obtain the statement of the lemma as a corollary of the following more general result.

Let us call a path α in $BG(P)$ feasible with the initial value $\langle n, \vec{v}\rangle$ (with the final value $\langle n', \vec{v}'\rangle$), if there exist a history ν along α such that $\langle n_0(\nu).\vec{v}_0(\nu)\rangle = \langle n, \vec{v}\rangle$ (respectively, $\langle n_*(\nu), \vec{v}_*(\nu)\rangle = \langle n', \vec{v}'\rangle$).

Lemma 4.18 *If an initial finite path α in $BG(P)$ is feasible with the final value $\langle n, \vec{v}'\rangle$ and a (finite or infinite) path β is feasible with the initial value $\langle n, \vec{v}''\rangle$ for some $\vec{v}'' \in [\![\vec{v}]\!]$, then the path $\alpha + \beta$ is feasible.*

Having proved the lemma, it is now completely straightforward to use the induction on the path length in order to obtain the proof of Lemma 4.17. The following is another interesting corollary of Lemma 4.18.

Corollary 4.19 *If two paths α and β in $BG(P)$ end with the same vertex, then for an arbitrary (finite or infinite) path γ in $BG(P)$ the path $\alpha + \gamma$ is feasible if and only if the path $\beta + \gamma$ is.*

Proof of Lemma 4.18: Let $C = [\![\vec{v}]\!] = [\![\vec{v}']\!] \in C_P$. The proof idea is to show that there is $\vec{v} \in C$, such that both α is feasible with the final value $\langle n, \vec{v}\rangle$ and β is feasible with $\langle n, \vec{v}\rangle$ as the initial value, this allows us to build explicitly a history along $\alpha + \beta$ in $BG(P)$. We begin with the notion of a P-uniform mapping and some its properties.

Definition 4.20 *We call a mapping $\rho : \mathbf{Z} \to \mathbf{Z}$ P-uniform, if for every $x, y \in \mathbf{Z}$*

- *$x < y$ implies $\rho(x) < \rho(y)$, i.e. ρ is strictly monotone, and*
- *$\rho(c_{min}) = c_{min}$ and $\rho(c_{max}) = c_{max}$, i.e. ρ preserves all the constants $c \in Cons(P) = \{c_{min}, \ldots, c_{max}\}$.*

We extend any mapping $\rho : \mathbf{Z} \to \mathbf{Z}$ in a polymorphic manner also to the structures, containing integers as elements in a way the mapping ρ replaces any integer component x with $\rho(x)$ and does not change the structure's components of other types, e.g., for a vector $\vec{v} \in \mathbf{Z}^m$

$$\rho(\vec{v}) = \langle \rho(\vec{v}.x_1), \rho(\vec{v}.x_2), \ldots, \rho(\vec{v}.x_m)\rangle.$$

Fact 4.21 *For a P-uniform mapping ρ and any $\vec{v} \in \mathcal{V}_P$ $[\![\vec{v}]\!] = [\![\rho(\vec{v})]\!]$.*

Proof: Follows from definitions. □

The following is the main point in the proof of Lemma 4.18:

Lemma 4.22 *For a P-uniform mapping ρ and an arbitrary history ν along a path γ in $BG(P)$ the sequence $\rho(\nu)$ is also a history of $BG(P)$ along γ.*

Proof: According to the definition of the history, it is enough to check that, if $\langle n_1, \vec{v}_1\rangle \xrightarrow{e} \langle n_2, \vec{v}_2\rangle$, then also $\langle n_1, \rho(\vec{v}_1)\rangle \xrightarrow{e} \langle n_2, \rho(\vec{v}_2)\rangle$, Fact 4.21 will ensure that the new history $\rho(\nu)$ goes along the same path in $BG(P)$ as the history ν.

Consider the language LBASE operator p, associated with the program P vertex n_1. It is straightforward according to the definitions of the semantic functions ϕ_p and ψ_p (see Chapter 3) that for p being an input operator

$$\rho(\phi_p(\vec{v}_1, c)) = \phi_p(\rho(\vec{v}_1, \rho(c)) \quad \text{and} \quad \psi_p(\vec{v}_1, c) = \psi_p(\vec{v}_1, \rho(c)),$$

and for p being an assignment, comparison or dummy operator

$$\rho(\phi_p(\vec{v}_1)) = \phi_p(\rho(\vec{v}_1)) \quad \text{and} \quad \psi_p(\vec{v}_1) = \psi_p(\rho(\vec{v}_1)).$$

The existence of the transition $\langle n_1, \rho(\vec{v}_1)\rangle \xrightarrow{c} \langle n_2, \rho(\vec{v}_2)\rangle$ (provided there exists the transition $\langle n_1, \vec{v}_1\rangle \xrightarrow{c} \langle n_2, \vec{v}_2\rangle$) now is straightforward from the definition of the transition relation $\longrightarrow$ in the program P state transition system.□

Definition 4.23 *Given two variable value vectors $\vec{v}_1$ and $\vec{v}_2$ satisfying the same configuration $C = [\![\vec{v}_1]\!] = [\![\vec{v}_2]\!]$, we call the vector $\vec{v}_1$ sparser than $\vec{v}_2$ if for any two program variables $x_i, x_j \in Vars(P)$*

$$x_i \leq_C x_j \text{ implies } \vec{v}_2.x_j - \vec{v}_2.x_i \leq \vec{v}_1.x_j - \vec{v}_1.x_i.$$

Fact 4.24 *For any two vectors $\vec{v}_1$ and $\vec{v}_2$ with $[\![\vec{v}_1]\!] = [\![\vec{v}_2]\!] = C$ there exists a vector $\vec{v}_3 \in C$ such that $\vec{v}_3$ is sparser than both given vectors $\vec{v}_1$ and $\vec{v}_2$.*

Proof: The vector $\vec{v}_3$ can be constructed, as follows. For all variables $y_1, y_2, \ldots, y_s$ greater than c_{max} in C and ordered in ascending order according to C (i.e. $c_{max} \leq_C y_1$, $y_i \leq_C y_{i+1}$ $\forall i$), define

$$\vec{v}_3.y_1 = \max\{\vec{v}_1.y_1, \vec{v}_2.y_1\} \text{ and } \vec{v}_3.y_{i+1} = \vec{v}_3.y_i + \max\{\vec{v}_1.y_{i+1} - \vec{v}_1.y_i,\ \vec{v}_2.y_{i+1} - \vec{v}_2.y_i\}.$$

One can similarly define also the $\vec{v}_3$-values for variables, less than c_{min} in C. It is obvious that the constructed vector $\vec{v}_3$ is sparser than both $\vec{v}_1$ and $\vec{v}_2$.□

Fact 4.25 *For any two variable value vectors $\vec{v}_1$ and $\vec{v}_2$, such that $\vec{v}_2$ is sparser than $\vec{v}_1$, there exists a P-uniform mapping ρ with $\rho(\vec{v}_1.a) = \vec{v}_2.a$ for every P variable or constant $a \in \mathcal{A}_P$.*

Proof: Define $\rho(c) = c$ for all $c \in Cons(P)$. For $x > c_{max}$ let $\vec{v}_1.a(x)$ be the maximal of values $\vec{v}_1.a$ for $a \in \mathcal{A}_P$, which is does not exceed x. Let us define

$$\rho(x) = \vec{v}_2.a(x) + (\vec{v}_1.a(x) - x).$$

For $x < c_{min}$ one defines

$$\rho(x) = \vec{v}_2.b(x) - (\vec{v}_1.b(x) - x), \text{ where}$$

$\vec{v}_1.b(x)$ is the minimal of values $\vec{v}_1.a$ which are not less than x. It is because of the sparseness of $\vec{v}_2$ w.r.t. $\vec{v}_1$ that the defined mapping ρ is strictly monotone. The P constant preservation by ρ is by definition.□

Given the variable value vectors $\vec{v}'$ and $\vec{v}''$ with $[\![\vec{v}']\!] = [\![\vec{v}'']\!] = C$ let $\vec{v} \in C$ be a vector which is sparser both than $\vec{v}'$ and $\vec{v}''$ (see Fact 4.24). Due to Fact 4.25 there are

P-uniform mappings ρ_1 and ρ_2 such that $\rho_1(\vec{v}') = \rho_2(\vec{v}'') = \vec{v}$. Let ν_1 be the history along α in $BG(P)$ with the final value $\langle n, \vec{v}' \rangle$. Lemma 4.22 guarantees that $\rho_1(\nu_1)$ also is a history along α in $BG(P)$, its final variable value vector is

$$\vec{v}_*(\rho_1(\nu_1)) = \rho_1(\vec{v}_*(\nu_1)) = \rho_1(\vec{v}') = \vec{v}.$$

In a similar way one establishes that $\rho_2(\nu_2)$ is a history along β with $\vec{v}$ as initial variable value vector: $\vec{v} = \vec{v}_0(\rho_2(\nu_2))$. It is according to the definition of the history that we can glue the obtained two histories together so obtaining a history along $\alpha+\beta$ in $BG(P)$, what means the feasibility of the path $\alpha + \beta$ in $BG(P)$.□

So we have completed the proof of Lemma 4.18. Since we have the result of Lemma 4.17 as a corollary, it remains for the proof of Theorem 4.2 to show only the effectivity of the graph $BG(P)$ construction from the text of the program P, what is done in Proposition 5.5 using the path inequality system techniques (see Chapter 5).

4.4 Basic Graphs for LBASQ programs

Although the projectivity of the set of all finite feasible paths for LTIM programs is demonstrated in the full generality in Section 4.5 and Section 5.2, we consider here an alternative approach for proving projective the sets of all finite feasible paths in LBASQ programs (LTIM programs without the real time counter, intuitively, the LBASE programs with the rational-valued variables).

Let us given a LBASQ program P define the partitioning C_P of the P variable value set $\mathcal{V}_P = \mathbf{Q}^m$, as it was done in Definition 4.4 for LBASE programs, by taking two variable value vectors $\vec{v}_1$ and $\vec{v}_2$ equivalent, if they coincide on P variable and constant value ordering (i.e., iff for any $a, b \in \mathcal{A}_P$ $\vec{v}_1.a \leq \vec{v}_1.b$ iff $\vec{v}_2.a \leq \vec{v}_2.b$).

We construct also the (LBASE-like) basic graph $BG(P)$ according to the partitioning C_P, as described in Section 4.2.

Lemma 4.26 *For every history ν along a finite path α in the basic graph $BG(P)$ of a LBASQ program P and every one-edge path $\beta = \langle n, C\rangle\tilde{e}, \langle n', C'\rangle$ with $\langle n, C\rangle = \langle n_*(\alpha), C_*(\alpha)\rangle$ there exists a history ν' along $\alpha' = \alpha + \beta$ with $\vec{v}_i(\nu').a = \vec{v}_i(\nu).a$ for all $i \in \mathcal{CR}d_o$ and $a \in \mathcal{A}_P$ (i.e. ν' is a continuation of ν).*

Proof: Since β is a path in $BG(P)$ we have for $e = l_E(BG(P))(\tilde{e})$ $\langle n, \vec{v}\rangle \xrightarrow{e} \langle n', \vec{v}'\rangle$ with $\vec{v} \in C$ and $\vec{v}' \in C'$. Since $\vec{v}_*(\nu) \in C$, we can exhibit a monotone mapping $\rho : \mathbf{Q} \to \mathbf{Q}$ with $\rho(\vec{v}.a) = \vec{v}_*(\nu).a$ for all $a \in \mathcal{A}_P$. Easy to see that the history ν, continued by $\langle n, \rho(\vec{v})\rangle\tilde{e}, \langle n, \rho(\vec{v}')\rangle$ is the desired history ν'.□

Corollary 4.27 *For every LBASQ program P its LBASE-like basic graph $BG(P)$ is perfect.*

Proof: The feasibility of every finite path in $BG(P)$ is straightforward by induction from Lemma 4.26.□

The effectiveness of the graph $BG(P)$ construction for LBASQ programs is obtained in a similar way as for LBASE programs in Section 5.1, we do not consider the details.

Observe, that an analogue of Lemma 4.26 for LBASE programs does not hold. In the case of integer-valued variables we cannot exhibit the corresponding monotone mapping $\rho : \mathbf{Z} \to \mathbf{Z}$ with $\rho(\bar{v}.a) = \bar{v}_*(\nu).a$ for all $a \in \mathcal{A}_P$ for in the case $\bar{v}.a = 17$, $\bar{v}.b = 19$ and $\bar{v}_*(\nu).a = 1992$, $\bar{v}_*(\nu).b = 1993$ we are required to define $\rho(17) = 1992$ and $\rho(19) = 1993$, but what to do with $\rho(18)$ (observe the monotonicity)? We have dealt with the problem of finding a history along the prolonged path in the case of LBASE programs by some *updating* of the given initial history by a P-uniform mapping in order to get the final history variable value vector "sufficiently sparse".

As a straightforward corollary from Lemma 4.26 we obtain also

Fact 4.28 *Every infinite path in the (LBASE-like) basic graph $BG(P)$ for a LBASQ program P is feasible.*

Proof: Given an infinite path α in $BG(P)$ we consider the sequence $\alpha_0, \alpha_1, \ldots$ of all its finite prefixes. According to Lemma 4.26 we have a sequence of finite histories $\nu_0, \nu_1, \ldots$, the history ν_i along the path α_i, with $\bar{v}_i(\nu_{i+j}) = \bar{v}_i(\nu_i)$ for all $i, j \in \mathbf{N}$. So we obtain a history ν along α by defining $\bar{v}_i(\nu) = \bar{v}_i(\nu_i)$ for all $i \in \mathbf{N}$ (intuitively, ν is a "pointwise limit" of the history sequence $\nu_0, \nu_1, \ldots$).□

As the analogue of Lemma 4.26 for LBASE programs does not hold we need to look for other techniques for studying the infinite path feasibility in LBASE programs, what is done in Chapter 6.

4.5 LTIM: Perfectness of BG(P)

Lemma 4.29 *Every finite path in the basic C_P-graph $BG(P)$ for given LTIM program P is feasible (i.e. for every LTIM program P the graph $BG(P)$ is perfect).*

Proof: First of all we introduce two kinds of mappings, which we need for the proof.

Definition 4.30 *We call a mapping $\rho : \mathbf{Q} \to \mathbf{Q}$ u-stable iff ρ is strongly monotone and $\rho(x) = x \ \forall x < u$.*
A strongly monotone mapping $\sigma : \mathbf{Q} \to \mathbf{Q}$ is called v-uniform iff

- *σ preserves all P constants (i.e. $\sigma(c) = c$ for all $c \in Cons(P)$) and*
- *$\sigma(x + c) = \sigma(x) + c$ for all $x \geq v$, $c \in \mathbf{N}$.*

As in the case of mappings $\mathbf{Z} \to \mathbf{Z}$, we extend the mappings $\mathbf{Q} \to \mathbf{Q}$ in a polymorphic manner to other structures, containing rationals as elements in a way these mappings apply to all components of the rational type and do not change any components of other types.

Lemma **4.31** *If an initial finite path α in $BG(P)$ has a history ν_1 along it and a (finite or infinite) path β with $(n_*(\alpha), C_*(\alpha)) = (n_0(\beta), C_0(\beta))$ has a history ν_2 along it, then there is a history ν_3 along $\alpha + \beta$ such that*

- *for all $i \in \mathcal{CR}d_\alpha$ $\vec{v}_i(\nu_3) = \rho(\vec{v}_i(\nu_1))$ and*
- *for k being the length of α $\forall j \in \mathcal{CR}d_\beta$ $\vec{v}_{k+j}(\nu_3) = \sigma(\vec{v}_j(\nu_2))$,*

where the mapping ρ is $\vec{v}_*(\nu_1).z + c_{max}$-stable and σ is $\vec{v}_0(\nu_2).z$-uniform. Moreover, the variable value vector $\vec{v}_i(\nu_3)$ can be effectively computed given the variable value vectors $\vec{v}_k(\nu_1)$, $\vec{v}_0(\nu_2)$, as well as $\vec{v}_i(\nu_1)$, if $i < k$, and $\vec{v}_{i-k}(\nu_2)$, if $i > k$.

Intuitively, the lemma is just a stronger version of an analogue of Lemma 4.18 for LTIM programs. We need the lemma in this formulation afterwards when we study the infinite path feasibility in LTIM programs.

Proof of **Lemma 4.31:**

Proposition **4.32** *Given a finite history $\nu = \langle n_0, \vec{v}_0\rangle e_0, \langle n_1, \vec{v}_1\rangle e_1, \ldots, \langle n_k, \vec{v}_k\rangle$ of a LTIM program P along a path α in $BG(P)$, for every $\vec{v}_k.z + c_{max}$-stable mapping ρ the sequence*

$$\rho(\nu) = \langle n_0, \rho(\vec{v}_0)\rangle e_0, \ldots, \langle n_k, \rho(\vec{v}_k)\rangle$$

is also a history along α.

Proof: First of all, since ρ is $\vec{v}_k.z + c_{max}$-stable, it follows from the definition of the configuration and the fact that $\forall i \leq k$ $\vec{v}_i.z + c_{max} \leq \vec{v}_k.z + c_{max}$ that for all $i \leq k$ $[\![\rho(\vec{v}_i)]\!] = [\![\vec{v}_i]\!]$. The check that a transition $\langle n, \rho(\vec{v})\rangle \xrightarrow{e} \langle n', \rho(\vec{v}')\rangle$ is in the P state transition system, given this transition system contains $\langle n, \vec{v}\rangle \xrightarrow{e} \langle n', \vec{v}'\rangle$ is straightforward from the transition system definition (i.e. the semantics of LTIM programs).□

Proposition **4.33** *For a finite or infinite history ν along α in $BG(P)$, starting with $(n_0, \vec{v}_0)$, for every $\vec{v}_0.z$-uniform mapping σ the sequence $\sigma(\nu)$ is also a history of $BG(P)$ along α.*

Proof: Similar to the proof of Proposition 4.32. For all $i \in \mathcal{CRD}_o$ $[\![\sigma(\vec{v}_i(\nu))]\!] = [\![\vec{v}_i(\nu)]\!]$ because σ is also $\vec{v}_i(\nu).z$-uniform, this leads to the coincidence of the base point sets $\mathcal{B}_P(\sigma(\vec{v}_i(\nu))) = \mathcal{B}_P(\vec{v}_i(\nu))$ (for all $x \in \mathcal{A}_P, c \in \mathbf{Z}$

$$\sigma(\vec{v}_i(\nu)).z \leq \sigma(\vec{v}_i(\nu)).x, \sigma(\vec{v}_i(\nu)).x + c \leq \sigma(\vec{v}_i(\nu)).z + c_{max}$$

if and only if

$$\vec{v}_i(\nu).z \leq \vec{v}_i(\nu).x, \vec{v}_i(\nu).x + c \leq \vec{v}_i(\nu).z + c_{max}),$$

as well as the base point orderings.

The transition relation preservation by σ is also straightforward from the definitions, notice that for the input statement $p(n)$ given $\langle n, \vec{v}\rangle \xrightarrow{e:a} \langle n', \vec{v}'\rangle$ one obtains $\langle n, \sigma(\vec{v})\rangle \xrightarrow{e:\sigma(a)} \langle n', \sigma(\vec{v}')\rangle$.□

Lemma 4.34 *For $\vec{v}_1, \vec{v}_2 \in C \in \mathcal{C}_P$ of a LTIM program P there exists a $\vec{v}_1.z + c_{max}$-stable mapping ρ and $\vec{v}_2.z$-uniform mapping σ such that $\rho(\vec{v}_1) = \sigma(\vec{v}_2) = \vec{v}_3 \in C$. The mappings ρ and σ can be computed given the vectors $\vec{v}_1$ and $\vec{v}_2$.*

Proof: First of all define the values $\sigma(\vec{v}_2.x_i) = \vec{v}_1.x_i$ for the base points $x_i \in \mathcal{B}_P(C)$ with $x_i \leq_C z + c_{max}$ and extend the definition of σ to the pair of intervals $]-\infty, \vec{v}_2.z + c_{max}] \rightarrow]-\infty, \vec{v}_1.z + c_{max}]$ in a piecewise linear way with the graph corner points at already defined values (taking into account also $\sigma(c) = c$ for $c \in Cons(P)$). It follows from the definition of the partitioning $\mathcal{C}_P$ that such a value definition does not violate the uniformity of σ. In what follows, the mapping σ is extended to $\mathbf{Q} \rightarrow \mathbf{Q}$ defining $\sigma(x) = \sigma(x-1) + 1$ for $x > \vec{v}_2.z + c_{max}$.

It is easy to see that also a $\vec{v}_1.z + c_{max}$-stable mapping ρ can be defined to meet $\rho(\vec{v}_1) = \sigma(\vec{v}_2)$.□

Lemma 4.34 together with Proposition 4.32 and Proposition 4.33 give an explicit way how to obtain every variable value vector of a history ν_3 demanded in Lemma 4.31, so giving also the proof of the lemma.□

Having proved Lemma 4.31, it is straightforward to follow the same induction arguments as in the LBASE case in order to obtain the proof of Lemma 4.29.□

Now in order to complete the proof of Theorem 4.1 it remains to show only the effectiveness of the basic $\mathcal{C}_P$-graph construction for LBASE and LTIM programs, what is dealt with by the techniques of Chapter 5. Perhaps the reader can find this effectivity rather obvious because the only thing needed is to determine automatically which paths of the length one (i.e. containing only one edge) in the maximal $\mathcal{C}_P$-graphs $MG(P)$ for LBASE and LTIM programs P are feasible.

Chapter 5

Path Inequality Systems

In this chapter we develop the path inequality system technique (for the paths in program path feasibility graphs) which allows both to demonstrate the effectivity of the basic graph construction for both LBASE and LTIM programs, so completing the proof of Theorem 4.1, and serves as the basis for the infinite path feasibility analysis in the programs (see especially Section 5.3).

5.1 Path Inequality Systems for LBASE

Let G be a C_P-graph for a LBASE program P. Let us fix an arbitrary finite or infinite path α in G. We obtain in this section some symbolic characteristic of the set of all histories along α in G in a form of inequality system, allowing to establish the effectivity of the $BG(P)$ construction from $MG(P)$ for LBASE programs P.

First of all we define for the path α the set $\mathcal{P}t_\alpha$ of *points* on α, each point being a pair $(a;r)$ for a program variable or constant $a \in \mathcal{A}_P$ and a path coordinate $r \in \mathcal{CR}d_\alpha$ (so $\mathcal{P}t_\alpha = \mathcal{A}_P \times \mathcal{CR}d_\alpha$). Given a history β along α in G one can associate with it a mapping $\Gamma_\beta : \mathcal{P}t_\alpha \to \mathbf{Z}$, defined

$$\Gamma_\beta((a;r)) = \vec{v}_r(\beta).a,$$

i.e. the mapping Γ_β gives for every point on α its "value" on the history β (recall that for a constant $c \in \mathcal{A}_P$ the notation $\vec{v}.c$ is used to denote simply the value of c). Let us call the mapping Γ_β the *generator* of the history β.

In a similar way, every mapping $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Z}$ is a generator of a sequence

$$\beta_\Gamma = \langle n_0(\alpha), \Gamma(\vec{x};0)\rangle e_0(\alpha), \langle n_1(\alpha), \Gamma(\vec{x};1)\rangle e_1(\alpha), \ldots [, \langle n_k(\alpha), \Gamma(\vec{x};k)\rangle]$$

for $k = card(\mathcal{CR}d_\alpha) - 1$ being the length of α and $\Gamma(\vec{x};i)$ for every i used as a shorthand of

$$\langle \Gamma(x_1;i), \Gamma(x_2;i), \ldots, \Gamma(x_m;i)\rangle.$$

Let us call the sequence β_Γ the sequence *generated by* the mapping Γ.

It is easy to see that not all sequences generated by mappings Γ are histories of G along α, however, some of them are. In particular, for any history β along the path α the sequence, generated by the generator of β is the history β itself (i.e. $\beta_{\Gamma_\beta} = \beta$).

The idea of the path α inequality system is to give a simple symbolic characteristic of all mappings Γ which correspond to (are generators of) the histories along α.

Let for an arbitrary program P operator p and an arbitrary $a \in \mathcal{A}_P$ we say that a **ub** p (*a unchanged by p*) if and only if p does not assign a new value to a (obviously, for any P constant c, c **ub** p, no matter which operator p is). The only cases when the relation a **ub** p does not hold is when a is the variable into which the input or assignment operator puts the value.

Let for a given program P vertex $n \in V(P)$ and edge label $l \in \{" + ", " - "\}$ for $x, y \in \mathcal{A}_P$

- $x <_{n,l} y$, if $p(n)$ is a comparison operator $\mathrm{x} < \mathrm{y}$ and $l = " + "$, and
- $y \leq_{n,l} x$, if $p(n)$ is a comparison operator $\mathrm{x} < \mathrm{y}$ and $l = " - "$, or else $y <_{n,l} x$.

The relation $\leq_\alpha$ in the set $\mathcal{P}t_\alpha$ of points along α is defined the following way:

- $(a;r) \leq_\alpha (b;r)$ for $a, b \in \mathcal{A}_P$ and $r, r+1 \in \mathcal{CR}d_\alpha$ whenever $a \leq_{n,l} b$ for $n = n_r(\alpha)$ and $l = l_{E(P)}(e_r(\alpha))$ (the inequalities requested by the statement exit labels on the path are reflected),
- if a **ub** $p(n_r(\alpha))$ for $a \in \mathcal{A}_P$, $r, r+1 \in \mathcal{CR}d_\alpha$, then $(a;r) \leq_\alpha (a;r+1)$ and $(a;r+1) \leq_\alpha (a;r)$,
- if $a \leq_{C_r(\alpha)} b$ for $a, b \in \mathcal{A}_P$, then $(a;r) \leq_\alpha (b;r)$ (after every execution step the program variable values satisfy the corresponding configuration),
- the relation $\leq_\alpha$ is transitive, i.e. for $w_1, w_2, w_3 \in \mathcal{P}t_\alpha$, if $w_1 \leq_\alpha w_2$ and $w_2 \leq_\alpha w_3$, then also $w_1 \leq_\alpha w_3$.

Let us call for an arbitrary path α the set of inequalities $\{w_1 \leq_\alpha w_2 \mid w_1, w_2 \in \mathcal{P}t_\alpha\}$ the *path α inequality system* and denote it by $\mathcal{N}_\alpha$.

Easy to see that for every α the relation $\leq_\alpha$ is a partial order. We define also an equivalence relation $=_\alpha$ such that $w =_\alpha w'$ for $w, w' \in \mathcal{P}t_\alpha$ if and only if $w \leq_\alpha w'$ and $w' \leq_\alpha w$. Let $w <_\alpha w'$ for $w, w' \in \mathcal{P}t_\alpha$ if and only if $w \leq_\alpha w'$ and the inverse $w' \leq_\alpha w$ does not hold. Let $w' \geq_\alpha w$ mean $w \leq_\alpha w'$ and $w' >_\alpha w$ be another notation for $w <_\alpha w'$.

Example 5.1 *Let P have 3 variables, denoted here by x, y and u, as well as the constants $c_{min} = 0, c_{max} = 1$. Let in a C_P- graph G the path*

$$\alpha = \langle n_0, C_0\rangle \tilde{e}_0, \langle n_1, C_1\rangle \tilde{e}_1, \langle n_2, C_2\rangle, \text{ where}$$

$p(n_0) = ?\mathrm{u}$ *and* $p(n_1) = \mathrm{x} \leftarrow \mathrm{y}$,
$ll(\tilde{e}_i) = l_{E(P)}(l_{E(G)}(\tilde{e}_i)) = " + "$ *for* $i = 0, 1$,
$0 <_{C_0} 1 <_{C_0} x <_{C_0} y <_{C_0} u$,
$0 <_{C_1} 1 <_{C_1} x <_{C_1} y <_{C_1} u$ *and*
$0 <_{C_2} 1 <_{C_2} x =_{C_2} y <_{C_2} u$.

Then the path inequality system $\mathcal{N}_\alpha$ *looks as follows:*
$(0;0) =_\alpha (0;1) =_\alpha (0;2) <_\alpha (1;0) =_\alpha (1;1) =_\alpha (1;2) <_\alpha (x;0)$,
$(x;0) =_\alpha (x;1) <_\alpha (x;2) =_\alpha (y;0) =_\alpha (y;1) =_\alpha (y;2)$,
$(y;2) <_\alpha (u;0)$, $(y;2) <_\alpha (u;1) =_\alpha (u;2)$, *but the points* $(u;0)$ *and* $(u;1)$ *are* $\leq_\alpha$ *- incomparable.*

Definition 5.2 *Given a path* α *in a* C_P*-graph* G*, we call a mapping* $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Z}$ *a solution of* $\mathcal{N}_\alpha$ *if and only if*

- $\forall c \in Cons(P) : \Gamma(c; r) = c$,
- *for any two points* $w_1, w_2 \in \mathcal{P}t_\alpha$*:*
  1. *if* $w_1 =_\alpha w_2$ *then* $\Gamma(w_1) = \Gamma(w_2)$
  2. *if* $w_1 <_\alpha w_2$ *then* $\Gamma(w_1) < \Gamma(w_2)$.

Proposition 5.3 *For every path* α *in a* C_P*-graph* G *every solution* Γ *of* $\mathcal{N}_\alpha$ *generates a history of* G *along* α.

Proof: Consider the sequence

$$\beta_\Gamma = \langle n_0(\alpha), \Gamma(\vec{x}; 0)\rangle e_0(\alpha), \langle n_1(\alpha), \Gamma(\vec{x}; 1)\rangle e_1(\alpha), \ldots [, \langle n_k(\alpha), \Gamma(\vec{x}; k)\rangle],$$

generated by some solution Γ of $\mathcal{N}_\alpha$. In order to prove that β_Γ is a history along α, it suffices to prove that

- for every $i \in \mathcal{CR}d_\alpha$ $\Gamma(\vec{x}; i) \in C_i(\alpha)$ and
- for every $i \in \mathcal{CR}d_\alpha$, $i \neq k$

$$\langle n_i(\alpha), \Gamma(\vec{x}; i)\rangle \xrightarrow{e_i(\alpha)} \langle n_{i+1}(\alpha), \Gamma(\vec{x}; i+1)\rangle,$$

what is straightforward from the definitions of the path α inequality system and the program P state transition system (observe, that we do not have "-"-labelled outgoing edges from the program nodes with attached input or assignment operators).□

Proposition 5.4 *Every history along α in a C_P-graph G is generated by some solution of the path α inequality system $\mathcal{N}_\alpha$.*

Proof: Follows from definitions.□

Proposition 5.5 *The construction of the basic graphs $BG(P)$ for LBASE programs P from the given program text is effective.*

Proof: First of all, given a LBASE program P it is easy to build for it the maximal C_P-graph $MG(P)$ (see Section 4.2). According to the definition of $BG(P)$, in order to obtain the graph $BG(P)$ from $MG(P)$ it suffices to determine and erase all infeasible edges in $MG(P)$ (i.e. the edges for which the path containing only this edge does not have any history ν along it (no points about any initiality of ν)).

One can easily check whether for given edge $\tilde{e} = \langle e, C_1, C_2\rangle$ the inequality system of the path $\alpha_{\tilde{e}} = \langle f_P(e), C_1\rangle\tilde{e}, \langle t_P(e), C_2\rangle$ in $MG(P)$ is contradictory or not. Due to the Proposition 5.3 and Proposition 5.4 this check will give also the answer about the path $\alpha_{\tilde{e}}$ feasibility.□

So we have completed the proof of Theorem 4.2.□

Lemma 5.6 *There exists an algorithm which, given a finite path α in $BG(P)$ for a LBASE program P exhibits a history ν of P along α.*

Proof: Any algorithm for solving the path inequality systems (i.e. linear inequality systems over integer variable values) yields some solution Γ of $\mathcal{N}_\alpha$ (since α is feasible, there is a solution of $\mathcal{N}_\alpha$). Take ν to be the history, generated by Γ.□

We do not consider here the methods for solving path inequality systems, let us note just that the very special form of the systems plays a crucial role for the solving process.

Corollary 5.7 *There exists an algorithm which, given an initial finite path α in a LBASE program P*

- *determines, whether α is feasible; and,*
- *if α is feasible, exhibits a program history along α.*

Proof: The path α is feasible if and only if it is a projection of some initial path α' in $BG(P)$ (Proposition 4.15 and Lemma 4.17). Any history along α' in $BG(P)$ is also an initial history along α.□

Note. Actually there is no need in the construction of the whole graph $BG(P)$ for deciding feasibility of a certain program P path α, it is enough to consider only that part of $BG(P)$ which has as projections all feasible initial subpaths of α. In [BBK74] even a simpler approach was chosen by associating a path inequality system *directly* with a path in the program.

Corollary 5.8 *There exists an algorithm which, given a vertex $n \in V(P)$ of a LBASE program P*

- *decides, whether n is reachable (f-correctly reachable); and*
- *if n is reachable (f-correctly reachable), exhibits an initial program P history ν along an (accepting) path α of P, which contains n, i.e. $n = n_i(\nu) = n_i(\alpha)$ for some $i \in C\mathcal{R}d_\alpha$.*

Proof: Given the vertex $n \in V(P)$ we can look in the graph $BG(P)$, whether there is a vertex $\langle n, C\rangle \in V(BG(P))$ for some $C \in \mathcal{C}_P$ which is contained in some initial (initial accepting) path α' in $BG(P)$ (an initial path in $BG(P)$ is called accepting, if it ends with $\langle n', C'\rangle$ for n' being an accepting vertex in P and $C' \in \mathcal{C}_P$). Let α be the projection of α'. Clearly, α is feasible. We obtain the needed history ν along α using the deciding algorithm of Proposition 5.7.□

Let us note that the actual graph $BG(P)$ building process should not necessarily go through the construction of the maximal graph $MG(P)$. One can for given configuration $C \in \mathcal{C}_P$, program vertex $n \in V(P)$ with its operator $p(n)$ and edge e s.t. $f_P(e) = n$ construct all configurations C' for which there exist $\vec{v} \in C$, $a \in \mathbf{Z}$ and $\vec{v}' \in C'$ such that $\langle n, C\rangle \xrightarrow{e} \langle t_P(e), C'\rangle$. In the case of the comparison or dummy operator one needs only to consider the configuration $C' = C$, for assignment operators also the configuration C' is uniquely determined. If $p(n)$ is an input operator, one must consider "all possible places" where the value a can be put w.r.t. the configuration C.

This method of inductive building of the graph $BG(P)$ allows to avoid the construction of a usually very large number of unneeded maximal graph vertexes, so saving a both the time and space resources for the reachability analysis. Actually also the "optimized" algorithms have in the worst case both the runtime and used memory exponential in the number m of the program variables, however only gathering all possible optimizations one can hope to obtain practically feasible program analysis algorithms (see Section B.4.1 in Appendix B for the outline of some further possible path feasibility graph reduction techniques).

5.2 Path Inequality Systems: LTIM

In a similar way as for LBASE programs, we consider a $\mathcal{C}_P$-graph G of a LTIM program P, and define an inequality system $\mathcal{N}_\alpha$ for every path α in G.

We define for the path α the set $\mathcal{P}t_\alpha$ of *points* on α, in a way $\mathcal{P}t_\alpha = \{(b; r) \mid r \in C\mathcal{R}d_\alpha,\ b \in \mathcal{B}_P(C_r(\alpha))\}$ (i.e. r is a coordinate on α and b is a base point of the rth configuration of α).

Given a history β along α in G we associate with it a mapping $\Gamma_\beta : \mathcal{P}t_\alpha \to \mathbf{Q}$, defined

$$\Gamma_\beta((a; r)) = \vec{v}_r(\beta).a$$

and called the *generator* of β.

Easy to see that every mapping $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Q}$ is a generator of the sequence

$$\beta_\Gamma = \langle n_0(\alpha), \Gamma(\vec{x}; 0)\rangle e_0(\alpha), \langle n_1(\alpha), \Gamma(\vec{x}; 1)\rangle e_1(\alpha), \ldots [, \langle n_k(\alpha), \Gamma(\vec{x}; k)\rangle]$$

for $k = card(\mathcal{CR}d_\alpha) - 1$, provided for every i

$$\Gamma(\vec{x}; i) = \langle \Gamma(z; i)\Gamma(x_1; i), \Gamma(x_2; i), \ldots, \Gamma(x_m; i)\rangle.$$

As in the LBASE case, we call the sequence β_Γ the sequence *generated by* the mapping Γ.

Also in the case of a LTIM program P it is easy to see that for any history β along the path α the sequence, generated by the generator of β is the history β itself (i.e. $\beta_{\Gamma_\beta} = \beta$).

Let for an arbitrary program P operator p and an arbitrary $a \in \mathcal{A}_P$ we say that a **ub** p (a *unchanged by* p) if and only if p does not assign a new value to a. We extend also the predicate **ub** p also to expressions in a way that, if $expr_1$ **ub** p and $expr_2$ **ub** p, then also $expr_1 + expr_2$ **ub** p and $expr_1 - expr_2$ **ub** p (it is especially important, that whenever a **ub** p for $a \in Vars(P)$ then for all $c \in \mathbf{Z}$ $a + c$ **ub** p).

Let for a given program P vertex $n \in V(P)$ and edge label $l \in \{" + ", " - "\}$ for $x, y \in \mathcal{A}_P$

- $x <_{n,l} y$, if $p(n)$ is a comparison operator $\mathbf{x} < \mathbf{y}$ and $l = " + "$,
- $y \leq_{n,l} x$, if $p(n)$ is a comparison operator $\mathbf{x} < \mathbf{y}$ and $l = " - "$,
- $x <_{n,l} z$, if $p(n)$ is a positive assignment operator $\mathbf{z} \stackrel{+}{\leftarrow} \mathbf{x}$ and $l = " - "$,
- $z \leq_{n,l} x$, if $p(n)$ is a positive assignment operator $\mathbf{z} \stackrel{+}{\leftarrow} \mathbf{x}$ and $l = " + "$.

We let also $x \leq_{n,l} y$ whenever $x <_{n,l} y$.

The relation $\leq_\alpha$ in the set $\mathcal{P}t_\alpha$ of points along α is defined similarly, as for LBASE programs:

- $(a; r) \leq_\alpha (b; r)$ for $a, b \in \mathcal{A}_P$ and $r, r+1 \in \mathcal{CR}d_\alpha$ whenever $a \leq_{n,l} b$ for $n = n_r(\alpha)$ and $l = l_{E(P)}(e_r(\alpha))$
- if a **ub** $p(n_r(\alpha))$ for $a \in \mathcal{B}_P(C_r(\alpha)) \cap \mathcal{B}_P(C_{r+1}(\alpha))$, $r, r+1 \in \mathcal{CR}d_\alpha$, then $(a; r) \leq_\alpha (a; r+1)$ and $(a; r+1) \leq_\alpha (a; r)$,
- if $a \leq_{C_r(\alpha)} b$ for $a, b \in \mathcal{B}_P(C_r(\alpha))$, then $(a; r) \leq_\alpha (b; r)$,
- the relation $\leq_\alpha$ is transitive, i.e. for $w_1, w_2, w_3 \in \mathcal{P}t_\alpha$, if $w_1 \leq_\alpha w_2$ and $w_2 \leq_\alpha w_3$, then also $w_1 \leq_\alpha w_3$.

The path α inequality system is defined as the set of inequalities $\mathcal{N}'_\alpha = \{w_1 \leq_\alpha w_2 \mid w_1, w_2 \in \mathcal{P}t_\alpha\}$.

Let also in the LTIM case $w =_\alpha w'$ for $w, w' \in \mathcal{P}t_\alpha$ if and only if $w \leq_\alpha w'$ and $w' \leq_\alpha w$, as well as, $w <_\alpha w'$ if and only if $w \leq_\alpha w'$ and the inverse $w' \leq_\alpha w$ does not hold. Let $w' \geq_\alpha w$ mean $w \leq_\alpha w'$ and $w' >_\alpha w$ be another notation for $w <_\alpha w'$.

Definition 5.9 *Given a path α in a $\mathcal{C}_P$-graph G of a LTIM program P and a set $A \subseteq \mathcal{P}t_\alpha$ we call a mapping $\Gamma^A : A \to \mathbf{Q}$ a solution of $\mathcal{N}_\alpha$ w.r.t. A if and only if*

- $\forall c \in Cons(P) : \Gamma^A(c;r) = c$,
- *for any two points $w_1, w_2 \in A$:*
  1. *if $w_1 =_\alpha w_2$ then $\Gamma^A(w_1) = \Gamma^A(w_2)$*
  2. *if $w_1 <_\alpha w_2$ then $\Gamma^A(w_1) < \Gamma^A(w_2)$, as well as*
- *for $(a;r), (b;r) \in A$, such that $b = a + c$ $(c \in \mathbf{Z})$, always $\Gamma^A(b) = \Gamma^A(a) + c$.*

A mapping $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Q}$ is a solution of $\mathcal{N}_\alpha$, if it is a solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{P}t_\alpha$.

Similarly, as in the LBASE case, it can be obtained from the definitions that the analogues of Proposition 5.3 and Proposition 5.4 hold also for $\mathcal{C}_P$-graphs of LTIM programs.

Proposition 5.10 *The construction of the graphs $BG(P)$ for LTIM programs P from the program text is effective.*

Proof: As in the proof of Proposition 5.5 we construct first the maximal $\mathcal{C}_P$-graph $MG(P)$. In order to obtain the needed graph $BG(P)$ it remains only to check whether for given edge $\tilde{e} = \langle e, C_1, C_2\rangle$ in $MG(P)$ the inequality system of the path $\alpha_{\tilde{e}} = (f_P(e), C_1)\tilde{e}, (t_P(e), C_2)$ in $MG(P)$ has a solution or not, and in the case of the solution existence effectively exhibit one. Indeed, the process of finding a solution for $\mathcal{N}_\alpha = \mathcal{N}_{\alpha_{\tilde{e}}}$ (as well as the process of finding a solution for any $MG(P)$ path inequality system) reduces to solving a simple system of linear equations and inequalities w.r.t. the values of the points $(a;r) \in \mathcal{P}t_\alpha$ with $a \in Vars(P)$ being program P variables (all equations and inequalities of the system are of the form $\Gamma(w_1)\lambda\Gamma(w_2)+c$ for $w \in \mathcal{P}t_\alpha$ and $c \in \mathbf{Z}$, $-c_{max} \leq c \leq c_{max}$, $\lambda \in \{=, <, \leq\}$). One can observe that also for LTIM programs we do not have the obviously infeasible "$-\infty$"-labelled edges, outgoing from vertexes with either input, assignment or variable activation $(x \leftarrow z + c)$ operators ascribed.□

Proposition 5.10 completes the proof of Theorem 4.3 (see Lemma 4.29), so resolving also last missing points in the proof of Theorem 4.1.□

As in the case of LBASE programs, we have also for LTIM programs the following result.

Lemma 5.11 *There exists an algorithm which, given a finite path α in $BG(P)$ for a LTIM program P exhibits a history ν of P along α.*

Proof: Any algorithm for solving the path inequality systems (i.e. linear inequality systems over rational variable values) yields some solution Γ of $\mathcal{N}_\alpha$ (since α is feasible, there is a solution of $\mathcal{N}_\alpha$). Take ν to be the history, generated by Γ.□

Concerning the general case of the analysis of LTIBA programs we have the following consequences of the proof of Theorem 4.1.

Corollary 5.12 *The LTIBA program vertex reachability problem is decidable.*

Proof: Build a projectivee H for the set of all finite feasible paths in given program P and look whether there is a path α in H from the initial vertex to some vertex $n' \in V(H)$, labelled by the given P vertex n.□

Corollary 5.13 *The set of all finite correct (accepting) paths for a LTIBA program with finite acceptance set is T-projective.*

Proof: As in the proof of Theorem 4.1, we can due to Lemma 2.8 consider the cases of LBASE and LTIM programs independently. For P being either a LBASE or LTIM program with the finite acceptance set $S_F(P) \subseteq V(P)$ let in the basic graph $BG(P)$ $S_F(BG(P)) = \{(n, C) \in V(BG(P)) \mid n \in S_F(P)\}$.

Due to Proposition 4.15 (note the perfectness of $BG(P)$ according to Lemma 4.17 (LBASE) and Lemma 4.29 (LTIM))

- every finite accepting feasible path α in P is a projection of some initial path α' in $BG(P)$, α' is accepting according to the definition of $S_F(BG(P))$ due to $n_*(\alpha) = n_*(\alpha')$;
- the projection of every finite initial accepting path in $BG(P)$ is a feasible accepting path in P.□

Corollary 5.14 *The f-correct reachability problem for LTIBA program vertexes is decidable.*

Proof: Build a T-projectivee H for the set of all finite feasible paths in given program P and look whether there is an accepting initial path α in H containing some vertex $n' \in V(H)$ which is labelled by the given P vertex n.□

Observing the fact that a path in a LTIBA program P is feasible if and only if it is feasible in both the programs P^B and P^T (see Section 3.2) and using the effectivity of the solving of path inequality system both for LBASE and LTIM program basic graphs (Lemma 5.6 and Lemma 5.11), we obtain also for LTIBA programs (see note in Section 5.1 after the proof of Corollary 5.7) the following result.

Corollary **5.15** *There exists an algorithm which, given an initial finite path α in a LTIBA program P*

- *determines, whether α is feasible; and,*
- *if α is feasible, exhibits a program history along α.*

Corollary **5.16** *There exists an algorithm which, given a vertex $n \in V(P)$ of a LTIBA program P*

- *decides, whether n is reachable (f-correctly reachable); and,*
- *if n is reachable (f-correctly reachable), exhibits an initial program P history ν along an (accepting) path α of P, which contains n.*

Proof: Follows from the proofs of Corollary 5.12 and Corollary 5.14 by Corollary 5.15.□

5.3 Point Classes

In what follows we develop some further techniques for infinite path feasibility analysis in both LBASE and LTIM programs.

From now on let us concentrate on the analysis of the basic graphs for LBASE and LTIM programs out of the whole family of C_P-graphs, defined in Section 4.2. The similar definitions of path inequality systems in basic graphs $BG(P)$ for the LBASE and LTIM programs allows to make some constructions for programs in these languages in a uniform way, we assume that P is a LL-program for LL being either LBASE or LTIM.

First of all we mention some simple and useful properties of the path inequality systems in the graphs $BG(P)$ for LL programs P.

Fact **5.17** *The inequality system $\mathcal{N}_\alpha$ for every finite path α in $BG(P)$ for a LL program P is not contradictory, i.e. there is a solution of $\mathcal{N}_\alpha$.*

Proof: Follows from Theorem 4.1 and Proposition 5.4 (consider also its analogue for LTIM programs).□

Fact **5.18** *For every path α in $BG(P)$ and every $r \in \mathcal{CR}d_\alpha$ $a, b \in BP(C_r(\alpha))$ the inequality $(a; r) \leq_\alpha (b; r)$ holds if and only if $a \leq_{C_r(\alpha)} b$ does.*

Proof: Follows from Fact 5.17: since $\mathcal{N}_\alpha$ is not contradictory, there cannot be any inequality of the sort $(a;r) \leq_\alpha (b;r)$ without the corresponding $a \leq_{C_r(\alpha)} b$ (notice that for all $a, b \in \mathcal{B}_P(C)$ always either $a <_C b$. or $b <_C a$ or $a =_C b$).□

Fact 5.18 eliminates the need to consider the inequalities $a \leq_\alpha b$ generated by $a \leq_{n_r(\alpha),l} b$ in the definition of the path inequality systems - every such inequality is already contained in $\mathcal{N}_\alpha$ because of the corresponding inequality in the graph vertex $(n_r(\alpha), C_r(\alpha))$ configuration $C_r(\alpha)$.

Let us for α being a path in $BG(P)$ for a LL program P and $r \in \mathcal{CR}d_\alpha$ define the set of points $\mathcal{P}t_\alpha(r)$ associated with the path α coordinate r in a way

$$\mathcal{P}t_\alpha(r) = \{(a;u) \in \mathcal{P}t_\alpha \mid r = u\}$$

(for a LBASE program P $\forall u\ \mathcal{P}t_\alpha(u) = \{(a;u) \mid a \in \mathcal{A}_P\}$, in the LTIM case $\mathcal{P}t_\alpha(u) = \{(a;u) \mid a \in \mathcal{B}_P(C_u(\alpha))\}$).

Fact 5.19 *If $(a;t) \leq_\alpha (b;t+r)$ for $(a;t), (b;t+r) \in \mathcal{P}t_\alpha$, then for every $i = 1, \ldots, r-1$ there exists $w \in \mathcal{P}t_\alpha(t+i)$ such that $(a;t) \leq_\alpha w \leq_\alpha (b;t+r)$.*

Proof: Let $(a;s) \leq'_\alpha (b;u)$ for $(a;s), (b;u) \in \mathcal{P}t_\alpha$ iff either $s = u$ and $a \leq_C b$ for $C = C_u(\alpha)$, or $a = b$, $s = u+1$ and a ub $p(n_u(\alpha))$, or else $a = b$, $u = s+1$ and a ub $p(n_s(\alpha))$. Since $\leq_\alpha$ is the transitive closure of $\leq'_\alpha$, $(a;t) \leq_\alpha (b;t+r)$ implies the existence of an inequality chain $(a;t) \leq'_\alpha w_1 \leq'_\alpha w_2 \leq'_\alpha \cdots \leq'_\alpha (b;t+r)$ easy to see that $w_u \in \mathcal{P}t_\alpha(t+i)$ for at least one w_u in the sequence.□

A similar result can be easily obtained also for the relation $\geq_\alpha$.

Fact 5.20 *If $w_1 =_\alpha w_2$ for $w_1 \in \mathcal{P}t_\alpha(t)$ and $w_2 \in \mathcal{P}t_\alpha(t+r)$ then for every $i = 1, \ldots, r-1$ there exists $w \in \mathcal{P}t_\alpha(t+i)$ such that $w =_\alpha w_1$.*

Proof: Easily obtained from Fact 5.19.□

The following simple result guarantees the "locality" of the relations $\leq_\alpha, \geq_\alpha$, so allowing to mix the usage of the inequality system of the whole path α with the inequality system of a path α fragment β when considering the points within the fragment β.

Fact 5.21 *Let β be a subpath of α in $BG(P)$ from ith to jth path α coordinate $(i, j \in \mathcal{CR}d_\alpha)$. Then for $w \in \mathcal{P}t_\alpha(i)$ and $w' \in \mathcal{P}t_\alpha(j)$ $w \leq_\alpha w'$ if and only if $w \leq_\beta w'$ and $w \geq_\alpha w'$ if and only if $w \geq_\beta w'$.*

Proof: Follows from Fact 5.18, Fact 5.19 and the definition of $\leq_\alpha$.□

We consider the set $\mathcal{P}t_\alpha$ of points along a path α in $BG(P)$ and define a partitioning $Q_\alpha = \{\langle\langle w\rangle\rangle_\alpha \mid w \in \mathcal{P}t_\alpha\}$ of $\mathcal{P}t_\alpha$ according to the equivalence relation $=_\alpha$, i.e. $\langle\langle w_1\rangle\rangle_\alpha = \langle\langle w_2\rangle\rangle_\alpha$ iff $w_1 =_\alpha w_2$ for any two $w_1, w_2 \in \mathcal{P}t_\alpha$. We often omit the index α in the denotation of the $=_\alpha$ - equivalence classes if it is clear from the context.

Definition 5.22 *We call any element of Q_α (i.e. any $=_\alpha$ - equivalence class) the point class along the path α.*

We introduce also the extended set of point classes $\overline{Q}_\alpha = Q_\alpha \cup \{-\infty, +\infty\}$. We extend the inequalities $<_\alpha, =_\alpha, \leq_\alpha$ to be defined also for the (extended) point classes along the given path α, we will allow ourselves even to mix the points and point classes in some such inequalities. By definition, $\forall \tilde{w} \in \overline{Q}_\alpha \quad -\infty \leq_\alpha \tilde{w} \leq_\alpha +\infty$, for $w \in Q_\alpha$ we have $-\infty <_\alpha w <_\alpha +\infty$.

The following notion of the point class stability is very important in studying the infinite path feasibility for LBASE programs.

Definition 5.23 *Given an infinite path α in $BG(P)$ we call a point class $\tilde{w} \in Q_\alpha$ stable along the path α if and only if there exists $u \in N$ such that for all $u' \geq u$ one can find some P variable or constant $a \in \mathcal{A}_P$ (a base point $a \in \mathcal{B}_P(C_{u'}(\alpha))$) with the point $(a; u') \in \tilde{w}$.*

Intuitively a point class corresponds to the "life-cycle" of a "value" in the program during its execution along the path α: the "value" is read from the input (at this time the first element of the point class appears in $\mathcal{P}t_\alpha$), afterwards it can "visit" (be assigned and re-assigned to) various variables, may be at some path coordinate the value "disappears" from the program since it becomes not held by any program variable or constant (the constants hold only their own fixed values, of course). A point class is stable if it represents a "value" which never disappears from the program while executing along the infinite path α (obviously, every point class, containing a constant at some stage, contains this constant forever and, so, is stable, however, there can be also other stable point classes along the path (e.g. when some variable keeps its "value" forever)).

Fact 5.24 *The number of stable point classes along an infinite path in $BG(P)$ for a LBASE program P does not exceed the number $card(\mathcal{A}_P)$ of program P variables and constants.*

Proof: Starting from some point in the path after every execution step every stable point class must be represented by some variable or constant (i.e. for some $u \in N$ and every $u' \geq u$ for every stable point class $\tilde{w}$ there exists $a \in \mathcal{A}_P$ with $(a; u') \in w$).□

Now we obtain some means for analyzing more carefully the relation $\leq_\alpha$ for a given path α. More precisely, given a point $(a; r)$ of the variable (constant) a at

some coordinate r in the path, we want to know for every natural s, which points at the coordinate $r+s$ are greater, which are less and which incomparable with $(a;r)$ according to the relations $\leq_\alpha, <_\alpha$ (these constructions are afterwards used in both the LBASE and LTIM program infinite path feasibility analysis).

Definition 5.25 *Given a path α and a point $(a;r) \in \mathcal{P}t_\alpha$ let us define for a natural s, such that $r+s \in \mathcal{CR}d_\alpha$*

- *the lower bound of $(a;r)$ at the coordinate $r+s$ on α as the point class $\inf_\alpha((a;r),s) \in \overline{Q}_\alpha$ such that :*
  1. *if $w \leq_\alpha (a;r)$ for no $w \in \mathcal{P}t_\alpha(s+r)$, then $\inf_\alpha((a;r),s) = -\infty$,*
  2. *otherwise, $\inf_\alpha((a;r),s) = \langle\!\langle (b;r+s) \rangle\!\rangle_\alpha$, where $(b;r+s)$ is the maximal of points at the coordinate $r+s$ with $(b;r+s) \leq_\alpha (a;r)$ (for every $(b',s+r) \in \mathcal{P}t_\alpha(s+r)$, if $b <_{C_{r+s}} b'$, then $(b';r+s) \leq_\alpha (a;r)$ is not contained in $\mathcal{N}_\alpha$);*
- *the upper bound of $(a;r)$ at the coordinate $r+s$ on α as the point class $\sup_\alpha((a;r),s) \in \overline{Q}_\alpha$ such that*
  1. *if $(a;r) \leq_\alpha w$ for no $w \in \mathcal{P}t_\alpha(s+r)$, then $\sup_\alpha((a;r),s) = +\infty$,*
  2. *otherwise, $\sup_\alpha((a;r),s) = \langle\!\langle (b;r+s) \rangle\!\rangle_\alpha$, where $(b;r+s)$ is the minimal of points at the coordinate $r+s$ with $(a;r) \leq_\alpha (b;r+s)$ (for every $(b',s+r) \in \mathcal{P}t_\alpha(s+r)$, if $b' <_{C_{r+s}} b$, then $(a;r) \leq_\alpha (b';r+s)$ is not contained in $\mathcal{N}_\alpha$).*

Example 5.26 *For the path α from Example 5.1*

$\inf_\alpha((x;0),2) = \langle\!\langle (1;2) \rangle\!\rangle_\alpha = \{(1;0),(1;1),(1,2)\}$
$\sup_\alpha((x;0),2) = \langle\!\langle (y;2) \rangle\!\rangle_\alpha = \{(x;2),(y;2),\ldots\}$
$\inf_\alpha((z;0),2) = \langle\!\langle (y;2) \rangle\!\rangle_\alpha$
$\sup_\alpha((z;0),2) = +\infty$.

Following there are some simple and useful properties of the point bounds. Let α be a fixed path in $BG(P)$ for a LL program P.

Fact 5.27 *If $(a;r) \leq_\alpha (b;r)$ for some a,b,r then for every s*

- $\inf_\alpha((a;r),s) \leq_\alpha \inf_\alpha((b;r),s)$,
- $\sup_\alpha((a;r),s) \leq_\alpha \sup_\alpha((b;r),s)$.

Proof: Follows from the transitivity of $\leq_\alpha$.□

Fact 5.28 *If $\langle\!\langle w \rangle\!\rangle = \langle\!\langle w' \rangle\!\rangle = \tilde{w} \in Q_\alpha$ for $w \in \mathcal{P}t_\alpha(t)$ and $w' \in \mathcal{P}t_\alpha(t+r)$ then also $\tilde{w} = \sup_\alpha(w,r) = \inf_\alpha(w,r)$.*

Proof: Straightforward from definitions.□

Fact 5.29 *If for some* $w \in \mathcal{P}t_\alpha$ *and some* r $\inf_\alpha(w,r) = \sup_\alpha(w,r)$ *then also* $\langle\!\langle w \rangle\!\rangle = \inf_\alpha(w,r) = \sup_\alpha(w,r)$.

Proof: Follows from Fact 5.20 and the definitions.□

Proposition **5.30** *For every* $w = (a;i) \in \mathcal{P}t_\alpha$, *for every* $j,k \in \mathbf{N}$, *such that* $i+j+k \in \mathcal{CR}d_\alpha$

- $\sup_\alpha(w,k+j) \geq_\alpha \sup_\alpha(w,k)$ *and*
- $\inf_\alpha(w,k+j) \leq_\alpha \inf_\alpha(w,k)$.

Proof: According to Fact 5.19 there exists $w' \in \mathcal{P}t_\alpha(k)$ such that $w \leq_\alpha w' \leq_\alpha \sup_\alpha(w,k+j)$. According to Fact 5.18 for the configuration $C_{i+k}(\alpha)$, the only possibility not to contradict the definition of $\sup_\alpha$ is to take $w' \geq_\alpha \sup_\alpha(w,k)$. The result for the upper bounds follows by the transitivity of $\geq_\alpha$. The case of the lower bounds is treated similarly.□

Proposition **5.31** *If* $(b;t+r) \in \sup_\alpha((a;t),r)$ *then* $\sup_\alpha((b;t+r),s) = \sup_\alpha((a;t),r+s)$ *for every* s. *Similarly, if* $(b;t+r) \in \inf_\alpha((a;t),r)$ *then* $\inf_\alpha((b;t+r),s) = \inf_\alpha((a;t),r+s)$ *for every* s.

Proof: Let $(b;t+r) \in \sup_\alpha((a;t),r)$. Due to Proposition 5.30 $\sup_\alpha((a;t),r+s) \geq_\alpha (b;t+r)$. If $w \geq_\alpha (b;t+r)$ for some $w \in \mathcal{P}t_\alpha(t+r+s)$ such that $w <_\alpha \sup_\alpha((a;t),r+s)$, Fact 5.19 gives $w' \in \mathcal{P}t_\alpha(t+r)$ with $(a;t) \leq_\alpha w' \leq_\alpha w$. In both cases, if either $w' <_\alpha (b;t+r)$, or not, a contradiction with the definition of upper bound follows, so $\sup_\alpha((a;t),r+s) = \sup_\alpha((b;t+r),s)$. The case of the lower bounds is treated similarly.□

Let us call this result the *transitivity* of lower and upper bounds of points.

Chapter 6

LBASE: Infinite Feasible Paths

It is easy to see that there can be infinite not feasible paths with all finite prefixes feasible in both LBASE and LTIM programs, e.g.

?x;"+"; ?y;"+"; 3: x < y;"+"; ?v;"+"; v < y;"+"; y ← v;"+"; goto 3: and

?x;"+"; 2: y < x;"+"; z $\overset{+}{\leftarrow}$ y;"+"; y ← z + 1;"+"; goto 2: .

So, at least for some LBASE and LTIM programs P their basic graphs contain infinite infeasible paths. Moreover, it is not obvious how to find out whether the given program has an infinite feasible path at all. This chapter addresses the problem for LBASE programs.

Theorem 6.1 *There is an algorithm which, given a LBASE program P, decides whether it has an infinite feasible path.*

Proof: According to Fact 4.12 and Fact 4.13, an infinite initial path in the program P is feasible if and only if it is a projection of some initial feasible path in $BG(P)$. So, instead of deciding the existence of an infinite feasible path in the program, we can decide the existence of such a path in its basic graph. In what follows, the variable value information after every execution step, contained in basic graph vertexes will be in essential use by the deciding algorithm.

Theorem 6.2 *There is an algorithm which, given the graph $BG(P)$ for a LBASE program P, decides whether it has an initial infinite feasible path.*

Proof: The idea of the proof is to find a property $\mathcal{PP}$ of finite paths in $BG(P)$ such that

- every infinite feasible path contains as subpath at least one path satisfying $\mathcal{PP}$,

- an infinite path in $BG(P)$, consisting of infinitely many times repeated path satisfying $\mathcal{PP}$, is feasible, and
- it is decidable whether there is a finite path satisfying $\mathcal{PP}$ in $BG(P)$.

6.1 Accomplished Loops

We start with the definitions yielding the notion of an accomplished loop which is afterwards proved to be able to serve as the abovementioned finite path property $\mathcal{PP}$.

Let us call any P variable and constant set $A \subseteq \mathcal{A}_P$ *constant holding*, if $Cons(P) \subseteq A$.

Given a constant holding (and so nonempty) set $A \subseteq \mathcal{A}_P$ and a configuration $C \in C_P$, let $M = M(A,C)$ be one of the elements of A, such that $\forall a \in A : a <_C M$ and $m = m(A,C) \in A$ such that $\forall a \in A : m <_C a$ (it may be the case that some of the variables M, m are not uniquely determined, in such a case take any suitable variable or constant).

Let us introduce a partitioning $\{B_{A,C}, U_{A,C}, L_{A,C}\}$ of the set $\mathcal{A}_P$ in a way:

- $B_{A,C} = \{a \in \mathcal{A}_P \mid m \leq_C a \leq_C M\}$, we call this set the *bounded interval* of C w.r.t. A;
- $U_{A,C} = \{a \in \mathcal{A}_P \mid M <_C a\}$, the *upper interval* of C w.r.t. A;
- $L_{A,C} = \{a \in \mathcal{A}_P \mid a <_C m\}$, the *lower interval* of C w.r.t. A.

Definition **6.3** *We call a finite path α with $\mathcal{CRd}_\alpha = \{0, 1, \ldots, k\}$ a (n, C, A, k)- loop for $(n, C) \in V(BG(P))$ and $A \subseteq \mathcal{A}_P$ being a constant holding set iff*

- $(n_0(\alpha), C_0(\alpha)) = (n_k(\alpha), C_k(\alpha)) = (n, C)$,
- $\langle\!\langle (a;0) \rangle\!\rangle_\alpha = \langle\!\langle (a;k) \rangle\!\rangle_\alpha$ *for all* $a \in A$.

A (n, C, A, k)-loop in $BG(P)$ is a path of the length k, starting and ending in the same vertex (n, C), provided for all $a \in A$

- the point classes (see Definition 5.22) containing $(a;0)$ have representants $(a';i) \in \langle\!\langle (a;0) \rangle\!\rangle_\alpha$ with $a' \in \mathcal{A}_P$ for all coordinates i along the path α, and
- these representants for i being the path's last coordinate k come back to the variables a, i.e. $\forall a \in A : (a;k) \in \langle\!\langle (a;0) \rangle\!\rangle_\alpha$.

A path in $BG(P)$ is called a (n, C, A)-loop, if it is a (n, C, A, k)-loop for some $k \in \mathbf{N}$.

Definition **6.4** *We call a (n, C, A, k) - loop α in $BG(P)$ accomplished if*

- *for every* $a \in B_{A,C} \cup U_{A,C}$ *the lower bound* $\inf_\alpha((a;0),k) = \langle\langle(b;k)\rangle\rangle$ *for some* $b \in A$,
- *for every* $a \in B_{A,C} \cup L_{A,C}$ *the upper bound* $\sup_\alpha((a;0),k) = \langle\langle(b;k)\rangle\rangle$ *for some* $b \in A$.

Intuitively, a path α is a (n, C, A)-loop whenever the variables and constants $a \in A$ have persistent values along α (these values can be stored in other variables for the intermediate path coordinates). The property of α being an accomplished loop tells that the upper and lower bounds of all other program variables at the beginning of the path reduce along the path to these variables with the persistent values (save the upper bounds of variables upper the largest variable from A and the lower bounds of variables below A).

Due to the finiteness of the sets $V(P), \mathcal{C}_P, \mathcal{A}_P$, the proof of the Theorem 6.2 now follows from the following three lemmas:

Lemma 6.5 *Every feasible infinite path in* $BG(P)$ *contains a fragment, which is an accomplished* (n, C, A, k) *- loop for some* $n \in V(P), C \in \mathcal{C}_P, k \in N$ *and* $A \subseteq \mathcal{A}_P$ *- constant holding set of* P *variables and constants.*

Lemma 6.6 *Every infinite path* α^* *in* $BG(P)$, *which after some initial fragment* γ *consists of infinitely many times repeated accomplished loop* α, *is feasible. Moreover, there exists an algorithm which given the initial fragment* γ *and the loop* α *in* $BG(P)$ *computes some history along* α^*.

Lemma 6.7 *There exists an algorithm which, given a vertex* (n, C) *in* $BG(P)$ *and a constant holding set* $A \subseteq \mathcal{A}_P$, *decides whether there is an accomplished* (n, C, A) *- loop in* $BG(P)$.

It is easy to see that, having proved these lemmas, one can look for the existence of an accomplished loop in $BG(P)$, which is reachable (in the graph theoretic sense) from the graph initial vertex, by using the algorithm yielded by Lemma 6.7 on all possible arguments n, C, A. If in some case an appropriate accomplished loop is found, we have got, according to Lemma 6.6, a cyclic infinite feasible path in the $BG(P)$ (and so, in the program P). If it turns out that in no case of n, C, A a corresponding accomplished loop exist, then according to Lemma 6.5 there is no infinite feasible path in $BG(P)$ (in P).

Before proving the lemmas let us note that we do not claim the feasibility of every infinite feasible path which infinitely many times *contains* an accomplished loop, the proof of Theorem 6.17 below gives a counterexample. Lemma 6.6 just asserts the feasibility of infinite paths which, starting from some point contain *nothing else* but some infinitely many times repeating accomplished loop.

6.2 Existence of Accomplished Loop

Proof of **Lemma 6.5**: Let us assume that we have in $BG(P)$ an infinite feasible path α. Let $\mathcal{W}_\alpha = \{\tilde{w}_1, \ldots, \tilde{w}_s\}$ be the set of *stable* point classes along α (the finiteness of the set $\mathcal{W}_\alpha$ is by Fact 5.24). Let for every $\tilde{w} \in \mathcal{W}_\alpha$ $\#(\tilde{w})$ be the least path α coordinate i for which the equation $\tilde{w} = \langle\!\langle (a;i) \rangle\!\rangle_\alpha$ holds for some $a \in \mathcal{A}_P$, let $\#_\alpha = \max\{\#(\tilde{w}) \mid \tilde{w} \in \mathcal{W}_\alpha\}$ ($\#_\alpha \in \mathcal{CR}d_\alpha$ is the least path α coordinate at which all stable point classes along α already have their representants, clearly, for every $\tilde{w} \in \mathcal{W}_\alpha$ and every $i \geq \#_\alpha$ there exists a representant $(a;i) \in \tilde{w}$).

Clearly, some of $BG(P)$ vertexes are repeated in α infinitely many times, let us take one of them, say $\langle n, C\rangle$, and let $j_1, j_2, \ldots$ be an infinite sequence of increasing indices j such that $j_1 \geq \#_\alpha$, which have the vertex $(n_j(\alpha), C_j(\alpha))$ coincident with $\langle n, C\rangle$.

Let $\chi(\tilde{w}, i) \in \mathcal{A}_P$ for $\tilde{w} \in \mathcal{W}_\alpha, i \in N$ denotes some P variable or constant a such that $\langle\!\langle (a;j_i) \rangle\!\rangle = \tilde{w}$. Let $\chi_i : \mathcal{W}_\alpha \to \mathcal{A}_P$ for every i be the mapping with $\chi_i(\tilde{w}) = \chi(\tilde{w}, i)$ for every $\tilde{w} \in \mathcal{W}_\alpha$.

Since the set $\mathcal{A}_P$ is finite, there exist a mapping $\chi : \mathcal{W}_\alpha \to \mathcal{A}_P$ which is (extensionally) equal with χ_i for infinitely many indices i. Let us fix one such mapping χ and define a subsequence $(l_s)_{s\in N} = (j_{i_s})_{s\in N}$ of $(j_i)_{i\in N}$ such that $\chi_{i_s} = \chi$ for every s.

We define the set $A \in \mathcal{A}_P$ by letting $a \in A$ for $a \in \mathcal{A}_P$ if and only if $a =_C b$ for some $b \in \chi(\mathcal{W}_\alpha) = \{\chi(\tilde{w}) \mid \tilde{w} \in \mathcal{W}_\alpha\}$.

Now consider the sequence of finite paths

$$\beta_s = \langle n_{l_1}(\alpha), C_{l_1}(\alpha)\rangle \tilde{e}_{l_1}, \ldots, \langle n_{l_s}(\alpha), C_{l_s}(\alpha)\rangle,$$

and prove that for some sufficiently large s the path β_s is an accomplished loop.

First of all, it is clear from the construction that for every s the path β_s is a $(n, C, A, l_s - l_1)$ - loop.

In order to prove that for some s the path β_s is accomplished $(n, C, A, l_s - l_1)$ - loop, assume the contrary. Without loosing generality, let for some $a \in B_{A,C} \cup U_{A,C}$ for every $u \in \mathbf{N}$ $\inf_\alpha((a;l_1), u)$ is not a stable point class along α (the case of unstable upper bound is analogical).

Observe that $a \in B_{A,C} \cup U_{A,C}$ means $a \geq_C m(A,C)$. Since $C_{l_1}(\alpha) = C$ we obtain that $(a;l_1) \geq_\alpha (m(A,C);l_1)$. Due to the monotonicity of the lower bounds (see Fact 5.27) for all u

$$\inf_\alpha((a;l_1), u) \geq_\alpha \inf_\alpha((m(A,C);l_1), u) = \langle\!\langle (m(A,C);l_1) \rangle\!\rangle_\alpha$$

(observe that $m(A,C) \in A$ by the definition of m, so the point class $\langle\!\langle (m(A,C);l_1) \rangle\!\rangle_\alpha$ is stable along α).

Due to the feasibility of the path α there exists a history ν along it, generated by some mapping $\Gamma_\nu : \mathcal{P}t_\alpha \to \mathbf{Z}$ which preserves the relations $\leq_\alpha, <_\alpha, =_\alpha$ (see Definition 5.2), let $\tilde{\mu} = \Gamma_\nu((m(A,C);l_1))$.

The non-stability of the point classes $\inf_\alpha((a;l_1),u)$, $u \in N$ means that for infinitely many coordinates u_i the classes $\inf_\alpha((a;l_1),u_i)$ are different. Moreover, due to the Proposition 5.30, for $u_j > u_i$ always

$$\inf_\alpha((a;l_1),u_j) <_\alpha \inf_\alpha((a;l_1),u_i).$$

Let $u_1, u_2, \ldots$ be such a sequence of the increasing coordinates u_i with all the point classes $w(u_i) = \inf_\alpha((a;l_1),u_i)$ different. Then the sequence

$$\Gamma_\nu(w(u_1)), \Gamma_\nu(w(u_2)), \ldots$$

is an infinite decreasing sequence of integers having the lower bound $\tilde{\mu} < \Gamma_\nu(w(u_i))$ for every i, a contradiction.□

6.3 Accomplished Loops: Sufficiency

Proof of Lemma 6.6:

Let us prove first that every path α^* in $BG(P)$ which consists of infinitely many times repeated accomplished (n, C, A, k) - loop α (without any pre-period) is feasible. First of all consider the inequality system $\mathcal{N}_\alpha$ of the path α. Let us call all the point classes $\langle\langle(a;0)\rangle\rangle_\alpha$ for all $a \in A$ *stable* along α (by the definition of an accomplished loop for every such a $\langle\langle(a;0)\rangle\rangle = \langle\langle(a;k)\rangle\rangle$, so also for every $i \leq k$ one can find $b \in \mathcal{A}_P$ with $\langle\langle(b;i)\rangle\rangle = \langle\langle(a;0)\rangle\rangle$. Recall that up to now we had defined stable point classes only along infinite paths).

Lemma 6.8 *The path α inequality system $\mathcal{N}_\alpha$ has a solution $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Z}$ such that for every $a \in B_{A,C}$ (i.e. for every variable or constant a in the bounded interval of C w.r.t. A) $\Gamma(a;0) = \Gamma(a;k)$.*

Let us introduce a new inequality system $\mathcal{N}'_\alpha$, which consists of all the $\mathcal{N}_\alpha$ inequalities, as well as of the ones $(a;0) =' (a;k)$ for all $a \in B_{A,C}$. To prove the lemma means to prove that the new system has a solution. Let us denote by $='_\alpha$ the transitive closure of the relation $(=_\alpha \cup =') \in \mathcal{P}t_\alpha \times \mathcal{P}t_\alpha$, as well as by $\leq'_\alpha$ the transitive closure of $\leq_\alpha \cup ='$, let $w <'_\alpha w'$, if $w \leq'_\alpha w'$ and not $w ='_\alpha w'$.

Definition 6.9 *Let us call sequence of $\mathcal{N}'_\alpha$ inequalities $w_1\lambda_1 w_2\lambda_2 \ldots w_n\lambda_n w_1$, where $w_i \in \mathcal{P}t_\alpha$ for every i the increasing loop, if for every $i \leq n$ λ_i is either $\leq_\alpha$ or $='$ and $\lambda_n = <_\alpha$.*

Obviously, did $\mathcal{N}'_\alpha$ contain an increasing loop, it were contradictory.

We prove the inverse, i.e., if $\mathcal{N}'_\alpha$ does not contain an increasing loop, it has a solution.

Let us show, how to define the values $\Gamma(w)$ for nonnegative $w \in \mathcal{P}t_\alpha$ (i.e. for w with $(0;0) \leq_\alpha w$). Obviously, any constant is $\leq_\alpha$ - comparable with every $w \in \mathcal{P}t_\alpha$, the case of $w \leq_\alpha (0;0)$ is analogous with the considered one.

For any $w \in \mathcal{P}t_\alpha$ whenever $w =_\alpha (c;0)$ for some $c \in Cons(P)$ let $\Gamma(w) = c$. Consider now the points from the set $W = \{w \mid w \geq_\alpha (c_{max};0)\}$. Due to the absence of an increasing loop in $\mathcal{N}'_\alpha$, one can order all the points $w \in W$ in a finite sequence $w_1, w_2, \ldots, w_s$ in a way to meet

- $w_j <'_\alpha w_i$ for no $i < j$ and,
- if $w_i ='_\alpha w_{i+s}$, then for all $r \leq s$ also $w_i ='_\alpha w_{i+r}$

(assume we have found which points $w \in W$ are to be taken as $w_1, w_2, \ldots, w_{i-1}$ and we want to determine w_i, the ith element of the sequence. In the case, if there is no not yet chosen points which are $='_\alpha$-equal with w_{i-1}, one can let w_i to be one of the points for which $w_j <'_\alpha w_i$ holds for no $w_j \in W \setminus \{w_1, \ldots, w_{i-1}\}$: did not such w_i exist, we could get an increasing loop).

Clearly, one can define now step by step the values $\Gamma(w_i)$, starting from $i = 1$ to $i = s$, letting $\Gamma(w_1) = c_{max} + 1$ and for every $i > 1$ $\Gamma(w_i) = \Gamma(w_{i-1})$, if $w_i ='_\alpha w_{i-1}$, otherwise $\Gamma(w_i) = \Gamma(w_{i-1}) + 1$, so obtaining the needed solution Γ of $\mathcal{N}'_\alpha$.

Now in order to prove Lemma 6.8, it remains to prove the absence of increasing loops in $\mathcal{N}'_\alpha$.

Lemma 6.10 *There is no increasing loop in $\mathcal{N}'_\alpha$.*

Proof: Assume the contrary: there is in $\mathcal{N}'_\alpha$ an increasing loop

$$w_1 \lambda_1 w_2 \lambda_2 \ldots w_n \lambda_n w_1.$$

Note that $\mathcal{N}_\alpha$ is not contradictory, so λ_i is $='$ for at least one i. Due to the transitivity of $\leq_\alpha$ and the implications

$$w_1 <_\alpha w_2 \leq_\alpha w_3 \Rightarrow w_1 <_\alpha w_3,$$

$$w_1 \leq_\alpha w_2 <_\alpha w_3 \Rightarrow w_1 <_\alpha w_3,$$

referred to further on as the transitivity of $<_\alpha$, one may assume that the given increasing loop consists only of boundary points, i.e. for every i either $w_i = (a;0)$ or $w_i = (a;k)$ for some a. Since $C_0(\alpha) = C_k(\alpha)$, then $(a;0) \leq_\alpha (b;0)$ if and only if $(a;k) \leq_\alpha (b;k)$ (recall Fact 5.18). So, if an increasing loop contains a fragment

$$(a;0) =' (a;k) \leq_\alpha (b;k) =' (b;0),$$

this fragment can be replaced by $(a;0) \leq_\alpha (b;0)$ and can be further reduced by the transitivity with the reduced loop still remaining to be increasing.

Hence, if there exists an increasing loop in $\mathcal{N}'_\alpha$, there also exists an increasing loop in one of the following two forms:

$$(a_1;0) =' (a_1;k) \leq_\alpha (a_2;0) =' (a_2;k) \leq_\alpha \ldots \leq_\alpha (a_t;0) =' (a_t;k) <_\alpha (a_1;0), \text{ or}$$

$$(a_1;k) =' (a_1;0) \leq_\alpha (a_2;k) =' (a_2;0) \leq_\alpha \ldots \leq_\alpha (a_t;k) =' (a_t;0) <_\alpha (a_1;k).$$

In the first case there are two possibilities: either $a_1 \leq_{C_0(\alpha)} a_t$, or $a_i >_{C_0(\alpha)} a_{i+1}$ for some i. For both alternatives we easily conclude that $(b;0) =' (b;k) <_\alpha (b;0)$ for some b (recall the definition of $='$), so, $(b;k) <_\alpha (b;0)$ for some $b \in B_{A,C}$. Were we taken the increasing loop of the second form, this would lead us to the inequality $(b;k) >_\alpha (b;0)$ for some $b \in B_{A,C}$. Because of the full symmetry, let us consider further only one case, say, that of $(b;k) <_\alpha (b;0)$.

According to the definition of an accomplished loop,

$$\inf((b;0),k) = \langle\langle(a;k)\rangle\rangle = \langle\langle(a;0)\rangle\rangle$$

for some $a \in A$. Clearly, $a \neq b$, easy to see also $\langle\langle(a;0)\rangle\rangle \neq \langle\langle(b;0)\rangle\rangle$ (otherwise, due to $C_0(\alpha) = C_k(\alpha)$, one could conclude also $\langle\langle(b;0)\rangle\rangle = \langle\langle(b;k)\rangle\rangle$, what contradicts $(b;k) <_\alpha (b;0)$).

By the definition of the lower bound $(a;k) \leq_\alpha (b;0)$, so since $(a;0) =_\alpha (a;k)$, also $(a;0) \leq_\alpha (b;0)$. Hence by Fact 5.18 $a \leq_{C_0(\alpha)} b$, so $a \leq_{C_k(\alpha)} b$, as well. Since $a \neq_{C_k(\alpha)} b$, one concludes $a <_{C_k(\alpha)} b$, this brings the inequality $(b;k) <_\alpha (b;0)$ in contradiction with the definition of $\inf((b;0),k) = \langle\langle(a;k)\rangle\rangle$. □□

Now let us continue the proof of Lemma 6.6, namely, the proof of the feasibility of the path α^*, consisting of the infinitely many times repeated accomplished loop α. Let us start to define the solution $\Gamma : \mathcal{P}t_{\alpha^*} \to \mathbf{Z}$, yielding some history along α^* (let us note just that the set $\mathcal{P}t_{\alpha^*}$ is infinite).

Lemma 6.8 ensures us about the possibility to assign in a consistent way the integer values for all stable point classes along α^* and all points between them, i.e. for all $w \in \mathcal{P}t_{\alpha^*}$ such that

$$\langle\langle(m(A,C);0)\rangle\rangle = \langle\langle(m(A,C);t \times k)\rangle\rangle \leq_{\alpha^*} w \leq_{\alpha^*} \langle\langle(M(A,C);0)\rangle\rangle = \langle\langle(M(A,C);t \times k)\rangle\rangle,$$

it is enough to consider one solution Γ_0 of $\mathcal{N}'_\alpha$ and for every point $w = (a;t \times k + r)$ with $r < k$ define $\Gamma(w) = \Gamma_0(a;r)$.

In order to exhibit a complete solution Γ of $\mathcal{N}_{\alpha^*}$ it remains to define $\Gamma(w)$ for

$$w >_{\alpha^*} w_{max} = \langle\langle(M(A,C);0)\rangle\rangle_{\alpha^*} \text{ and } w <_{\alpha^*} w_{min} = \langle\langle(m(A,C);0)\rangle\rangle_{\alpha^*}.$$

Easy to see that these are independent tasks, due to the full symmetry let us consider only the case of the points greater than w_{max}.

Following the argument similar to that used in the proof of Lemma 6.8. if we could exhibit a sequence $w_1, w_2, \ldots$ of all points $w_i >_{\alpha^\bullet} w_{max}$ such that $w_j <_{\alpha^\bullet} w_i$ for no $i < j$ and $w_i =_{\alpha^\bullet} w_j$ implying $w_i =_{\alpha^\bullet} w_s$ for all s between i and j, we could step by step define the values $\Gamma(w)$ for all $w >_{\alpha^\bullet} w_{max}$ in a way $\Gamma(w_{i+1}) = \Gamma(w_i)$, if $w_{i+1} =_{\alpha^\bullet} w_i$, and $\Gamma(w_{i+1}) = \Gamma(w_i) + 1$ otherwise.

We exhibit the sequence $w_1, w_2, \ldots$ and define the values $\Gamma(w_1)$, $\Gamma(w_2), \ldots$ by scanning inductively all the path α^ω coordinates $0, 1, 2, \ldots$ and at each coordinate k define the points $w_i, w_{i+1}, \ldots, w_{i+s}$ (provided the points upto w_{i-1} were defined before the kth coordinate) to cover all the points $(a; t) >_{\alpha^\bullet} w_{max}$ with $t \leq k$ for which $\inf_{\alpha^\bullet}((a; t), k - t) = w_{max}$ and which are not defined to be w_j for $j < i$ (i.e. which have $\inf_{\alpha^\bullet}((a; t), k - 1 - t) >_{\alpha^\bullet} w_{max}$ (clearly, $t < k$)).

Since the inequality system $\mathcal{N}_\alpha$ is not contradictory, we can for every k order all the points $w_i, w_{i+1}, \ldots, w_{i+s}$ to be defined at the path α^ω coordinate k and define their values $\Gamma(w)$ in a consistent way with $\leq_{\alpha^\bullet}$. It follows from the definition of $\inf_{\alpha^\bullet}$ and the stability of w_{max} that also $w \leq_{\alpha^\bullet} w'$ is not possible for w defined at a coordinate later than w', hence the sequence w_i with $w_j <_{\alpha^\bullet} w_i$ for no $i < j$ and $w_i =_{\alpha^\bullet} w_j$ implying $w_i =_{\alpha^\bullet} w_s$ for all s between i and j can be (effectively) exhibited and the solution Γ, effectively defined for all w in this sequence.

The existence of the history ν along α^ω is finally demonstrated by observing that every point $w >_{\alpha^\bullet} w_{max}$ sometimes appears in the sequence $(w_i)_{i \in \mathbf{N}}$. Really, according to the definition of an accomplished loop and the transitivity of $\inf_{\alpha^\bullet}$ (see Proposition 5.31) we have for every $(a; t) >_{\alpha^\bullet} w_{max}$ that $\inf_{\alpha^\bullet}((a; t), r) = w_{max}$ at least when one full α - loop is contained between the coordinates t and $t + r$ in α^ω.

Observe that we have provided an algorithm defining a solution $\Gamma : \mathcal{P}t_{\alpha^\bullet} \to \mathbf{Z}$ of the path α^ω inequality system $\mathcal{N}_{\alpha^\bullet}$, so demonstrating also the effectivity of the generation of a history along α^ω, let us denote the history, generated by Γ, by ν_Γ.

If the path α^ω in $BG(P)$ is prefixed by some initial path γ, so obtaining an infinite path $\gamma + \alpha^\omega$, the existence of a history along $\gamma + \alpha^\omega$ is by Lemma 4.18 (a history ν^γ along γ is by Lemma 4.17 and Lemma 5.6).

For the algorithm, generating a history along $\gamma + \alpha^\omega$, observe that the uniform mappings ρ_i in the proof of Lemma 4.18 can be computed effectively from $\vec{v}_*(\nu^\gamma)$ and $\vec{v}_0(\nu_\Gamma)$ (see Fact 4.24 and Fact 4.25). □

6.4 Accomplished Loops: Decidability

Proof of **Lemma 6.7**: We call a path α in $BG(P)$ a $\langle n, C\rangle$-path, if α is starting from (n, C) (i.e., if $\langle n_0(\alpha), C_0(\alpha)\rangle = \langle n, C\rangle$).

Let for arbitrary finite $\langle n, C\rangle$-path α of the length $k = card(\mathcal{CR}d_\alpha - 1)$ for all $x_i \in Vars(P)$

- $\sup_\alpha.x_i = \{a \in \mathcal{A}_P \mid (a; k) \in \sup_\alpha((x_i; 0), k)\}$,

- $\inf_\alpha.x_i = \{a \in \mathcal{A}_P \mid (a;k) \in \inf_\alpha((x_i;0),k)\}$.

Let $\overrightarrow{\sup}_\alpha \stackrel{\text{def}}{=} \langle \sup_\alpha.x_1, \ldots, \sup_\alpha.x_m \rangle$ and $\overrightarrow{\inf}_\alpha \stackrel{\text{def}}{=} \langle \inf_\alpha.x_1, \ldots, \inf_\alpha.x_m \rangle$.

Proposition 6.11 *If two paths α and β end with the same vertex $\langle n', C' \rangle$ in $BG(P)$ and $\overrightarrow{\inf}_\alpha = \overrightarrow{\inf}_\beta$, as well as $\overrightarrow{\sup}_\alpha = \overrightarrow{\sup}_\beta$, then for every γ, starting from $\langle n', C' \rangle$,*

$$\overrightarrow{\inf}_{\alpha+\gamma} = \overrightarrow{\inf}_{\beta+\gamma}, \quad \textit{as well as} \quad \overrightarrow{\sup}_{\alpha+\gamma} = \overrightarrow{\sup}_{\beta+\gamma}.$$

Proof: Follows from the transitivity of bounds, see Proposition 5.31.□

Proposition 6.12 *A $\langle n, C \rangle$-path α in $BG(P)$ is an accomplished (n, C, A, k)-loop for some $k \in \mathbf{N}$ if and only if*

- *$a \in \sup_\alpha.a$, $a \in \inf_\alpha.a$ for all $a \in A \cap Vars(P)$,*
- *for every variable $x \in B_{A,C} \cup U_{A,C}$ the set $\inf_\alpha.x$ contains at least one $a \in A$,*
- *for every variable $x \in B_{A,C} \cup L_{A,C}$ the set $\sup_\alpha.x$ contains at least one $a \in A$.*

Proof: Follows from the definition of an accomplished loop.□

Fact 6.13 *The set of all possible distinct pairs $(\overrightarrow{\sup}_\alpha, \overrightarrow{\inf}_\alpha)$ for α being a $\langle n, C \rangle$-path in $BG(P)$ is finite.*

Proof: Follows from the finiteness of $\mathcal{A}_P$.□

Lemma 6.14 *The set of all accomplished (n, C, A)-loops in the graph $BG(P)$ for given $\langle n, C \rangle \in V(BG(P))$ and a constant holding set $A \subseteq \mathcal{A}_P$ is T-projective.*

Proof: Follows from Lemma 2.10 according to Proposition 6.11, Proposition 6.12 and Fact 6.13.□

The result of Lemma 6.7 now is a direct consequence from Lemma 6.14.□

So we have proved all the lemmas, needed for the proof of Theorem 6.2, so proving also the theorem itself, as well as Theorem 6.1.□□

Corollary 6.15 *The problem of whether a given LBASE program P has an infinite feasible path containing a given vertex n in its graph, is decidable.*

Proof: Test for the existence of an accomplished loop in the basic graph $BG(P)$ which is reachable (in the graph theoretic sense) from the $BG(P)$ initial vertex via some vertex $\langle n, C \rangle$ for some $C \in C_P$.□

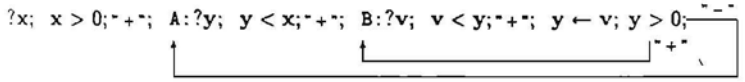
Corollary 6.16 *If a LBASE program P contains an infinite feasible path, then it contains also an infinite periodic feasible path (possibly with some pre-period).*

Proof: Follows from Lemma 6.5 and Lemma 6.6.□

For a real time system the existence of a periodic infinite feasible path is a rather natural feature saying that the system has a normal "execution cycle".

6.5 Feasible Fair Paths

As one can see from the following example, the set of all infinite feasible paths in given LBASE program may be not characterizable as the set of projections of all fair paths in some finite graph:



an infinite path in this program is feasible iff the sequence α of A and B labels in it belongs to the set $(AB^+)^\omega$ and $\exists k$ such that there are no more than k adjacent B's in α. So we have proved the following result.

Theorem 6.17 *The set of all feasible infinite paths in a given LBASE program P is not necessarily F-projective.*

At the same time it is possible to check, whether a given LBASE program has an infinite feasible path satisfying some fairness conditions.

Theorem 6.18 *There exists an algorithm which, given a LBASE program P with a fairness set $FS \subseteq V(P)$, decides whether there is an infinite feasible fair path in P (i.e. a path containing some vertex $n \in FS$ infinitely often).*

Proof: We define the fairness set $BFS \subseteq V(BG(P))$ for the vertexes of $BG(P)$ in a way $\langle n, C\rangle \in BFS$ iff $n \in FS$. An accomplished loop in $BG(P)$ is called F-accomplished iff it contains at least one vertex from BFS.

Lemma 6.19 *Every infinite feasible fair path in the basic graph $BG(P)$ of a LBASE program P contains at least one F-accomplished loop.*

Proof: Given an infinite feasible fair path α in $BG(P)$, consider the sequence of paths β_s defined in the proof of Lemma 6.5. Let $\beta_{s'}$ be that of these paths which is an accomplished loop. Clearly, the paths β_s are accomplished loops also for all $s \geq s'$. Since α is a fair path, the path β_s for some s contains a vertex $\langle n', C'\rangle \in BFS$, so β_s is F-accomplished.□

Now, in a full analogy with the proof of Theorem 6.2, we decide, whether there is a F-accomplished (n, C, A)-loop in $BG(P)$, which is reachable in the graph-theoretic sense from the $BG(P)$ initial vertex. If some such F-accomplished loop α is found, the path α^* consisting of infinitely many times repeated loop α with some necessary pre-period is both feasible (according to Lemma 6.6, taking into account Lemma 4.17 and Lemma 4.18) and fair (since there is at least one vertex from BFS in every instance of the loop α). If the answer is that there is no (reachable) F-accomplished

loops in $BG(P)$, we obtain according to Lemma 6.19 that there is no infinite feasible fair path in $BG(P)$.

As to the deciding of the existence of a F-accomplished (n, C, A)-loop for given $\langle n, C\rangle \in V(BG(P))$ and a constant holding set $A \subseteq \mathcal{A}_P$, notice that both the sets of accomplished (n, C, A)-loops and paths containing at least one vertex from BFS are T-projective (see Lemma 6.14 for the case of the accomplished loops). Lemma 2.8 guarantees the T-projectivity also of these set intersection - the set of all F-accomplished (n, C, A)-loops, so obtaining the decidability of the F-accomplished (n, C, A)-loop existence problem.□

Let us note that using the infinite feasible fair path existence decision algorithm, yielded by the proof of Theorem 6.18, we can decide for LBASE programs also the correct statement reachability problems, considered in Section 3.3.

Corollary 6.20 *The problem of ω-correct vertex reachability for LBASE programs is decidable.*

Corollary 6.21 *The problem of correct vertex reachability for LBASE programs is decidable.*

The corresponding results with some proof ideas demonstrated are given in Section 7.5 for the more general case of LTIBA programs, here we do not focus on the proof of the corollaries.

Chapter 7

Infinite Path Feasibility: LTIM and LTIBA

As observed already in Chapter 6, there can be infinite not feasible paths with all finite prefixes feasible also in LTIM programs. In this chapter we obtain the characteristics of the set of all infinite feasible paths in LTIM programs in the terms of F-projectivity, as well as discuss the decidable infinite path feasibility problems for the programs in the "united" language LTIBA.

Theorem 7.1 *The set of all infinite feasible paths for every LTIM program P is F-projective.*

Observe that we can using the already established results easily have a more general result for a sublanguage of LTIM:

Fact 7.2 *The set of all infinite feasible paths for every LBASQ program P is ω-projective.*

Proof: Follows from Fact 4.28 and the effectivity of the LBASE-like basic graph construction for LBASQ programs (see Section 5.1).□

For the characteristic of the set of all infinite feasible paths in LTIM_0 programs, see Corollary 7.12.

As in the case of LBASE and LBASQ programs, we study the infinite path feasibility in the basic graph of the given program P. We obtain the result of Theorem 7.1 from the following

Theorem 7.3 *There exists an algorithm which given a graph $BG(P)$ for a LTIM program P constructs an F-projectivee for the set of all infinite initial feasible paths in $BG(P)$.*

Moreover, if the graph $BG(P)$ is provided with a fairness set $FS \subseteq V(BG(P))$, an F-projectivee can be constructed for the set of all feasible fair paths in $BG(P)$.

Proof: In order to prove that some infinite path in a given basic graph $BG(P)$ is feasible, we follow the same general idea of exhibiting a sequence of "pointwise convergent" histories along the finite prefixes of the given path, which turned useful for LBASQ program basic graphs in the proof of Fact 4.28. However, since we do not have the analogue of Lemma 4.26 for LTIM programs, we are to define the objects, converging to the overall history in more tricky ways, which are applicable only to certain classes of $BG(P)$ paths (see the proofs of Lemma 7.7 and Lemma 7.11). We are giving also a symbolic characteristic of the set of all feasible paths in $BG(P)$ (Lemma 7.5) what allows to obtain the needed F-projectivity of the feasible path set (see Section 7.4).

First of all, if the program P does not have positive constants and, so, does not have variable activation operators (except the ones of the form $\mathrm{t} \leftarrow \mathrm{z}$), every infinite path in $BG(P)$ becomes obviously feasible (actually the LTIM programs without positive constants are even simpler than LBASQ programs), so the graph $BG(P)$ with the infinite path acceptance set taken to be $V(BG(P))$ serves as the desired F-projectivee (actually, even as an ω-projectivee).

7.1 Progressing and Conservative Paths

Let us call a LTIM program nontrivial, if it contains at least one positive constant $c > 0$ and consider in this section only the LTIM programs of this kind. We start to obtain some symbolic characteristic of every feasible initial infinite path in $BG(P)$.

Definition 7.4 *An infinite path α in $BG(P)$ for a LTIM program P is said to be:*

- *progressing, if for every natural k and every point $w \in \mathcal{P}t_\alpha(k)$ there exists $j \in \mathbf{N}$ such that $\inf_\alpha(w; j) \leq_\alpha (z + c_{max}; k + j)$;*
- *conservative, if there exists $j \in \mathbf{N}$ such that $\sup_\alpha((z+1; j), k)) >_\alpha (z; j + k)$ for every $k \in \mathbf{N}$ (i.e. the inequality $(z+1; j) \leq_\alpha (z; j + k)$ does not hold for any k).*

Lemma 7.5 *An infinite initial path in the feasibility graph $BG(P)$ for a nontrivial LTIM program P is feasible if and only if it is either progressing or conservative.*

Proof: We prove the following:

- every infinite initial feasible path in $BG(P)$ is either progressing or conservative,
- every initial conservative path in $BG(P)$ is feasible, and
- every initial progressing non-conservative path in $BG(P)$ is feasible.

Lemma 7.6 *Every feasible initial path in $BG(P)$ for a nontrivial LTIM program P is either progressing or conservative.*

Proof: Consider an infinite initial feasible path α in $BG(P)$ as well as an initial history ν of $BG(P)$ along α. There are two possibilities: either the sequence of values $(\vec{v}_i(\nu).z)_{i\in\mathbf{N}}$ is bounded by some constant M (i.e. $\vec{v}_i(\nu).z < M$ for all $i \in \mathbf{N}$), or for every $M \in \mathbf{Q}$ there exists $i \in \mathbf{N}$ such that $\vec{v}_i(\nu).z > M$.

In the first case there exists $i \in \mathbf{N}$ with $\vec{v}_j(\nu).z + 1 > \vec{v}_{j+k}(\nu).z$ for all $k \in \mathbf{N}$, so it is easy to conclude that the path α is conservative (were it not so, some inequality of the form $(z+1;k) \leq_\alpha (z;k+j)$ would be present in the α inequality system $\mathcal{N}_\alpha$, what implies that the mapping Γ_ν generating the history ν is not a solution of $\mathcal{N}_\alpha$, a contradiction.

As to the case of $\vec{v}_i(\nu).z > M$ for all M, it is easy to see in a similar way that the path α must be progressing.□

7.2 Feasibility of Conservative Paths

Lemma 7.7 *Every initial conservative path in $BG(P)$ for a nontrivial LTIM program P is feasible.*

Proof: A path α in $BG(P)$ is called *fully conservative*, if the inequality $(z+1;0) \leq_\alpha (z;k)$ does not hold for any k. We prove that every fully conservative path in $BG(P)$ is feasible, the general result of the feasibility of arbitrary conservative paths follows from Lemma 4.31 and the feasibility of all finite paths in $BG(P)$ (Lemma 4.29).

Let α be a fully conservative path in $BG(P)$. We obtain a solution (see Definition 5.9) $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Q}$ of its inequality system $\mathcal{N}_\alpha$ as the pointwise limit of mappings $\Gamma_i : \mathcal{P}t_{\alpha_i} \to \mathbf{Q}$ for α_i being the prefix (initial subpath) of α with the length i and Γ_i being the solution of $\mathcal{N}_\alpha$ w.r.t. the point set $\mathcal{P}t_{\alpha_i}$.

Let Γ_0 be some solution of the "path" $\alpha_0 = \langle n_0(\alpha), C_0(\alpha)\rangle$ inequality system satisfying the inequality

$$\Gamma_0(x;0) > \Gamma_0(z + c_{max};0) + 1$$

for every $x \in B_P(C_0(\alpha))$ with $x >_{C_0(\alpha)} z + c_{max}$ (easy to see that $x \in Vars(P)$ for every such x).

Let δ be the shortest distance between any two non-equal values $\Gamma_0(x;0)$, $\Gamma_0(y;0)$ for $x, y \in B_P(C_0(\alpha))$, we define $\delta_0 = \delta/3$ as well as for all i $\delta_{i+1} = \delta_i/2$.

Let us define for every $i \in \mathbf{N}$ the set $\mathcal{L}_i$ in a way

$$\mathcal{L}_i = \{(x;k) \in \mathcal{P}t_\alpha \mid k \leq i \;\&\; (x;k) \leq_\alpha (z + c_{max};i)\}.$$

The sequence of mappings $(\Gamma_i)_{i\in\mathbf{N}}$ is defined inductively in a way that for all i

- for $w \in \mathcal{L}_i$ $\Gamma_i(w) < \Gamma_0(z + c_{max}; 0) + 1$ and
- for $w \in \mathcal{P}t_{\alpha_i} \setminus \mathcal{L}_i$ $\Gamma_i(w) > \Gamma_0(z + c_{max}; 0) + 1$.

Assume that we have already defined a mapping Γ_i which is a solution of $\mathcal{N}_\alpha$ w.r.t. the point set $\mathcal{P}t_{\alpha_i}$ and satisfies these requirements, we are to show, how to define a corresponding mapping Γ_{i+1}.

We let ν_i be the history along the initial fragment α_i of the path α, generated by Γ_i. Since there exists a history along the one-edge path

$$\alpha' = \langle n_i(\alpha), C_i(\alpha)\rangle e_i(\alpha), \langle n_{i+1}(\alpha), C_{i+1}(\alpha)\rangle$$

then according to Lemma 4.31 there exists also a history ν_{i+1} along the path $\alpha_{i+1} = \alpha_i + \alpha'$, such that for all points $(x; k) \in \mathcal{P}t_{\alpha_i}$ $\vec{v}_k(\nu_{i+1}).x = \rho(\vec{v}_k(\nu_i).x)$ for some $\vec{v}_i(\nu_i).z + c_{max}$-stable mapping ρ. Let us denote the generator of the history ν_{i+1} by Γ^* ($\Gamma^* : \mathcal{P}t_{\alpha_{i+1}} \to \mathbf{Q}$ already is a solution of $\mathcal{N}_\alpha$ w.r.t. the point set $\mathcal{P}t_{\alpha_{i+1}}$, we will obtain the mapping Γ_{i+1} from the mapping Γ^* by changing some its values slightly, in order for Γ_{i+1} to meet the additional requirements, imposed on the mappings Γ_i).

For the sake of comprehensibility let us introduce the notation $\overline{z}_i$ for the points $(z + c_{max}; i)$ along the path α.

Due to the stability of the mapping ρ we have

- $\Gamma^*(w) = \Gamma_i(w)$ for all $w \in \mathcal{L}_i$ (observe that $w \leq_\alpha \overline{z}_i$ for all $w \in \mathcal{L}_i$) and
- $\Gamma^*(w) > \Gamma_i(\overline{z}_i)$ for all $w \in \mathcal{P}t_{\alpha_i} \setminus \mathcal{L}_i$ (observe the *strong* monotonicity of ρ).

Let us define $\Gamma_{i+1}(w) = \Gamma^*(w) = \Gamma_i(w)$ for all $w \in \mathcal{L}_i$.

Now we want to define the values $\Gamma_{i+1}(w)$ for $w \in \Delta = \mathcal{L}_{i+1} \setminus \mathcal{L}_i$ (the definition of $\Gamma_{i+1}(w)$ for $w \notin \mathcal{L}_{i+1}$ is done afterwards). The point set Δ consists, in general, of two disjoint point sets

$$\Delta^0 = \mathcal{L}_{i+1} \cap \mathcal{P}t_\alpha(i+1)$$

(the points at the new coordinate $i + 1$) and

$$Y = \Delta \cap \mathcal{P}t_{\alpha_i}$$

(the points $w \in \mathcal{P}t_{\alpha_i}$ for which $w \leq_\alpha \overline{z}_{i+1}$ but not $w \leq_\alpha \overline{z}_i$).

Let $U \subseteq \Delta^0$ be the least set of points $(x; i+1)$ with either $(x; i+1) =_\alpha (b; i)$ for some $(b; i) \in \mathcal{L}_i$, or $x = y + c$ for $c \in \mathbf{Z}$ and $(y; i+1) \in U$ (intuitively, the set U is the set of points w at the coordinate $i+1$ the values $\Gamma(w)$ of which are unequivocally determined for any solution Γ of the path α_{i+1} inequality system by the definition of the values $\Gamma(w') = \Gamma_{i+1}(w')$ for $w' \in \mathcal{L}_i$), let us call every point $w \in U$ *inherited*.

Fact 7.8 *For every $\Gamma^*(z; i+1)$-uniform mapping σ (see Definition 4.30) satisfying $\sigma(\Gamma^*(w)) = \Gamma^*(w)$ for all $w \in U$ and for $w = \overline{z}_i$ the mapping $\Gamma^+ : \mathcal{L}_{i+1} \to \mathbf{Q}$ defined*

- $\Gamma^+(w) = \Gamma^*(w)$ *for* $w \in \mathcal{L}_i$ *and*
- $\Gamma^+(w) = \sigma(\Gamma^*(w))$ *for* $w \in \Delta$,

is a solution of $\mathcal{N}_\alpha$ *w.r.t. the point set* $\mathcal{L}_{i+1}$.

Proof: Follows from the definition of the path inequality system solution (Definition 5.9) by observing that $\sigma(\Gamma^*(w)) > \Gamma^*(\overline{z}_i)$ for all $w \in Y$, observe also $\overline{z}_{i+1} \in U$.□

Let $\overline{U} = \Delta^0 \setminus U$ be the set of points, called *new* at the coordinate $i+1$. We define for every new point w its *basement* $bas(w)$ to be the largest (w.r.t. $\leq_\alpha$) point among the inherited ones and $\overline{z}_i$, which are less than w (clearly, all the points, considered here, are mutually comparable w.r.t. $\leq_\alpha$).

It can be obtained from the definition of the partitioning $\mathcal{C}_P$ for LTIM programs that one can define the $\Gamma^*(z; i+1)$-uniform mapping σ considered in Fact 7.8 in a way to meet $\Gamma^+(w) < \Gamma^+(bas(w)) + \delta_{i+1}$ for all $w \in \overline{U}$, let us define $\Gamma_{i+1}(w) = \Gamma^+(w)$ for $w \in \mathcal{L}_{i+1}$ for one such fixed mapping Γ^+.

Regarding the already defined values of Γ_{i+1} we can prove

Fact 7.9 $\Gamma_{i+1}(z + c_{max}; i+1) < \Gamma_0(z + c_{max}; 0) + 1$.

Proof: We prove that $\Gamma_{i+1}(z; i+1) < \Gamma_0(z+1; 0)$, what is equivalent to the statement of the fact.

Let for every $t = 0, 1, \ldots, i+1$ $\mathrm{binf}(t)$ be the maximal of points $w \in \mathcal{P}t_\alpha(t)$ with $w \leq_\alpha (z; i+1)$ ($\mathrm{binf}(t)$ can be looked at as a "backwards lower bound" of $(z; i+1)$ at the path α coordinate t), (i.e. we require in addition to $w \leq_\alpha (z; i+1)$ that for all $w' \in \mathcal{P}t_\alpha(t)$ whenever $w' \leq_\alpha (z; i+1)$, then also $w' \leq_\alpha w$ (we do not demand the syntactical uniqueness of $\mathrm{binf}(t)$, just take one point which meets the requirements of the definition)). Let

$$\mathrm{binf}(t) = (b_t; t) \in \mathcal{P}t_\alpha(t) \text{ for all } t.$$

Due to the non-decreasing nature of the program real time counter z for all t $z \leq_C b_t$ for $C = C_t(\alpha)$. Since α is fully conservative, we obtain also that for all t $b_t \leq_C z+1$, were it not so, we could obtain

$$(z; i+1) \geq_\alpha \mathrm{binf}(t) \geq_\alpha (z+1; t) \geq_\alpha (z+1; 0),$$

a contradiction with the full conservativity of α.

According to the definition of the mappings Γ_t (observe also $\Gamma_{j+k}(w) = \Gamma_j(w)$ for all $w \in \mathcal{L}_j$) we have that for all $t < i+1$

$$\Gamma_{t+1}(\mathrm{binf}(t+1)) - \Gamma_t(\mathrm{binf}(t)) < \delta_{t+1},$$

indeed, always $\mathrm{binf}(t) \in \mathcal{L}_t$. So we obtain

$$\Gamma_{i+1}(z;i+1) = \Gamma_{i+1}(\mathrm{binf}(i+1)) < \Gamma_0(\mathrm{binf}(0)) + (\delta_1 + \delta_2 + \cdots + \delta_{i+1}) < \\ < \Gamma_0(z+1;0) - \delta + \tfrac{1}{3}\delta < \Gamma_0(z;0) + 1,$$

as requested (observe that

$$\Gamma_0(\mathrm{binf}(0)) < \Gamma_0(z+1;0) - \delta \text{ due to } \mathrm{binf}(0) <_\alpha (z+1;0)$$

according to the definition of Γ_0 and δ).□

According to the definition of the set $\mathcal{L}_{i+1}$ we obtain from Fact 7.9 that $\Gamma_{i+1}(w) < \Gamma_0(\overline{z}_0)$ for all $w \in \mathcal{L}_{i+1}$.

For $(x;k) \notin \mathcal{L}_{i+1}$ we define the values $\Gamma_{i+1}(x;k)$ the following way:

- for $k \leq i$ always $\Gamma_{i+1}(x;k) = \Gamma_i(x;k)$, as well as
- $\Gamma_{i+1}(x;i+1) = \Gamma_i(b;i)$ whenever $(x;i+1) =_{\alpha_{i+1}} (b;i)$ for some b. Otherwise define $\Gamma_{i+1}(b;i+1) = \Omega(x)$ for some value

  $$\Omega(x) > \Gamma_0(z + c_{max};0) + 1$$

  (such values $\Omega(x)$ can always be chosen in a way Γ_{i+1} becomes a solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{P}t_{\alpha_{i+1}} \setminus \mathcal{L}_{i+1}$ since

  $$\Gamma_i(b;i) > \Gamma_0(z + c_{max};0) + 1 \text{ for all } (b;i) \in \mathcal{P}t_{\alpha_i} \setminus \mathcal{L}_{i+1}$$

  and for any two rationals values $x, y \in \mathbf{Q}$ such that $x < y$ there always exists $v \in \mathbf{Q}$ such that $x < v < y$).

Now it is easy to see also that Γ_{i+1} is the solution of $\mathcal{N}_\alpha$ w.r.t. the whole point set $\mathcal{P}t_{\alpha_{i+1}}$ since the definition of the set $\mathcal{L}_{i+1}$ does not admit any inequality $w \leq_\alpha w'$ to hold for $w' \in \mathcal{L}_{i+1}$ and $w \in \mathcal{P}t_{\alpha_{i+1}} \setminus \mathcal{L}_{i+1}$, observe also that the points $(b;i+1) \in \mathcal{P}t_{\alpha_{i+1}} \setminus \mathcal{L}_{i+1}$ are not involved in any relation $b = b' + c$ for $b' \in \mathcal{B}_P(C_{i+1}(\alpha))$.

In order to show that we obtain the solution of $\mathcal{N}_\alpha$ as a limit of the sequence $(\Gamma_i)_{i \in \mathbf{N}}$ it remains to show the convergence of $(\Gamma_i(w))_{i \in \mathbf{N}}$ for every $w \in \mathcal{P}t_\alpha$, what on its turn follows from the fact that whenever $\Gamma_i(w) \in \mathbf{Q}$ for some i, we have

- if $w \in \mathcal{L}_i$ then for every j both $w \in \mathcal{L}_{i+j}$ and $\Gamma_{i+j}(w) = \Gamma_i(w)$,
- else, i.e., if $w \in \mathcal{P}t_{\alpha_i} \setminus \mathcal{L}_i$ then either
  - for every j $w \notin \mathcal{L}_{i+j}$ and, so, $\Gamma_{i+j}(w) = \Gamma_i(w)$, or else
  - $w \in \mathcal{L}_{i+j}$ for some j, what is the case considered above.

So we have completed the proof of Lemma 7.7.□

Lemma 7.10 *There exists an algorithm which for every cyclic conservative path $\alpha' = \gamma + \beta^*$ in $BG(P)$ given its pre-period γ and period β computes some history along α'.*

Proof: For $\alpha' = \gamma + \beta^*$ being an infinite conservative path in $BG(P)$ consisting of the pre-period γ and infinitely many times repeated loop (path) β we have according to the α' conservativity that there exists a coordinate k starting from which the tail of α' is fully conservative. Due to the cyclic nature of β^* we conclude that fully conservative is already the path $\alpha = \beta^*$.

Let us show an algorithm which computes a history ν along the fully conservative cyclic path α, the computability of some history along the overall initial path α' will then follow according to Lemma 4.31 and Lemma 5.11.

For α being any fully conservative path in $BG(P)$ the construction of some appropriate mapping Γ_{i+1} from Γ_i in the proof of Lemma 7.7 can be easily done effectively. The only problem with computing a history along any fully conservative path in $BG(P)$ is the uncertainty for a given point $w \in \mathcal{P}t_{\alpha_i} \setminus \mathcal{L}_i$ whether or not $w \in \mathcal{L}_k$ for some $k \in \mathbf{N}$. This uncertainty can be resolved positively, in principle, at *arbitrarily large* ks. We show how to overcome this problem for the cyclic fully conservative path α in $BG(P)$.

Let $\langle n, C\rangle = \langle n_0(\beta), C_0(\beta)\rangle = \langle n_*(\beta), C_*(\beta)\rangle$. We denote by k the length of the loop β.

Let for $x \in BP(C)$ with $x >_C z + c_{max}$ (equivalently, $(x;kq) \notin \mathcal{L}_{kq}$ for all $q \in \mathbf{N}$; clearly, we have $x \in Vars(P)$) define $b(x) \in Vars(P) \cup \{+\infty\}$ to meet

- $(b(x);k) =_\alpha \sup_\alpha((x;0),k)$, if $\sup_\alpha((x;0),k) >_\alpha (z + c_{max};k)$, and
- $b(x) = z$, if $\sup_\alpha((x;0),k) \leq_\alpha (z + c_{max};k)$.

According to the definition of the sets $\mathcal{L}_i$ we easily obtain that for $q \in \mathbf{N}$ $(x;kq) \notin \mathcal{L}_i$ for all i if and only if for $b^j(x) = z$ for no $j \in \mathbf{N}$, what can be easily tested due to the monotonicity of b and the finiteness of the set of considered variables x. The monotonicity and transitivity of the point bounds (Fact 5.27 and Proposition 5.31) allow to determine whether $w \in \mathcal{L}_i$ for some i also for other points along α.□

7.3 Feasibility of Progressing Paths

Lemma 7.11 *Every initial progressing non-conservative path α in $BG(P)$ for a non-trivial LTIM program P is feasible. Moreover, there exists an algorithm which computes a history along every such α.*

Proof: Let α be an initial progressing non-conservative path in $BG(P)$. Let $\alpha_0, \alpha_1, \ldots$ be the sequence of all finite prefixes (initial subpaths) of α.

In order to prove the feasibility of α we exhibit a solution (see Definition 5.9) $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Q}$ of the path α inequality system $\mathcal{N}_\alpha$. We define for every $k \in \mathbf{N}$ the sets

$$\mathcal{L}_k = \{(x;i) \in \mathcal{P}t_\alpha \mid i \leq k \ \& \ \inf_\alpha((x;i), k-i) <_\alpha (z + c_{max}; k)\} \text{ and}$$

$$\mathcal{E}_k = \{(x;i) \in \mathcal{P}t_\alpha \mid i \leq k \ \& \ (x;i) =_\alpha (z + c_{max}; k)\},$$

let $\forall k \ \mathcal{L}'_k = \mathcal{L}_k \cup \mathcal{E}_k$.

Since α is progressing, for every natural k and every point $w \in \mathcal{P}t_\alpha(k)$ there exists $j \in \mathbf{N}$ with

$$\inf_\alpha(w, j) \leq_\alpha (z + c_{max}; k + j)$$

. Due to the non-conservativity of α

$$(z + c_{max}; k + j) <_\alpha (z + c_{max}; k + j')$$

for some $j' > j$, so, according to the definition of $\inf_\alpha$, for every natural k and every point $w \in \mathcal{P}t_\alpha(k)$ one can find $j' \in \mathbf{N}$ with

$$\inf_\alpha(w, j') <_\alpha (z + c_{max}; k + j'), \text{ so}$$

$\mathcal{P}t_\alpha = \bigcup_{k \in \mathbf{N}} \mathcal{L}_k$. Clearly, also $\mathcal{P}t_\alpha = \bigcup_{k \in \mathbf{N}} \mathcal{L}'_k$, as well as $\mathcal{L}'_k \subseteq \mathcal{L}'_{k+1}$ for all $k \in \mathbf{N}$.

We obtain the solution Γ as a pointwise limit of the sequence of mappings $(\Gamma_k)_{k \in \mathbf{N}}$ such that for all $k \ \ \Gamma_k : \mathcal{L}'_k \to \mathbf{Q}$, each Γ_k being a solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{L}'_k$ (see Definition 5.9). Easy to see that, if we could exhibit such a convergent sequence of mappings Γ_k, the limit mapping $\Gamma : \mathcal{P}t_\alpha \to \mathbf{Q}$ would be a solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{P}t_\alpha$.

Now let us define the mappings Γ_k satisfying for every k the additional requirement that for all $w \in \mathcal{L}'_k \ \ \Gamma_k(w) \leq \Gamma_k(\overline{z}_k)$ (recall that we use the shorthand $\overline{z}_i$ to denote the point $(z + c_{max}; i) \in \mathcal{P}t_\alpha$).

First of all for every variable $x \in Vars(P)$ and constant $c \in Cons(P)$ let $\Gamma_0(x + c; 0) = c \in \mathbf{N}$, as well as $\Gamma_0(x; 0) = 0$ and $\Gamma_0(c; 0) = c$.

Assume that we have defined an appropriate mapping $\Gamma_i : \mathcal{L}'_i \to \mathbf{Q}$ - a solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{L}'_i$; we show, how to define $\Gamma_{i+1} : \mathcal{L}'_{i+1} \to \mathbf{Q}$ - a corresponding solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{L}'_{i+1}$.

First of all let $\Gamma^0_i : \mathcal{P}t_{\alpha_i} \to \mathbf{Q}$ be a solution of the path α inequality system $\mathcal{N}_\alpha$ w.r.t. the point set $\mathcal{P}t_{\alpha_i}$, which is consistent with Γ_i for $w \in \mathcal{L}'_i$ (clearly the appropriate values $\Gamma^0_i(w)$ for $w >_\alpha \overline{z}_i$ can be defined).

According to Lemma 4.31 and the relations between the program histories and their generator mappings we obtain a mapping $\Gamma^* : \mathcal{P}t_{\alpha_{i+1}} \to \mathbf{Q}$ which is a solution of $\mathcal{N}_\alpha$ w.r.t. the point set $\mathcal{P}t_{\alpha_{i+1}}$ and meets $\Gamma^*(w) = \Gamma^0_i(w) = \Gamma_i(w)$ for all $w \in \mathcal{L}'_i$ (it is of importance that for all $w \in \mathcal{L}'_i$ we have $\Gamma^0_i(w) = \Gamma_i(w) \leq \Gamma_i(\overline{z}_i)$).

The mapping Γ^* can be "almost" taken as the seeked mapping Γ_{i+1} with the only possible exceptions that $\Gamma^*(w) > \Gamma^*(\overline{z}_{i+1})$ for some $w \in \mathcal{L}'_{i+1}$. Since $\Gamma^*(w)$ is a solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{P}t_{\alpha_{i+1}}$, we easily conclude that in this case $w \in \Delta$ for

$$\Delta \subseteq \mathcal{P}t_{\alpha_i} \cap (\mathcal{L}_{i+1} \setminus \mathcal{L}'_i)$$

being the set of points w which are $\leq_\alpha$-incomparable with $\overline{z}_{i+1}$.

For $w \in \mathcal{L}'_{i+1} \setminus \Delta$ we define $\Gamma_{i+1}(w) = \Gamma^*(w)$. As to $w \in \Delta$, observe first that the set Δ can be non-empty only if $\overline{z}_{i+1} >_\alpha \overline{z}_i$ (equivalently, if $(z; i+1) >_\alpha (z; i)$). In this case the corrections made to the values $\Gamma^*(w)$ before taking them as the corresponding values $\Gamma_{i+1}(w)$ are the following.

Let w_0 be the largest point (w.r.t. $\leq_\alpha$) from the set

$$\{w \in \mathcal{P}t_{\alpha_{i+1}} \mid w < \overline{z}_{i+1}\} \cup \{\overline{z}_i\}$$

(clearly, all points from this set are $\leq_\alpha$-comparable). We define a strongly monotone mapping $\pi : \mathbf{Q} \to \mathbf{Q}$ satisfying $\pi(x) \in]\Gamma^*(w_0), \Gamma^*(\overline{z}_{i+1})[$ for all $x \in \mathbf{Q}$ and let for $w \in \Delta$

$$\Gamma_{i+1}(w) = \pi(\Gamma^*(w)).$$

It is easy to see that any mapping $\Gamma_{i+1} : \mathcal{L}'_{i+1} \to \mathbf{Q}$ defined the above described way both is a solution of $\mathcal{N}_\alpha$ w.r.t. $\mathcal{L}'_{i+1}$ and meets $\Gamma_{i+1}(w) \leq \Gamma_{i+1}(\overline{z}_{i+1})$ for all w.

So we have completed the inductive step of the construction of an appropriate mapping Γ_{i+1} from Γ_i. Clearly, the construction of some mapping Γ_{i+1} can be done effectively from the given mapping Γ_i. So we have completed both the proofs of Lemma 7.11 and Lemma 7.5. □□

We can apply the obtained result of Lemma 7.5 to the analysis of the programs in the sublanguage LTIM_0 of LTIM by observing that every path in the basic graph $BG(P)$ for a LTIM_0 program P is either conservative (in this case the value of z is kept constant forever along the path starting from some path coordinate), or progressing (x is the only variable for which the inequality $(x; k) >_\alpha (z + c_{max}; k)$ can hold at some path coordinate k and whenever the positive assignment operator is executed at the coordinate $k+l$, the lower bound of $(x; k)$ becomes obviously equal with $(z + c_{max}; l)$). So we have obtained the following result.

Corollary 7.12 *Every infinite path in the graph $BG(P)$ for a LTIM_0 program P is feasible (the set of infinite feasible paths for a LTIM_0 program P is ω-projective).*

Let us just note that actually the theory of progressing and conservative paths was not fully needed in order to obtain this corollary, it was just an easy way to state it in the presence of the general theory, developed mostly for other purposes.

7.4 F-projectivity of Path Sets

Lemma 7.13 *There exists an algorithm which, given a graph $BG(P)$ for a nontrivial LTIM program P constructs an F-projectivee for all initial conservative paths in $BG(P)$.*

Proof: First of all we show, how to build F-projectivees for all fully conservative paths starting from one fixed vertex $\langle n, C\rangle \in V(BG(P))$ (i.e. fully conservative $\langle n, C\rangle$-paths in $BG(P)$), the construction of the F-projectivee required by the lemma will be given afterwards.

Let us associate with every finite $\langle n, C\rangle$-path α in $BG(P)$ a characteristic $S(\alpha) = \sup_\alpha((z+1;0), k)$ for k being the length of α. It is easy to see according to the definition and properties of the upper point bounds (see Section 5.3) that whenever two $\langle n, C\rangle$-paths α and β end with the same vertex $\langle n', C'\rangle \in V(BG(P))$ and have $S(\alpha) = S(\beta)$, then for every $\langle n', C'\rangle$-path γ in $BG(P)$ $S(\alpha+\gamma) = S(\beta+\gamma)$. Clearly, also the set of all possible $S(\alpha)$ values for various α is finite (always

$$S(\alpha) \in \mathcal{B}_P = \{b \in \mathcal{B}_P(C) \mid C \in \mathcal{C}_P\}).$$

So, we can construct in a similar way, as in the proof of Lemma 2.10, a finite labelled graph $H^{\langle n,C\rangle}$ with the vertexes $(\langle n', C'\rangle, b\rangle$ for $\langle n', C'\rangle \in V(BG(P))$ and $b \in \mathcal{B}_P(C')$ such that for every $(\langle n, C\rangle, z+1)$-path α', ending with $(\langle n', C'\rangle, b\rangle$ in $H^{\langle n,C\rangle}$ the projection $\alpha = proj(\alpha')$ is a $\langle n, C\rangle$-path in $BG(P)$, ending with $\langle n', C'\rangle$, and having the characteristic $S(\alpha) = b$.

We associate with the graph $H^{\langle n,C\rangle}$ the infinite acceptance (fairness) set

$$S_I(H^{\langle n,C\rangle}) = \{(\langle n', C'\rangle, b\rangle \mid b >_{C'} z\}.$$

Let us also delete the "garbage" consisting of the vertexes $(\langle n', C'\rangle, b\rangle$ with $b \leq_{C'} z$ together with all edges incoming in and outgoing from these vertexes. It is easy to see that the obtained graph $H^{\langle n,C\rangle}$ serves as the desired F-projectivee for all fully conservative $\langle n, C\rangle$-paths in $BG(P)$.

In order to have an F-projectivee for all initial conservative paths in $BG(P)$ let us take first the graph $BG(P)$ itself and label every vertex and edge in it by *itself*). For every edge e in the considered graph, leading from $\langle n, C\rangle$ to $\langle n', C'\rangle$, let us introduce a new edge from $\langle n, C\rangle$ to the initial vertex of $H^{\langle n',C'\rangle}$, labelled also by e. Let the initial vertex of the newly obtained graph H be that of $BG(P)$ and the infinite acceptance set be the union of all infinite acceptance sets for all graphs $H^{\langle n,C\rangle}$ for $\langle n, C\rangle \in V(BG(P))$. Since an initial path in $BG(P)$ is conservative if and only if some its postfix (a terminal subpath) is fully conservative, the graph H can be taken as the requested F-projectivee for all initial conservative paths in $BG(P)$. Observe also that all constructions made in order to obtain the graph H from $BG(P)$ are effective.□

Lemma 7.14 *There exists an algorithm which, given a graph $BG(P)$ for a nontrivial LTIM program P constructs an F-projectivee for all initial progressing paths in $BG(P)$.*

Proof: Let us show how to build the F-projectivees for the sets of all progressing $\langle n, C\rangle$-paths which contain the vertex $\langle n, C\rangle$ *infinitely often*. The F-projectivees for initial paths in $BG(P)$, containing some specific vertex $\langle n, C\rangle$ infinitely often are easily obtained due to the fact that the addition of an initial fragment to the path in $BG(P)$ does not change the path property of being or being not progressing (observe Fact 5.27 and Proposition 5.31). An F-projectivee of the set of all initial progressing paths in $BG(P)$ can be built according to Lemma 2.9 (observe the finiteness of $V(BG(P))$, every infinite progressing path in $BG(P)$ contains at least one graph vertex infinitely often).

Assume that we have the vertex $\langle n, C\rangle$ fixed and consider the $\langle n, C\rangle$-paths in $BG(P)$. Let $m \in \mathcal{B}_P(C)$ be the *maximal* base point of $\mathcal{B}_P(C)$, the only nontrivial case to consider is $m >_C z + c_{max}$ (otherwise the construction of the needed F-projectivee is trivial).

Let for every finite $\langle n, C\rangle$-path α in $BG(P)$ the characteristic $S(\alpha) = \inf_\alpha(\langle m; 0\rangle, k)$ for k being the length of α. It is easy to see according to the definition and properties of the lower point bounds (see Section 5.3) that whenever two $\langle n, C\rangle$-paths α and β end with the same vertex $\langle n', C'\rangle \in V(BG(P))$ and have $S(\alpha) = S(\beta)$, then for every $\langle n', C'\rangle$-path γ in $BG(P)$ $S(\alpha + \gamma) = S(\beta + \gamma)$. Clearly, also the set of all possible $S(\alpha)$ values for various α is finite. So, we can construct in a similar way, as in the proof of Lemma 2.10, a finite labelled graph H with the vertexes $\langle\langle n', C'\rangle, b\rangle$ for $\langle n', C'\rangle \in V(BG(P)$ and $b \in \mathcal{B}_P(C')$, such that for every $\langle\langle n, C\rangle, m\rangle$-path α', ending with $\langle\langle n', C'\rangle, b\rangle$ in H the projection $\alpha = proj(\alpha')$ is a $\langle n, C\rangle$-path in $BG(P)$, ending with $\langle n', C'\rangle$ and having the characteristic $S(\alpha) = b$.

We introduce in the graph H one more vertex $\langle\langle n, C\rangle, *\rangle$, labelled by $\langle n, C\rangle$ and having the following incoming and outgoing edges:

- an incoming edge, labelled by $e \in E(BG(P))$ from $x \in V(H)$ whenever an edge labelled by e is drawn in H from x to $\langle\langle n, C\rangle, b\rangle$ for some $b \leq_C z + c_{max}$;
- outgoing edges with the same labels and targets, as the edges leading from $\langle\langle n, C\rangle, m\rangle$.

Let the graph H initial vertex be $\langle\langle n, C\rangle, m\rangle$ and the infinite acceptance set $S_I(H) = \{\langle\langle n, C\rangle, *\rangle\}$.

In order to prove that H is an F-projectivee for the set of all progressing $\langle n, C\rangle$-paths which contain the vertex $\langle n, C\rangle$ infinitely often, let us observe first that due to Fact 5.27 and Proposition 5.31 an $\langle n, C\rangle$-path, containing the vertex $\langle n, C\rangle$ infinitely often is progressing if and only if there exists a sequence of the path α vertexes $(\langle n_{i_j}(\alpha), C_{i_j}(\alpha)\rangle)_{j\in\mathbf{N}}$ such that

- for all j $\langle n_{i_j}(\alpha), C_{i_j}(\alpha)\rangle = \langle n, C\rangle$ and

- $S(\alpha^j) \leq_C z + c_{max}$ for every path α^j defined as the fragment of α from $\langle n_{i_j}(\alpha), C_{i_j}(\alpha)\rangle$ to $\langle n_{i_{j+1}}(\alpha), C_{i_{j+1}}(\alpha)\rangle$

(since the vertex $\langle n, C\rangle$ is contained in α infinitely often we can wait for $\langle n_{i_j}(\alpha), C_{i_j}(\alpha)\rangle$ insertion in the defined sequence of path α vertexes until $S(\alpha^{j-1}) \leq_C z + c_{max}$, it is due to the progressivity of α that such a moment will be *eventually* reached at some α coordinate).

Now it is easy to see that both

- for the projection $\alpha = proj(\alpha')$ of any accepting path α' in H the abovedefined sequence of the path α coordinates i_j can be chosen (let us take the sequence of indices i_j for which the i_jth vertex of α' is $\langle\langle n, C\rangle, *\rangle$), and

- for every path α with the abovedefined sequence of coordinates i_j there exists an initial path α' in H such that both $\alpha = proj(\alpha')$ and for every i_j the i_jth vertex of α' is $\langle\langle n, C\rangle, *\rangle$.

So we conclude the proof of Lemma 7.14.□

The proof of Theorem 7.3, and so also the proof of Theorem 7.1 now is obtained from Lemma 7.5, Lemma 7.13 and Lemma 7.14 using also Lemma 2.9 for building the needed F-projectivees for the sets of paths being either conservative or progressing in the graphs $BG(P)$. Lemma 2.9 is used also to build the F-projectivee for the set of all feasible *fair* paths in $BG(P)$.□□

There is one more interesting path set in every LTIM program (the graph $BG(P)$ for a program P), namely the set of all paths feasible with no bound on the limit value of the real time counter z, let us call these paths *non-Zeno feasible* both in a program P and its graph $BG(P)$. It turns out that the set of all non-Zeno feasible paths in every graph $BG(P)$ for a LTIM program P (and, so, in the program P itself) is also F-projective. We do not consider the detailed proof, just note the following results:

Lemma 7.15 *An infinite path α in $BG(P)$ for a LTIM program P is non-Zeno feasible if and only if for every $k \in \mathbf{N}$ the inequality $(z; k + l) \leq_\alpha (z + 1; k)$ does not hold for some l (i.e. $\sup_\alpha(z; k + l) >_\alpha (z + 1; k)$ for some l).*

Proof idea: Clearly, for every non-Zeno feasible path for every k the corresponding l can be found.

In order to prove the reverse, show that for every history ν_k along the prefix α_k of α there exists a continuation ν_{k+l} of ν_k (perhaps with some values $\vec{v}_s(\nu_k).x > \vec{v}_k(\nu_k).z + c_{max}$ modified) along α_{k+l} such that $\vec{v}_{k+l}(\nu_{k+l}).z > \vec{v}_k(\nu_k).z + 1$.□

Lemma 7.16 *There exists an algorithm which, given a graph $BG(P)$ for a LTIM program P constructs an F-projectivee for all initial paths α in $BG(P)$ such that for all k $\sup_\alpha(z; k + l) >_\alpha (z + 1; k)$ for some l.*

Proof: Similar to the proof of Lemma 7.14.□

7.5 Infinite Path Feasibility in LTIBA

In this section we sum up the results obtained both for LBASE and LTIM programs in order to characterize the possibilities to analyze the infinite path feasibility in the programs in the language LTIBA containing the means for the description of both data and time dependencies.

Theorem 7.17 *There is an algorithm which, given a LTIBA program P, decides, whether it has an infinite feasible path. There is an algorithm which, given a LTIBA program P with an infinite acceptance set, decides, whether it has an infinite feasible fair path.*

If the program P contains an infinite feasible (fair) path, the deciding algorithm can be asked to compute a history along one such path.

Proof: Consider the LTIM program P^T which is obtained from P by replacing all LBASE operators of P by the dummy LTIM operator NOP and the LBASE program P^B which is obtained from P by replacing all LTIM operators of P by the LBASE operator NOP (see Section 3.2). Consider the basic graph $BG(P^T)$ of the program P^T. According to Theorem 7.3, let us build an F-projectivee $H(P)$ for the set of all infinite initial feasible (fair) paths in $BG(P^T)$.

Every vertex $\tilde{n} \in V(H(P))$ is labelled by some program P vertex $n = l_V(\tilde{n}) \in V(P)$ (actually, is labelled by some $BG(P^T)$ vertex, which is on its turn labelled by $n \in V(P)$), as well as every edge $\tilde{e} \in E(H(P))$ has a label $l_E(\tilde{e}) \in E(P)$.

We construct a LBASE program H^B from $H(P)$ by replacing every $H(P)$ vertex label $n \in V(P)$ by its P^B operator $p(n)$ (if the vertex n has a LTIM operator p associated with it in P then $p(n)$ is the dummy LBASE operator NOP) and every $H(P)$ edge label $e \in E(P)$ by the edge e label $l_E(P) \in \{"+", "-"\}$ in P.

According to Theorem 6.18 one can decide whether the program H^B has an infinite feasible fair path (the fairness set for the program H^B is inherited from the graph $H(P)$).

If some infinite fair path α in H^B is found, we can take, according to Lemma 6.5 and Lemma 6.6 α to be a path consisting of infinitely many times repeated accomplished (F-accomplished) loop. According to Lemma 6.6 we can generate a history ν^B along the H^B path α.

Consider the path α in $H(P)$ (the vertexes and edges of $H(P)$, as well as the vertex and edge incidence are in $H(P)$ *the same*, as in H^B), let α' be the projection of α in $BG(P^T)$. Since α is cyclic in $H(P)$, so is α' in $BG(P^T)$. Since $H(P)$ is an F-projectivee of the set of all initial infinite feasible (fair) paths in $BG(P^T)$, we have that the path α' is feasible (and fair). According to Lemma 7.5, the path α' is either progressing, or conservative. Since α' is cyclic, it is easily decidable, whether it is conservative, or not, so in each case one can generate a history ν^T along α' (see Lemma 7.10 and Lemma 7.11).

Let α'' be the projection of the $BG(P^T)$ path α' in P^T (and, so, in P). Clearly, we have that ν^T is a history along α'' in P^T. It is also straightforward from the construction of H^B and the construction of the path α'' that the defined H^B history ν^B is also a history along α'' in the LBASE program P^B, so we have demonstrated the existence of an infinite feasible (fair, according to the construction) path α'' in P together with an algorithm generating a history along α''.

On the other hand, if there is an infinite feasible (fair) path α in P, it is a projection of some initial fair path α' in $BG(P^T)$, what on its turn is a projection of some initial fair path α'' in $H(P)$. Observe that α'' is also an initial fair path in H^B, as well as that the H^B path α'' vertex and edge label sequence (which is the only information which determines the feasibility of a path) coincides with that of the P^B path α. Since α is feasible in P^B, also α'' is feasible in H^B; so we conclude that α'' is an initial feasible fair path in H^B.□

Corollary 7.18 *There exists an algorithm which given a vertex $n \in V(P)$ for a LTIBA program P decides whether there exists an infinite feasible path in P containing the vertex n infinitely often.*

Proof: Follows from Theorem 7.17 by defining the fairness set for the program P to be $\{n\}$.□

As to the possibilities to characterize the set of all infinite feasible paths in LTIBA programs, we can note the following

Corollary 7.19 *The set of all feasible infinite paths in a given LTIBA program P is not necessarily F-projective.*

Proof: Follows from Theorem 6.17 since every LBASE program is also a LTIBA program.□

However, as in the case for both LBASE and LTIM programs, we have

Corollary 7.20 *If a LTIBA program P contains an infinite feasible path, then it contains also an infinite periodic feasible path (possibly with some pre-period).*

Proof: Lemma 6.5 and Lemma 6.6 allow to find a periodic feasible path in the program H^B from the proof of Theorem 7.17.□

Let us note that using the infinite feasible fair path existence decision algorithm, yielded by the proof of Theorem 7.17, we can decide for LTIBA programs also the correct statement reachability problems, considered in Section 3.3.

Corollary 7.21 *The problem of ω-correct vertex reachability for LTIBA programs is decidable. Moreover, if some vertex n in a program P is ω-correctly reachable, there exists an algorithm generating a history of P along an infinite accepting path α containing the vertex n.*

Proof: Construct for the given program P and vertex $n \in V(P)$ an F-projectivee $F(P)$ of the set of all fair paths in P containing n (at this point we have no claim about the *feasibility* of the paths, clearly, such a projectivee $F(P)$ can be constructed). We define a program P' as the one obtained from $F(P)$ by replacing every $F(P)$ vertex label $n = l_{V(F(P))}(\tilde{n}) \in V(P)$ by the P operator $p(n)$ and every edge label $e \in E(P)$ by $l_{E(P)}(e) \in \{"+", "-"\}$. Clearly, there exists an infinite feasible fair path, containing n, in P if and only if there exists an infinite feasible fair path in P', what is decidable according to Theorem 7.17. Moreover, every history along an infinite feasible fair path α in P', generated by the algorithm of Theorem 7.17, is also a history along the projection α' of α in P, what is a fair path in P and contains the given vertex n.□

Corollary 7.22 *The problem of whether a given LTIBA program P has an infinite feasible path containing a given vertex n in its graph, is decidable.*

Proof: Follows from Corollary 7.21.□

Corollary 7.23 *The problem of correct vertex reachability for LTIBA programs is decidable. Moreover, if some vertex n in a program P is correctly reachable, there exists an algorithm generating a history of P along an accepting path α containing the vertex n.*

Proof: Follows from Corollary 5.14, Corollary 5.16 and Corollary 7.21.□

Chapter 8

Programs With Integer Counters

The main aim of this chapter is to prove the undecidability of the vertex reachability problem for programs in the language LTIM', what is the variant of LTIM over integer-valued variables. We give also some background illustrating the decidability and undecidability results for various program classes with integer counters known so far. An interesting result, yielding a subclass of LTIM' programs with decidable vertex reachability problem, is considered in Section 8.3.

8.1 Undecidability of Reachability for LTIM'

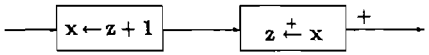
We define the language LTIM' the following way.

Every program in LTIM' is assumed to have a finite number of *integer-valued* ordinary variables $x_1, x_2, \ldots, x_m$ and a *counter* z with the following operators permitted:

- all LBASE operators (input, assignment, comparison) over the ordinary variables x_i;
- $\mathbf{z} < \mathbf{x}$ and $\mathbf{x} < \mathbf{z}$ - the LBASE comparison between z and ordinary variables;
- a counter increasement $\mathbf{z} \leftarrow \mathbf{z} + 1$, increasing the value of the counter z by 1.

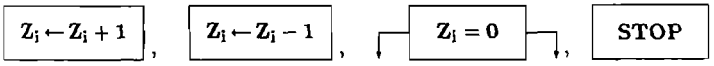
The program begins its execution from its initial vertex and all variable values set to 0. The formal semantics of LTIM' programs is given in a similar way, as for LBASE and LTIM programs in Chapter 3.

We have included in the system of operators of LTIM' only those which are needed in order to prove the undecidability of the vertex reachability problem. As to the counter increasement operator $\mathbf{z} \leftarrow \mathbf{z} + 1$, it can be easily modelled by operators which syntactically more resemble LTIM operators (variable activation and positive assignment) as a block:



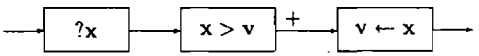
Theorem 8.1 *The statement (vertex) reachability problem for LTIM' programs is undecidable.*

Proof: The proof follows from the well-known result (see [Min67]) that for every recursive function φ_i there exists a so-called Minsky machine which computes the function φ_i. Every Minsky machine is a program which uses only two counters Z_1 and Z_2 in the following allowed operations:



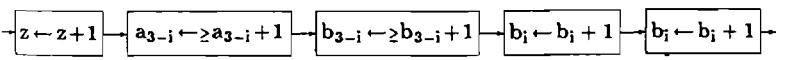
Without the loss of generality we can consider only those Minsky machines which have 0 as the initial value for both its counters Z_1 and Z_2. We demonstrate the undecidability of the vertex reachability problem for LTIM' programs by showing, how to construct for every considered Minsky machine M a corresponding LTIM' program $P(M)$ which has a special end vertex $@ \in V(P(M))$ reachable if and only if M stops its execution with the final counter values $Z_1^* = Z_2^* = 0$.

For every Minsky machine M the corresponding LTIM' program $P(M)$ will have 5 ordinary internal variables a_1, a_2, b_1, b_2 and x, as well as the counter z. For $v \in \{a_1, a_2, b_1, b_2\}$ we define the macro v ← ≥v+1 to stand for the following sequence of operators:

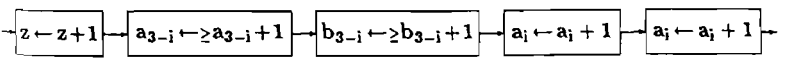


First of all we show, how to construct for every Minsky machine's instruction its modelling block of LTIM' operators:

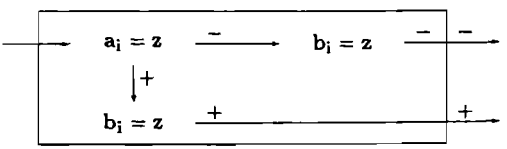
1. $Z_i \leftarrow Z_i + 1$, the counter increasment $(i = 1, 2)$:



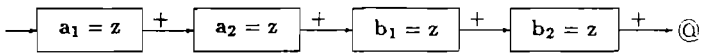
2. $Z_i \leftarrow Z_i - 1$, the counter decreasment $(i = 1, 2)$:



3. $Z_i = 0$, the test for 0 $(i = 1, 2)$:



4. **STOP**, the halting operator:



The program $P(M)$ is constructed from the given Minsky machine M by replacing every M instruction by the its modelling block. Let $V(M)$ be the set of the machine M vertexes, we define a *configuration* of M as a triple (n, p_1, p_2) for $n \in V(M)$ and $p_1, p_2 \in \mathbf{N}$ being the values of the counters Z_1 and Z_2, respectively. We say that a program $P(M)$ state $(n', \vec{v})$ for $n \in V(P(M))$ and $\vec{v} \in \mathbf{N} \times \mathbf{Z}^5$ *models* the given configuration (n, p_1, p_2) of the machine M, if $n' = n \in V(M)$ and

$$\vec{v}.b_1 - \vec{v}.z = \vec{v}.z - \vec{v}.a_1 = p_1; \;\; \vec{v}.b_2 - \vec{v}.z = \vec{v}.z - \vec{v}.a_2 = p_2.$$

If the machine M stops with the values of counters Z_1, Z_2 being $p_1 = p_2 = 0$, then $P(M)$ can reach its vertex @ by a history ν proceeding along the same path, as M, and executing every macro-operator $\mathrm{v} \leftarrow \geq \mathrm{v}{+}1$ as $\mathrm{v} \leftarrow \mathrm{v} + 1$ (i.e. the corresponding value read from the input into the variable x in each case must be just $v + 1$), in this case for the sequence $c_1, c_2, \ldots, c_k$ of M configurations there is a sequence $s_1, s_2, \ldots, s_k$ of $P(M)$ states in ν such that s_i models c_i for all $i \leq k$, and $s_k = (@, \vec{v}_0)$ for some $\vec{v}_0$ with

$$\vec{v}_0.a_1 = \vec{v}_0.a_2 = \vec{v}_0.b_1 = \vec{v}_0.b_2 = \vec{v}_0.z,$$

so the vertex @ is reachable in $P(M)$.

On the other hand, if the vertex @ is reachable in $P(M)$, then consider an initial history ν along a path α in $P(M)$, which contains @.

Observe that in every history ν' of $P(M)$ we have according to the $P(M)$ construction that, if $n_i(\nu') \in V(M)$ (i.e., if the ith vertex of the history ν' is also the machine M vertex), then both

$$\vec{v}_i.b_1 - \vec{v}_i.z \geq \vec{v}_i.z - \vec{v}_i.a_1, \text{ and } \vec{v}_i.b_2 - \vec{v}_i.z \geq \vec{v}_i.z - \vec{v}_i.a_2.$$

As to the history ν, if for some $n_i(\nu) \in V(M)$ we have either

$$\vec{v}_i.b_1 - \vec{v}_i.z = (\vec{v}_i.z - \vec{v}_i.a_1) + d, \text{ or } \vec{v}_i.b_2 - \vec{v}_i.z = (\vec{v}_i.z - \vec{v}_i.a_2) + d,$$

for $d > 0$ then for all $j \geq i$ with $n_j(\nu) \in V(M)$ also

$$\vec{v}_j.b_1 - \vec{v}_j.z \geq (\vec{v}_j.z - \vec{v}_j.a_1) + d, \text{ or } \vec{v}_j.b_2 - \vec{v}_j.z \geq (\vec{v}_j.z - \vec{v}_j.a_2) + d.$$

Since at the end of the history ν we have

$$\vec{v}.b_1 - \vec{v}.z = \vec{v}.z - \vec{v}.a_1 = 0, \text{ and } \vec{v}.b_2 - \vec{v}.z = \vec{v}.z - \vec{v}.a_2 = 0.$$

we conclude also

$$\vec{v}_i.b_1 - \vec{v}_i.z = \vec{v}_i.z - \vec{v}_i.a_1, \text{ and } \vec{v}_i.b_2 - \vec{v}_i.z = \vec{v}_i.z - \vec{v}_i.a_2$$

for all i with $n_i(\nu) \in V(M)$.

This means that in the history ν every $P(M)$ macro-operator $\mathbf{v} \leftarrow \geq \mathbf{v}+1$ was actually executed by increasing the value of v just by 1.

Clearly, the path α consists of consequent fragments, each fragment modelling one machine M instruction, let α' be the path in M which corresponds to the $P(M)$ path α. It is straightforward to prove by induction using the construction of $P(M)$ that the machine M executes along the path α' and that for $s_1, s_2, \ldots, s_k$ being the sequence of states $s_i = (n_i, \vec{v}_i)$ in ν for which $n_i \in V(M)$ and $c_1, c_2, \ldots, c_k$ being the sequence of M configurations each c_i is modelled by s_i.

Since the vertex ◎ appears in $P(M)$ only in the modelling fragment of the M halting operator, we establish also that M indeed stops with 0 final values of its counters Z_1 and Z_2.□

8.2 Comparison with Background

In this section we formulate some already known decidability and undecidability results for the vertex reachability problem for some program classes which permit the use of counters in the programs.

Let us define the language L_1 by allowing programs in it to have all the facilities of LBASE, as well as one special counter variable z with the permitted operations being the assignments $\mathbf{z} \leftarrow \mathbf{c}$ for $c \in \mathbf{Z}$, the standard two-way counter operations: $\mathbf{z} \leftarrow \mathbf{z}+1$, $\mathbf{z} \leftarrow \mathbf{z}-1$ and the test $\mathbf{z} = \mathbf{0}$ with 2 exits: "+" and "−".

The language L_2 is defined by enriching the LBASE program facilities in another way: every program in L_2 is allowed to have in the addition to all LBASE facilities a special counter variable z with the permitted operations $\mathbf{z} \leftarrow \mathbf{c}$ for $c \in \mathbf{Z}$, the counter increasment $\mathbf{z} \leftarrow \mathbf{z}+1$, and the test $\mathbf{z} < \mathbf{x}$ ($\mathbf{x} < \mathbf{z}$) with 2 exits: "+" and "−" for x being an ordinary program variable. Notice that there is no counter decreasment operations in the language L_2, the counter used by any program in the language is "one-way".

Theorem 8.2 *[BBK77]. The vertex reachability problem is decidable for L_1 programs, but undecidable for L_2 programs.*

It can be learned from the theorem that the comparison of the counter variable with an ordinary variable (which can receive its value from the input) is rather lethal for the deciding of the reachability (it is possible to obtain the undecidability even if considering only one one-way counter in the programs). The proof of the theorem,

given in [BBK77], uses the modelling of a Minsky machine's configuration sequence by the input to the program, the counter is used just to check that the sequence of values appearing on the input encodes such a sequence (what can be easily done, if the counter can be reset to the constants as many times, as needed).

The result of Theorem 8.1, obtained originally in this work, is stronger since it allows to get the undecidability of the reachability problem out of the program class for which the comparison between the counter and ordinary (input) variables is allowed, but which *do not* have the possibility to decrease the counter in any way.

8.3 "Positive" LTIM Programs

Comparing the time constraint specification formalisms over the discrete (integer) and dense (rational) time variable value domains, the result of Theorem 8.1 vs. the Corollary 5.12 seems at first sight rather surprising for the "normal" intuition tells us that the discrete case should be the subject of an *easier* analysis automation (as it is in the most of other timed specification formalisms). What is the reason for the undecidability of the vertex reachability for LTIM' programs, is the *infinite* discrete structure with every element of it being "individual" in some sense (this "individuality" can be characterized by the property that whenever $x < y$ for $x, y \in \mathbf{Z}$, then also $x \leq y - 1$). If one disables this "individuality" property of integers in one or another way, the vertex reachability problem for the programs of the restricted kind may again become decidable, as it is shown by the following result (outlined).

Let us call a path in a LTIM program "positive", if every exit from a comparison operator in it is labelled by "$-$" (recall that we have only strong inequalities $\mathbf{x} < \mathbf{y}$, $\mathbf{x} < \mathbf{c}$, $\mathbf{x} < \mathbf{c}$ admitted in the LTIM comparison operators) and every exit from the positive assignment operator is labelled by "$+$". Let us call a history β of a LTIM program P *integer-valued*, if for every $i \in \mathcal{CR}d_\beta$ and every $x \in Vars(P)$ we have $\vec{v}_i(\beta).x \in Z$. We call a path α in a LTIM program *integer feasible*, if there is an integer-valued history along it.

Lemma 8.3 *Every feasible positive path in a LTIM program P with all explicit constants being integers is also integer feasible.*

Proof: Consider a feasible path α in a given program P with all explicit constants being integers, let β be some history along α. The integer valued history β' along α is obtained from the history β by defining for all $i \in \mathcal{CR}d_\alpha$ and $a \in Vars(P)$

$$\vec{v}_i(\beta').a = \lfloor \vec{v}_i(\beta).a \rfloor$$

(by $\lfloor x \rfloor$ we denote the "integral part" of the real number x (i.e. the largest integer which does not exceed x)). Since for $x \leq y$ we have always also $\lfloor x \rfloor \leq \lfloor y \rfloor$, as well as

for an integer constant c always $\lfloor x+c \rfloor = \lfloor x \rfloor + c$, we obtain that β' is a history along α from the semantics (state transition system) definition for the LTIM programs (the positivity of the path α guarantees that at no point at the path the further control flow is passed in accordance with $x < y$ for some variable values x and y, in which case the analogue $\lfloor x \rfloor < \lfloor y \rfloor$ would not necessarily hold).□

Let us call a LTIM program "positive", if all the constants used in its comparison and variable activation operators are integers and it has only positive paths. Clearly, we obtain from Lemma 8.3 the following result

Theorem 8.4 *The set of all feasible paths in a positive LTIM program coincides with the set of its integer feasible paths.*

Observe that a path in a LTIM' program is feasible if and only if it is integer feasible in "the same" LTIM program (just having the variable values interpreted over the variable value space of rational numbers), so, in fact the result of Theorem 8.4 together with Theorem 4.3 allows to give the projectivity characteristics to the set of all finite feasible paths in a given positive LTIM' program, what implies the decidability of the vertex reachability for positive LTIM' programs.

Regarding the path feasibility analysis for positive LTIM_0 programs, the program feasible path set coincidence with the set of all integer feasible paths allows to use for this analysis, in fact, just simple FSM state space enumeration techniques, which apply to the integer valued analogues of the LTIM_0 programs.

Part II

Models With Real Time Semantics

Chapter 9

Parallel Timer Processes

We begin the consideration of real time specification models by the Parallel Timer Processes (see Section 1.5 for some motivation) for which the decidability of both the vertex reachability and the strong and weak bisimulation equivalence problems (see Chapter 11 and Chapter 12) are demonstrated.

In this chapter we introduce the basic model of PTPs (Section 9.1) and show the decidability of the reachability problem for PTPs via modelling of PTPs by LTIM_0 programs (see Section 9.2). Further on a number of possible enrichments of the basic PTP model which still retain decidable at least the vertex reachability problem are considered in Chapter 10.

9.1 The Basic PTP Model

Assume that we have a predefined finite set L of *events*. We define the basic model (model class) of Parallel Timer Processes over the event set L.

Let $G = \langle V, E, f, t, L, lab \rangle$ be a finite edge-labelled graph with the set of vertexes V, the set of edges E, the set of edge labels L being the set of events, and the edge labelling function $lab : E \to L$ (when compared with labelled graphs from Section 2.1, the graph G does not have vertex labels, the graph's initial vertex also is not specified yet).

We assume that in the graph G every edge $e \in E$ is coloured either *red* (instantaneous) or *black* (possibly waiting), we denote the set of red edges of G by R and that of black ones by B.

Given such a graph G and a finite set of timers (time variables) $\mathcal{T}$, we define a *timer automaton* by associating with every $e \in E$:

- a set $\gamma(e) \subseteq \mathcal{T}$ of timers, called the edge e *condition* (on what timers the transitions along e depend) and

- a *timer setting function* $\phi(e) : \mathcal{T} \to \mathcal{T} \cup \mathbf{Q}^{+0}$ (every timer can be set either to some nonnegative rational constant, or to some (other) timer value, some timer values may remain unchanged).

We denote a concrete timer setting function $\phi : \mathcal{T} \to \mathcal{T} \cup \mathbf{Q}^{+0}$ by a vectorial assignment

$$\langle t_1, t_2, \ldots, t_m \rangle \leftarrow \langle \phi(t_1), \phi(t_2), \ldots, \phi(t_m) \rangle,$$

or, if no confusion can arise, also as a vector of simple assignments

$$t_1 \leftarrow \phi(t_1),\ t_2 \leftarrow \phi(t_2),\ \ldots\ ,\ t_m \leftarrow \phi(t_m).$$

Moreover, it for some $t \in \mathcal{T}$ we have $\phi(t) = t$, the corresponding simple assignment can be omitted (see below the semantics for the intuitive justification).

For $\Phi = \langle V, E, f, t, L, lab, \mathcal{T}, \gamma, \phi \rangle$ being a timer automaton we define the set of its *states* to be

$$\mathcal{S}^{\Phi} = \{ \langle v, \delta \rangle \mid v \in V,\ \delta : \mathcal{T} \to \mathbf{Q}^{+0} \}.$$

The *parallel timer process* (PTP, for short, called also *timed process*, if no confusion can arise) is defined as a pair $P = \langle \Phi, s \rangle$, where $\Phi = \langle V, E, f, t, L, lab, \mathcal{T}, \gamma, \phi \rangle$ is a timer automaton and $s \in \mathcal{S}^{\Phi}$ is defined to be the process P *initial state*.

For the process $\langle \Phi, s \rangle$ with the set of timers $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ let $\forall i \leq m$ $\delta_i \stackrel{\text{def}}{=} \delta(t_i)$ and $\vec{\delta} = \langle \delta_1, \ldots, \delta_m \rangle$.

Example 9.1 In order to have some illustration for the constructs, used in the Parallel Timer Process definition, consider a process P with

Vertexes	v_1, v_2;	
Timers	$\mathcal{T} = \{t_1, t_2, t_3, t_4\}$;	
Edges	e_1	from v_1 to v_2, red, labelled by a, condition $\{t_1, t_3\}$, timer setting $\langle t_1,\ t_2,\ t_3,\ t_4 \rangle \leftarrow \langle t_2,\ 7,\ t_2,\ 5 \rangle$;
	e_2	from v_2 to v_1, red, labelled by a, condition $\{t_1, t_4\}$, timer setting $\langle t_1,\ t_2,\ t_3 \rangle \leftarrow \langle 3,\ 6,\ t_2 \rangle$;
	e_3	from v_2 to v_1, black, labelled by b, condition $\{t_3\}$, timer setting $\langle t_3 \rangle \leftarrow \langle t_4 \rangle$ and

Initial state $s_0 = \langle v_1, \langle 0.7, 3.14, 2, 0 \rangle \rangle$ (the vector $\langle 0.7, 3.14, 2, 0 \rangle$ is the encoding of the function δ with $\delta(t_1) = 0.7, \delta(t_2) = 3.14$, etc.).

The pictorial representation of P is given in Figure 9.1. Here and further on we follow the convention to represent the black edges as dashed. Notice also the way, how the initial state of the process is depicted.

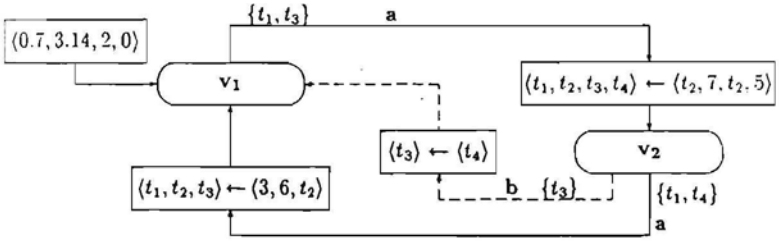


Figure 9.1: A Parallel Timer Process

For a timer automaton Φ we interpret all its edge labels $\sigma \in L$ as the real events which may occur during the "life time" of the processes, associated with Φ (such an action can be either a sending or receiving a signal, reading or writing some data, some kind of synchronization, or any other kind of communication with the automaton's environment; we abstract here from the nature of the actions).

For every real action σ there are certain rules (see below the formal definition) according to which a process, which is allowed to perform σ, is changed to another process (associated with the same automaton Φ) while performing the action (the automaton Φ changes its state while executing the action σ). To formalize this behaviour we define below for every $\sigma \in L$ the transition relation $\xrightarrow{\sigma}$ in the set $\mathcal{S}^{\Phi} \times \mathcal{S}^{\Phi}$ with $P \xrightarrow{\sigma} Q$ meaning that the process P can perform the action σ and then become Q

Every process $P = \langle \Phi, s \rangle$ is assumed to work in time, to formalize this behaviour we define also, as in [Wan90], a special kind of *delay* actions $c(d)$ with $d \in \mathbf{Q}^{+0}$ for the processes, meaning by the relation $P \xrightarrow{c(d)} Q$ that the process P can become Q just by letting time to pass for d units.

Every real (non-delay) action, occurring in the process, is associated with the change of the process initial vertex along some edge of the timer automaton, so we define first the relations $\xrightarrow{e:\sigma}$ for $e \in E$ and $\sigma \in L$.

Let $\Phi = \langle V, E, f, t, L, lab, \mathcal{T}, \gamma, \phi \rangle$ be a timer automaton, then

$$\langle \Phi, \langle v, \delta \rangle \rangle \xrightarrow{e:\sigma} \langle \Phi, \langle v', \delta' \rangle \rangle,$$

if the edge $e \in E$ is leading from v to v' and is labelled by $lab(e) = \sigma$, and

- for every $t \in \gamma(e)$ $\delta(t) = 0$ (timing enabling condition; the transition of the process along the edge is possible only when the values of timers this edge depends on have reached 0) and

- for every timer $t \in \mathcal{T}$ its new value $\delta'(t)$ is computed by the setting $\phi(e)$ in a way:
  - if $\phi(e)(t) = c \in \mathbf{Q}^{+0}$, then $\delta'(t) = c$,
  - if $\phi(e)(t) = t' \in \mathcal{T}$, then $\delta'(t) = \delta(t')$.

Let $\langle \Phi, \langle v, \delta \rangle \rangle \xrightarrow{\sigma} \langle \Phi, \langle v', \delta' \rangle \rangle$ iff $\langle \Phi, \langle v, \delta \rangle \rangle \xrightarrow{e:\sigma} \langle \Phi, \langle v', \delta' \rangle \rangle$ for some $e \in E$.

For delay transitions:

$$\langle \Phi, \langle v, \delta \rangle \rangle \xrightarrow{\epsilon(d)} \langle \Phi, \langle v, \delta' \rangle \rangle \quad \text{iff}$$

- for every red edge $e \in R$ outgoing from v (i.e. having $f(e) = v$) there exists $t \in \gamma(e)$ with $\delta(t) \geq d$ (no red edge will be enabled during the waiting of d seconds. If a transition along a red edge is enabled, a longer delay of the process at the current vertex is not possible (either this transition, or some other, must fire immediately)),
- for every $t \in \mathcal{T}$ $\delta'(t) = \delta(t) \ominus d$, where $x \ominus y \stackrel{\text{def}}{=} max\{0, x - y\}$ for all x, y (the values of all timers are synchronously decreasing downto 0 along the passage of time).

Let for $s, s' \in \mathcal{S}^{\Phi}$ $s \xrightarrow{\nu} s'$ whenever $\langle \Phi, s \rangle \xrightarrow{\nu} \langle \Phi, s' \rangle$ for ν being either $\sigma \in L$ or $\epsilon(d)$ with $d \in \mathbf{Q}^{+0}$.

Example 9.2 For the timed process from Example 9.1 the following chain of transitions is possible:

$$s_0 \xrightarrow{\epsilon(2)} s_1 \xrightarrow{a} s_2 \xrightarrow{\epsilon(1.14)} s_3 \xrightarrow{b} s_4 \xrightarrow{\epsilon(3.86)} s_5 \cdots, \quad \text{where}$$

$$\begin{aligned}
s_0 &= \langle v_1, \langle 0.7,\ 3.14,\ 2,\ 0 \rangle \rangle, \\
s_1 &= \langle v_1, \langle 0,\ 1.14,\ 0,\ 0 \rangle \rangle, \\
s_2 &= \langle v_2, \langle 1.14,\ 7,\ 1.14,\ 5 \rangle \rangle, \\
s_3 &= \langle v_2, \langle 0,\ 5.86,\ 0,\ 3.86 \rangle \rangle, \\
s_4 &= \langle v_1, \langle 0,\ 5.86,\ 3.86,\ 3.86 \rangle \rangle, \\
s_5 &= \langle v_1, \langle 0,\ 2,\ 0,\ 0 \rangle \rangle.
\end{aligned}$$

Since there are no enabled red edges at s_3, the process could, starting from it, develop also as

$$s_3 \xrightarrow{\epsilon(2.25)} s_6 \xrightarrow{b} s_7 \xrightarrow{\epsilon(1.61)} s_5 \cdots, \quad \text{or}$$

$$s_3 \xrightarrow{\epsilon(3.86)} s_8 \xrightarrow{a} s_9 \xrightarrow{\epsilon(3)} s_{10} \cdots, \quad \text{where}$$

$$s_6 = \langle v_2, \langle 0,\ 3.61,\ 0,\ 1.61\rangle\rangle,$$
$$s_7 = \langle v_1, \langle 0,\ 3.61,\ 1.61,\ 1.61\rangle\rangle,$$
$$s_8 = \langle v_2, \langle 0,\ 2,\ 0,\ 0\rangle\rangle,$$
$$s_9 = \langle v_1, \langle 3,\ 6,\ 2,\ 0\rangle\rangle,$$
$$s_{10} = \langle v_1, \langle 0,\ 3,\ 0,\ 0\rangle\rangle.$$

Concerning the labelled transition system semantics of Parallel Timer Processes the following important properties can be observed:

- *time determinacy* ([Wan90]) meaning that, if $P \xrightarrow{\epsilon(d)} P'$ and $P \xrightarrow{\epsilon(d)} P''$, then $P' = P''$;
- *time continuity* ([Wan90]), meaning that $P \xrightarrow{\epsilon(d+e)} P'$ if and only if $P \xrightarrow{\epsilon(d)} P'' \xrightarrow{\epsilon(e)} P'$ for some P'';
- *time-stop freeness* (this property is similar to the deadlock-freeness considered in [NSY91]), meaning that for every PTP P always
  - either $P \xrightarrow{\epsilon(d)} P(d)$ for every $d \in \mathbf{Q}^{+0}$ for some $P(d)$, or
  - $P \xrightarrow{\epsilon(d)} P' \xrightarrow{\sigma} P''$ for some $d \in \mathbf{Q}^{+0}$ and some $\sigma \in L$.

Let us assume that the given set of L of events contains a special event (action) $\tau \in L$ assuming the transitions $\xrightarrow{\tau}$ to be *internal* transitions of the process, they are invisible for any external observer of the process. We define $L_0 = L \setminus \{\tau\}$ be the set of all *visible* events from the set L (this set is usually ranged over by α).

The presence or absence of the τ labels in the processes is absolutely irrelevant when one studies the vertex *reachability* problem in the processes (see Section 9.2). The τ actions become important when one studies various equivalences between processes (e.g., the *weak bisimulation equivalence* defined in Section 11.1) which want to consider two processes as equivalent, if they exhibit the same *observable* behaviour, no matter how many and what internal actions each one of these processes performs. Also the compositional operators for processes (see Chapter 14) may distinguish between visible and internal actions of the processes.

It is *not* claimed that Parallel Timer Processes is a *completely new* model; it has a number of constructions, similar to the models of the Timed Graph family (see [AD90], [ACD90] and [NSY91]). Some discussion on the relations between PTPs and other timed specification formalisms is done in Section 1.6 and Appendix A.

9.1.1 A Simple Example

Let us show how the PTPs can be used in the description of a simple real time process in telephone exchanges, which controls the timed aspects of the dialling of phone numbers by the abonents. It is assumed that a phone number is a sequence of digits containing at least two digits. The abonent dials all the number's digits one after another and leaves some time between the dialling of any two digits. The controlling process is assumed to interrupt the number dialling in any of the following three cases:

- the first digit of the number does not arrive in 30 seconds after the beginning of the dialling (picking up the receiver);
- the current digit which is not the first does not arrive in 20 seconds after the arrival of the previous digit; and
- the total time delay from the beginning of the number dialling reaches 60 seconds.

The process shown in Figure 9.2 has the following external events as the labels on its edges: "Call", "Digit", "Tim" and "Connect". The number dialling begins with the event "Call", the reception of every digit is modelled by the event "Digit". The event "Tim" means the interruption of the number dialling (the corresponding signal is sent to the abonent) and the event "Connect" stands for the successful completion of the number dialling. The process timer set is $\mathcal{T} = \{D, T\}$, where D controls the current digit arrival time (observe that the timer D is set on different edges to different values (timeout periods)), and the timer T controls the total number dialling time. The process vertexes are "SLEEP", "WFD" (Waiting First Digit), "WD" (Waiting Digits) and "CONN". We define also every edge labelled by the label "Call" or "Digit" to be *black* (these events are normally initiated by the environment of the process), all edges in the process labelled by either "Tim" or "Connect" are *red*, the transitions along these edges (timeout, or succesful completion) must occur as soon as they are enabled. The initial state of the process is assumed to be the pair consisting of the vertex "SLEEP" and the timer value assignment δ with $\delta(t) = 0$ for all $t \in \mathcal{T}$.

9.2 Modelling of PTPs by LTIM Programs

In this section we show, how the Parallel Timer Processes, introduced in Section 9.1, can be modelled by LTIM_0 programs. So we both illustrate one possible methodology of the language LTIM (see Section 3.2) construct interpretation in a model with real time semantics and obtain some useful results about the path set projectivity in PTPs (implying the decidability of the reachability problem for PTPs).

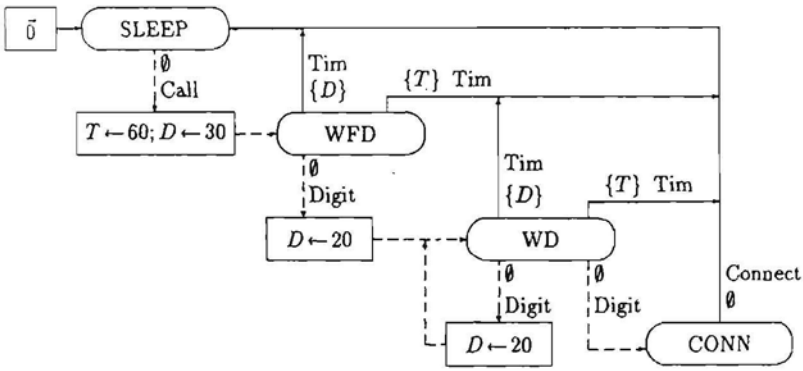


Figure 9.2: A Dialling Control Process

It should be noted that the vertex reachability problem for a model similar to PTPs (Timed Graphs) is already solved in [ACD90] by the method of time regions, without any use of intermediate models (like $LTIM_0$ programs in our case). The primary interest in the PTP as LTIM program modelling methodology is in showing the relations between the analysis of programs (processes) with real time semantics (expressed in the terms of the constructs like the relations $\xrightarrow{c(d)}$) and programs with other data structures. Some further points about this relation can be found in Chapter 10, where by a similar process modelling techniques the vertex reachability problem is proved decidable for more general model classes with the real time semantics.

Let us introduce first some notation. Let for a PTP $A = \langle \Phi, s \rangle$ with

$$\Phi = \langle V, E, f, t, L, lab, \mathcal{T}, \gamma, \phi \rangle \text{ and } s = \langle v, \delta \rangle$$

- $\Phi = \Phi(A)$, and $s_0^A = s$ together with $v_0^A = v$ and $\delta^A = \delta$;
- $V(A) = V(\Phi) = V$, $E(A) = E(\Phi) = E$, $\mathcal{T}(A) = \mathcal{T}(\Phi) = \mathcal{T}$, $\gamma^A = \gamma$, $\phi^A = \phi$, and
- $\mathcal{S}^A = \mathcal{S}^\Phi$ (i.e. the set of the process A states is defined to coincide with the set of the automaton Φ states).

Let Δ^A be the set of all mappings $\mathcal{T} \to \mathbf{Q}^{+0}$ (so, $\mathcal{S}^A = V(A) \times \Delta^A$).

For a PTP $A = \langle \Phi, s \rangle$ we call a sequence

$$(\langle s_i, e_i : (d_i, \sigma_i), s_{i+1} \rangle)_{i \in \mathbf{N}} \quad (\text{or, } (\langle s_i, e_i : (d_i, \sigma_i), s_{i+1} \rangle)_{i<k})$$

an infinite (resp. finite) initial history of A work provided

- $s_0 = s$ and $s_i \in \mathcal{S}^A$ for all i,
- $s_i \xrightarrow{c(d_i)} \xrightarrow{e_i:\sigma_i} s_{i+1}$ with $d_i \in \mathbf{Q}^{+0}$, $\sigma_i \in L$ for all i.

Associated with these histories there are paths $v_0e_0, v_1e_1, \ldots$ and $v_0e_0, v_1e_1, \ldots, v_k$ in the graph (see Section 2.1) $A^G = (V, E, f, t, L, lab, v_0^A)$ of the process A (see Section 2.1, let us say further on "in the process A"), with $s_i = (v_i, \delta^i)$ for all i. Let us say, as in the case of LTIBA programs, that the defined histories go *along* the associated paths.

We call a path in the timed process A *initial*, if it begins with the vertex v_0^A.

An initial path in the process A is called *feasible*, if there is an initial history along it. A vertex $v \in V(A)$ is called *reachable*, if it is contained in some feasible initial path of A.

Theorem 9.3 *There exists an algorithm which, given a Parallel Timer Process A constructs a projectivee for the set of all (both finite and infinite) feasible paths in A.*

Corollary 9.4 *The vertex reachability problem for Parallel Timer Processes is decidable.*

Proof of Theorem 9.3: Given a PTP A we construct for it first the modelling LTIM_0 program $M(A)$.

We let the program $M(A)$ vertex set to contain, first, the set of all process A vertexes: $V(A) \subseteq V(M(A))$ (some other vertexes of $M(A)$ will appear later on). We define for every A timer $t \in \mathcal{T}$ a corresponding variable x^t in the program $M(A)$. For every program $M(A)$ variable value vector $\vec{v} \in \mathcal{V}_{M(A)}$ let $\theta(\vec{v})$ be the mapping $\delta \in \Delta^A$ defined

$$\delta(t) = \vec{v}.x^t \ominus \vec{v}.z \text{ for all } t \in \mathcal{T}(A)$$

(z is the real time counter of the program $M(A)$ and $\ominus$ is the positive minus operation). Let us say that the program variable value vector $\vec{v}$ *is modelling* the process timer value assignment $\theta(\vec{v})$ (it can be the case that for one process timer value assignment there are more than one modelling program variable value vectors).

We say also that the program $M(A)$ state $s = (n, \vec{v})$ for $n \in V(A)$ (recall $V(A) \subseteq V(M(A))$) *models* the process A state

$$\pi_S(s) \stackrel{\text{def}}{=} (n, \theta(\vec{v})).$$

We complete the modelling program construction by building for every process A edge $e \in E(A)$ leading from a vertex n to n' a program fragment of the LTIM_0 operators which is to be put between the vertexes n and n' in $M(A)$ and allows a modelling program history (transition chain) through the fragment from the program

state $s = (n, \vec{v})$ to some state $s' = (n', \vec{v}')$ if and only if the process A can do a transition along e (possibly preceded by some waiting) from $\pi_S(s)$ to $\pi_S(s')$ (see Lemma 9.5 for the modelling statement expressed more precisely).

Assume that we have an edge $e \in E(A)$, leading from n to n' in A, let $\gamma(e) = \{t_1, \ldots, t_s\}$. The edge e is modelled in $M(A)$ by the program fragment, having the following block structure:

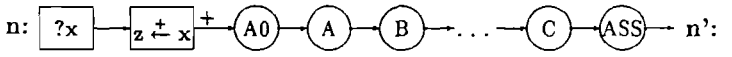
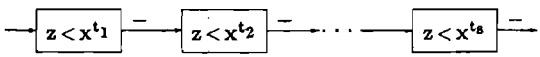


Figure 9.3: Modelling Block Structure

where x is the input variable for $M(A)$ (recall the definition of the programming language LTIM_0 in Section 3.2) and the blocks $A0$, ASS and $A, B, \ldots, C$ are defined in the following.

We clarify simultaneously also the roles which are plaid in the modelling by every block, so we assume that the program is executing the modelling fragment, starting from some state $(n, \vec{v}_n)$ which corresponds to the process state (n, δ) for $\delta = \theta(\vec{v}_n)$. Let the program variable value vectors at the beginning of every block $A0, A, B, \ldots$ be denoted by $\vec{v}_{A0}, \vec{v}_A, \vec{v}_B, \ldots$ respectively.

- The block $A0$ checks that after waiting $d = \vec{v}_{A0}.z - \vec{v}_n.z$ units of time in the state (n, δ) the transition along the edge e in A is enabled (i.e. for all $t \in \gamma(e)$ $\delta(t) \leq d$). It is defined to be the chain



- The blocks $A, B, \ldots, C$ are associated every one with one outgoing *red* edge from the vertex n in the process. Every of these blocks checks that the corresponding red edge did not become enabled *before* waiting d units of time (so preventing the process ability to wait d units and then execute a transition along the edge e). The block, corresponding to a red edge e' with $\gamma(e') = \{u_1, \ldots, u_r\}$ is defined as

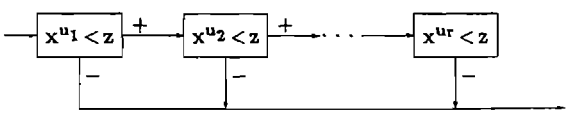


Figure 9.4: The A,B,...,C blocks

- The block ASS models the timer setting $\phi(e)$ by the changing the program $M(A)$ variable values in order to obtain $\delta' = \theta(\vec{v}_{n'})$. It contains for every A timer $t \in \mathcal{T}$ such that $\phi(e)(t) \neq t$ the assignment operator

  - $\mathbf{x^t} \leftarrow \mathbf{z} + \mathbf{c}$, if $\phi(e)(t) = c \in \mathbf{Q^{+0}}$, and
  - $\mathbf{x^t} \leftarrow \mathbf{x^u}$, if $\phi(e)(t) = u \in \mathcal{T}$

  (since the timer setting $\phi(e)$ has the vectorial semantics, it can be needed in some cases to replace some of these assignment operators by a pair of operators $\mathbf{x^0} \leftarrow \mathbf{x^u}$ and $\mathbf{x^t} \leftarrow \mathbf{x^0}$ where x^0 is an auxiliary variable, the first of these operators must be put in the assignment sequence before any assignment with x^u as the lefthand side).

We add into the program $M(A)$ also the initializing fragment, containing the assignments $\mathbf{x^t} \leftarrow \delta^{\mathbf{A}}(\mathbf{t})$ for all $t \in \mathcal{T}(A)$.

Let us define for every program $M(A)$ edge e the corresponding process A edge $\tilde{e} = \pi_E(e)$ to be the edge in the modelling fragment of which the program edge e lies.

Let for $n, n' \in V(A)$, $d \in \mathbf{Q^{+0}}$ and $\tilde{e} \in E(A)$

$$\langle n, \vec{v} \rangle \xrightarrow{d;\tilde{e}} \langle n', \vec{v}' \rangle \text{ mean that}$$

- $\vec{v}'.z = \vec{v}.z + d$ and
- there exists a finite history ν of $M(A)$ with $\langle n, \vec{v} \rangle$ as the initial value and with $\langle n', \vec{v}' \rangle$ as the final value such that $\pi_E(e_i(\nu)) = \tilde{e}$ for all $i < card(\mathcal{CR}d_\nu)$.

Lemma 9.5 *If* $\langle n, \delta \rangle \xrightarrow{\epsilon(d)} \xrightarrow{e;\sigma} \langle n', \delta' \rangle$ *in the process* A, *and the program variable value vector* $\vec{v} \in \mathcal{V}_{M(A)}$ *is taken such that* $\theta(\vec{v}) = \delta$, *then we have also* $\langle n, \vec{v} \rangle \xrightarrow{d;e} \langle n', \vec{v}' \rangle$ *in the program* $M(A)$ *for some* $\vec{v}'$ *with* $\theta(\vec{v}') = \delta'$.

Reverse, if $\langle n, \vec{v} \rangle \xrightarrow{d;e} \langle n', \vec{v}' \rangle$ *in the modelling program* $M(A)$ *and* $\sigma = lab(e) \in L$, *then* $\langle n, \theta(\vec{v}) \rangle \xrightarrow{\epsilon(d)} \xrightarrow{e;\sigma} \langle n', \theta(\vec{v}') \rangle$.

Proof: Follows from the construction of the modelling program.□

Let us introduce for the program $M(A)$ the finite and infinite acceptance sets $S_F(M(A)) = S_I(M(A)) = V(A)$.

We define for every initial accepting path α in $M(A)$ a corresponding path $\pi_P(\alpha)$ to be the path $\alpha' = n_0 e_0, \ldots [, n_k]$ in the process A for which

- $n_0, n_1, \ldots [, n_k]$ is the sequence of all path α vertexes which belong to $V(A)$, and
- for every two vertexes n_i and n_{i+1} in α' every program $M(A)$ edge e which lies in the path α between n_i and n_{i+1} has $\pi_E(e) = e_i$.

Lemma 9.6 *An initial path α' in the process A is feasible if and only if there exists a feasible initial accepting $M(A)$ path α with $\alpha' = \pi_P(\alpha)$.*

Proof: Let α be a (finite or infinite) feasible accepting path in $M(A)$ and $\alpha' = \pi_P(\alpha)$.

Let us denote by $i_1, i_2, \ldots [, i_k]$ the sequence of all indices i for which the ith vertex of the path α $n_i(\alpha) \in V(A)$. Then for every history ν along α in $M(A)$ we have

$$\langle n_{i_j}(\nu), \vec{v}_{i_j}(\nu)\rangle \xrightarrow{d;\tilde{e}_j} \langle n_{i_{j+1}}(\nu), \vec{v}_{i_{j+1}}(\nu)\rangle, \text{ where}$$

- $\tilde{e}_j = \pi_E(e_{i_j}(\alpha))$ and
- $d = \vec{v}_{i_{j+1}}.z - \vec{v}_{i_j}.z$ for every two adjacent indices i_j and i_{j+1}.

According to Lemma 9.5 the sequence

$$\nu' = (\langle\langle \tilde{n}_j, \delta_j\rangle, \tilde{e}_j : \sigma_j, \langle \tilde{n}_{j+1}, \delta_{j+1}\rangle\rangle)$$

for $\langle \tilde{n}_j, \delta_j\rangle = \pi_S(\langle n_{i_j}(\nu), \vec{v}_{i_j}(\nu)\rangle)$ and $\tilde{e}_j = \pi_E(e_{i_j})$, $\sigma_j = lab(\tilde{e}_j)$ for all j is a history of the process A along α'. The initiality of ν' in A is by the construction of $M(A)$ and the initiality of ν in $M(A)$.

Reverse, if the path α' is feasible in A, the existence of an appropriate feasible initial path α in $M(A)$ also is straightforward from Lemma 9.5.□

Finally, we show how to construct for the process A the graph $G(A)$ which is the projectivee for both the sets of all finite and infinite feasible paths in A. Having built the modelling program $M(A)$ for the process A, let us build a projectivee $BG(M(A))$ for the set of all initial finite feasible paths of $M(A)$ (see Theorem 4.3). Since $M(A)$ is a LTIM_0 program, the graph $BG(M(A))$ is also a projectivee (ω-projectivee) for the set of all infinite feasible paths in $M(A)$ (see Lemma 7.12).

Given the graph $BG(M(A))$, the construction of $G(A)$ is done, as follows:

- $V(G(A)) = \{\langle n, C\rangle \in V(BG(M(A))) \mid n \in V(A)\}$, a vertex $\langle n, C\rangle$ in $G(A)$ is labelled by $n \in V(A)$;
- there is an edge from $\langle n, C\rangle$ to $\langle n', C'\rangle$, labelled by $\tilde{e} \in E(A)$, if and only if there exists a path β in $BG(M(A))$ from $\langle n, C\rangle$ to $\langle n', C'\rangle$ with $\pi_E(e_i(\beta)) = \tilde{e}$ for all $i < card(\mathcal{CR}d_\beta)$ (i.e. for all program $M(A)$ edges in the projection of β).
- the graph $G(A)$ initial vertex is $\langle v^A, C_0^A\rangle$ where $C_0^A = [\![\vec{v}_0^A]\!]$ for the $M(A)$ variable value vector $\vec{v}_0^A$ which is obtained after the executing the initializing fragment with the assignments $\mathbf{x^t} \leftarrow \delta^{\mathbf{A}}(\mathbf{t})$ for all $t \in T(A)$ (clearly, $\theta(\vec{v}_0^A) = \delta^A$, it is according to the definition of configuration for LTIM programs that the set C_0^A is singleton (the scaling of number line must be done before in order to make all program constants integers)).

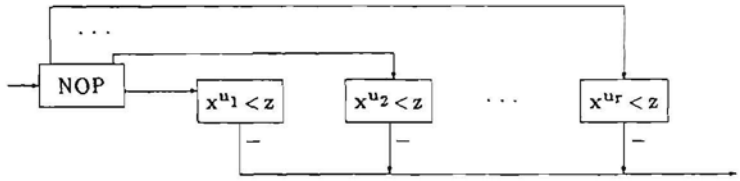


Figure 9.5: The "Positive" A,B,...,C blocks

The proof that $G(A)$ is the needed initial projectivee for the set of all feasible paths in A is straightforward.□

Note. The presented modelling methodology of PTPs by $LTIM_0$ programs in general admits also non-positive paths in the obtained modelling programs (see Section 8.3 for the definition of a "positive" LTIM program). However, if one changes the transition impossibility controlling blocks A,B,...,C (depicted in Figure 9.4), as shown in Figure 9.5 (in fact we just introduce a somewhat wide nondeterminism in the programs), it can be easily seen that for every PTP A its modelling program $M(A)$ is a positive $LTIM_0$ program (all the modelling characterizing results remain true also for the "new" modelling program).

It is easy to define the notion of an integer valued history and integer feasible paths (as well as integer reachable vertexes) also for PTPs, as it was done for LTIM programs in Section 8.3. Using the positivity of the modelling program, as well as the result of Theorem 8.4 we easily obtain the following result characterizing the set of all feasible paths in a PTP.

Theorem 9.7 *The set of all feasible paths in an arbitrary PTP with all explicit constant used in the timer settings and the initial timer value assignment being integers is feasible if and only if it is integer feasible.*

This result gives yet another way for the deciding the vertex reachability problem for Parallel Timer Processes.

Chapter 10

Enrichments of PTPs

In this chapter we consider some enrichments of the Parallel Timer Process model which can be introduced still retaining the possibility of the automated reachability analysis. We look also for the ways how the additional specification constructs can be modelled by means of the languages LTIM and LTIBA. Of particular importance in this chapter is Section 10.4, where the data dependent enrichments of PTPs are considered and studied w.r.t. their analysis automation possibilities.

10.1 Processes with Inactive Timers

In various timed specification formalisms, the programming language SDL [CC89] being a typical example, a timer uses to be an abstract device which can be either active (set up to produce a timeout after some period of time), or *inactive* (never going to produce a timeout without a prior activation). Only if a timer is active, it has in a given process state an associated time value showing the time remaining before its timeout. The inactive timer idea can be easily adopted also for PTPs the following way (we call the newly obtained processes the *PTPs with Undefined values*, or simply "PTPU"s, for short).

Let $\mathbf{u}$ be a special timer value ("u" for "undefined"), $\mathbf{u} \notin \mathbf{Q}$. We let $\mathbf{u} \ominus d = \mathbf{u}$ for all $d \in \mathbf{Q}^{+0}$ (the values $\mathbf{u} \ominus \mathbf{u}$ and $d \ominus \mathbf{u}$ for $d \in \mathbf{Q}^{+0}$ are not defined), let $\overline{\mathbf{Q}}^{+0} = \mathbf{Q}^{+0} \cup \{\mathbf{u}\}$. The value $\mathbf{u}$ is assumed to be also incomparable with the numbers on being less or greater (i.e. no one of the relations $\mathbf{u} < d$, $d < \mathbf{u}$, $\mathbf{u} = d$ holds for any $d \in \mathbf{Q}^{+0}$).

The definition of PTPUs now is simply obtained from the basic definition of PTPs (see Section 9.1) by, first, allowing every process state to be a pair $\langle v, \delta \rangle$ for $v \in V(P)$ and $\delta : \mathcal{T} \to \overline{\mathbf{Q}}^{+0}$ (instead of $\delta : \mathcal{T} \to \mathbf{Q}^{+0}$), and, second, allowing the value $\mathbf{u}$ to appear in the timer settings associated with the process edges (i.e. for every edge e of a PTPU A $\phi(e) : \mathcal{T} \to \mathcal{T} \cup \overline{\mathbf{Q}}^{+0}$).

Figure 10.1 shows, how the process controlling the timing aspects of the phone number dialling, described in Section 9.1.1, can be rewritten to exploit the undefined timer value possibility to reduce the process vertex set cardinality (observe that we have here an auxiliary timer L which can assume only the values 0 and u). For the sake of illustration we replace the only timer D, controlling in Figure 9.2 the current digit arrival time by two timers: F for controlling the first digit arrival time and N for the "next" (not the first) digit arrival time.

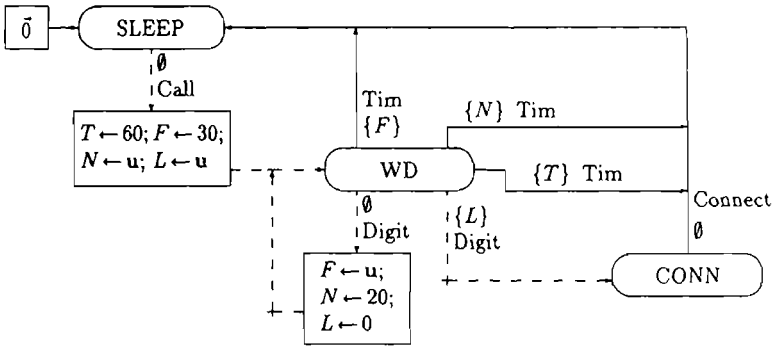


Figure 10.1: A PTPU for Dialling Control

It is easy to see that the proof of Theorem 9.3 can be easily generalized to apply also to the processes with undefined timer values, what means also the decidability of the vertex reachability problem for PTPUs.

The idea of dealing with the undefined timer values consist in modelling of them by the corresponding program variables assuming a certain fixed otherwise not used value, say, -1. So, every edge e timer setting $\phi(e)$ with $\phi(e)(t) = \mathrm{u}$ causes the operator $\mathbf{x}^t \leftarrow (-1)$ to be included in the corresponding modelling program fragment (the block ASS associated with e).

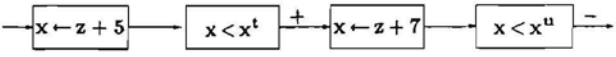
It is not difficult to define the appropriate mapping $\theta : \mathcal{V}_{M(A)} \rightarrow \Delta^A$ and to invent the modelling fragments in LTIM_0 which should be used as the blocks $A0, A, B, \ldots, C$ and ASS in order to obtain the analogue of Lemma 9.5 holding, we do not consider the details.

10.2 Processes with Extended Time Conditions

One particular peculiarity of the PTP model is that the timer value conditions which allow a transition along a given edge to be performed are of the form $\delta(t_i) = 0$ for

all $t_i \in \gamma(e)$. It seems useful sometimes to have these conditions of a more general kind. allowing, as in the Timed Graph model [ACD90, AD90] them to be boolean expressions over the predicates $t_i \lambda c_i$ for $t_i \in \mathcal{T}$, $\lambda \in \{<, \leq, =, \geq, >\}$ and $c_i \in \mathbf{Q}^{+0}$. A transition along an edge e with such condition in the generalized process, called EPTP, is enabled in a state $\langle v, \delta \rangle$ for $v = f(e)$ if and only if the condition's boolean expression interpreted in the standard way over the timer values $\delta(t_1), \ldots, \delta(t_k)$ evaluates to true. Again, whether a transition along an enabled edge *must* be executed, or a further delay of the process is allowed, is the matter of the edge e colour (i.e. either e is red or black). In order to obtain the time-stop freeness of the process labelled transition systems let us require that every predicate b over the timer values, which is associated with a *red* edge, is *closed* in the sense that only the non-strong inequalities, joined by the positive connectives (i.e. & or ∨) are allowed to define b.

The extended time conditions do not cause any very principal problems in applying the above considered schema of the deciding the vertex reachability problem for the processes by reducing it to the finite and infinite path set projectivity of LTIM_0 programs. The most important new point, if compared with the modelling of the basic PTPs, here appears in the constructions for the check, whether in the current process state $\langle n, \delta \rangle$ the transition, requiring the conditions, say, $\delta(t) > 5$ and $\delta(u) \leq 7$, can be performed, or not. As in the proof of Theorem 9.3 (see Figure 9.3) we input first the (absolute) time moment of the transition firing (?x), then assign it positively to z. It remains to check now that the current value of x^t is greater than $z + 5$ and that the value of x^u is not greater than $z + 7$, what is done, as follows (observe that the variable x does not carry any essential information at this moment and we are free to use it as an auxiliary variable):



10.3 Processes with Nondeterministic Timer Settings

The ability to ascribe the same external behavior (=the same label) to both red (instantaneous) and black (possibly waiting) edges in a PTP gives a possibility to express in the PTP model nondeterministic interval-like delays saying that, for example, an action a is going to be executed in a process P between 3 and 5 seconds passed after some other action b was performed. These are constructs, typical in Time Petri Nets [MF76, GMMP89] (see also Section 1.6 for some discussion).

Here we look at another possibility to introduce delays of nondeterministic length

in the PTP model. First of all, let $\mathbf{I}(\mathbf{Q}^{+0})$ be the set of all intervals of $\mathbf{Q}^{+0}$, i.e.

$$\mathbf{I}(\mathbf{Q}^{+0}) = \{[c_1, c_2], [c_1, c_2[,]c_1, c_2],]c_1, c_2[\mid c_1 \in \mathbf{Q}^{+0},\ c_2 \in \mathbf{Q}^{+0} \cup \{+\infty\},\ c_1 \leq c_2\}.$$

We obtain the definition of a PTP with Nondeterministic timer setting (a PTPN, for short) from the definition of PTP by letting for every process edge $e \in E$ to have the timer setting

$$\phi(e) : \mathcal{T} \to \mathcal{T} \cup \mathbf{I}(\mathbf{Q}^{+0})$$

(intuitively, every timer can be assigned either the value of other timer, or some value from a specified time value interval).

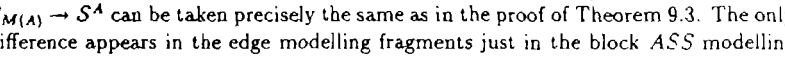
The semantics of such a timer setting is defined to have early binding of the concrete t_i value from the specified interval at the moment of the setting execution (the moment of a transition firing along the edge), i.e. for $\phi(e)(t) = [c_1, c_2]$ we require in the definition of the transition

$$\langle v, \delta \rangle \xrightarrow{e:\sigma} \langle v', \delta' \rangle$$

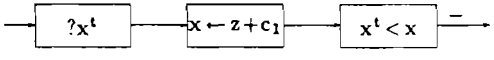
that $\delta'(t_i) \in [c_1, c_2] \cap \mathbf{Q}^{+0}$, (in the other cases the corresponding interval (i.e. $]c_1, c_2]$, etc.) must be taken instead of $[c_1, c_2]$).

We call a PTPN *bounded*, if all nondeterministic timer settings on its edges have finite upper delay bounds $c_2 < +\infty$. The model of bounded PTPNs appears to have theoretically approximately the same expressive power, as standard PTPs and can be analyzed both w.r.t. reachability, path feasibility and equivalence by almost the same methods, as developed for PTPs. However, the unboundedness of the nondeterminism brings essentially new problems in the process analysis (PTPNs can not be any longer modelled by LTIM_0 programs (in a sense of retaining a homomorphism between the state sets), more general LTIM constructs are needed). In the case of the unbounded nondeterminism some infinite not feasible paths with all finite prefixes feasible may appear in the processes and their modelling programs (see e.g. Figure 10.2), so eliminating also the possibility of modelling PTPNs by LTIM_0 programs also in the sense of the path feasibility.

For modelling PTPNs by LTIM programs one can use the same schema as in the modelling of PTPs by LTIM_0 programs. Also the state modelling relation $\pi_S : V(A) \times \mathcal{V}_{M(A)} \to \mathcal{S}^A$ can be taken precisely the same as in the proof of Theorem 9.3. The only difference appears in the edge modelling fragments just in the block ASS modelling the timer setting, when for $\phi(e)(t) = [c_1, c_2[$ we have the modelling fragment



For, say, $\phi(e)(t) = [c_1, +\infty[$ the corresponding modelling fragment is



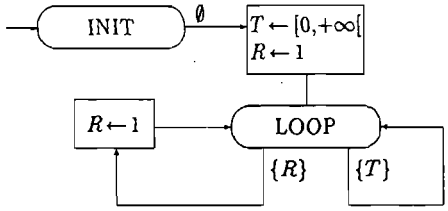


Figure 10.2: Infinite Infeasible Path: An Example

One can observe that in the case of the only bounded nondeterminism in a PTPN A all infinite paths in the modelling program $M(A)$ path feasibility graph $BG(M(A))$ are *progressing* in the sense of Section 7.1, so, according to Lemma 7.5 we again have the initial projectivity of the sets of both finite and infinite feasible path sets in A. As to the unbounded nondeterminism, a counterexample to the infinite feasible path set initial projectivity is straightforward (Figure 10.2 again).

It is possible to prove for bounded PTPNs also the decidability of bisimulation equivalences (see Section 13.1), this decidability is hardly believable for the unbounded PTPNs, even in the simplest case of the strong bisimulation (it is still an open problem).

10.4 Processes with Data Parameters

In this section we consider a more principal possibility to enrich the PTP model by introducing in it the integer-valued data (we call the newly obtained processes the PTPBs ("B" for the LBASE system of commands over the parameters)).

Assuming predefined the set finite L of events the processes can participate in, we define a PTPB process to be a pair $P = \langle \Phi, s \rangle$ for Φ being a TD-automaton ("T" for "time" and "D" for "data"):

$$\Phi = \langle V, E, L, lab, \mathcal{T}, \mathcal{W}, lab_W, \gamma_T, \gamma_W, \phi_T, \phi_W \rangle, \text{ where}$$

- V, E, L and $\mathcal{T}$ are the sets of Φ vertexes, edges, edge labels and timers, respectively, $lab : E \to L$ is the Φ edge labelling function (see the definition of PTPs in Section 9.1);
- for every $e \in E$ $\gamma_T(e)$ and $\phi_T(e)$ are the timer condition and timer setting, associated with e, also defined already for PTPs;
- $\mathcal{W} = \{x_1, x_2, \ldots, x_n\}$ is a finite set of integer-valued variables;

- $lab_W : E \to \mathcal{W}$ ascribes to every process edge a variable;
- for every automaton's edge $e \in E$ $\gamma_W(e)$ is called the *variable value condition* associated with e, and is an arbitrary boolean formula built by the connectives $\&$, $\vee$ and $\neg$ from the elementary predicates of the form $a < b$, where every of the elements a and b is either a process variable $x \in \mathcal{W}$, or an integer constant $c \in \mathbf{Z}$ (let us call the formulas of this kind *linear*);
- for every automaton's edge $e \in E$ $\phi_W(e) : \mathcal{W} \to \mathcal{W} \cup \mathbf{Z}$ is the variable setting function (analogous to the timer settings $\phi_T(e')$); we have also

$s = \langle v_0^P, \delta_T^P, \delta_W^P \rangle$ for $v_0^P \in V$, $\delta_T^P : \mathcal{T} \to \mathbf{Q}^{+0}$ and $\delta_W^P : \mathcal{W} \to \mathbf{Z}$ being the process *initial state*. Every state of the process also is a triple $\langle v, \delta_T, \delta_W \rangle$ for $v \in V$, $\delta_T : \mathcal{T} \to \mathbf{Q}^{+0}$ and $\delta_W : \mathcal{W} \to \mathbf{Z}$.

The idea behind the semantics of PTPBs is that every real transition of the process along an edge $e \in E$ is associated both with the event $lab(e) \in L$ and with the input of some integer value into the variable $lab_W(e) \in \mathcal{W}$. Given an edge $e \in E$ with $lab(e) = \sigma \in L$, the transition $\stackrel{e:\sigma,u}{\longrightarrow}$ for $u \in \mathbf{Z}$ from the state $\langle v, \delta_T, \delta_W \rangle$ s.t. $v = f(e)$ along e with the input of the value u into the variable $lab_W(e) = x_i$ is possible iff both

- $\delta_T(t_i) = 0$ for all $t_i \in \gamma_T(e)$ and
- the *interpretation* of the formula $\gamma_W(e)$ over the variable values $\delta_W^0(x)$ for $x \in \mathcal{W}$, defined

$$\delta_W^0(x) = \delta_W(x) \text{ for } x \neq x_i, \text{ and } \delta_W^0(x_i) = u$$

evaluates to `true` (treating the meaning of the inequality signs and the logical connectives in the standard way). Observe that variable value condition is interpreted taking into account the *new* value of the input variable $x_i = lab_W(e)$.

The target state of such a transition along e is $\langle t(e), \delta_T', \delta_W' \rangle$ where $t(e)$ is the target vertex of e and δ_T', δ_W' are obtained from δ_T and δ_W^0 by the settings $\phi_T(e)$ and $\phi_W(e)$ in a standard way:

- $\delta_T'(t) = \delta_T(t')$, if $\phi_T(e)(t) = t' \in \mathcal{T}$, and
- $\delta_T'(t) = c$, if $\phi_T(e)(t) = c \in \mathbf{Q}^{+0}$, as well as
- $\delta_W'(x) = \delta_W^0(y)$, if $\phi_W(e)(x) = y \in \mathcal{W}$, and
- $\delta_W'(x) = c$, if $\phi_W(e)(x) = c \in \mathbf{Z}$.

We retain for PTPBs the edge colouring (red and black edges) and assume that a transition along a red edge must occur as soon as it is enabled (if there are some values $u \in \mathbf{Z}$ for which the transition is enabled and some others for which the

condition $\gamma_W(e)$ forbids the transition, one of the enabling values must be taken and the transition along the edge has to be executed immediately).

The definition of the semantics of PTPBs is completed by saying that they admit also the delay transitions $\xrightarrow{c(d)}$ during which no variable values are changed and the timers change their values by synchronous decreasing downto 0 as in the case of basic PTPs. More precisely,

$$\langle \Phi, \langle v, \delta_T, \delta_W \rangle \rangle \xrightarrow{c(d)} \langle \Phi, \langle v, \delta'_T, \delta_W \rangle \rangle \text{ iff}$$

- for every red edge $e \in R$ outgoing from v either
  - there exists $t \in \gamma_T(e)$ with $\delta_T(t) \geq d$, or
  - *for all* $u \in \mathbf{Z}$ the variable value condition $\gamma_W(e)$ interpreted over the values $\delta^0_W(x)$ yields `false` (if for some $u \in \mathbf{Z}$ a transition along a red edge can be executed, no further delay is possible);
- for every $t \in T$ $\delta'_T(t) = \delta_T(t) \ominus d$.

Using the facilities offered by PTPBs we give in Figure 10.3 a slightly more detailed "specification" of the phone number dialling control process (see Section 9.1.1, as well as Figure 9.2 and Figure 10.1). First of all we replace the "fictive" timer L from the PTPU specification (Figure 10.1) by an (integer-valued) variable x. More importantly, every signal "Digit", incoming into the process now carries an integer parameter d which is to be bound between 0 and 9 (*which* digit has been dialled). In our toy example it is not possible to deliver any reasonable information about *all* the dialled digits to the connection seeking process (initiated by the event "Connect"), so we deliver just the *last* digit of the dialled number. Formally, this delivery is done by the "input" of some value into another variable, n, however the variable value condition requires this new value to coincide with the current value of d. So, if the connection seeking process "synchronizes" with our process on the event "Connect" by offering any variable value for this synchronization, the synchronization will necessary be performed on the value of d.

At first sight it seems rather obvious to generalize the modelling methodology, given in the proof of Theorem 9.3 (see Lemma 9.5) also to the case of the PTPB modelling by LTIBA programs. However, the situation turns so that we have the following, at first sight rather surprising result:

Theorem **10.1** *The vertex reachability problem for PTPBs is undecidable.*

Proof sketch: The construction, presented in Figure 10.4, allows to increase the value of an arbitrary variable a by 1 every time when the path fragment from the vertex U to the vertex U' is executed (@ is an auxiliary vertex from which no other

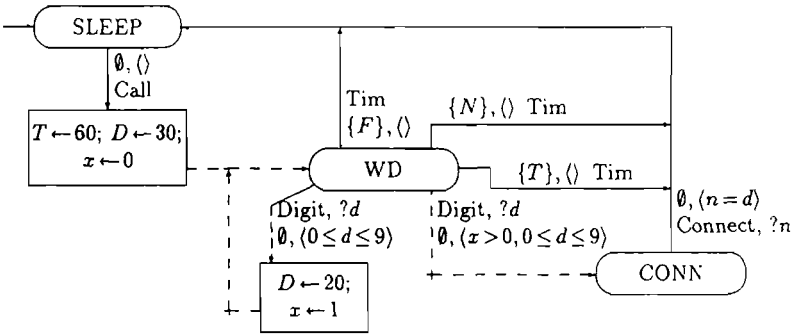


Figure 10.3: The PTPB phone

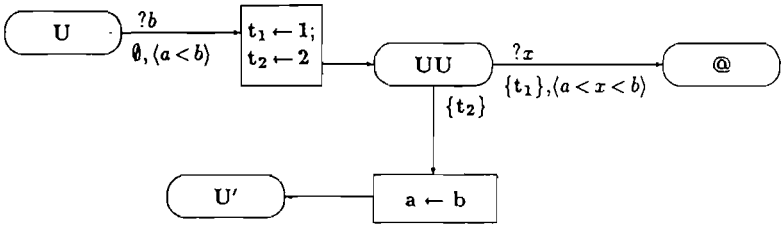


Figure 10.4: A Variable Increasment in PTPB

vertex is reachable, every edge in this fragment is *red*): had b received any *other* value (i.e. *not* that of $a+1$) greater than the value of a, the process execution would reach the vertex @ since the transition, pending on the timer t_1 *becomes enabled* in 1 second after the process reaching the *UU* vertex. So, such a block can be used as the operation a ← a + 1 in any machine with counters. The variable value decreasment is done in a similar way, recall for the undecidability that in PTPBs we have unlimited possibilities to use the variable *comparison* operation.□

It turns out that we can restrict the PTPB model just slightly in order to retain the decidability of the vertex reachability (and the finite feasible path set projectivity) for the restricted model, called PTPB_0.

First of all, given a linear formula (predicate) γ_W over the variables $x \in \mathcal{W}$, let us call γ_W *semi-closed* w.r.t. $x_i \in \mathcal{W}$, if the formula contains positively (i.e. within an even number of negations) *no more than one* occurrence of inequalities $x_i < a$

and $a < x_i$ for a being either a variable $x \in \mathcal{W}$, or an integer constant (the number of negative occurrences of these inequalities (corresponding naturally to the positive occurrences of inequalities $x_i \geq a$ and $a \geq x_i$) is not limited).

We call a PTPB A a PTPB_0, if for every *red* edge $e \in E(A)$ its variable condition $\gamma_W(e)$ is semi-closed w.r.t. the edge input variable $lab_W(e) \in \mathcal{W}$ (the variable value conditions on black edges are not constrained by any additional requirements).

Even a simpler natural way of restricting PTPBs when seeking the decidability of the vertex reachability problem would be rather acceptable: one could simply forbid *any* use of the input variable value in the inequality system determining the possibility of the transitions along the edge. However, as the further results show, this is not necessary, so we manage to save a considerable modelling power of the decidable specification formalism (also the phone number dialling process, described in Figure 10.3 is a PTPB_0).

We define for PTPBs the notions of the process graph, path, history, feasible path, etc. the same way as it is done for PTPs in Section 9.2.

Given a PTPB A we let Δ_T^A to be the set of all mappings $\delta_T : \mathcal{T}(A) \to \mathbf{Q}^{+0}$ and Δ_B^A to be the set of all mappings $\delta_W : \mathcal{W}(A) \to \mathbf{Z}$.

Theorem **10.2** *There exists an algorithm which, given a $PTPB_0$ A constructs a projectivee for the set of all finite feasible paths in A.*

Corollary **10.3** *The vertex reachability problem for $PTPB_0$s is decidable.*

Proof of **Theorem 10.2:** Given a PTPB_0 A we construct for it the modelling LTIBA program $M(A)$ following similar lines, as in the proof of Theorem 9.3 for the PTP modelling.

We let first the program $M(A)$ vertex set to contain the set of the process A vertexes: $V(A) \subseteq V(M(A))$ (some other vertexes of $M(A)$ will appear later on).

We define for every A timer $t \in \mathcal{T}$ a corresponding LTIM variable x_t^T in the program $M(A)$, for every A variable $y \in \mathcal{W}$ the LBASE variable x_y^B is defined in $M(A)$, as well.

For every program $M(A)$ variable value vector $\vec{v} = (\vec{v}^B, \vec{v}^T) \in \mathcal{V}_{M(A)}$ let $\theta^T(\vec{v}^T)$ be the mapping $\delta_T \in \Delta_T^A$ defined

$$\delta_T(t) = \vec{v}^T.x_t^T \ominus \vec{v}^T.z \text{ for all } t \in \mathcal{T}(A),$$

let $\theta^B(\vec{v}^B) = \delta_B \in \Delta_B^A$ be defined by

$$\delta_B(y) = \vec{v}^B.x_y^B \text{ for all } y \in \mathcal{W}(A).$$

Let us say that the program variable value vector $\vec{v}$ *is modelling* the process timer and variable value assignment $\theta(\vec{v}) = (\theta^T(\vec{v}^T), \theta^B(\vec{v}^B))$.

We say also that the program $M(A)$ state $s = \langle n, \vec{v} \rangle$ for $n \in V(A)$ (recall $V(A) \subseteq V(M(A))$) is modelling the process A state

$$\pi_S(s) \stackrel{\text{def}}{=} \langle n, \theta(\vec{v}) \rangle.$$

As in the case of modelling PTPs by LTIM_0 programs, we define the modelling program fragments, corresponding to the process A edges in order to meet

Lemma 10.4 *If* $\langle n, \delta \rangle \xrightarrow{c(d)} \xrightarrow{c:\sigma} \langle n', \delta' \rangle$ *in the PTPB process* A*, and the program variable value vector* $\vec{v} \in \mathcal{V}_{M(A)}$ *is taken such that* $\theta(\vec{v}) = \delta$*, then we have also* $\langle n, \vec{v} \rangle \xrightarrow{d:c} \langle n', \vec{v}' \rangle$ *in the program* $M(A)$ *for some* $\vec{v}'$ *with* $\theta(\vec{v}') = \delta'$.

Reverse, if $\langle n, \vec{v} \rangle \xrightarrow{d:c} \langle n', \vec{v}' \rangle$ *in the modelling program* $M(A)$ *and* $\sigma = lab(e) \in L$*, then* $\langle n, \theta(\vec{v}) \rangle \xrightarrow{c(d)} \xrightarrow{c:\sigma} \langle n', \theta(\vec{v}') \rangle$.

(The definition of the relation $\longrightarrow$ is just the same, as in the proof of Theorem 9.3, take into account that now every $M(A)$ variable value vector $\vec{v}$ is a pair $\langle \vec{v}^B, \vec{v}^T \rangle$ and that the existence of the history ν in $M(A)$ is equivalent to the existence of corresponding histories ν^B and ν^T in the programs $M(A)^B$ and $M(A)^T$ (see Section 3.2 for the definitions of the programs P^B and P^T from given LTIBA program P)).

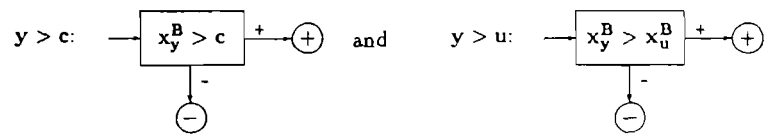
The modelling of one edge $e \in E(A)$ is also done following similar lines, as the modelling of PTP edges in the proof of Theorem 9.3, however, some technical details are additional.

Lemma 10.5 *For every linear formula* γ *over the variable set* $\mathcal{W}$ *there exists a LBASE program block* $B(\gamma)$ *with one entry and two exits (labelled by "+" and "−"), such that the computation of the block starting from the variable values*

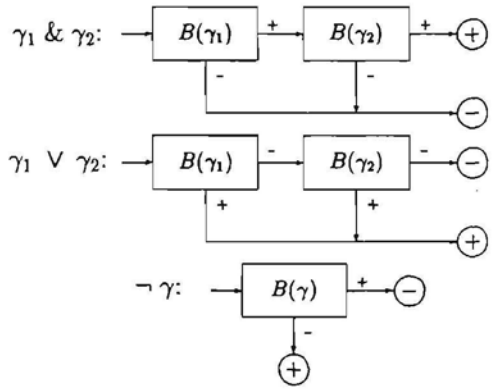
$$\vec{v}.x_y^B = \delta_B(y) \text{ for all } y \in \mathcal{W}$$

leads to the exit "+" if and only if the process variable values δ_B *satisfy the formula* γ*; otherwise the computation always leads to the "−" exit.*

Proof: The elementary predicates are translated, as follows (here and further on the little labels on the edges denote the exits of the modelling block components (program operators; blocks, corresponding to subformulae), while the labels, shown as the edge targets denote the exits of the whole block):



As to the composed formulas, their translations by the program blocks are obtained inductively on the subformula structure the following way (here $B(\gamma)$ denotes the block for translating the formula γ):



The correctness of the translation is straightforward.□

As in the modelling of PTPs, we consider an edge $e \in E(A)$, leading from n to n' in a PTPB A, let $\gamma(e) = \{t_1, \ldots, t_s\}$. The modelling block structure for the edge e in $M(A)$ is similar to that in Figure 9.3, however, some slight additions and changes are present (the input of a LBASE variable x_y^B which corresponds to the process variable $y = labw(e) \in \mathcal{W}$, the block $A0$ is moved to the end of the modelling chain for it makes use of the new value of x_y^B while the $A, B, \ldots, C$ blocks need the "old" value):

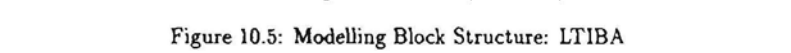
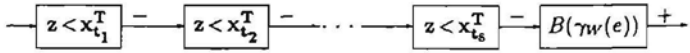


Figure 10.5: Modelling Block Structure: LTIBA

The blocks $A0$, ASS and $A, B, \ldots, C$ have the same principal roles as in Figure 9.3,

The block $A0$ checks the satisfication of the edge e conditions (both $\delta(t_i) = 0$ for $t_i \in \gamma_T(e)$ after waiting of d time units and γ_W (observe the interpretation of γ_W over the variable value assignment δ_W^0, including the value, received from the input):



The block ASS is obtained from the settings $\phi_T(e)$ and $\phi_W(e)$ as in the modelling of PTPs (for every variable x_y^B the assignment $x_y^B \leftarrow x_u^B$ for $u = \phi_W(e)(y) \in \mathcal{W}$ $x_y^B \leftarrow c$ for $c = \phi_W(e)(y) \in \mathbf{Z}$) is present (similarly for variables x_t^T for $t \in \mathcal{T}$)).

Up to this point the constructions made were possible also for the whole class of PTPBs. The problem in the modelling PTPB processes is that we cannot in general build for given red edge e a corresponding LTIBA block which allows a control flow through it if and only if *there does not exist* any input value for which the transition along e becomes enabled (because of the fulfillment of the condition $\gamma_W(e)$), the example in Theorem 10.1 is based just on this point, see Figure 10.4.

In the case of $PTPB_0$s we obtain the modelling possibility from the following important result (combined with Lemma 10.5).

Lemma 10.6 *For every linear formula γ over the set $\mathcal{W}$ of integer-valued variables, if γ is semi-closed w.r.t. $y \in \mathcal{W}$, then there exists a corresponding linear formula γ^0 over the set of variables $\mathcal{W}^0 = \mathcal{W} \setminus \{y\}$ such that*

- *γ^0 is true for fixed variable values $\delta(x)$, $x \in \mathcal{W}^0$ if and only if*
- *there exists a value $u = \delta(y) \in \mathbf{Z}$ such that γ is true for the variable values $\delta(x)$, $x \in \mathcal{W}$.*

Moreover, for every such γ the corresponding formula γ^0 can be effectively obtained from γ.

Proof: Exclude the variable y from the formula γ by replacing every chain of relations $a\lambda_1 y\lambda_2 b$ for $x_1, x_2 \in \mathcal{W}^0 \cup \mathbf{Z}$ and $\lambda_i \in \{<, \leq\}$ by $a\lambda b$, where λ is the strongest of the relations λ_i. Due to the semi-closeness of γ no more than one of λ_i is strong inequality ($<$), so in the case of the validity of γ^0 over some variable value assignment δ, also the value of y yielding the validity of γ can be found, we do not consider further details.□

Now the modification of the transition impossibility controlling blocks (Figure 9.4) is straightforward: the block, corresponding to a red edge e' with $\gamma_T(e') = \{u_1, \ldots, u_r\}$ is defined as:

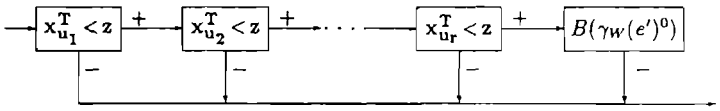


Figure 10.6: Transition Impossibility Controlling Blocks

We add into the program $M(A)$ also the initializing fragment, containing the assignments $x^T_t \leftarrow \delta^A_T(t)$ for all $t \in T(A)$ and $x^B_y \leftarrow \delta^A_B(y)$ for all $y \in W(A)$.

The rest of the proof of the theorem is very similar to the case of the basic PTPs (Theorem 9.3), its detailed consideration is omitted (since the graph $BG(M(A))$ has

not been defined for LTIBA programs, we have to use simply *some* T-projectivee for the set of all finite feasible accepting paths in $M(A)$ (obtained in the proof of Theorem 4.1 from the graphs $BG(M(A)^B)$ and $BG(M(A))^T$) by Lemma 2.8).□

Via the modelling of $PTPB_0$s by LTIBA programs (Lemma 10.4) on can prove also various interesting results about the infinite path feasibility in $PTPB_0$s (see Section 7.5 for some list of results in LTIBA program analysis).

There is also another alternative for introducing the data in the parallel timer model: we obtain the definition of a PTP with Rational data (PTPQ, for short) if we replace the integer data type of the non-timer variables in the definition of PTPBs by the *rational* one.

For the linear formulas over rational-valued variables the requirement of semi-closeness is not necessary for an analogue of Lemma 10.6 to hold. So, using a similar schema, as in the $PTPB_0$ modelling, one can model every *PTPQ* in the language, called, say, $LTIQ_0$, every program in which is allowed to have two complects of rational-valued variables: one of them with allowed $LTIM_0$ commands over them (used in handling the timed aspects of the process behaviour) and the other without the real time counter and allowed LBASQ commands (used in handling the data aspects of the process behaviour), any interaction between both the variable complects is forbidden (clearly, every $LTIQ_0$ program is also a LTIM program).

Since the programs in both the languages $LTIM_0$ and LBASQ have ω-projective infinite feasible path sets (see Corollary 7.12 and Fact 7.2), the ω-projectivity result holds also for the sets of all infinite feasible paths in $LTIQ_0$ programs due to Lemma 2.9.

Using the modelling of PTPQs by $LTIQ_0$ programs one shows the ω-projectivity of the set of all infinite feasible paths also for PTPQs (an analogue of Lemma 9.5 and Lemma 10.4 is useful for the modelling correctness proof, we do not consider the details).

In Section 13.2 we give also the principal schema of the deciding bisimulation equivalences for PTPQs (in fact we consider for the sake of simplicity only the strong equivalence), what is interesting to mention is that a similar algorithm if tried on the deciding bisimulation equivalences for $PTPB_0$s, runs into serious problems about which it is not clear for the moment, how they can be dealt with at all.

All the above considered enrichments of the PTP model still retained the decidability of, at least, the process vertex reachability problem (as it will be seen from the proofs of Chapter 11 and Chapter 12, the most of the enriched models (save the cases of not bounded PTPNs and the integer data dependent formalisms PTPB, $PTPB_0$) permit also the automated bisimulation equivalence decision). However, it is very easy to extend the PTP model also in a way to have all the interesting analysis problems, including the vertex reachability, *undecidable*, a very natural (and rather "similar" to PTPQs) such extension is considered in Chapter 15.

Chapter 11

Deciding Bisimulation Equivalences

This chapter is devoted to the deciding bisimulation equivalences for Parallel Timer Processes. After the definitions of the equivalences in Section 11.1 we consider the notion of region (symbolic) processes which play a crucial role in the deciding algorithms (Section 11.2) and show the algorithm of deciding the strong bisimulation equivalence in Section 11.3. The further arguments, needed in the deciding of the weak bisimulation equivalence, are postponed to the next chapter.

11.1 Strong and Weak Equivalences

Let $\Lambda \stackrel{\text{def}}{=} L \cup \{\epsilon(d) | d \in \mathbf{Q}^{+0}\}$ be the set of all actions ranged over by ν.

We define the strong *timed bisimulation equivalence* in the set $\mathcal{P}$ of timed processes in the usual way (see [HLW91]):

Definition 11.1 *Let $F(R)$ be the set of all $\langle P, Q\rangle \in \mathcal{P} \times \mathcal{P}$ satisfying*

i) whenever $P \xrightarrow{\nu} P'$ then $Q \xrightarrow{\nu} Q'$ with $\langle P', Q'\rangle \in R$ for some Q',
ii) whenever $Q \xrightarrow{\nu} Q'$ then $P \xrightarrow{\nu} P'$ with $\langle P', Q'\rangle \in R$ for some P'.

Then R is a timed bisimulation if $R \subseteq F(R)$. We define the timed bisimulation equivalence, written $\sim$, to be the greatest fixpoint of F.

We prove in this chapter the following theorem.

Theorem 11.2 *There is an algorithm which, given two Parallel Timer Processes A and B, decides whether $A \sim B$ or not.*

It has been proved to be often useful to abstract from certain actions when observing a process behavior, this leads to the notion of weak (abstracted) bisimulation equivalence in process algebras. In our case with the semantics of processes given by the timed transition systems we face two principal alternatives whether to abstract from *timed* transitions in the process behaviour or not.

Here we choose to have the timed transitions (i.e. $\xrightarrow{\epsilon(d)}$) non-abstractable (as it was done in [Wan91b]). As to the time abstracted case we just note that the equivalence obtained this way still depends on the timing constraints on the behaviour of the process: these constraints can eliminate some still observable action sequences as impossible. The reader may himself convince that this equivalence can be shown decidable by using the path set projectivity techniques, demonstrated for PTPs in Section 9.2, combined with the standard bisimulation deciding algorithms for Finite State Machines (see e.g. [KS83]).

Let $P \stackrel{\epsilon}{\Longrightarrow} Q$ if and only if $P(\xrightarrow{\tau})^* Q$ (i.e. P may become Q by doing a *sequence* of $\xrightarrow{\tau}$-transitions). Define for $\alpha \in L_0$ the relation $P \stackrel{\alpha}{\Longrightarrow} Q$ as $P \stackrel{\epsilon}{\Longrightarrow} \xrightarrow{\alpha} \stackrel{\epsilon}{\Longrightarrow} Q$. For delay transitions let, as in [Wan90] $P \stackrel{\epsilon(d)}{\Longrightarrow} Q$ whenever

$$P \stackrel{\epsilon}{\Longrightarrow} \xrightarrow{\epsilon(d_1)} \stackrel{\epsilon}{\Longrightarrow} \cdots \xrightarrow{\epsilon(d_k)} \stackrel{\epsilon}{\Longrightarrow} Q$$

for some $d_1, d_2, \ldots, d_k$ with $d_1 + d_2 + \cdots d_k = d$.

Intuitively, $\stackrel{\epsilon(d)}{\Longrightarrow}$ is rather complex relation for it allows the abstracted waiting to be split into elementary waitings by the τ-actions at arbitrary time moments and without a priori bound on the number of the splitting moments (we afterwards show such an a posteriori bound, see Lemma 12.4).

Letting ν to range over $L_0 \cup \{\epsilon(d) \mid d \in \mathbf{Q}^{+0}\}$ we can now define the weak timed bisimulation for timed processes (observe that $P \stackrel{\epsilon}{\Longrightarrow} Q$ iff $P \stackrel{\epsilon(0)}{\Longrightarrow} Q$):

Definition **11.3** *Let $F(R)$ be the set of all $\langle P, Q\rangle \in \mathcal{P} \times \mathcal{P}$ satisfying*

i) whenever $P \stackrel{\nu}{\Longrightarrow} P'$ then $Q \stackrel{\nu}{\Longrightarrow} Q'$ with $\langle P', Q'\rangle \in R$ for some Q',
ii) whenever $Q \stackrel{\nu}{\Longrightarrow} Q'$ then $P \stackrel{\nu}{\Longrightarrow} P'$ with $\langle P', Q'\rangle \in R$ for some P'.

Then R is a weak timed bisimulation, if $R \subseteq F(R)$. We define the weak timed bisimulation equivalence, written $\approx$, to be the greatest fixpoint of F.

Theorem **11.4** *There is an algorithm which, given two Parallel Timer Processes A and B, decides whether $A \approx B$ or not.*

Observe that the result of Theorem 11.2 follows from that of Theorem 11.4 in the case of no edges in the processes A and B are labelled with the τ-actions.

Let for a given finite process $P = \langle \Phi, \langle v, \delta \rangle \rangle$ $\tilde{d}(P)$ be the set of processes $\langle \Phi, \langle v', \delta' \rangle \rangle$ with v' being a vertex in Φ graph and $\delta'(t) \leq c^P$ for every P timer t, where c^P is defined to be a constant which exceeds both all timer values from δ and all constants used in Φ edge assignments. It is easy to see that every P derivative (= a process which can be obtained from P by performing some sequence of transitions) falls into the set $\tilde{d}(P)$ (the converse might not be true). So, without loosing generality in deciding whether $A \sim B$, $A \approx B$ one can consider $\sim$ and $\approx$ to be the maximal bisimulations in the set $\tilde{d}(A) \times \tilde{d}(B) \subseteq \mathcal{P} \times \mathcal{P}$.

11.2 Symbolic Processes

Without the loss of generality we assume that all the explicit constants c, used in the edge timer value assignment vectors $\phi(e)$ in the graphs of A and B, are *integers* (were it not so one could change the scale of the number line to ensure it; easy to see that the behaviour of the processes is not affected by the scale change).

In order to decide whether $A \sim B$, $A \approx B$ we give an effective characteristic (by the means of a finite partitioning) of all bisimilar process pairs within $\tilde{d}(A) \times \tilde{d}(B)$ in a way described below (it turns out to be too rough for the proof to consider partitionings of $\tilde{d}(A)$ and $\tilde{d}(B)$ independently: the proof cornerstone Lemma 11.7 does not hold for *any* nontrivial partitioning of $\tilde{d}(A) \times \tilde{d}(B)$ which is obtained as the product of independent partitionings of $\tilde{d}(A)$ and $\tilde{d}(B)$).

For $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ being a finite set of timers, let us represent every $\mathcal{T}$ timer value assignment $\delta : \mathcal{T} \to \mathbf{Q}^{+0}$ as the vector $\langle \delta(t_1), \ldots, \delta(t_m) \rangle \in (\mathbf{Q}^{+0})^m$. Let $\cong$ be an equivalence relation in the set $\Delta_{\mathcal{T}}$ of $\mathcal{T}$ timer value assignments such that $\delta^1 \cong \delta^2$ if and only if

- $\lfloor \delta^1(t_i) \rfloor = \lfloor \delta^2(t_i) \rfloor$ for every $i = 1, \ldots, n$ and
- for every i, j
  - $\{\delta^1(t_i)\} \geq \{\delta^1(t_j)\}$ if and only if $\{\delta^2(t_i)\} \geq \{\delta^2(t_j)\}$
  - $\{\delta^1(t_i)\} = 0$ if and only if $\{\delta^2(t_i)\} = 0$

(here $\lfloor x \rfloor$ denotes the "integral part" of x, i.e. the largest integer, which is not greater than x, and $\{x\}$ stands for the fractional part of x (i.e. $\{x\} = x - \lfloor x \rfloor$)).

Given $\delta \in \Delta_{\mathcal{T}}$, let us denote the equivalence class $C \subseteq \Delta_{\mathcal{T}}$ w.r.t. $\cong$ with $\delta \in C$ by $C(\delta)$ and call it the time region of $\mathcal{T}$, corresponding to δ.

Given a timer value assignment one can easily compute its corresponding time region (one can use the time region representations, say, by linear inequality systems in order to make all computations with them effective).

Example 11.5 *If* $\mathcal{T} = \{t_1, t_2, \ldots, t_7\}$ *and* $\delta = (0.7,\ 1,\ 1.23,\ 4,\ 17.23,\ 17.75,\ 17.75)$, *then the time region* $C(\delta)$ *can be described as an inequality system*
$C(\delta) = (0 < t_1 < 1 = t_2 < t_3 < 2 < 4 = t_4 < 17 < t_5 < t_6 = t_7 < 18,$
$0 = \{t_2\} = \{t_4\} < \{t_3\} = \{t_5\} < \{t_1\} < \{t_6\} = \{t_7\}).$

Clearly, if for every $t \in \mathcal{T}$ and every timer value assignment $\delta \in \Delta \subseteq \Delta_{\mathcal{T}}$ always $\delta(t) \in [0, c] \subseteq \mathbf{Q}^{+0}$, then the set of the corresponding time regions $\{C(\delta) \mid \delta \in \Delta\}$ is *finite*.

The presented time region construction is actually the same, as used in [ACD90] for demonstrating the effectivity of the model checking procedure over Timed Graphs. One may see also [ABBCK91] to find out other situations, where similar ideas of variable value space partitioning have worked in deciding reachability for various classes of data-dependent programs.

Definition 11.6 *Let* $P_i = \langle \Phi_A, \langle v^{P_i}, \delta^{P_i} \rangle \rangle \in \tilde{d}(A)$, $Q_i = \langle \Phi_B, \langle v^{Q_i}, \delta^{Q_i} \rangle \rangle \in \tilde{d}(B)$, *we say that* $\langle P_1, Q_1 \rangle \cong \langle P_2, Q_2 \rangle$ *iff*

- $v^{P_1} = v^{P_2}$ *and* $v^{Q_1} = v^{Q_2}$ *(i.e. the vertexes of corresponding processes coincide) and*
- $\vec{\delta}^{P_1} :: \vec{\delta}^{Q_1} \cong \vec{\delta}^{P_2} :: \vec{\delta}^{Q_2}$, *where* :: *denotes the concatenation of two vectors.*

It is important to notice that for $\langle P_1, Q_1 \rangle \cong \langle P_2, Q_2 \rangle$ it is not enough to have $v^{P_1} = v^{P_2}, v^{Q_1} = v^{Q_2}$ and $\delta^{P_1} \cong \delta^{P_2}, \delta^{Q_1} \cong \delta^{Q_2}$, one needs also the timer values in P_1 to be ordered w.r.t. the timer values in Q_1 the same way as the timer values in P_2 are ordered w.r.t. those in Q_2.

Lemma 11.7 *Let* $P_1, P_2 \in \tilde{d}(A)$, $Q_1, Q_2 \in \tilde{d}(B)$, *such that* $\langle P_1, Q_1 \rangle \cong \langle P_2, Q_2 \rangle$. *Then* $P_1 \approx Q_1$ *if and only if* $P_2 \approx Q_2$.

Proof: Let us give another characteristic of the equivalent process pairs:

Definition 11.8 *We call a mapping* $\rho : \mathbf{Q}^{+0} \to \mathbf{Q}^{+0}$ *uniform if*

- ρ *is strongly monotone (*$x > y$ *implies* $\rho(x) > \rho(y)$*),*
- $\rho(x) + c = \rho(x + c)$ *for every natural* c *and every* $x \in \mathbf{Q}^{+0}$,
- $\rho(0) = 0$.

We extend any mapping $\rho : \mathbf{Q}^{+0} \to \mathbf{Q}^{+0}$ in a polymorphic manner to any structures containing nonnegative rationals as elements in such a way that the mapping

ρ is applied to every component $a \in \mathbf{Q}^{+0}$ of the structure and does not change any component of other type, for example.

$$\rho(\delta_1, \delta_2, \ldots, \delta_m) = (\rho(\delta_1), \rho(\delta_2), \ldots, \rho(\delta_m)),$$

as well as for $P \in \tilde{d}(A)$, $Q \in \tilde{d}(B)$ $\rho(P,Q\rangle = \langle P', Q'\rangle$, where P' and Q' have the same vertices as P and Q respectively, but the corresponding timer vector $\vec{\delta}^{P'} :: \vec{\delta}^{Q'} = \rho(\vec{\delta}^{P} :: \vec{\delta}^{Q})$, etc. The proofs of the following facts easily follow from definitions:

Fact 11.9 *$\langle P_1, Q_1\rangle \cong \langle P_2, Q_2\rangle$ if and only if there exists a uniform mapping ρ, such that $\langle P_1, Q_1\rangle = \rho(\langle P_2, Q_2\rangle)$.*

Fact 11.10 *Whenever $\rho : \mathbf{Q}^{+0} \to \mathbf{Q}^{+0}$ is a uniform mapping, then for every $d \in \mathbf{Q}^{+0}$ the mapping ρ_d, defined $\rho_d(x) = \rho(x + d) - \rho(d)$ for every x, is also uniform.*

Fact 11.11 *Whenever $\rho : \mathbf{Q}^{+0} \to \mathbf{Q}^{+0}$ is a uniform mapping, then for every $d, d' \in \mathbf{Q}^{+0}$ and every $x \in \mathbf{Q}^{+0}$*

$$(\rho_d)_{d'}(x) = \rho_{d+d'}(x).$$

Proof:

$(\rho_d)_{d'}(x) = \rho_d(x + d') - \rho_d(d') =$
$= \rho(x + d + d') - \rho(d) - (\rho(d + d') - \rho(d)) =$
$= \rho(x + d + d') - \rho(d + d') = \rho_{d+d'}(x).\square$

Proposition 11.12 *Whenever $P_2 = \rho(P_1)$ for the processes $P_1, P_2 \in \tilde{d}(A) \cup \tilde{d}(B)$ and some uniform mapping ρ, then,*

- *if $P_1 \xrightarrow{a} P_1'$, then $P_2 \xrightarrow{a} \rho(P_1')$,*
- *if $P_1 \xrightarrow{\epsilon(d)} P_1'$, then $P_2 \xrightarrow{\epsilon(\rho(d))} \rho_d(P_1')$,*
- *if $P_1 \stackrel{a}{\Longrightarrow} P_1'$, then $P_2 \stackrel{a}{\Longrightarrow} \rho(P_1')$,*
- *if $P_1 \stackrel{\epsilon(d)}{\Longrightarrow} P_1'$, then $P_2 \stackrel{\epsilon(\rho(d))}{\Longrightarrow} \rho_d(P_1')$.*

Proof: Consider first the untimed transitions. Since P_1 and P_2 have the same graph vertex, as well as the same timers with 0 values and, so, the same transitions along the same edges enabled, the result follows by observing that for every possible newly appearing timer value $c \in N$ (remember scaling!) $\rho(c) = c$ (because of the ρ uniformity), use induction along the $(\xrightarrow{a})^*$ derivation for the transition $\stackrel{a}{\Longrightarrow}$.

As to the timed transitions consider first the case $P_1 \xrightarrow{\epsilon(d)} P_1'$. Let P_1 have a state $\langle v, \delta\rangle$, then the state of P_2 is $\langle v, \rho(\delta)\rangle$. By the definition of $\xrightarrow{\epsilon(d)}$ for every red edge e.

outgoing from v, there exists $t_i \in \gamma(e)$, such that $\delta(t_i) \geq d$. By the monotonicity of ρ for every such t_i $\rho(\delta(t_i)) \geq \rho(d)$, so $P_2 \xrightarrow{\epsilon(\rho(d))} P_2'$ for some P_2'.

In order to prove that $P_2' = \rho_d(P_1')$ it remains to consult the definitions of the transition relation $\xrightarrow{\epsilon(d)}$ and the mapping ρ_d (for every P_1 timer t_i consider 2 cases whether $\delta(t_i) \leq d$ or $\delta(t_i) > d$, in both the result follows easily).

The general case of $P_1 \stackrel{\epsilon(d)}{\Longrightarrow} P_1'$ follows from Fact 11.11 by the induction along the elementary transition chain in the derivation $\stackrel{\epsilon(d)}{\Longrightarrow}$.□

Let $P \approx' Q$ if and only if $P_1 \approx Q_1$ for some $\langle P_1, Q_1 \rangle \cong \langle P, Q \rangle$. By showing that $\approx'$ is a weak bisimulation we complete the proof of the lemma.

Take some $P \approx' Q$, let $P_1 \approx Q_1$ and $\langle P_1, Q_1 \rangle \cong \langle P, Q \rangle$, then $\langle P_1, Q_1 \rangle = \rho(\langle P, Q \rangle)$ for some uniform ρ (Fact 11.9). By Proposition 11.12 whenever $P \stackrel{o}{\Longrightarrow} P'$ then $P_1 \stackrel{o}{\Longrightarrow} \rho(P')$. Since $P_1 \approx Q_1$, then also $Q_1 \stackrel{o}{\Longrightarrow} Q_1'$ for some Q_1' with $\rho(P') \approx Q_1'$. Since the inverse of a uniform mapping is also uniform, Proposition 11.12 gives $Q \stackrel{o}{\Longrightarrow} \rho^{-1}(Q_1')$, easy to see that $\langle P', \rho^{-1}(Q_1') \rangle \cong \langle \rho(P'), Q_1' \rangle$ and so $P' \approx' \rho^{-1}(Q_1')$, as requested.

All the other cases (including the timed ones) are very similar to the considered one, their detailed analysis is omitted.□

Now let us consider a partitioning $\mathcal{X}_{A,B}$ of the set $\tilde{d}(A) \times \tilde{d}(B)$, generated by $\cong$, easy to see that it is finite (for every $P \in \tilde{d}(A) \cup \tilde{d}(B)$ any its timer value does not exceed $max\{c^A, c^B\}$). For arbitrary $P \in \tilde{d}(A)$, $Q \in \tilde{d}(B)$ let us denote by $X(P, Q)$ the element in this partitioning to which the pair $\langle P, Q \rangle$ belongs to and call it a *region process*, corresponding to $\langle P, Q \rangle$ (one could call the region processes also symbolic processes due to their symbolic nature).

For the completion of the proof of Theorem 11.4 it suffices to check whether for a given set $\mathcal{X} \subseteq \mathcal{X}_{A,B}$ the corresponding set of process pairs $R_{\mathcal{X}} = \{\langle P, Q \rangle \mid X(P, Q) \in \mathcal{X}\}$ is a bisimulation (observe that there is only a finite number of different symbolic process sets $\mathcal{X} \subseteq \mathcal{X}_{A,B}$), we deal with these problems in Section 11.3 for the case of strong bisimulation and in Chapter 12 for the weak bisimulation (see especially Section 12.2).

11.3 Deciding Strong Equivalence

For the sake of simplicity we consider first the deciding of the strong (i.e. non-abstracted) bisimulation equivalence. We begin with some results, characterizing the "waiting behaviour" of the processes.

Let for $P \in \mathcal{P}$ $P(d)$ be the process which is obtained from the process P by letting time to pass for d units ($P \xrightarrow{\epsilon(d)} P(d)$) provided P can perform such a waiting (observe

that according to the semantics of timed processes (time-determinacy property) such process $P(d)$ is always *unique* for every P and d).

We let $\mu(P)$ for $P \in \mathcal{P}$ to denote the minimal nonzero P timer value fractional part (if all timer values in P are integers, let $\mu(P) = 1$), let $\mu(P,Q) = min\{\mu(P), \mu(Q)\}$. We call a process P *stable*, written $P \xrightarrow{WT}$, if and only if there exists $d > 0$, such that $P \xrightarrow{\epsilon(d)} P(d)$.

Fact 11.13 *For $P \in \mathcal{P}$, if $P \xrightarrow{WT}$, then for all $d \leq \mu(P)$ $P \xrightarrow{\epsilon(d)} P(d)$.*
For $P \in \tilde{d}(A)$ and $Q \in \tilde{d}(B)$, if $P \xrightarrow{WT}$ and $Q \xrightarrow{WT}$ then for all $d, d' \in]0, \mu(P,Q)[$ always $P(d), Q(d), P(d'), Q(d')$ exist and $X(P(d), Q(d)) = X(P(d'), Q(d'))$.

Proof: Follows from the semantics of processes and the definition of region processes (equivalence relation $\cong$).□

Definition 11.14 *Let for $X = X(P,Q) \in \mathcal{X}_{A,B}$ with $P \xrightarrow{WT}$ and $Q \xrightarrow{WT}$*

- $next_0(X) = X(P(\mu/2), Q(\mu/2))$ *and*
- $next_1(X) = X(P(\mu), Q(\mu))$, *where* $\mu = \mu(P,Q)$.

According to Proposition 11.12 and since for $\langle P', Q' \rangle = \rho(\langle P'', Q'' \rangle)$ for uniform ρ always $\mu(P', Q') = \rho(\mu(P'', Q''))$, we obtain that the operation $next_1$ for region processes is well (uniquely) defined. In order to obtain the well-definedness of $next_0$, consider also the monotonicity of ρ and Fact 11.13.

The following result is used below (see Lemma 11.17) to obtain a symbolic characteristic of waiting for region processes.

Lemma 11.15 *Given a set $R_{\mathcal{X}} = \{\langle P,Q \rangle \mid X(P,Q) \in \mathcal{X}\}$ with $\mathcal{X} \subseteq \mathcal{X}_{A,B}$, the following two statements are equivalent:*

1. *for every $\langle P,Q \rangle \in R_{\mathcal{X}}$ and every $d > 0$ whenever $P \xrightarrow{\epsilon(d)} P(d)$ then also $Q \xrightarrow{\epsilon(d)} Q(d)$ and $\langle P(d), Q(d) \rangle \in R_{\mathcal{X}}$; and*
2. *for every $X = X(P,Q) \in \mathcal{X}$, if $P \xrightarrow{WT}$, then both $next_0(X) \in \mathcal{X}$ and $next_1(X) \in \mathcal{X}$.*

Proof: Consider (1.) $\Rightarrow$ (2.) and assume $\langle P,Q \rangle$ fixed. Since $P \xrightarrow{WT}$, according to Fact 11.13 for all $d \in [0, \mu(P,Q)]$ $P \xrightarrow{\epsilon(d)} P(d)$. According to (1.) we obtain that also $Q \xrightarrow{\epsilon(d)} Q(d)$ for all $d \in [0, \mu(P,Q)]$, moreover $\langle P(\mu(P,Q)/2), Q(\mu(P,Q)/2) \rangle \in R_{\mathcal{X}}$, hence $next_0(P,Q) \in \mathcal{X}$, and $\langle P(\mu(P,Q)), Q(\mu(P,Q)) \rangle \in R_{\mathcal{X}}$, so, $next_1(P,Q) \in \mathcal{X}$.

As to (2.) $\Rightarrow$ (1.), assume $\langle P,Q \rangle \in R_{\mathcal{X}}$ and $d \in \mathbf{Q}^{+0}$ with $P \xrightarrow{\epsilon(d)} P(d)$ fixed, and consider the sequence

$$\langle P_0, Q_0 \rangle, \langle P_1, Q_1 \rangle, \langle P_2, Q_2 \rangle, \ldots \text{ where}$$

- $P_0 = P$ and $Q_0 = Q$, as well as
- $P_{i+1} = P_i(\mu_i)$ and $Q_{i+1} = Q_i(\mu_i)$ (we abbreviate $\mu_i = \mu(P_i, Q_i)$).

According to (2.) we inductively establish that $\langle P_i, Q_i \rangle \in R_{\mathcal{X}}$ for all i all the time the elements $\langle P_i, Q_i \rangle$ can be defined. If at some stage either $P_k(\mu_k)$, or $Q_k(\mu_k)$ does not exist, we have $\langle P_k, Q_k \rangle \in R_{\mathcal{X}}$, so we conclude according to (2.) and Fact 11.13 that the process P cannot continue its waiting: $P_k \stackrel{WT}{\not\longrightarrow}$, in this case we have $\mu_0 + \mu_1 + \ldots + \mu_{k-1} \geq d$.

Since for every i the sum $\mu_0 + \mu_1 + \ldots + \mu_i$ can be expressed as $f + c$ for f being a fractional part of some P or Q timer value and $c \in \mathbf{N}$ (it follows from the semantics of timed processes), as well as both the processes P and Q have just finite number of different timer value fractional parts, no more than finite number of the elements in the defined sequence can occur before the time moment d is reached.

So we have either $P(d) = P_i$ and $Q(d) = Q_i$ for some i, or $P(d) = P_i(\mu')$ and $Q(d) = Q_i(\mu')$ with $\mu' < \mu_i$, in both cases we obtain $\langle P(d), Q(d) \rangle \in R_{\mathcal{X}}$ according to (2.) (observe that $X(P_i(\mu'), Q_i(\mu')) = X(P_i(\mu_i/2), Q_i(\mu_i/2))$ according to Fact 11.13).□

Let us introduce in the set $\mathcal{X}_{A,B}$ of region processes the relations $\stackrel{a}{\longrightarrow}$ for $\sigma \in L$ in a way $X \stackrel{a}{\longrightarrow} X'$ iff there exist $\langle P, Q \rangle \in X$ and $\langle P', Q' \rangle \in X'$ such that $P \stackrel{a}{\longrightarrow} P'$ and $Q \stackrel{a}{\longrightarrow} Q'$. We use the following notion of symbolic bisimulation in order to give an efficient characteristic of the set of all bisimilar process pairs $\langle P, Q \rangle \in \tilde{d}(A) \times \tilde{d}(B)$:

Definition 11.16 *The set $\mathcal{X} \in \mathcal{X}_{A,B}$ is a strong symbolic bisimulation if and only if for all $X = X(P, Q) \in \mathcal{X}$*

- *whenever $P \stackrel{a}{\longrightarrow} P'$ then $X \stackrel{a}{\longrightarrow} X(P', Q') \in \mathcal{X}$ for some Q';*
- *whenever $Q \stackrel{a}{\longrightarrow} Q'$ then $X \stackrel{a}{\longrightarrow} X(P', Q') \in \mathcal{X}$ for some P';*
- *whenever $P \stackrel{WT}{\longrightarrow}$, or $Q \stackrel{WT}{\longrightarrow}$, then both $next_0(X) \in \mathcal{X}$ and $next_1(X) \in \mathcal{X}$.*

Since every (strong) bisimulation $R_{\mathcal{X}} \subseteq \tilde{d}(A) \times \tilde{d}(B)$ contains together with every pair of timed processes $\langle P, Q \rangle$ also all process pairs $\langle P', Q' \rangle$ with $X(P', Q') = X(P, Q)$ (see Lemma 11.7) and the set $\mathcal{X}_{A,B}$ of all region processes is finite, the following two results complete the proof of Theorem 11.2 (decidability of strong timed bisimulation equivalence for PTPs).

Lemma 11.17 *The set $R_{\mathcal{X}} = \{\langle P, Q \rangle \mid X(P, Q) \in \mathcal{X}\}$ is a strong timed bisimulation if and only if the set $\mathcal{X}$ is a strong symbolic bisimulation.*

Proof: Easily obtained from Lemma 11.15, Proposition 11.12 and the definitions.□

Lemma 11.18 *It is decidable whether a given set $\mathcal{X} \subseteq \mathcal{X}_{A,B}$ is a strong symbolic bisimulation.*

Proof: In order to check, whether $\mathcal{X}$ is a symbolic bisimulation (i.e. whether $R_{\mathcal{X}}$ is a strong timed bisimulation), we take for every $X \in \mathcal{X}$ a representative $\langle P, Q \rangle \in X$. Since for every $\sigma \in L$ the set of σ-derivatives P' of every given timed process P_0 is finite (i.e. the set $\{P' \in \mathcal{P} \mid P_0 \xrightarrow{\sigma} P'\}$), the check whether for all P', σ with $P \xrightarrow{\sigma} P'$ there exists Q' with both $Q \xrightarrow{\sigma} Q'$ and $X(P', Q') \in \mathcal{X}$ (as well as the case with Ps and Qs in changed roles) is effective. As to the case of timed transitions, the check, whether for given $X \in \mathcal{X}_{A,B}$ with either $P \xrightarrow{WT}$ or $Q \xrightarrow{WT}$ for some (=for every) representative $\langle P, Q \rangle \in X$ the processes $next_0(X)$ and $next_1(X)$ exist and belong to $\mathcal{X}$ are clearly effective.□

Chapter 12

Deciding Weak Equivalence

In this chapter we complete the proof of Theorem 11.4 (decidability of weak bisimulation equivalence for Parallel Timer Processes). We begin with some technical points about the timer value inheritance.

12.1 Timer Value Inheritance

In this section we briefly consider some technical points regarding the timer value maintenance along the process derivations. The results of this section will be used to prove the correctness of the weak bisimulation deciding algorithms below.

Definition 12.1 *Let $P \in \mathcal{P}$ and $P \stackrel{c(d)}{\Longrightarrow} P_d$. We say that a process P_d timer t inherits the value from the process P timer $t^0 \in \mathcal{T}(P)$ along the given derivation $P \stackrel{c(d)}{\Longrightarrow} P_d$, if for every process P' contained in this derivation and having $P \stackrel{c(d')}{\Longrightarrow} P'$ there exists a timer $t' \in \mathcal{T}(P)$ with $\delta^{P'}(t') = \delta^{P}(t^0) - d'$ (so, also $\delta^{P_d}(t) = \delta^{P}(t^0) - d$).*

Proposition 12.2 *If $P \stackrel{c(d)}{\Longrightarrow} P_d$ for $d < 1$, then for every P_d timer $t \in \mathcal{T}(P_d) = \mathcal{T}(P)$ with $0 < \{\delta^{P_d}(t)\} < 1 - d$ the timer t inherits its value from some P timer $t^0 \in \mathcal{T}(P)$ with $\{\delta^{P}(t^0)\} > 0$.*

Proof: Assume the contrary and look for the *earliest* process P' in the given derivation $P \stackrel{c(d)}{\Longrightarrow} P_d$ such that the P_d timer t inherits its value from some P' timer t', let $P' \stackrel{c(d')}{\Longrightarrow} P_d$ for $d' \leq d$. Since P' is the earliest such process, we infer, according to the semantics of timed processes and the agreement to have all constants used in P timer settings integers, that $\delta^{P'}(t') \in \mathbf{N}$. The definition of the timer value inheritance gives us $\delta^{P_d}(t) = \delta^{P'}(t') - d'$, due to $d' \leq d$ we obtain $\{\delta^{P_d}(t)\} \geq 1 - d$, a contradiction (the case $d' = 0$ is excluded by the proposition requirements).□

Fact 12.3 *Whenever* $P \stackrel{\epsilon(d)}{\Longrightarrow} P_d$, $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ *for* $d < \mu(P,Q)$ *we have* $\mu(P_d, Q_d) \geq \mu(P,Q) - d$.

Proof: Straightforward from Proposition 12.2 and the definition of $\mu(P,Q)$.□

Lemma 12.4 *Let for a sequence of process pairs* $((P_i, Q_i))_{i \in I}$ *with* $I = \mathbf{N}$, *or* $I = 0, 1, \ldots, s$ *for some* $s \in \mathbf{N}$, *we have* $P_i \stackrel{\epsilon(d_i)}{\Longrightarrow} P_{i+1}$ *and* $Q_i \stackrel{\epsilon(d_i)}{\Longrightarrow} Q_{i+1}$ *with* $d_i = \mu(P_i, Q_i)$ *for all* i. *Then for* $k^P = card(\mathcal{T}(P_0))$, $k^Q = card(\mathcal{T}(Q_0))$ *and* $i > u(k^P + k^Q + 1)$ *with* $u \in \mathbf{N}$-*arbitrary we have* $d_0 + d_1 + \cdots + d_{i-1} \geq u$.

Proof: Let for every i $d_0 + d_1 + \cdots + d_{i-1} = p_i$. According to Proposition 12.2 we obtain that, if $p_{i+1} = p_i + \mu(P_i, Q_i) < 1$, then every P_i timer t' with $\{\delta^{P_i}(t')\} = \mu(P_i, Q_i)$ inherits its value from some P_0 timer t with $\{\delta^{P_0}(t)\} > 0$, as well as every Q_i timer t' with $\{\delta^{Q_i}(t')\} = \mu(P_i, Q_i)$ inherits its value from some Q_0 timer t with $\{\delta^{Q_0}(t)\} > 0$.

Since the processes P_0 and Q_0 have together no more than $k^P + k^Q$ different timer value fractional parts, the delays $p_i \in]0,1[$ can be for no more that $k^P + k^Q$ different indices i, so we have $p_{k^P+k^Q+1} > 1$. We can now repeat the construction by taking $\langle P_{k^P+k^Q+1}, Q_{k^P+k^Q+1} \rangle$ instead of $\langle P_0, Q_0 \rangle$ as many times, as needed, so obtaining the result of the lemma.□

12.2 Deciding Weak Equivalence

For the deciding whether the processes A and B are weakly bisimilar, we follow the same principal lines, as in the deciding strong equivalence, considered in Section 11.3. We define first the generalized ("weak") versions of the *next* operators for process pairs.
Let for $P \in \tilde{d}(A)$, $Q \in \tilde{d}(B)$ for $\mu = \mu(P,Q)$

- whenever $P \stackrel{WT}{\longrightarrow}$ then $\mathcal{N}_0^A(P,Q) = \{X(P(d), Q_d) \mid Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d \ \& \ 0 < d < \mu\}$ and $\mathcal{N}_1^A(P,Q) = \{X(P(\mu), Q') \mid Q \stackrel{\epsilon(\mu)}{\Longrightarrow} Q'\}$, as well as
- whenever $Q \stackrel{WT}{\longrightarrow}$ then $\mathcal{N}_0^B(P,Q) = \{X_d, Q(d)) \mid P \stackrel{\epsilon(d)}{\Longrightarrow} P_d \ \& \ 0 < d < \mu\}$ and $\mathcal{N}_1^B(P,Q) = \{X(P', Q(\mu)) \mid P \stackrel{\epsilon(\mu)}{\Longrightarrow} P'\}$

Fact 12.5 *For* $\langle P,Q \rangle \cong \langle P', Q' \rangle$ *always* $\mathcal{N}_j^I(P,Q) = \mathcal{N}_j^I(P', Q')$, *where* $I = A, B$ *and* $j = 0, 1$.

Proof: Follows from Proposition 11.12.□

Fact 12.6 *For every $X \in \mathcal{N}_0^A(P,Q)$ and for every $d \in]0, \mu(P,Q)[$ there exists Q_d with both $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ and $\langle P(d), Q_d\rangle \in X$.*

Proof: According to the definition of $\mathcal{N}_0^A$ and the transition relation $\stackrel{\epsilon(d)}{\Longrightarrow}$ we have

$$Q \stackrel{\epsilon}{\Longrightarrow} \stackrel{\epsilon(d_1)}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} \stackrel{\epsilon(d_2)}{\longrightarrow} \ldots \stackrel{\epsilon(d_x)}{\longrightarrow} \stackrel{\epsilon}{\Longrightarrow} Q_{d'}$$

with $d_i > 0$ and $d_1 + d_2 + \ldots + d_x = d'$ for some $d' \in]0, \mu(P,Q)[$.

It is easy to see by the induction along the $Q_{d'}$ derivation that if we change in the derivation of $Q_{d'}$ the values of d_i to d_i's in a way $d_i' > 0$ and $d_1' + d_2' + \ldots + d_x' = d$, we obtain the needed derivation of Q_d with $X(P(d), Q_d) = X(P(d'), Q_{d'})$.□

A similar result can be obtained also for the set $\mathcal{N}_0^B$.

Note: In the case both the processes A and B satisfy the *maximal progress* assumption, stating that τ-labels in the processes can be ascribed only to red (instantaneous) edges (i.e. the internal actions in the processes *must* occur as soon as enabled), the defined sets $\mathcal{N}_0^A(P,Q)$, $\mathcal{N}_1^A(P,Q)$, $\mathcal{N}_0^B(P,Q)$ and $\mathcal{N}_1^B(P,Q)$, and, so, also all the deciding algorithm and its correctness proof given below become much simpler (we could have, for instance, that $\mathcal{N}_A^0(P,Q) = \{X(P(d), Q_d) \mid Q \stackrel{\epsilon}{\Longrightarrow} Q' \stackrel{\epsilon(d)}{\longrightarrow} Q_d \;\&\; 0 < d < \mu\}$). However, since the overall proof follows the same principal ideas as the proof of Theorem 11.2 (see Section 11.3), we consider just the general case and leave to find the simplifications, obtained in the maximal progress case, for the reader himself.

The following lemma plays an important role in obtaining symbolic characteristics of weak bisimulations $R_{\mathcal{X}} \subseteq \tilde{d}(A) \times \tilde{d}(B)$.

Lemma 12.7 *Given a set $\mathcal{X} \subseteq \mathcal{X}_{A,B}$ with $R_{\mathcal{X}} = \{\langle P,Q\rangle \mid X(P,Q) \in \mathcal{X}$ the following two statements are equivalent:*

1. *for every $\langle P,Q\rangle \in R_{\mathcal{X}}$ and every $d > 0$, if $P \stackrel{\epsilon(d)}{\longrightarrow} P(d)$ then $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ for some Q_d with $\langle P(d), Q_d\rangle \in R_{\mathcal{X}}$; and*
2. *for every $X(P,Q) \in \mathcal{X}$, if $P \stackrel{WT}{\longrightarrow}$, then both $X' \in \mathcal{N}_0^A \cap \mathcal{X}$ and $X'' \in \mathcal{N}_1^A \cap \mathcal{X}$ for some X', X''.*

Proof: (1.) ⇒ (2.) follows from Fact 11.13 and the definitions of $\mathcal{N}_0^A$ and $\mathcal{N}_1^A$.
As to (2.) ⇒ (1.) assume $\langle P,Q\rangle \in R_{\mathcal{X}}$ and consider the sequence

$$\langle P_0, Q_0\rangle,\ \langle P_1, Q_1\rangle,\ \langle P_2, Q_2\rangle, \ldots, \text{ where}$$

- $P_0 = P$ and $Q_0 = Q$ (so, $X(P_0, Q_0) \in \mathcal{X}$), as well as

- $P_{i+1} = P_i(\mu_i)$ and $Q_i \stackrel{\epsilon(\mu_i)}{\Longrightarrow} Q_{i+1}$ with $X(P_{i+1}, Q_{i+1}) \in \mathcal{N}_1^A(P_i, Q_i) \cap \mathcal{X}$ (we abbreviate $\mu_i = \mu(P_i, Q_i)$), such Q_{i+1} can always be taken due to (2.) provided $X(P_i, Q_i) \in \mathcal{X}$ and $P_i \stackrel{WT}{\longrightarrow}$.

Clearly, if $P \stackrel{\epsilon(d)}{\longrightarrow} P(d)$, then either the sequence of $\langle P_i, Q_i \rangle$ is infinite, or $P_i = P(d')$ for $d' \geq d$ for some i (the end of the sequence can be reached at P_k only when $P_k \stackrel{WT}{\not\longrightarrow}$). According to Lemma 12.4 $P_i = P(d')$ for $d' \geq d$ for some i in every case.

So, for a fixed value d with existing $P(d)$ we have either

- $P(d) = P_i$ for some i, in this case $\langle P(d), Q_i \rangle \in R_{\mathcal{X}}$ as well as $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_i$; or
- $P(d) = P_i(\mu')$ with $\mu' < \mu_i$ again we have $X(P_i, Q_i) \in \mathcal{X}$ with $Q \stackrel{\epsilon(d-\mu')}{\Longrightarrow} Q_i$, so we obtain $\langle P(d), Q_d \rangle \in R_{\mathcal{X}}$ for some Q_d with $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ from (2.) using Fact 12.6.□

In what follows we give a symbolic characteristic of every weak bisimulation $R_{\mathcal{X}} \subseteq \tilde{d}(A) \times \tilde{d}(B)$.

We introduce for $\nu \in L_0 \cup \{\epsilon\}$ in the set $\mathcal{X}_{A,B}$ of region processes the relations $\stackrel{\nu}{\Longrightarrow}$ in a way $X \stackrel{\nu}{\Longrightarrow} X'$ iff there exist $\langle P, Q \rangle \in X$ and $\langle P', Q' \rangle \in X'$ such that $P \stackrel{\nu}{\Longrightarrow} P'$ and $Q \stackrel{\nu}{\Longrightarrow} Q'$.

Definition 12.8 *Let for any $\mathcal{X} \subseteq \mathcal{X}_{A,B}$ $F^*(\mathcal{X})$ be the set of all $X(P, Q)$ satisfying*

- *if $P \stackrel{\nu}{\Longrightarrow} P'$, then $X(P, Q) \stackrel{\nu}{\Longrightarrow} X(P', Q') \in \mathcal{X}$ for some Q';*
- *if $Q \stackrel{\nu}{\Longrightarrow} Q'$, then $X(P, Q) \stackrel{\nu}{\Longrightarrow} X(P', Q') \in \mathcal{X}$ for some P';*
- *if $P \stackrel{WT}{\longrightarrow}$, then both $X' \in \mathcal{N}_0^A(P, Q) \cap \mathcal{X}$ and $X'' \in \mathcal{N}_1^A(P, Q) \cap \mathcal{X}$ for some X', X''; and*
- *if $Q \stackrel{WT}{\longrightarrow}$, then both $X' \in \mathcal{N}_0^B(P, Q) \cap \mathcal{X}$ and $X'' \in \mathcal{N}_1^B(P, Q) \cap \mathcal{X}$ for some X', X''.*

Then $\mathcal{X}$ is a weak symbolic bisimulation, if $\mathcal{X} \subseteq F^*(\mathcal{X})$.

The proof of Theorem 11.4 (the decidability of the weak bisimulation equivalence) now can be completed by showing the following two results.

Lemma 12.9 *For $\mathcal{X} \in \mathcal{X}_{A,B}$ and $R_{\mathcal{X}} = \{\langle P, Q \rangle \mid X(P, Q) \in \mathcal{X}_{A,B}\}$ $\mathcal{X}$ is a weak symbolic bisimulation if and only if $R_{\mathcal{X}}$ is a weak (timed) bisimulation.*

Lemma 12.10 *It is decidable, whether a given set $\mathcal{X} \subseteq \mathcal{X}_{A,B}$ is a weak symbolic bisimulation.*

Corollary 12.11 *It is decidable, whether there exists a weak symbolic bisimulation $\mathcal{X}$ with $X(A,B) \in \mathcal{X}$.*

Proof of Lemma 12.9: Let $\mathcal{X}$ be a weak symbolic bisimulation. In order to prove that $R_{\mathcal{X}}$ is a weak (timed) bisimulation, take $\langle P,Q\rangle \in R_{\mathcal{X}}$. Since $\mathcal{X}$ is a weak symbolic bisimulation and due to Lemma 11.7 all α for $\alpha \in L_0$ and ϵ- moves by P can be matched by the corresponding moves of Q, and vice versa (whenever $P \stackrel{\alpha}{\Longrightarrow} P'$, then $X(P,Q) \stackrel{\alpha}{\Longrightarrow} X(P',Q') \in \mathcal{X}$, according to the definition of $\stackrel{\alpha}{\Longrightarrow}$ for region processes $Q \stackrel{\alpha}{\Longrightarrow} Q'$. $\langle P',Q'\rangle \in R_{\mathcal{X}}$ since $X(P,Q) \in \mathcal{X}$).

We show, how to find a corresponding match Q' with $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q'$ and $\langle P',Q'\rangle \in R_{\mathcal{X}}$ for P' with $P \stackrel{\epsilon(d)}{\Longrightarrow} P'$ (the case of Ps and Qs in opposite roles is analogous).

Since $P \stackrel{\epsilon(d)}{\Longrightarrow} P'$, we have a derivation

$$P = P_0 \stackrel{\epsilon}{\Longrightarrow} P_0' \stackrel{\epsilon(d_1)}{\longrightarrow} P_1 \stackrel{\epsilon}{\Longrightarrow} P_1' \stackrel{\epsilon(d_2)}{\longrightarrow} P_2 \cdots Q_{x-1}' \stackrel{\epsilon(d_x)}{\longrightarrow} Q_x \stackrel{\epsilon}{\Longrightarrow} Q_x' = Q_d$$

for $d_1 + \cdots + d_x = d$ and $d_i > 0$ for all i. We inductively establish that there exist Q_i, Q_i' for all i such that

- $\langle P_i, Q_i\rangle \in R_{\mathcal{X}}$ and $\langle P_i', Q_i'\rangle \in R_{\mathcal{X}}$, as well as
- $Q \stackrel{\epsilon(p_i)}{\Longrightarrow} Q_i$ and $Q \stackrel{\epsilon(p_i)}{\Longrightarrow} Q_i'$ for $p_i = d_1 + d_2 + \ldots + d_i$.

$Q_i \stackrel{\epsilon}{\Longrightarrow} Q_i'$ with $\langle P_i', Q_i'\rangle \in R_{\mathcal{X}}$ provided $\langle P_i, Q_i\rangle \in R_{\mathcal{X}}$ is proved already above; $Q_i' \stackrel{\epsilon(d_{i+1})}{\Longrightarrow} Q_{i+1}$ with $\langle P_{i+1}, Q_{i+1}\rangle \in R_{\mathcal{X}}$ for $\langle P_i', Q_i'\rangle \in R_{\mathcal{X}}$ follows from Lemma 12.7.

The proof that for every weak timed bisimulation $R_{\mathcal{X}} = \{\langle P,Q\rangle \mid X(P,Q) \in \mathcal{X}\}$ the set $\mathcal{X}$ is a weak symbolic bisimulation is straightforward from the definitions, Fact 12.5 and Lemma 12.7.□

Proof of Lemma 12.10: Since for every pair of timed processes $\langle P,Q\rangle \in \tilde{d}(A) \times \tilde{d}(B)$ the set of process pairs $\langle P',Q'\rangle$ with $P \stackrel{\nu}{\Longrightarrow} P'$ and $Q \stackrel{\nu}{\Longrightarrow} Q'$ for every $\nu \in L_0 \cup \{\epsilon\}$ is finite and effectively computable from $\langle P,Q\rangle$ (though the set of all corresponding derivations can be infinite due to the repeating τ-loops, observe that the all newly appearing timer values in the processes in these derivations are integers from a bounded interval), Proposition 11.12 guarantees the decidability of the untimed match existence for all processes $X(P,Q)$ with either $P \stackrel{\nu}{\Longrightarrow}$, or $Q \stackrel{\nu}{\Longrightarrow}$.

Let us demonstrate the effectivity of the check, whether for given $X \in \mathcal{X}$ with $P \stackrel{WT}{\longrightarrow}$ for some (=for all, see Proposition 11.12) $\langle P,Q\rangle \in X$ both the sets

$$\mathcal{N}_0^{\mathcal{A}}(P,Q) \cap \mathcal{X} \text{ and } \mathcal{N}_1^{\mathcal{A}}(P,Q) \cap \mathcal{X}$$

are not empty. Due to the finiteness of $\mathcal{X}_{A,B}$ it suffices to show the algorithms, *generating* the sets $\mathcal{N}_0^A(P,Q)$ and $\mathcal{N}_1^A(P,Q)$ from the given processes $P \in \tilde{d}(A)$ and $Q \in \tilde{d}(B)$.

For this purpose we generalize slightly the notion of the region process (compare the following definition with Definition 11.6).

Definition 12.12 *Let for a given set $\mathcal{T}$ of timers we denote by $\Delta^{\mathcal{T}}$ the set of all timer value assignments $\delta : \mathcal{T} \to \mathbf{Q}^{+0}$. Let for $P_1, P_2 \in \tilde{d}(A)$, $Q_1, Q_2 \in \tilde{d}(B)$ and $\delta^1, \delta^2 \in \Delta^{\mathcal{T}}$ we say that $\langle P_1, Q_1, \delta^1\rangle \cong \langle P_2, Q_2, \delta^2\rangle$ iff*

- $v^{P_1} = v^{P_2}$ *and* $v^{Q_1} = v^{Q_2}$, *and*
- $\delta^{P_1} :: \delta^{Q_1} :: \delta^1 \cong \delta^{P_2} :: \delta^{Q_2} :: \delta^2$

(see Section 11.2 for the definition of the relation $\cong$ for timer value vectors).

Let $\mathcal{X}_{A,B}^{\mathcal{T}}$ be the partitioning of $(\tilde{d}(A) \times \tilde{d}(B)) \times \Delta^{\mathcal{T}}$, generated by the relation $\cong$, let for every triple $\langle P, Q, \delta\rangle \in \tilde{d}(A) \times \tilde{d}(B) \times \Delta^{\mathcal{T}}$ $X(P,Q,\delta)$ denotes the element of the partitioning $\mathcal{X}_{A,B}^{\mathcal{T}}$ to which the triple belongs to (i.e. $\langle P, Q, \delta\rangle \in X(P,Q,\delta)$) and is called a (refined) region process, corresponding to $\langle P, Q, \delta\rangle$.

Let us mention the following simple properties of the refined region processes.

Fact 12.13 *Whenever $X(P,Q,\delta) = X(P',Q',\delta')$ for any δ, δ', then always $X(P,Q) = X(P',Q')$.*

Proof: Follows from definitions.□

Fact 12.14 *$\langle P_1, Q_1, \delta^1\rangle \cong \langle P_2, Q_2, \delta^2\rangle$ if and only if there exists a uniform mapping ρ, such that $\langle P_1, Q_1, \delta^1\rangle = \rho(\langle P_2, Q_2, \delta^2\rangle)$.*

Proof: Follows from definitions.□

Let for a given triple $\langle P, Q, \delta\rangle$ $\mu(P,Q,\delta)$ be the minimal timer value fractional part among either P or Q timer values, or the values $\delta(t)$ for $t \in \mathcal{T}$.

Fact 12.15 *For $P \in \tilde{d}(A)$, $Q \in \tilde{d}(B)$ and $\delta \in \Delta^{\mathcal{T}}$, if $P \xrightarrow{WT}$ and $Q \xrightarrow{WT}$, then for all $d, d' \in]0, \mu(P,Q,\delta)[$ always $P(d), Q(d), P(d'), Q(d')$ exist and*

$$X(P(d), Q(d), \delta \ominus d) = X(P(d'), Q(d'), \delta \ominus d').$$

Proof: Follows from the semantics of processes and the definition of refined region processes.□

Definition 12.16 *We define the processes $next_0(X)$ and $next_1(X)$ for $X(P,Q,\delta) \in \mathcal{X}_{A,B}^{\mathcal{T}}$ with $\mu(P,Q,\delta) = \mu$ in a way*

- $next_1(X(P,Q,\delta)) = X(P(\mu), Q(\mu), \delta \ominus \mu)$ *and*
- $next_0(X(P,Q,\delta)) = X(P(\mu/2), Q(\mu/2), \delta \ominus \mu/2)$.

Fact 12.17 *For every region process $X \in \mathcal{X}^{\mathcal{T}}_{A,B}$ the processes $next_0(X)$ and $next_1(X)$ are unique (i.e. the operations $next_0$ and $next_1$ over the refined region processes are well-defined).*

Proof: Assume $X(P,Q,\delta) = X(P',Q',\delta')$. According to Fact 12.14 $\langle P',Q',\delta'\rangle = \rho(\langle P,Q,\delta\rangle)$ for some uniform ρ. So we have $\mu(P',Q',\delta') = \rho(\mu(P,Q,\delta))$, denote $\mu = \mu(P,Q,\delta)$.

Clearly, $(\delta^P :: \delta^Q :: \delta) \ominus \mu \cong \rho(\delta^P :: \delta^Q :: \delta) \ominus \rho(\mu)$, so

$$X(P(\mu), Q(\mu), \delta \ominus \mu) = X(P'(\rho(\mu)), Q'(\rho(\mu)), \delta' \ominus \rho(\mu)),$$

what means the well-definedness of $next_1$.

In a similar way

$$X(P(\mu/2), Q(\mu/2), \delta \ominus \mu/2) = X(P'(\rho(\mu/2)), Q'(\rho(\mu/2)), \delta' \ominus \rho(\mu/2)).$$

According to Fact 12.15 and the uniformity of ρ we have also

$$X(P(\mu/2), Q(\mu/2), \delta \ominus \mu/2) = X(P'(\rho(\mu)/2), Q'(\rho(\mu)/2), \delta' \ominus \rho(\mu)/2),$$

i.e. the well-definedness of $next_0$.□

Now let us continue the proof of Lemma 12.10. Let us define $\mathcal{T} = \{t_0\}$ and let $\delta^0(t_0) = 1 + \mu(P,Q)$. Let for every region process $X \in \mathcal{X}^{\mathcal{T}}_{A,B}$ the transitions $\xrightarrow{\bullet}$, $\xrightarrow{\bullet\bullet}$ and $\xrightarrow{e}$ for the process Q edge $e \in E(Q)$ (=for the process B edge) with $lab(e) = \tau$ be defined as

- $X \xrightarrow{\bullet} next_0^{\mathcal{T}}(X)$,
- $X \xrightarrow{\bullet\bullet} next_1^{\mathcal{T}}(X)$ and
- $X(P',Q',\delta) \xrightarrow{e} X(P',Q'',\delta)$ for $e \in E(Q')$ with $Q' \xrightarrow{e:\tau} Q''$.

Let $R^0_{P,Q}$ be the set of region processes $X(P',Q',\delta)$ which are reachable from $X(P,Q,\delta_0)$ using the defined transitions $\xrightarrow{\bullet}$ and $\xrightarrow{e}$. Let $R_{P,Q} \subseteq R^0_{P,Q}$ be the set of processes with a derivation containing at least one $\xrightarrow{\bullet}$ transition. We let also $R'_{P,Q}$ to be the set of region processes, reachable from processes of $R^0_{P,Q}$ by *one* $\xrightarrow{\bullet\bullet}$ transition, followed by a number of $\xrightarrow{e}$ transitions.

Easy to see that both the sets $R_{P,Q}$ and $R'_{P,Q}$ can be computed in a finite time from the given processes $P \in \tilde{d}(A)$ and $Q \in \tilde{d}(B)$.

The proof of Lemma 12.10 now is completed by providing a simple decision algorithm via the following result.

Proposition 12.18 $\mathcal{N}_A^0(P,Q) = \{X(P',Q') \mid \exists \delta :\ X(P',Q',\delta) \in R_{P,Q}\}$ *and* $\mathcal{N}_A^1(P,Q) = \{X(P',Q') \mid \exists \delta :\ X(P',Q',\delta) \in R'_{P,Q}\}$.

Proof: Consider first $X(P(d),Q_d) \in \mathcal{N}_0^A(P,Q)$. According to the definition of $\mathcal{N}_0^A$ we have $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ for some $d \in]0,\mu(P,Q)[$, so according to the definition of $\stackrel{\epsilon(d)}{\Longrightarrow}$ we can obtain

$$Q = Q_0 \stackrel{\epsilon}{\Longrightarrow} Q'_0 \stackrel{\epsilon(d_1)}{\longrightarrow} Q_1 \stackrel{\epsilon}{\Longrightarrow} Q'_1 \stackrel{\epsilon(d_2)}{\longrightarrow} Q_2 \cdots Q'_{x-1} \stackrel{\epsilon(d_x)}{\longrightarrow} Q_x \stackrel{\epsilon}{\Longrightarrow} Q'_x = Q_d$$

with $d_1 + \cdots + d_x = d$ and $d_i > 0$ for all i. Easy to see that for $p_i = d_1 + \cdots + d_i \ \forall i$

- for every $i \leq x \ \ X(P(p_i),Q_i,\delta^0 - p_i) \stackrel{e_1}{\longrightarrow} \stackrel{e_2}{\longrightarrow} \ldots \stackrel{e_u}{\longrightarrow} X(P(p_i),Q'_i,\delta^0 - p_i)$ with $e_j \in E(Q)$ for $j = 1,2,\ldots,u$, as well as
- for every $i < x \ \ X(P(p_i),Q'_i,\delta^0 - p_i) \stackrel{\tau}{\longrightarrow} X(P(p_{i+1}),Q_{i+1},\delta^0 - p_{i+1})$ (according to Fact 12.3 and the definition of δ^0 we have $\mu(P(p_i),Q'_i,\delta^0 - p_i) \geq d_{i+1}$),

so we inductively along the derivation of Q_d conclude that all the abovementioned region processes $X(P(p_i),Q_i,\delta^0 - p_i)$ and $X(P(p_i),Q'_i,\delta^0 - p_i)$ belong to the set $R^0_{P,Q}$, so also

$$X(P(d),Q_d,\delta^0 - p_x) \in R^0_{P,Q}.$$

Since $d > 0$, we have $x > 0$, so

$$X(P(d),Q_d,\delta^0 - p_x) \in R_{P,Q},$$

as requested.

As to $X(P(\mu),Q_\mu) \in \mathcal{N}_0^1(P,Q)$, consider again a derivation

$$Q = Q_0 \stackrel{\epsilon}{\Longrightarrow} Q'_0 \stackrel{\epsilon(d_1)}{\longrightarrow} Q_1 \stackrel{\epsilon}{\Longrightarrow} Q'_1 \stackrel{\epsilon(d_2)}{\longrightarrow} Q_2 \cdots Q'_{x-1} \stackrel{\epsilon(d_x)}{\longrightarrow} Q_x \stackrel{\epsilon}{\Longrightarrow} Q'_x = Q_\mu$$

with $d_1 + \cdots + d_x = \mu$ and $d_i > 0$ for all i.

As in the above case we establish $X(P(p_{x-1}),Q'_{x-1},\delta^0 - p_{x-1}) \in R^0_{P,Q}$. Fact 12.3 gives us

$$\mu(P(p_{x-1}),Q'_{x-1},\delta^0 - p_{x-1}) \geq \mu(P,Q,\delta^0) - p_{x-1},$$

according to the definition of δ^0 we obtain

$$\mu(P(p_{x-1}),Q'_{x-1},\delta^0 - p_{x-1}) = \mu(P,Q,\delta^0) - p_{x-1},$$

so, according to the definition of $\stackrel{\tau}{\longrightarrow}$, we obtain $X(P(\mu),Q_x,\delta^0 - \mu) \in R'_{P,Q}$, hence also $X(P(\mu),Q_\mu,\delta^0 - \mu) \in R'_{P,Q}$, as requested.

Assume now $X \in R^0_{P,Q}$, let us prove that $\langle P(d), Q_d, \delta^0 - d\rangle \in X$ for some $d \in [0, \mu[$ and some Q_d with $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$.

Since $X \in R^0_{P,Q}$, it has a derivation from $X(P, Q, \delta^0)$ by the defined $\stackrel{\bullet}{\longrightarrow}$ and $\stackrel{\epsilon}{\longrightarrow}$ transitions. We find the delay d and the process Q_d inductively along the derivation of X. For the empty derivation take $d' = 0$ and $Q_0 = Q$.

Regarding the induction step for $X \in R^0_{P,Q}$ assume $X = X(P(d), Q_d, \delta^0 - d)$ with $d \in [0, \mu[$ and $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$. Consider 2 cases:

- $X \stackrel{\epsilon}{\longrightarrow} X' = X(P(d), Q', \delta^0 - d)$, in this case $Q_d \stackrel{\epsilon:\tau}{\longrightarrow} Q'$ and, so, $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q'$ follows from the definition of the transition relation $\stackrel{\epsilon}{\longrightarrow}$;
- $X \stackrel{\bullet}{\longrightarrow} X'$, according to Fact 12.5 we have

$$X' = X(P(d + d'), Q_d(d'), \delta^0 - (d + d')) \text{ for } d' = (\mu(P(d), Q_d, \delta^0 - d))/2,$$

clearly $Q \stackrel{\epsilon(d+d')}{\Longrightarrow} Q_d(d')$.

It is easy to see that, if there is at least one $\stackrel{\bullet}{\longrightarrow}$ transition in the derivation of X, we have also the corresponding delay $d > 0$.

For $X \in R'_{P,Q}$ consider again the derivation of X by $\stackrel{\bullet}{\longrightarrow}$, $\stackrel{\epsilon}{\longrightarrow}$ and $\stackrel{\bullet\bullet}{\longrightarrow}$ from $X(P, Q, \delta^0)$, the existence of appropriate Q_μ again is straightforward.□

Chapter 13

Equivalences for PTP Enrichments

In this chapter we outline the bisimulation equivalence decidability proofs for two enrichments of the basic PTP model. Regarding the processes with nondeterministic timer settings (see Section 13.1) the obtained result appears to be mostly of the theoretical interest for it is just outlining a slightly more general model with timing with the decidable both strong and weak bisimulation equivalences. The second result regarding the processes with the dependencies on rational-valued external data could be, at least in principle, of larger practical importance for the principally new modelling abilities possessed by the processes (however, only a brief outline of the proof is given in Section 13.2).

13.1 Processes with Nondeterministic Timer Settings

In this section we consider the deciding bisimulation equivalences for bounded PTPNs (see Section 10.3). The PTPNs have already got labelled transition system semantics, the definition of bisimulation equivalences for them is standard (see Definition 11.1 and Definition 11.3).

Regarding the deciding the strong equivalence for PTPNs, it does not contain any new ideas, if compared with the basic PTP case. Since the decidability of the strong equivalence follows from the decidability of the weak one, we further on focus only on the weak equivalence for PTPNs.

Also the proof of the decidability of the weak bisimulation equivalence for PTPNs appears just to be a generalization of the corresponding proof for the basic PTPs and follows the same schema. What turns the situation with PTPNs more complicated, is that we cannot give any upper bound for the number of times per time unit when

some timer in the process reaches the 0 value (so, possibly enabling a transition along a new edge), Lemma 12.4 does not generally hold for PTPNs.

Theorem 13.1 *There exists an algorithm which, given two bounded PTPNs A and B, decides whether $A \sim B$, $A \approx B$, or not.*

Proof outline: First of all observe that the generalization of Lemma 11.7 and Proposition 11.12 to the case of PTPNs is completely straightforward (one does not need even the boundedness at this stage) with the only new point in showing that, if $P \xrightarrow{e:\sigma} P'$, then for a uniform mapping ρ also $\rho(P) \xrightarrow{e:\sigma} \rho(P')$, what can be dealt with by observing that all interval bounds in the edge e timer setting are (taken) integers. The finiteness of the intervals in the edge timer settings is important to ensure the *finiteness* of the region process set $\mathcal{X}_{A,B}$.

Let us show, how, given a set $\mathcal{X} \subseteq \mathcal{X}_{A,B}$, check, whether the corresponding set

$$R_{\mathcal{X}} = \{\langle P, Q\rangle \mid X(P,Q) \in \mathcal{X}\} \subseteq \tilde{d}(A) \times \tilde{d}(B)$$

is a weak bisimulation (in the case of the strong bisimulation the proof continue to go precisely along the same lines, as in the case of PTPs, and is demonstrated in Section 11.3).

Clearly, given $X \in \mathcal{X}$ and a representative $\langle P, Q\rangle \in X$, the test whether all real (i.e. $\xrightarrow{\sigma}$ or $\stackrel{\sigma}{\Longrightarrow}$) transitions of P into P' can be matched by corresponding transitions of Q with the resulting pair of processes in $R_{\mathcal{X}}$ is straightforward, though in the case of nondeterministic settings there can be infinitely many different target states after performed some real transition, these states can be easily grouped w.r.t. the region processes the pairs $\langle P', Q\rangle$ belong to.

Using the techniques similar to those of the proofs in Section 12.2 one can reduce the proof of the decidability of whether $R_{\mathcal{X}}$ for given $\mathcal{X}$ is a bisimulation to the following lemma.

Lemma 13.2 *Given $\langle P, Q\rangle \in R_{\mathcal{X}}$ such that $P \xrightarrow{WT}$, it is decidable whether for all $d \in [0, \mu(P)]$ there exists Q_d with both $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ and $\langle P(d), Q_d\rangle \in R_{\mathcal{X}}$.*

Note: Observe that we are using here $\mu(P)$ instead of $\mu(P,Q)$ used in the formulation of Lemma 12.4 for Lemma 12.4 being not generally true for PTPNs. The building of the sequence

$$\langle P_0, Q_0\rangle,\ \langle P_1, Q_1\rangle,\ \langle P_2, Q_2\rangle, \ldots,$$

as in the proof of Lemma 12.7 now is based on the idea of $P_{i+1} = P_i(\mu(P_i))$ and is used together with the result of Lemma 13.2 *directly* in the proof of the decidability whether for a given pair of processes $\langle P, Q\rangle \in X \in \mathcal{X}$ all $\xrightarrow{\epsilon(d)}$ moves of one process can be matched by $\stackrel{\epsilon(d)}{\Longrightarrow}$ moves of another process.

Proof: Let $\mu_1, \mu_2, \ldots, \mu_s$ be all different P and Q timer value fractional parts with $0 < \mu_1 < \ldots < \mu_s = \mu(P)$.

We define $\mathcal{T} = \{t_1, t_2, \ldots, t_s\}$ and let $\delta^0(t_i) = 1 + \mu_i$ for all $i = 1, 2, \ldots, s$. Let us define for every refined region process $X \in \mathcal{X}^{\mathcal{T}}_{A,B}$ the transitions $X \xrightarrow{\cdot} next^{\mathcal{T}}_0(X)$, $X \xrightarrow{\cdot\cdot} next^{\mathcal{T}}_i(X)$ and $X(P', Q', \delta) \xrightarrow{e} X(P', Q'', \delta)$ for $e \in E(Q')$ with $Q' \xrightarrow{e:\tau} Q''$ (observe that the transition $\xrightarrow{e}$ of the region processes can still have more than one target for the same source process). Let $\xrightarrow{\omega}$ denotes any sequence of $\xrightarrow{\cdot}$ and $\xrightarrow{\cdot\cdot}$ transitions.

Let $R_{P,Q}$ be the set of region processes $X(P', Q', \delta)$ which

- are reachable from $X(P, Q, \delta_0)$ using the defined transitions $\xrightarrow{\cdot}$, $\xrightarrow{\cdot\cdot}$ and $\xrightarrow{e}$, and
- have $\delta(t_s) \geq 1$ (i.e. we consider the waiting only until the least P timer value fractional part reduces to 0).

Easy to see that the set $R_{P,Q}$ can be computed in a finite time from the given processes $P \in \tilde{d}(A)$ and $Q \in \tilde{d}(B)$.

Fact 13.3 *Whenever $\langle P(d), Q', \delta \rangle \in X \in R_{P,Q}$ we have $\delta = \delta^0 - d$.*

Proof: Follows from definitions.□

Let $\overline{\mathcal{X}} = \{X(P', Q', \delta) \mid X(P', Q') \in \mathcal{X}\}$. Easy to see that the proof of Lemma 13.2 can be obtained from the following result.

Lemma 13.4 *For $Q' \in \tilde{d}(B)$ $Q \stackrel{\epsilon(\mu_i)}{\Longrightarrow} Q'$ and $\langle P(\mu_i), Q_{\mu_i} \rangle \in R_{\mathcal{X}}$ if and only if $X(P(\mu_i), Q', \delta) \in R_{P,Q} \cap \overline{\mathcal{X}}$ for some δ with $\delta(t_i) = 1$.*

Given d with $\mu_i < d < \mu_{i+1}$ (resp. $d < \mu_1$) there exists Q_d with $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ and $\langle P(d), Q_d \rangle \in R_{\mathcal{X}}$ if and only if $X(P(d'), Q', \delta) \in R_{P,Q} \cap \overline{\mathcal{X}}$ for some Q', d', δ with $\delta(t_i) < 1 < \delta(t_{i+1})$ (resp. $1 < \delta(t_1)$).

Note: Have one proved the lemma, it would be possible for every interval $]\mu_i, \mu_{i+1}[$ (and every fixed delay value μ_i) to look in the generated set $R_{P,Q}$ whether it contains a region process with corresponding values $\delta(t_j)$, which belongs to $\overline{\mathcal{X}}$, so finding out the answer about the timed match Q_d with $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ and $\langle P(d), Q_d \rangle \in \mathcal{X}$ existence for all $d \leq \mu(P) = \mu_s$.

Proof of Lemma 13.4: Let for $d, d' \leq \mu(P) = \mu_s$ $d \simeq d'$ whenever $d \lambda \mu_i$ iff $d' \lambda \mu_i$ for $\lambda \in \{\leq, \geq\}, i \in \{1, \ldots, s\}$. Consider the following two propositions.

Proposition 13.5 *If $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ for some $d \in [0, \mu_s]$, then $X(P(d), Q_d, \delta^0 - d) \in R_{P,Q}$.*

Proof: According to the definition of $\stackrel{\epsilon(d)}{\Longrightarrow}$ we have

$$Q = Q_0 \stackrel{\epsilon}{\Longrightarrow} Q'_0 \stackrel{\epsilon(d_1)}{\longrightarrow} Q_1 \stackrel{\epsilon}{\Longrightarrow} Q'_1 \stackrel{\epsilon(d_2)}{\longrightarrow} Q_2 \cdots Q'_{x-1} \stackrel{\epsilon(d_x)}{\longrightarrow} Q_x \stackrel{\epsilon}{\Longrightarrow} Q'_x = Q_d$$

with $d_1 + \cdots + d_x = d$ and $d_i > 0$ for all i. Easy to see that for $p_i = d_1 + \cdots + d_i \ \ \forall i$

- for every $i \leq x \ \ X(P(p_i), Q_i, \delta^0 - p_i) \stackrel{e_1}{\longrightarrow} \stackrel{e_2}{\longrightarrow} \ldots \stackrel{e_u}{\longrightarrow} X(P(p_i), Q'_i, \delta^0 - p_i)$ with $e_j \in E(Q)$ for $j = 1, 2, \ldots, u$, as well as
- for every $i < x \ \ X(P(p_i), Q'_i, \delta^0 - p_i) \stackrel{\omega}{\longrightarrow} X(P(p_{i+1}), Q_{i+1}, \delta^0 - p_{i+1})$.□

Proposition 13.6 *Given $d \in [0, \mu_s]$, whenever $\langle P(d^0), Q', \delta^0 - d^0 \rangle \in X \in R_{P,Q}$ for some $d^0 \simeq d$, there exists Q_d with $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ and $\langle P(d), Q_d, \delta^0 - d \rangle \in X$.*

Proof: We prove first that $\langle P(d'), Q_{d'}, \delta^0 - d' \rangle \in X$ with $Q \stackrel{\epsilon(d')}{\Longrightarrow} Q'_d$ for some $d' \in [0, \mu_s]$. Since $X \in R_{P,Q}$, it has a derivation from $X(P, Q, \delta^0)$ by the defined $\stackrel{*}{\longrightarrow}$, $\stackrel{**}{\longrightarrow}$ and $\stackrel{e}{\longrightarrow}$ transitions. We find the delay d' and the process $Q_{d'}$ inductively along the derivation of X. For the empty derivation take $d' = 0$ and $Q_0 = Q$.

Assume for $X \in R_{P,Q} \ \ X = X(P(d'), Q'_d, \delta^0 - d')$ with $Q \stackrel{\epsilon(d')}{\Longrightarrow} Q'_d$. Consider 3 cases:

- $X \stackrel{e}{\longrightarrow} X' = X(P(d'), Q', \delta^0 - d')$, in this case $Q'_d \stackrel{\epsilon:\tau}{\longrightarrow} Q'$ and, so, $Q \stackrel{\epsilon(d')}{\Longrightarrow} Q'$ follows from the definition of the transition relation $\stackrel{\epsilon}{\longrightarrow}$;
- $X \stackrel{*}{\longrightarrow} X'$, we have $X' = X(P(d' + d''), Q'_d(d''), \delta^0 - (d' + d''))$ for $d'' = (\mu(P(d'), Q'_d, \delta^0 - d'))/2$ and $Q \stackrel{\epsilon(d'+d'')}{\Longrightarrow} Q'_d(d'')$;
- $X \stackrel{**}{\longrightarrow} X'$, we have $X' = X(P(d' + d''), Q'_d(d''), \delta^0 - (d' + d''))$ for $d'' = \mu(P(d'), Q'_d, \delta^0 - d')$, again $Q \stackrel{\epsilon(d'+d'')}{\Longrightarrow} Q'_d(d'')$.

Observe that in the case of $d = \mu_i$ we have also $d^0 = \mu_i$, the proved result gives us already the needed Q_d with $Q \stackrel{\epsilon(d)}{\Longrightarrow} Q_d$ for in this case $\langle P(d'), Q_{d'}, \delta^0 - d' \rangle \in X$ requires $\delta^0(t_i) - d' = 1$ (we have

$$\langle P(d^0), Q', \delta^0 - d^0 \rangle \in X \text{ and } \delta^0(t_i) - d^0 = \delta^0(t_i) - \mu_i = 1,$$

hence $\delta^0(t_i) - d' = 1$ (recall the definition of region processes), so, $d' = \mu_i$.

Since $Q \stackrel{\epsilon(d')}{\Longrightarrow} Q_{d'}$, we have a derivation

$$Q = Q_0 \stackrel{\epsilon}{\Longrightarrow} Q'_0 \stackrel{\epsilon(d_1)}{\longrightarrow} Q_1 \stackrel{\epsilon}{\Longrightarrow} Q'_1 \stackrel{\epsilon(d_2)}{\longrightarrow} Q_2 \cdots Q'_{x-1} \stackrel{\epsilon(d_x)}{\longrightarrow} Q_x \stackrel{\epsilon}{\Longrightarrow} Q'_x = Q_{d'}$$

for $d_1 + \cdots + d_x = d'$ and $d_i > 0$ for all i. The idea is to change the delays and the timer values in this derivation slightly in order to have it ending with Q_d and have $X(P(d), Q_d, \delta^0 - d) = X(P(d'), Q_{d'}, \delta^0 - d')$.

Since in the case $\mu_i < d < \mu_{i+1}$ we also have $\mu_i < d^0 < \mu_{i+1}$ and, so, $\mu_i < d' < \mu_{i+1}$

$$(\delta^0(t_i) - d^0 < 1 < \delta^0(t_{i+1}) - d^0, \text{ so } \delta^0(t_i) - d' < 1 < \delta^0(t_{i+1}) - d',$$

we let $\mu_0 = 0$ to allow to treat the case $d < \mu_1$ uniformly), we can take a uniform mapping $\rho : \mathbf{Q}^{+0} \to \mathbf{Q}^{+0}$ (see Definition 11.8) satisfying

- $\rho(x) = x$ for all $x \in [\mu(P), 1]$ and for $x = \mu_i$, $i \leq s$;
- $\rho(d') = d$.

It follows from Fact 11.10, Fact 11.11 and Proposition 11.12 that we obtain the derivation of Q_d with $X(P(d), Q_d, \delta^0 - d) = X(P(d'), Q_{d'}, \delta^0 - d')$ by changing every timer value $g = \delta^{Q_u}(t)$ of the process Q_u, placed in the derivation of $Q_{d'}$ in a way $Q \stackrel{\epsilon(d'')}{\Longrightarrow} Q_u$, to $\rho_{d''}(g)$.□

The result of Lemma 13.4 and, so Lemma 13.2, now can be obtained from Proposition 13.5, Proposition 13.6 and Fact 13.3.□□□

13.2 Processes with Data Dependencies

In this section we briefly outline the bisimulation deciding algorithms for the processes which can depend both on quantitative time constraints and *rational-valued* external data, namely, we consider PTPQs from Section 10.4.

We assume that PTPQs have already got the semantics in the terms of labelled transition systems (analogically with PTPBs) over the transitions $\stackrel{\sigma,u}{\longrightarrow}$ for $\sigma \in L$, $u \in \mathbf{Q}$, and $\stackrel{\epsilon(d)}{\longrightarrow}$ for $d \in \mathbf{Q}^{+0}$. So, the definitions of the bisimulation equivalences for PTPQs is standard (for the sake of simplicity we consider only the *strong* bisimulations, defined to be the relations $R \subseteq \mathcal{P}^{\mathcal{Q}} \times \mathcal{P}^{\mathcal{Q}}$ such that for every $\langle P, Q\rangle \in R$ and every $\nu \in (L \times \mathbf{Q}) \cup \{\epsilon(d) | d \in \mathbf{Q}^{+0}\}$

i) whenever $P \stackrel{\nu}{\longrightarrow} P'$ then $Q \stackrel{\nu}{\longrightarrow} Q'$ with $\langle P', Q'\rangle \in R$ for some Q',
ii) whenever $Q \stackrel{\nu}{\longrightarrow} Q'$ then $P \stackrel{\nu}{\longrightarrow} P'$ with $\langle P', Q'\rangle \in R$ for some P'

and the strong bisimulation equivalence, written $\sim$, defined as the largest strong bisimulation).

Theorem 13.7 *There exists an algorithm which given two PTPQs A and B decides whether $A \sim B$, or not.*

Proof outline: The proof follows the same principal lines, as the proof of Theorem 11.2, we outline the different points.

First of all we construct the region processes. For this purpose we assume all *timed* constants (those used in the settings $\gamma_T(e)$) in both the graphs of A and B integers.

We define also an equivalence relation $\cong_L$ in the set of the data variable value vectors $\delta_W : (\mathcal{W}(A) \cup \mathcal{W}(B)) \to \mathbf{Q}$ in a way that $\delta^1 \cong_L \delta^2$ if for every t, t'

- $\delta^1(t) \leq \delta^1(t')$ iff $\delta^2(t) \leq \delta^2(t')$ and
- $\delta^1(t) \leq c$ $(\delta^1(t) \geq c)$ iff $\delta^2(t) \leq c$ $(\delta^2(t) \geq c)$, here c is any constant used either in A, B graph edge variable settings $\phi_W(e)$, or in conditions $\gamma_W(e)$.

Two process pairs $\langle P_1, Q_1 \rangle$ and $\langle P_2, Q_2 \rangle$ for $P_i \in \tilde{d}(A)$ and $Q_i \in \tilde{d}(B)$ with the corresponding states

$$\langle v^{P_i}, \delta_T^{P_i}, \delta_W^{P_i} \rangle \text{ and } \langle v^{Q_i}, \delta_T^{Q_i}, \delta_W^{Q_i} \rangle$$

are called *equivalent*, written $\langle P_1, Q_1 \rangle \cong_P \langle P_2, Q_2 \rangle$, if and only if

- $v^{P_1} = v^{P_2}$ and $v^{Q_1} = v^{Q_2}$,
- $\delta_T^{P_1} :: \delta_T^{Q_1} \cong \delta_T^{P_2} :: \delta_T^{Q_2}$ (cf. Definition 11.6) and
- $\delta_W^{P_1} :: \delta_W^{Q_1} \cong_L \delta_W^{P_2} :: \delta_W^{Q_2}$.

A *region process* is a class of equivalent process pairs.

The uniform mapping technique is also used to characterize the equivalent process pairs, in fact we have that $\langle P_1, Q_1 \rangle \cong_P \langle P_2, Q_2 \rangle$ if and only if

- $v^{P_1} = v^{P_2}$ and $v^{Q_1} = v^{Q_2}$,
- $\delta_T^{P_1} :: \delta_T^{Q_1} = \rho(\delta_T^{P_2} :: \delta_T^{Q_2})$ and
- $\delta_W^{P_1} :: \delta_W^{Q_1} = \rho_L(\delta_W^{P_2} :: \delta_W^{Q_2})$

for a uniform ρ (see Definition 11.8) and a strongly monotone, A and B data constant preserving, mapping ρ_L.

The analogues of Proposition 11.12 and Lemma 11.7 for PTPQs with the defined equivalence relation $\cong_P$ are obtained in a similar way, as the original results are for PTPs. Observe that the data part of the PTPQs is insensitive to the process waiting, so also the symbolic characteristic of the bisimulations is obtained for PTPQs in a rather similar way, as for PTPs (see Definition 11.16), one just defines for $P \in \mathcal{P}^Q$ $P \xrightarrow{\sigma} P'$ whenever $P \xrightarrow{\sigma,u} P'$ for some $u \in \mathbf{Q}$. Also the decidability of the symbolic bisimulation predicate for PTPQ region processes can be obtained easily.□

If one tries to obtain in a similar way a bisimulation deciding algorithm for PTPB_0s the first problem encountered would be the unawareness, how to obtain an appropriate finite partitioning of the data variable value space to have an analogue of Lemma 11.7 holding. Intuitively, the difficulty lies in the "individuality" of every integer number, which can be sooner or later forced to appear in the possibility or impossibility of some chains of transitions.

It is hardly believable that the bisimulation equivalence problems for $PTPB_0$s could be decidable by any method, however, this hypotheses still needs some kind of proof statement. Perhaps one could look also to some very restricted cases of PTPBs, e.g., to the processes, permitting the comparison of the data variables in the edge conditions only on being equal or not equal, (actually, for just this class the decidability of the bisimulation equivalence problems seems to be the case).

Chapter 14

Compositional Properties of PTPs

The Parallel Timer Process model allows a number of useful algebraic operations to be defined over the processes, these are mostly various kinds of static process algebra combinators, such as hiding, renaming, parallel composition, etc. (see [Hoa85, Mil89] for the use of combinators in process algebras).

In this chapter we show how two kinds of parallel composition operators can be defined for PTPs: one in the spirit of the CSP parallel composition ([Hoa85]), see Section 14.1, the other one (considered in Section 14.2) resembling the parallel composition of Timed CCS ([Mil89], [Wan90]). We define in Section 14.2 also other CCS-style static process algebra combinators and discuss how to use these compositionality properties for the modelling of a large class of Timed CCS processes from [Wan90] by PTPs. Such a modelling allows us to decide bisimulation equivalences for the considered class of TCCS processes (called here TCCS nets).

For the sake of simplicity we do not consider in this chapter the processes with value-passing (such as PTPBs and PTPQs from Section 10.4), we restrict our attention just to the basic PTP model.

Let for a Parallel Timer Process A we write $v \overset{e:\sigma}{\longmapsto}_A v'$ for $v, v' \in V(A)$ and $e \in E(A)$ with $f(e) = v$, $t(e) = v'$ and $lab(e) = \sigma$.

Technically, if we have a mapping $\phi : \mathcal{T} \to \mathcal{T} \cup \mathbf{Q}^{+0}$ for some finite set $\mathcal{T}$ then define for a finite set X the mapping

$$(\phi \times X) : (\mathcal{T} \times X) \to (\mathcal{T} \times X) \cup \mathbf{Q}^{+0}$$

in a way that for every $x \in X$

- $(\phi \times X)((t, x)) = (\phi(t), x)$ whenever $\phi(t) \in \mathcal{T}$, and
- $(\phi \times X)((t, x)) = \phi(t)$, if $\phi(t) \in \mathbf{Q}^{+0}$.

For $\mathcal{T}_1$ and $\mathcal{T}_2$ being disjoint sets and the mappings $\phi_1 : \mathcal{T}_1 \to \mathcal{T}_1 \cup \mathbf{Q}^{+0}$ and $\phi_2 : \mathcal{T}_2 \to \mathcal{T}_2 \cup \mathbf{Q}^{+0}$ let the mapping

$$\phi = (\phi_1 \star \phi_2) : (\mathcal{T}_1 \cup \mathcal{T}_2) \to (\mathcal{T}_1 \cup \mathcal{T}_2) \cup \mathbf{Q}^{+0}$$

be defined as $\phi(t) = \phi_1(t)$ for $t \in \mathcal{T}_1$ and $\phi(t) = \phi_2(t)$ for $t \in \mathcal{T}_2$.

14.1 CSP-like composition

Recall that we have L to be the set of events for PTPs (used as edge labels), as well as $L = L_0 \cup \{\tau\}$ for $\tau \notin L_0$ being an *invisible* (internal) event.

Let for a timed process A $rlabs(A) \subseteq L$ denote the set of labels which are ascribed to the red edges in A (i.e. $rlabs(A) = \{\sigma \in L \mid \exists e \in R \subseteq E(A) \text{ with } lab(e) = \sigma\}$).

Let A and B be timed processes and $K \subseteq L_0$ with

$$K \cap (rlabs(A) \cup rlabs(B)) = \emptyset \ .$$

be some set of visible events. We define the process $C = A \mid [K] \mid_{CSP} B$ called the (CSP-like) parallel composition of A and B with the synchronization on the event set K in a way

- $V(C) = V(A) \times V(B)$ and
- $\mathcal{T}(C) = \{\langle t, 1\rangle \mid t \in \mathcal{T}(A)\} \cup \{\langle t, 2\rangle \mid t \in \mathcal{T}(B)\}$,

as well as $\langle v, w\rangle \overset{\tilde{e}:\sigma}{\longmapsto}_C \langle v', w'\rangle$ for $\sigma \in L$ if and only if one of the following holds:

- $\sigma \in L \setminus K$, $w = w' \in V(B)$ and $v \overset{e:\sigma}{\longmapsto}_A v'$ for some $e \in E(A)$, the colour of $\tilde{e}$ is inherited from that of e and
  - $\gamma^C(\tilde{e}) = \gamma^A(e) \times \{1\}$,
  - $\phi^C(\tilde{e}) = \phi^A(e) \times \{1\}$; or
- $\sigma \in L \setminus K$, $v = v' \in V(A)$ and $w \overset{e:\sigma}{\longmapsto}_B w'$ for some $e \in E(B)$, the colour of $\tilde{e}$ is inherited from that of e and
  - $\gamma^C(\tilde{e}) = \gamma^B(e) \times \{2\}$,
  - $\phi^C(\tilde{e}) = \phi^B(e) \times \{2\}$; or
- $\sigma \in K$ and $v \overset{e:\sigma}{\longmapsto}_A v'$, $w \overset{e':\sigma}{\longmapsto}_B w'$ for some $e \in E(A)$, $e' \in E(B)$ (in this case the colour of $\tilde{e}$ necessarily is black) and
  - $\gamma^C(\tilde{e}) = \gamma^A(e) \times \{1\} \cup \gamma^B(e') \times \{2\}$ and
  - $\phi^C(\tilde{e}) = (\phi^A(e') \times \{1\}) \star (\phi^B(e'') \times \{2\})$.

The initial state of C is defined in a way $v^C = \langle v^A, v^B \rangle$ and δ^C is a mapping with $\delta^C(\langle t, 1 \rangle) = \delta^A(t)$ for $t \in T(A)$ and $\delta^C(\langle t, 2 \rangle) = \delta^B(t)$ for $t \in T(B)$.

Intuitively, two processes, working in parallel, are assumed to synchronize on all real visible actions $\sigma \in K$ (they necessarily must be ascribed to black edges), in all other cases the actions can be performed by the processes A and B independently.

The soundness of the defined parallel composition operator w.r.t. the rules defining the operational semantics for the Timed CSP processes (see [Sch91]) is justified by the following results which follow immediately from the composite process definition and the semantics (labelled transition system) of PTPs.

Theorem 14.1 *$A \mid [K] \mid_{CSP} B \xrightarrow{\alpha} C$ for $\alpha \in K$ if and only if $C = A' \mid [K] \mid_{CSP} B'$ and $A \xrightarrow{\alpha} A'$, $B \xrightarrow{\alpha} B'$ for some A', B'.*

Theorem 14.2 *$A \mid [K] \mid_{CSP} B \xrightarrow{\sigma} C$ for $\sigma \in L \setminus K$ if and only if $C = A' \mid [K] \mid_{CSP} B'$ with either*

- *$\sigma \in L$, $B = B'$ and $A \xrightarrow{\sigma} A'$ for some A', or*
- *$\sigma \in L$, $A = A'$ and $B \xrightarrow{\sigma} B'$ for some B'.*

Theorem 14.3 *$A \mid [K] \mid_{CSP} B \xrightarrow{\epsilon(d)} C$ if and only if $C = A' \mid [K] \mid_{CSP} B'$ with $A \xrightarrow{\epsilon(d)} A'$, $B \xrightarrow{\epsilon(d)} B'$.*

14.2 CCS-like composition

In this section we describe another parallel composition operator for a subset of Parallel Timer Processes, resembling the parallel composition (synchronization) of the process algebra CCS [Mil80, Mil89] treating the time modelling, as in [Wan90, Wan91a, Wan91b].

For this purpose we assume that for every visible event $\alpha \in L_0$ there exists a complementary event $\overline{\alpha} \in L_0$ such that $\overline{\overline{\alpha}} = \alpha$.

We call a Parallel Timer Process A a TCCS-like process, if an edge e in A automaton is red if and only if it is labelled by the invisible action $\tau \in L$ (this guarantees, by the way, the property of the processes, called in [Wan90] the *maximal progress*, stating that every internal action in a process must occur as soon as it becomes enabled).

For A and B being TCCS-like timed processes, let the process $C = A \mid\mid_{CCS} B$ called the (CCS-like) parallel composition of A and B be defined to have

- $V(C) = V(A) \times V(B)$,
- $T(C) = \{\langle t, 1 \rangle \mid t \in T(A)\} \cup \{\langle t, 2 \rangle \mid t \in T(B)\}$ and

- $\langle v, w\rangle \xrightarrow{\tilde{e}:\sigma}_C \langle v', w'\rangle$ for a $\sigma \in L$ in the following three cases (we do not discuss the edge colours separately since they are determined by the edge labels):

  - $w = w' \in V(B)$ and $v \xrightarrow{e:\sigma}_A v'$ for some $e \in E(A)$, as well as $\gamma^C(\tilde{e}) = \gamma^A(e) \times \{1\}$ and $\phi^C(\tilde{e}) = \phi^A(e) \times \{1\}$; or
  - $v = v' \in V(A)$ and $w \xrightarrow{e:\sigma}_B w'$ for some $e \in E(B)$, as well as $\gamma^C(\tilde{e}) = \gamma^B(e) \times \{2\}$, $\phi^C(\tilde{e}) = \phi^B(e) \times \{2\}$; or
  - $\sigma = \tau$ and $v \xrightarrow{e:\alpha}_A v'$, $w \xrightarrow{e':\overline{\alpha}}_B w'$ for some $e \in E(A)$, $e' \in E(B)$ and $\alpha, \overline{\alpha} \in L_0$ provided
    - $\gamma^C(\tilde{e}) = \gamma^A(e) \times \{1\} \cup \gamma^B(e') \times \{2\}$ and
    - $\phi^C(\tilde{e}) = \phi^A(e') \times \{1\} * \phi^B(e'') \times \{2\}$.

As in the case of the CSP-communication, the initial state of C is defined in a way $v^C = \langle v^A, v^B\rangle$ and δ^C being a mapping with $\delta^C(\langle t, 1\rangle) = \delta^A(t)$ for $t \in \mathcal{T}(A)$ and $\delta^C(\langle t, 2\rangle) = \delta^B(t)$ for $t \in \mathcal{T}(B)$.

Intuitively, two processes, working in parallel, are assumed to be able to synchronize on all complementary pairs of visible actions $\alpha, \overline{\alpha} \in L_0$ (unlike the case of the CSP-like parallel composition, in this case the actions $\alpha \in L_0$ still *can* be executed without the participation of the other component, what makes some restrictions on such an independent execution is the necessary instantaneous performance of the internal action whenever both the "complementary" actions become available). Some further operations like *restriction* of CCS ([Mil89]) can be used to prevent a process to exhibit the transitions associated with certain visible actions, so, for instance, enforcing in some cases the process to wait for the complementary action to become enabled (if no other alternatives are offered).

The soundness of the defined parallel composition operator w.r.t. the rules defining the operational semantics for the Timed CCS processes (see, for example [Wan90]) is justified by the following results which follow immediately from the composite process definition and the semantics (labelled transition system) of PTPs.

Fact 14.4 *A CCS-like parallel composition of two TCCS-like timed processes also is a TCCS-like timed process.*

Let for given PTP P for every edge $e \in E(P)$ outgoing from the P initial vertex (i.e. $f(e) = v^P$) $\tilde{\mu}(P, e) = max\{\delta^P(t) \mid t \in \gamma(e)\}$ (according to the definition of the processes $\tilde{\mu}(P, e)$ is the earliest time when a transition of P along e may occur provided no other transitions fire before). Let for $\alpha \in L$

$$E(\alpha, P) = \{e \in E(P) \mid f(e) = v^P \ \&\ lab(e) = \alpha\}$$

($E(\alpha, P)$ is the set of the α-labelled P edges, outgoing from its initial vertex). We define for every $\alpha \in L_0$ the earliest α-enabledness time of P (provided no other transitions fire before) to be

$$\mu(P,\alpha) = min\{\tilde{\mu}(P,e) \mid e \in E(\alpha,P)\}$$

The following results again are straightforward from the definitions.

Theorem 14.5 *$A \,||_{CCS}\, B \xrightarrow{\epsilon(d)} C$ if and only if*

- *for every $\alpha \in L_0$ either $\mu(A,\alpha) \geq d$, or $\mu(B,\overline{\alpha}) \geq d$, and*
- *$C = A' \,||_{CCS}\, B'$ with $A \xrightarrow{\epsilon(d)} A'$, $B \xrightarrow{\epsilon(d)} B'$.*

Theorem 14.6 *$A \,||_{CCS}\, B \xrightarrow{\alpha} C$ for $\alpha \in L_0$ if and only if $C = A' \,||_{CCS}\, B'$ for some A', B' with either*

- *$B = B'$ and $A \xrightarrow{\alpha} A'$, or*
- *$A = A'$ and $B \xrightarrow{\alpha} B'$.*

Theorem 14.7 *$A \,||_{CCS}\, B \xrightarrow{\tau} C$ if and only if $C = A' \,||_{CCS}\, B'$ for some A', B' with either*

- *$B = B'$ and $A \xrightarrow{\tau} A'$, or*
- *$A = A'$ and $B \xrightarrow{\tau} B'$, or*
- *$A \xrightarrow{\alpha} A'$ and $B \xrightarrow{\overline{\alpha}} B'$ for some $\alpha, \overline{\alpha} \in L_0$.*

In particular, these results show that, if one succeeds to interpret two Timed CCS processes (see [Wan90]) as PTPs, their parallel composition also can be interpreted as a PTP retaining the consistence with the Timed CCS process semantics.

We define also relabelling and restriction combinators of CCS (inherited also by Timed CCS) for TCCS-like Parallel Timer Processes.

Let A be a TCCS-like PTP and $g : L_0 \to L_0$ be a bijective mapping which for every $\alpha \in L_0$ satisfies $g(\overline{\alpha}) = \overline{g(\alpha)}$. We assume also $g(\tau) = \tau$ and define the process $A[g]$ (the process A, relabelled by the function g) in a way:

- $V(A[g]) = V(A)$, $\mathcal{T}(A[g]) = \mathcal{T}(A)$ and $E(A[g]) = E(A)$ (also the edge incidence functions f and t, as well as the conditions γ and timer settings ϕ are inherited by $A[g]$ from A);

- for every $e \in E(A)$ the label of e in $A[g]$

$$lab_{A[g]}(e) = g(lab_A(e)).$$

As to the restriction combinator, given a process A and a label set $K \subseteq L_0$ such that $\alpha \in K$ implies $\overline{\alpha} \in K$, the process $A\backslash K$ (A with the restriction not to participate in the events from K) is defined by erasing all edges e from A automaton which are labelled by $lab(e) \in K$.

Clearly, we have the following results which justify the soundness of the defined operators w.r.t. the original semantics of Timed CCS, given in [Wan90] (neither the bijectivity of g, nor the complement preservation is important for the proofs of Theorem 14.8 and Theorem 14.9. These properties, already present in the TCCS calculi, just guarantee some nice laws to hold in the obtained algebra):

Theorem 14.8 *For every $\sigma \in L$ $A[g] \xrightarrow{\sigma} B$ if and only if $A \xrightarrow{\sigma'} B'$ for $\sigma = g(\sigma')$, $B = B'[g]$.*

Theorem 14.9 *$A[g] \xrightarrow{\epsilon(d)} B$ if and only if $A \xrightarrow{\epsilon(d)} B'$ with $B = B'[g]$.*

Theorem 14.10 *For $\sigma \in L$ $A\backslash K \xrightarrow{\sigma} B$ if and only if $A \xrightarrow{\sigma} B'$, $B = B'\backslash K$ and $\sigma \notin K$.*

Theorem 14.11 *$A\backslash K \xrightarrow{\epsilon(d)} B$ if and only if $A \xrightarrow{\epsilon(d)} B'$ and $B = B'\backslash K$.*

Proof: Observe that while constructing $A\backslash K$ we have not eliminated any red edge from the process A graph.□

So we have defined for PTPs all the Timed CCS static combinators (parallel composition, restriction and relabelling). Let us consider also the class of *regular* TCCS processes which is constructed by the combinators

- NIL, the process constant which can do nothing but idle:
  $NIL \xrightarrow{\epsilon(d)} NIL$ for every $d \in \mathbf{Q}^{+0}$;

- action prefix $\sigma.P$ for $\sigma \in L$ and a process P denotes the process which first does σ and then behaves as P (for $\sigma = \alpha \neq \tau$ also an arbitrary delay of $\sigma.P$ is possible):

$$\sigma.P \xrightarrow{\sigma} P \quad \text{and} \quad \alpha.P \xrightarrow{\epsilon(d)} \alpha.P,$$

- *summation* $P + Q$ which has the following semantic rules:

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}, \quad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'} \text{ and}$$

$$\frac{P \xrightarrow{c(d)} P', \quad Q \xrightarrow{c(d)} Q'}{P + Q \xrightarrow{c(d)} P' + Q'},$$

as well as mutually recursive guarded defining equations (see [Wan90], or [HLW91] for the class of regular TCCS processes described more precisely).

Let us call a TCCS net every process which is obtained from regular processes by (possibly repeated) use of parallel composition, restriction and relabelling combinators (with the semantics, as defined in [Wan90]).

The modelling of regular TCCS processes as PTPs clearly can be done effectively in one or other way (one can even manage to exhibit such a translation into one-timer PTPs, if considering the semantics modelling up to the weak bisimulation equivalence), we do not consider the details. Let us just note that, observing the decidability of both strong and weak bisimulation equivalences for PTPs (Theorem 11.2 and Theorem 11.4), we have the outline of the proof of the following result.

Theorem 14.12 *The strong and weak bisimulation equivalences for Timed CCS nets are decidable.*

It is not difficult to "cook" also various other parallel composition operators for PTPs, perhaps also some making more essential use of the red edges with visible labels in the processes. If one considers EPTPs, introduced in Section 10.2 the parallel composition with the *broadcast* communication between processes, introduced in [Pra91], can also be easily modelled (actually, as the result of this modelling some kind of *timed* broadcasting process model will appear).

For some undecidability results concerning the analysis of Timed CCS processes see also Chapter 15.

Chapter 15

Timed Processes with Memory

Though the PTP model, introduced in Section 9.1, and its enrichments, considered in Chapter 10, allow to describe infinite behaviour in time with overlapping time constrained intervals (simultaneous decreasing of timers for a number of parallel system's components), these models still are not able to express every reasonable timed behaviour. As a very simple example of such behaviour one can consider an ice-hockey timer [Wan91b] which can be interrupted at any time after the initial start-up and later restarted with the value of the remaining time interval (i.e. the time left before the time-out) the same it possessed at the moment of the interruption. One can easily add to the PTP model some additional means, say, the memory cells for storing the current timer values and retrieving them afterwards in appropriate moments (it is reasonable to associate such moments with the real transitions in the process), which would allow to capture also this kind of timed behaviour.

More precisely, one obtains a parallel timer process with memory (PTPM, for short), if one adds to a given PTP

$$A = \langle\langle V, E, L, lab, T, \gamma, \phi\rangle, \langle v, \delta\rangle\rangle$$

a set $\mathcal{M}$ of *memory cells* and extends every edge timer setting $\phi(e)$ for $e \in E$ to be the timer and memory cell setting

$$\phi(e) : (T \cup \mathcal{M}) \to (T \cup \mathcal{M}) \cup \mathbf{Q}^{+0}.$$

Intuitively, the new operations to be performed while executing the transitions along some process edges are of the sort of "remember": $m_i \leftarrow t_j$ and "retrieve": $t_j \leftarrow m_i$ for $t_j \in T, m_i \in \mathcal{M}$ (the assignment of constants to memory cells and the value of one memory cell to another is of less interest).

The semantics (labelled transition system) of PTPMs can be easily obtained as a generalization of that of PTPs: every state for a PTPM A consists of its graph's vertex and a value assignment $\delta : T \cup \mathcal{M} \to \mathbf{Q}^{+0}$ both for timers and memory cells.

Every timer and memory cell setting $\phi(e)$ is "executed" every time when a transition along the edge e fires, and it works, as for PTPs, by defining the new values $\delta'(u)$ for $u \in \mathcal{T} \cup \mathcal{M}$ in a way

- $\delta'(u) = \delta(u')$, if $u' = \phi(e)(u) \in \mathcal{T} \cup \mathcal{M}$, and
- $\delta'(u) = c$, if $c = \phi(e)(u) \in \mathbf{Q}^{+0}$.

The main difference of the memory cells from the timers is that the values of the memory cells *do not decrease during the passage of time* as the timer values do (whenever

$$\langle \Phi, \langle v, \delta \rangle \rangle \xrightarrow{c(d)} \langle \Phi, \langle v, \delta' \rangle \rangle,$$

then $\delta'(m) = \delta(m)$ for every $m \in \mathcal{M}$).

The following results show that the bisimulation equivalence deciding algorithms developed for PTPs cannot be extended to cope also with PTPMs.

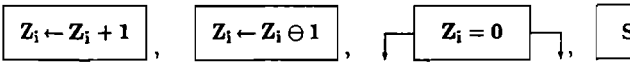
Let for any PTPM A $d(A)$ denote the set of *derivatives* of A (the least set containing A and closed under the transition relation).

We call a vertex $v \in V$ in a given PTPM $A = \langle \Phi, s_0 \rangle$ *reachable* if $\langle \Phi, \langle v, \delta \rangle \rangle \in d(A)$ for some timer and memory cell value assignment δ.

Theorem 15.1 *The vertex reachability problem for PTPMs with* 5 *timers and* 1 *memory cell is undecidable.*

Corollary 15.2 *The bisimulation equivalence problem PTPMs with* 5 *timers and* 1 *memory cell is undecidable.*

Proof of Theorem 15.1: We consider a variant of Minsky machines (defined originally in [Min67]) with the allowed operations for every counter Z_i $(i = 1, 2)$ being (compare with the proof of Theorem 8.1 in Section 8.1)



, where by the definition $x \ominus y = max\{0, x - y\}$. We assume also that every Minsky machine has 0 as the initial values of its counters. The proof of the theorem is done by reducing the halting problem of the Minsky machines to the vertex reachability problem for PTPMs.

It is easy to see that one can effectively construct for a given Minsky machine M with 0 initial values for its counters a corresponding *bounded machine* $B(M)$, with variables x_1, x_2, x_3 and allowed operations over them

- $x_i \leftarrow x_i/2$ for $i = 1, 2, 3$,
- $x_i \leftarrow x_i + x_1$, $x_1 \leftarrow x_i \ominus x_1$ for $i = 2, 3$,

- tests $x_i = 0$ with 2 exits (labelled " + " and " − "), $i = 2, 3$,

setting the variable values initially as $x_1 = 1/2$, $x_2 = 0$, $x_3 = 0$, in a way that

- every M vertex v has its corresponding vertex $b(v)$ in $B(M)$ graph, reachable in $B(M)$ iff v is reachable in M,
- during the $B(M)$ execution all variable x_i values remain strictly less than 1,
- if the value of x_i $(i = 2, 3)$ differs from the value of x_j or from some of constants $0, 1$, then the corresponding difference it is not less than the value of x_1.

The idea of the modelling of M work in $B(M)$ consists in representing M counter values Z_1 and Z_2 as the values of fractions x_2/x_1 and x_3/x_1 (x_1 works as the "unit" to measure the values of the "counters" x_2 and x_3). In any case, when M executes a counter increase instruction, $B(M)$ first decreases all its variable values by half and executes the corresponding increase (by x_1) instruction afterwards.

Now in order to complete the proof it remains to show how to model (in the sense of the vertex reachability) the bounded machine's $B(M)$ work by a finite PTPM $P(M)$ with 5 timers and 1 memory cell.

Let us denote the $P(M)$ timers by t_0, t_1, t_2, t_3, t_4 and the memory cell by m. The set of $P(M)$ graph vertexes contains, first, all $B(M)$ vertexes (the other vertexes will appear further on during the modelling construction). Let $P(M)$ have also a special vertex @ from which no other vertexes are reachable. The construction of $P(M)$ is done by the independent building of fragments corresponding to the edges in $B(M)$ in a way described below.

Also during the runtime $P(M)$ is intended to simulate every $B(M)$ instruction execution by executing the fragment which corresponds to this instruction. In order to achieve such a simulation one of $P(M)$ timers, t_0, is used for giving time-outs after every full initially fixed time unit. The timers t_1, t_2, t_3 are intended to have at the t_0 time-out moments the corresponding values of $B(M)$ variables x_1, x_2, x_3 (the timer t_4 and the memory cell m are playing an auxiliary role in the modelling constructions).

The key idea of the modelling is in the way how to keep the value of some timer t_i unchanged in some execution step (between two adjacent t_0 timeouts; notice that the timers always decrease synchronously with the passage of time). The solution to this problem consists in the setting of the value of this timer to 1 (by a transition along a red edge) immediately after it reaches the 0 value (the time intervals between adjacent t_0 timeouts are also precisely 1). Further on, if the value of some timer is the subject of change during some execution step, the moments of transition firing in the process can be used to make the necessary timer and memory cell value setting (to make all the assignments contained in this setting).

Every edge in $P(M)$ is described below by its condition and timer and memory cell setting, as well as its source and target vertexes (we do not count for edge labels, as they do not have any effect on the vertex reachability). Unless otherwise stated, any edge in $P(M)$ is assumed to be *red* (every transition along the edge must be executed as soon as it is enabled). Let us call an edge e, with the condition $\gamma(e) = \{t_i\}$ (stating that a transition along e can be executed whenever $\delta(t_i) = 0$) and setting $t_i \leftarrow 1$ a *simple edge* with t_i. When a simple edge has the same source and target vertexes, it is called also a *simple loop*. Now we can describe the modelling construction itself.

If $B(M)$ has an edge from v_a to v_b, associated with the execution of an operation $x_i \leftarrow x_i + x_1$ $(i = 2, 3)$, in $P(M)$ this edge is modelled by:

- the assignment $m \leftarrow t_1$ (remember) added to every edge, incoming into v_a;
- simple loops with t_1 and t_{5-i} around v_a;
- an edge from v_a to v'_a with the condition $\{t_i\}$ and the setting $t_i \leftarrow m$ (retrieve);
- simple loops with t_1, t_2 and t_3 around v'_a;
- a simple edge from v'_a to v_b with t_0.

If $B(M)$ has an edge from v_a to v_b, associated with the execution of an operation $x_i \leftarrow x_i \ominus x_1$ (i=2,3), in $P(M)$ this edge is modelled by, first, the fragment for checking that $\delta(t_i) \neq 1$ (it can be the case that t_i has been set to 1 right before the process $P(M)$ enters the vertex v_a (by executing some other transition at the same time moment), we have to eliminate such a possibility), which consists of:

- the assignment $m \leftarrow t_1$ added to every edge, incoming into v_a;
- a simple edge with t_i from v_a to v'_a, setting also $t_4 \leftarrow m$;
- edges with the condition $\{t_0\}$ from v'_a to @ and from v_a to @;
- an edge with the condition $\{t_4\}$ from v'_a to v''_a (here we use the idea that, if $\delta(t_i) < 1$, then also $\delta(t_i) + \delta(t_1) \leq 1$ ($\delta(t)$ here denotes the timer's t value at the last t_0 timeout moment), which in its turn is justified by the abovementioned invariant relation between the values of x_i's and t_i's at t_0 timeout moments; the x_i's cannot be too close to 1 according to the $B(M)$ construction);
- simple loops with t_1 and t_{5-i} around v_a, v'_a, v''_a;
- a simple edge with t_0 from v''_a to w_a.

The following $P(M)$ fragment from w_a to v_b performs the actual subtraction, provided that the value of t_i initially does not happen to be 1:

- a simple loop with t_{5-i} around w_a;
- simple edges from w_a to w'_a with t_1 and t_i, both containing also assignments $m \leftarrow t_i$;
- simple loops with t_1, t_2, t_3 around w'_a;
- a simple edge from w'_a to v_b with t_0, assigning also $t_i \leftarrow m$ (retrieve).

If $B(M)$ has an edge from v_a to v_b associated with the operation $x_i \leftarrow x_i/2$, it is modelled in $P(M)$ the following way:

- simple loops with t_1, t_2, t_3 around v_a;
- a black edge with $\emptyset$ condition and setting $t_4 \leftarrow 1$ around v_a;
- a simple edge with t_0 from v_a to v'_a, setting also $m \leftarrow t_4$ (the memory cell m now (in the moment of the execution along this edge) contains the actual (relative) moment of the black transition firing, it will turn out that the further normal execution of the process is possible only if $\delta(t_i) \neq 1$ and $\delta(t_i) = 2 \times \delta(m)$);
- an edge with the condition $\{t_i\}$ from v'_a to @;
- an edge with the condition $\{t_4\}$ and setting $t_4 \leftarrow m$ (retrieve) from v'_a to v''_a;
- edges with the conditions $\{t_4\}$ and $\{t_i\}$ from v''_a to @;
- an edge with the condition $\{t_i, t_4\}$ and setting $t_4 \leftarrow 1$ from v''_a to v'''_a;
- a simple edge with t_0 from v'''_a to w_a, assigning also $t_i \leftarrow m$;
- simple loops with t_j, t_k (those of t_1, t_2, t_3 which are not t_i) around v'_a, v''_a, v'''_a;
- fragment for checking that $\delta(t_4) \neq 1$, as described above (in the successful case leads from w_a to v_b).

If $B(M)$ has edges from v_a to v_b, if $x_i = 0$, and to v_c, if $x_i \neq 0$, $(i = 2, 3)$, then $P(M)$ contains the following fragment for modelling this decision:

- from v_a to w_a a test for $\delta(t_i) \neq 1$;
- a simple loop with t_{5-i} around w_a;
- a simple edge with t_1 from w_a to w'_a (in the case if the process has entered w'_a, clearly, $\delta(t_i) \geq \delta(t_1) > 0$ ($\delta(t)$ here stand for timer values at the last t_0 timeout moment));

- simple loops around w'_a with t_2, t_3;
- a simple edge with t_0 from w'_a to v_c;
- a simple edge with t_i from w_a to w''_a;
- simple loops with t_1, t_{5-i} around w''_a;
- an edge with the condition $\{t_0\}$ from w''_a to @;
- an edge with the condition $\{t_0, t_i\}$ and the setting $t_0 \leftarrow 1$ from w''_a to v_b.

One defines the initial vertex of $P(M)$ to be that of $B(M)$, the initial timer value vector for $P(M)$ is $\delta_0 = 1$, $\delta_1 = 0.5$, $\delta_i = 0$ for $i = 2, 3, 4$, $\delta(m) = 0$. Now it is not difficult to see that a vertex v_x in $B(M)$ is reachable if and only if the same vertex v_x is reachable in $P(M)$, so the decidability of the vertex reachability problem for timed processes with memory cells would imply the decidability of the reachability (halting) problem for Minsky machines, a contradiction.□

A similar undecidability result can be obtained for PTPMs with 2 timers and 4 memory cells (3 memory cells can simulate the bounded machine's $B(M)$ variables x_1, x_2 and x_3, the timers and the fourth memory cell are enough for performing the bounded machine's operations). For PTPMs with 1 timer and n memory cells both the strong and weak bisimulation equivalence problems are easily decidable.

In fact, the undecidability result obtained here does not depend crucially on the exact PTPM model chosen here, but is inevitable for every sufficiently general timed formalism in which the timers are not *uniform*, i.e. they can change their values with different speeds, (here - some timers can be held, i.e. stored into memory cells (with values decreasing with the speed 0), the others decrease synchronously with the passage of time (their value decreasing speed is 1 (or, the value change speed is -1, if one wants to measure positively the increasing timers (clocks))).

The assignment operations between timers and memory cells, used in the modelling of the bounded machine $B(M)$ in the proof above, can be easily replaced by simple "holding" and "releasing" of timers (any timer setting $t_1 \leftarrow t_2$ can be done by setting both t_1 and t_2 to 1 at the moment when t_2 has reached 0 in the modelling schema above).

As a particular example of such a "different speed timer" formalism one can consider the process calculi Timed CCS (TCCS) (the version with time variables, defined in [Wan91a, Wan91b], not the simplest version of TCCS nets, considered in Section 14.2 and defined in accordance with [Wan90]). We assume the TCCS processes provided with the process constants with explicit time parameters (or, equivalently,

the **rec** combinator for time-parameterized processes)[1]. A typical sequential Timed CCS process is (compare Section 14.2)

$$\epsilon(2).a.\epsilon(3).b@t.P(t),$$

where a and b are action names, $\epsilon(d)$ stands for a delay of d time units, the $b@t$ construction records into the time variable t the time of the actual b action firing w.r.t. the moment, when it was enabled (offered to the environment) first. After the execution of $b@t$ the free variable t in $P(t)$ is bound by the value recorded by $@t$. The TCCS contain also all usual CCS (see [Mil80, Mil89]) combinators (sum, parallel composition, hiding, renaming), for the precise definition of TCCS one can look in, say, [Wan91a].

In order to obtain the undecidability of the bisimulation equivalence for TCCS processes it is enough to consider the class $\mathcal{R}$ of "regular" processes (built without the parallel composition, restriction and relabelling) with time variables (one can allow only left-associative "positive minus" $\ominus$ operation in time expressions).

Really, given a PTPM P, one can associate with every vertex v of P one Timed CCS process constant $K(v)$ (with time parameters, carrying the P timer value information at the states, associated with v). The defining equation for this constant is obtained as the sum of expressions corresponding to all outgoing edges from v (we do not consider the technical details).

One of the reasons for introducing the time variables in the Timed CCS [Wan91a], clearly, was the desire to obtain algebraic timed process calculi with interleaving semantics (the *expansion theorem*, allowing to eliminate (up to the bisimulation equivalence) the parallel composition from the process specifications). However, as it can be seen from Theorem 14.12 and the results of this chapter, the proposed method in [Wan91a] of obtaining algebraically nicer calculi destroys other nice property of them, namely, the decidability of the bisimulation equivalence (as well as any other nontrivial algorithmic problems over the processes).

So, a dilemma of expansion theorem vs. decidability for timed specification formalisms appears. It looks likely to be the case that there cannot be any *interleaving* timed specification formalism which would be rich enough (PTPs, or TCCS-nets included) and have decidable at least the vertex reachability (or, stronger, bisimulation equivalence) problems, however, this requires some further investigation.

[1]There are a number of principal examples of using process constants with time parameters in TCCS [Wan91b] (including also the abovementioned example of ice-hockey timer), though the original formal syntax and semantics of the calculi in [Wan91a, Wan91b] for the sake of simplicity eliminate these processes.

Chapter 16

Conclusions

In the thesis work a variety of results characterizing the decidability of the vertex reachability, infinite behaviour possibility and bisimulation equivalences are obtained for various classes of data and time dependent real time system specification formalisms (see Introduction to the thesis for a detailed outline of the obtained results and some discussion on the related work). Here we give just a brief summary of the obtained results with some points made also to the possible emerging applications and methodological interpretation, some possible future research lines are also mentioned. In Section 16.1 we give some points about the most important proof methods, used to obtain the results of the thesis.

The first group of the thesis results is devoted to effective characterizations of the sets of finite and infinite feasible paths in simple theoretical programming languages LBASE, LTIM, LTIBA and some their modifications. Regarding the program vertex reachability problem and the effective symbolic characterizations of the sets of all finite feasible program paths for these languages, a rather complete solution is obtained (observe the path set projectivity property, proved in Theorem 4.1). We give also some evidence for the "unextendability" of the obtained results to a "more powerful" specification formalism (as the language LTIM' appears to be).

As to the infinite path feasibility in the programs, the obtained positive result tell us about the decidability of the infinite feasible path *existence* in a given program in any of these languages (also the existence of an infinite feasible fair path, etc. can be decided, see Section 7.5 for the result summary). Regarding the infinite feasible path set characterizations, these path sets only in LTIM programs obey, in general, some kind of the projectivity property. An interesting theoretical problem for future investigation could be the looking, if there exists for the infinite feasible path sets in LBASE programs some effective characteristics of *any kind* (not just the projectivity (and F-projectivity) disproved here for this purpose), one can observe the relations of this problem to the inductive solving possibilities of simple infinite inequality systems over integers.

We have investigated also the algorithms for computing the histories along the finite and infinite feasible paths in LTIBA programs (see Corollary 5.16 and Corollary 7.23). These algorithms can be used in the automatic generation of complete test sets for *deterministic* LTIBA programs (we assume the testing completeness criterion to be C_1 – the covering of all feasible branches of the program). Let us consider the LTIBA programs with acceptance conditions (both for finite and infinite paths) and call a test for a program (a sequence of the values, appearing on the program input gate) correct, if it guides the program behaviour along a finite or infinite *accepting* path. We call a program branch *correctly feasible*, if it is executed on some correct test for the program. According to Corollary 7.23 (in fact, we use just a slightly modified version of it) we can for every program branch determine whether it is correctly feasible, and in the case it is, exhibit an initial (finite or infinite) program history along an accepting path, which contains the given branch. Next, it is easy to get for every correctly feasible branch a test forcing the program execution in accordance with the corresponding computed history. This way we obtain the algorithm for generating a finite set of correct tests, covering all correctly feasible branches of the given LTIBA program (one may prefer either the generating algorithms for infinite correct tests in the spirit of Definition 3.3, or the finite initial fragments of these tests to be present in the finite correct CTS, either of these requirements can be satisfied).

The results obtained regarding the models with real time semantics fall again in two principal groups. First, we have the various results characterizing the decidability of the process vertex reachability and related problems both for the considered basic PTP model and a number of its enrichments. The positive results of this kind are obtained by modelling the considered real time system specification formalisms in the (various sublanguages of) the above studied programming language LTIBA. This modelling is, first, illustrative in that it allows to see the relations between the analysis of formalisms with real time semantics and that of programs with simple "ordinary" data structures (variables, counters). Perhaps more important point in this modelling is that it allows to transfer to the considered PTP enrichments (notice especially the data-dependent formalism $PTPB_0$) the analysis automation algorithms, developed for LTIBA programs.

It should be noted that regarding the PTPB model family (PTPBs, $PTPB_0$s and PTPQs) some further work is to be done in order to provide them with the compositional semantics, however, already the obtained results, showing the decidability of the vertex reachability problem for $PTPB_0$s and undecidability for PTPBs seem to be imposing rather natural constraints on data and time dependent r.t.s. specification formalisms with real time semantics, which are to be met in order to have decidable at least the vertex reachability problem.

In the same "reachability result" group for the specification models with real

time semantics fall also in a slightly more intuitive way presented approach to the SDL process analysis and testing automation, considered in Appendix B. One can find in this appendix the illustration of the application of some (already previously existing) general program reachability graph minimization methods in "tailoring" the elaborated theoretical LTIBA program analysis algorithms to some slightly more practical situations.

The second group of results regarding the specification formalisms with real time semantics deal with another kind of very natural real time system analysis problem, i.e. we show the decidability of strong and weak bisimulation equivalences for Parallel Timer Processes and some of their enrichments. The results of this group are obtained without any use of intermediate models, however, the used methods are, in principle, the same, as exploited to obtain the LBASE and LTIM program analysis results (finite partitioning of process state sets (though with some nuances here), the uniform mapping technique, see Section 16.1 for some further comments). It is interesting to point out also that the bisimulation equivalence can be proved decidable for all considered real time specification formalisms with the "limit closed" sets of feasible paths (PTPs, bounded PTPNs, PTPQs, also ATGs, as it can be seen from Appendix A - the models in which every infinite path is feasible whenever all its finite prefixes are feasible; for PTPUs and EPTPs the bisimulation equivalence can also be shown decidable in a similar way). As to the r.t.s. specification formalisms with the decidable vertex reachability problem, but not limit closed path sets ($PTPB_0$s and unbounded PTPNs) the bisimulation equivalence deciding problems are left open here, actually their decidability is hardly believable.

In the thesis we have considered a number of similar specification formalisms over discrete (integer) and dense (rational) data and time "domains" (the languages LTIM and LTIM', LBASE and LBASQ, the data-dependent formalisms PTPB and PTPQ, etc.). As one can observe, for all such situations in the case of the dense variable value space *stronger* analysis automation results can be obtained (save the exception of "positive" LTIM and LTIM' programs, considered in Section 8.3). The seeming contradiction with usually the opposite situation when some discrete systems can be analyzed by the standard FSM "state exploration" techniques, while the corresponding dense systems cannot due to their "essentially infinite" state space, can be removed observing that the difficulties in the discrete system analysis are brought in by the presence of *infinite* discrete structures (such as the integer numbers are), see Section 8.3 and Section 13.2 for some additional comments regarding this comparison. We just note that for an analogue of PTPMs over the discrete time domain all the problems, which have proved undecidable for PTPMs themselves (as defined over the dense domain in Chapter 15), can be decided by rather standard FSM analysis techniques.

It can be noted also that the thesis does not discuss the *complexity* issues of the considered deciding algorithms. In fact, all the principal obtained algorithms (vertex reachability and infinite behaviour possibility for LTIBA programs, strong and weak bisimulation equivalences for PTPs and their extensions) are, roughly speaking, worst-case polynomial in the size of the programs (processes) and exponential in the program variable (timer) set cardinality. The calculation of the theoretical worst-case bounds for the algorithm complexity is an interesting problem, however, it seems to contain no principally new ideas. These bounds also seem to be not very illuminating for the algorithm work characteristic in real situations, for these reasons, as well as for the sake of brevity, they are omitted.

Regarding the practical aspects of the possible obtained algorithm implementation and optimization some points are given in Appendix B. There exists also a wide variety of further more or less general resource-saving methods in the state-space exploration algorithms, one can see [BKM91], [Lar92] and [VC91] to obtain at least some intuition about some possible methods of this kind. Hopefully, some of these methods could be used also in the optimization of the deciding algorithms presented here.

16.1 Proof Techniques

In this section we give some points about the main methods used to obtain the results of the thesis.

Regarding the considered undecidability results, all of them were obtained by reducing some well-known unsolvable problem to the examined one, this is the most common way to cope with this kind of problems in general.

More important, some interesting methods can be found in the proofs showing the *decidability* of one or other algorithmic problem.

First, most of the obtained principal results use the idea of the infinite program or process state set partitioning into a finite set of classes, each class of the states having all "essential" properties in common, so making it possible to consider such a class as *one vertex* in a finite model of the system (e.g., its path feasibility graph), see Section 4.1, Section 11.2 and Section 13.2 for some applications. This principle is not in general novel and has already been successfully used, for instance, in [Auz84b] and [ACD90]. A novel its application in the thesis is associated with the partitioning of *process pair state sets* when deciding bisimulation equivalences for PTPs (see Section 11.2 for details).

Second, we have proposed in the thesis an original method of *uniform mappings* (see Section 4.3, Section 4.5, Section 11.2 and Section 13.2 for some applications) for formalizing the relations between the program/system work histories which contain at some point different elements from one program state set equivalence class (consider especially Proposition 11.12, as well as Lemma 4.22, Proposition 4.32 and Proposition 4.33).

Third, a number of important results in Part I were formulated in the terms of the path set *projectivity*, introduced in Section 2.2. It seems that the path set projectivity could be just that abstraction in the terms of which the feasible path sets of programs with data dependencies should be characterized in order to have the possibility to obtain as the corollary the applicability of various FSM analysis techniques to the considered data dependent formalisms. The projective path sets in the graphs are shown to obey simple algebraic properties, as illustrated in Section 2.2. However, this methodology is applied in the thesis just to obtain concrete results, no general investigation about its suitability is performed.

Bibliography

[AD90] R. Alur and D.Dill, *Automata for Modelling Real-Time Systems*, LNCS 443, 1990.

[ACD90] R. Alur, C.Courcoubetis and D.Dill, *Model-Checking for Real-Time Systems*, Proceedings from LICS'90 pp. 414-425, 1990.

[ABBCK91] A. Auziņš, J. Bārzdiņš, J. Bičevskis, K. Čerāns and A. Kalniņš, *Automatic Construction of Test Sets: Theoretical Approach*, in Baltic Computer Science, LNCS, No. 502, 1991.

[Auz84a] A.I.Auzins, *On the Construction of Complete Sample Systems*, Dokl. Akad. Nauk SSSR, Vol.288, No.3, 1984 (In Russian).

[Auz84b] A.I.Auzins, *Decidability of Reachability for the Relational Push-Down Automata*, Programmirovanie, No.3, 1984 (In Russian).

[BB90] J. Baeten and J. Bergstra, **Real Time Process Algebra**, Technical Report, Center for Mathematics and Computer Science, Amsterdam, 1990.

[BBK74] J.M.Barzdin, J.J.Bicevskis and A.A.Kalninsh, *Construction of Complete Sample Systems for Program Testing*, in Latv. Gosudarst. Univ. Uch. Zapiski, Vol.210, 1974 (In Russian).

[BBK77] J.M.Barzdin, J.J.Bicevskis and A.A.Kalninsh, *Automatic Construction of Complete Sample Systems for Program Testing*, Proc. IFIP Congress, 1977, North-Holland 1977.

[BKA89] J.M.Barzdin, A.A.Kalnins and M.I.Auguston, *SDL Tools for Rapid Prototyping and Testing*, in SDL'89: The Language at Work, North-Holland, 1989

[BKB74] J.M.Barzdin, A.A.Kalninsh and J.J.Bicevskis, *Decidable and Undecidable Cases of the Problem of Construction of Complete Sample Systems*, in Latv. Gosudarst. Univ. Uch. Zapiski, Vol.210, 1974 (In Russian).

[BKM91] J.Borzovs, A. Kalniņš and I.Medvedis *Automatic Construction of Test Sets: Practical Approach*, in Baltic Computer Science, LNCS, No. 502, 1991.

[BM83] B.Berthomieu and M.Menasche, *An Enumerative Approach for Analyzing Time Petri Nets*, Proc. IFIP Congress, 1983, North-Holland, 1983.

[BM79] R. Boyer and J.S.Moore, **A Computational Logic**, Academic Press, New York, 1979.

[CC89] CCITT, **Functional Specification and Description Language (SDL)**, Recommendations Z.100, 1989.

[Cer91] K. Čerāns, *Decidability of Bisimulation Equivalence for Parallel Timed Processes*, to appear in Proc. of Chalmers Workshop on Concurrency, Gōteborg, 1991.

[Cer92a] K. Čerāns, *Feasibility of Finite and Infinite Paths in Data Dependent Programs*, to appear in Proc. of LFCS'92, Russia, Tver, 1992.

[Cer92b] K. Čerāns, **Decidability of Bisimulation Equivalences for Processes with Parallel Timers**, Technical report, Institute of Mathematics and Computer Science, University of Latvia, Riga, 1992.

[Cer92c] K. Čerāns, *Decidability of Bisimulation Equivalences for Parallel Timer Processes*, to appear in Proc. of CAV'92, Montreal, 1992.

[CES86] E.M.Clarke, E.A.Emerson and A.P.Sistla, *Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, Vol.8, No.2, 1986.

[Che91] L. Chen, *Decidability and Completeness in Real Time Processes*, LFCS, Edinburgh University, 1991.

[Dan92] M.Daniels, *Modelling Real-Time Behaviour with an Interval Time Calculus*, Proc. of Conference on Formal Techniques in RT and FT Systems, Nijmegen, 1992

[Em91] E.A.Emerson, *Temporal and Modal Logic*, in Handbook of Theoretical Computer Science, B, The MIT Press/Elsevier, 1991.

[GMMP89] C.Ghezzi, D.Mandrioli, S.Morasca and M.Pezze *A General Way To Put Time in Petri Nets*, ACM SIGSOFT Eng. Notes, Vol. 14, No. 3, 1989.

[GMW79] M.Gordon, R.Milner and C.Wadsworth, **Edinburgh LCF**, LNCS No.78, Springer-Verlag, 1979.

[HLW91] U. Holmer, K.Larsen and Yi Wang, *Deciding Properties for Regular Real Timed Processes*, in Proc. of CAV'91, 1991.

[Hoa85] C.A.R. Hoare, **Communicating Sequential Processes**, Prentice Hall, 1985.

[HR90] M.Hennesy and T.Regan, *A Temporal Process Algebra*, Technical Report 2/90, University of Sussex, 1990.

[ISO89a] ISO, **LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour**, ISO 8807, 1989.

[ISO89b] ISO, **Estelle: A Formal Description Technique Based on an Extended State Transition Model**, ISO 9074, 1989.

[Kal91] A.Kalniņš, *Global State Based Automatic Test Case Generation for SDL*, in SDL'91: Evolving Methods, North-Holland, 1991.

[KS83] P.C. Kannelakis and S.A. Smolka, *CCS Expressions, Finite State Processes and Three Problems of Equivalence*, ACM Symposium on Principles of Distributed Computing, 1983.

[KE89] J.Karlson and A.Ek, *SSI - An SDL Simulation Tool*, in SDL'89: The Language at Work, North-Holland, 1989.

[Lar92] K.G.Larsen, *Efficient Local Correctness Checking*, to appear in Proc. of CAV'92, Montreal, 1992.

[MF76] P. Merlin and D.J. Farber, *Recoverability of Communication Protocols*, IEEE Trans. on Communication Protocols, Vol. COM-24, No. 9, 1976.

[Mil80] R. Milner, *A Calculus of Communicating Systems*, LNCS No. 92, 1980

[Mil89] R. Milner, **Communication and Concurrency**, Prentice Hall International Series in Computer Science, 1988.

[Min67] M.Minsky, **Finite and Infinite Machines**, Prentice Hall, Englewood Cliffs, N.Y., 1967.

[MT90] F.Moller and C.Tofts, *A Temporal Calculus of Communicating Systems*, LNCS No. 458, 1990.

[NSY91] X.Nicollin, J.Sifakis and S.Yovine, *From ATP to Timed Graphs and Hybrid Systems*, in Proc. of REX Workshop "Real-Time: Theory in Practice", The Netherlands, 1991.

[Par81] D.Park, *Concurrency and Automata on Infinite Sequences*, LNCS No. 104, 1981.

[Pe62] C.A.Petri, **Kommunikation mit Automaten**, Schriften des Institutes für Instrumentelle Matematik, Bonn, 1962.

[Pra91] K.V.S. Prasad, *A Calculus of Broadcasting Systems*, TAPSOFT'91, in LNCS No. 493, 1991.

[Re85] W.Reisig, **Petri Nets. An Introduction**, Springer-Verlag, 1985.

[RR86] G.M. Reed and A.W. Roscoe, *A Timed Model for Communicating Sequential Processes*, LNCS No. 226, 1986.

[Sch91] S.A.Schneider, *An Operational Semantics for Timed CSP*, Oxford University Technical Report TR 1-91, 1991.

[SVW87] A.P.Sistla, M.Y.Vardi, P.Wolper, *The Complementation Problem for Büchi Automata With Applications to Temporal Logic*, Theoretical Computer Science No.49, 1987.

[VC91] A.Valmari and M.Clegg, *Reduced Labelled Transition Systems Save Verification Effort*, Proc. of CONCUR'91, Amsterdam, 1991.

[Wan90] Yi Wang, *Real Time Behaviour of Asynchronous Agents*, Proc. of CONCUR'90, LNCS No. 458, 1990.

[Wan91a] Yi Wang, *CCS + Time = an Interleaving Model for Real Time Systems*, ICALP'91, Madrid, 1991.

[Wan91b] Yi Wang, **A Calculus of Real Time Systems**, Ph.D. theses, Chalmers Univ. of Technology, Göteborg, 1991.

Appendix A

Bisimulations for Action Timed Graphs

We have already obtained some results about the deciding bisimulation equivalences for timed specification formalisms, other than PTPs (see Theorem 14.12).

In this chapter we consider one more model, the Action Timed Graphs, introduced in [NSY91] and being perhaps the closest specification formalism to PTP. We show, how the strong and weak bisimulation deciding algorithms, developed for PTPs in Chapter 11 and Chapter 12, can be modified to apply for deciding bisimulation equivalences for ATGs. Some relations between the ATG model and the PTPs are already discussed in Section 1.6.

We have some slight technical changes in the presentation of the ATG model, if compared with the original paper [NSY91], made mostly for the sake of some uniformity with other presented material. Our definition of ATGs is also *more specific* than in that [NSY91] for we describe precisely the class of graphs being the subject of automated analysis (in the way, borrowed from the definition of Timed Graphs in [ACD90]), this class of graphs is rich enough to cover all the applications, considered in [NSY91].

As in the case of PTPs, we assume L to be a predefined set of events. Let $G = (V, E, f, t, L, lab, v_0^G)$ be a finite edge-labelled graph with v_0^G as the initial vertex.

A *clock* is defined to be a variable taking its values from $\mathbf{Q}^{+0}$. For G being the edge-labelled graph and $\mathcal{T} = \{t_1, \ldots, t_n\}$ being a finite set of clocks, an *Action Timed Graph* is defined by associating with every edge $e \in E$

- a *time condition* $b(e) \in \mathcal{B}(\mathcal{T})$ for $\mathcal{B}(\mathcal{T})$ being the set of all *linear* predicates over the clock value set $(\mathbf{Q}^{+0})^n$ (every predicate $b \in \mathcal{B}(\mathcal{T})$ is represented by a boolean formula built by the standard logical connectives $\vee, \&$ and $\neg$ from the elementary predicates in the forms $t_i \lambda c$, $t_i \lambda t_j$ for $t_i, t_j \in \mathcal{T}$, $c \in \mathbf{Q}^{+0}$ and $\lambda \in \{<, \leq, \geq, >\}$); and

- a clock *reset* set $r(e) \subseteq \mathcal{T}$.

Given an arbitrary clock value vector $\vec{t} = \langle t_1^0, \ldots, t_n^0 \rangle \in (\mathbf{Q}^{+0})^n$ and a predicate $b \in \mathcal{B}(\mathcal{T})$, the interpretation of b over $\vec{t}$ yields either the value true, or the value false in a standard way.

Let for any clock value vector $\vec{t} = \langle t_1^0, \ldots, t_n^0 \rangle$ and $d \in \mathbf{Q}^{+0}$ the vector $\vec{t} + d$ be $\langle t_1^0 + d, \ldots, t_n^0 + d \rangle$.

Let $\hat{b}$ for $b \in \mathcal{B}(\mathcal{T})$ be a predicate which is true for given clock value vector $\vec{t}$ iff $b(\vec{t} + d)$ holds for some $d \in \mathbf{Q}^{+0}$ (intuitively, $\hat{b}$ holds sometimes in the future, see the operational semantics). It can be proved that for b being a linear predicate, the predicate $\hat{b}$ also can be represented as linear.

Following [NSY91] we define for every vertex $v \in V$ the predicate $\mathbf{en}(v)$ over the clock value vectors as

$$\mathbf{en}(v) \stackrel{\text{def}}{=} \bigcup \{b(e) \mid f(e) = v\}$$

(the predicate $\mathbf{en}(v)$ gives all clock value vectors for which the predicate $b(e)$ for *some* edge e outgoing from v is satisfied).

Given the set of clocks $r \subseteq \mathcal{T}$ let $\phi_r : (\mathbf{Q}^{+0})^n \to (\mathbf{Q}^{+0})^n$ be the function which resets the values of the clocks $t \in r$ to 0 and does not change the values of other clocks, i.e. we have

$$\phi_r(\langle t_1^0, \ldots, t_n^0 \rangle) = \langle t_1^1, \ldots, t_n^1 \rangle \text{ for}$$

- $t_i^1 = 0$, if $t_i \in r$, and
- $t_i^1 = t_i^0$, if $t_i \in (\mathcal{T} \setminus r)$.

The operational semantics of ATGs ([NSY91]) is given as a labelled transition system based on the relations $\stackrel{\sigma}{\longrightarrow}$ for $\sigma \in L$ and $\stackrel{\epsilon(d)}{\longrightarrow}$ for $d \in \mathbf{Q}^{+0}$. It is defined to have the states $\langle v, \vec{t} \rangle$ for $v \in V$ and $\vec{t} \in (\mathbf{Q}^{+0})^n$ (the initial state is $\langle v_0^G, \vec{0} \rangle$) and the transitions in accordance with the following two rules

$$\frac{v = f(e), v' = t(e), b(\vec{t})}{\langle v, \vec{t} \rangle \stackrel{\sigma}{\longrightarrow} \langle v', \phi_{r(e)}(\vec{t}) \rangle} \quad \text{and} \quad \frac{\widehat{\mathbf{en}(v)}(\vec{t} + d)}{\langle v, \vec{t} \rangle \stackrel{\epsilon(d)}{\longrightarrow} \langle v, \vec{t} + d \rangle}.$$

Intuitively, the first semantic rule tells that a real transition can be executed from the state $\langle v, \vec{t} \rangle$ to the state $\langle v', \vec{t}' \rangle$ at possibly another node, if the clock value vector $\vec{t}$ satisfies the predicate b, and that, during this execution the clock value vector changes from $\vec{t}$ to $\phi_{r(e)}(\vec{t})$ (some clocks are reset to 0). As to the second rule, it expresses both the condition on the possible execution remaining (waiting) at the same node (stating that some transition from this state must remain *eventually enabled*) and the rule for the clock value changing during the waiting (synchronous increasing).

We outline the bisimulation equivalence deciding algorithm for ATGs just by stressing the different points in it, if compared with the bisimulation equivalence deciding for PTPs, demonstrated in Chapter 11 and Chapter 12.

The weak bisimulation equivalence (based on the relations $\stackrel{\sigma}{\Longrightarrow}$ and $\stackrel{\epsilon(d)}{\Longrightarrow}$) for ATGs is defined by abstracting from some "internal" actions in the standard way (e.g. as for PTPs in Section 11.1), we do not consider the details here.

Given two ATGs $G_1 = (N_1, T_1, n_0^1, \longrightarrow_1)$ and $G_2 = (N_2, T_2, n_0^2, \longrightarrow_2)$, observe first that, the constants, used in the predicates $b \in B(T_i)$ are integers by definition.

As in the case of PTPs we define the time regions $R(s_1, s_2)$ (symbolic processes) of state pairs (s_1, s_2) for $s_i = (n_i, \vec{t}_i)$ being a state of G_i with the only difference that we do not care for the ordering of the clock values, greater than c_{max} - the largest constant, used in the given graph transition clock value predicates (this idea is explained in details for the time regions built for one TG in [ACD90]).

In order to obtain the analogues of Proposition 11.12, Fact 11.9 and Lemma 11.7, which form the cornerstone for deciding bisimulation equivalences in the PTP case, we have to change the definition of the uniform mapping slightly.

Definition A.1 *A mapping $\rho : \mathbf{Q} \to \mathbf{Q}$ is called c_{max}-uniform, iff*

- *ρ is strongly monotone,*
- *$\rho(x - c) = \rho(x) - c$ for $x \leq c_{max}$ and $c \in \mathbf{N}$ (it is important that this equality holds also for $c > x$), and*
- *$\rho(0) = 0$.*

Let for a c_{max}-uniform mapping ρ and $x, d \in \mathbf{Q}$

$$\overline{\rho}(d) = c_{max} - \rho(c - d) \text{ and } \rho_d(x) = \rho(x - d) + \overline{\rho}(d).$$

Proposition A.2 *For s_1 and s_2 being the states of either G_1, or G_2, satisfying $s_2 = \rho(s_1)$ for the c_{max}-uniform mapping ρ, we have*

- *whenever $s_1 \stackrel{\sigma}{\Longrightarrow} s_1'$, then $s_2 \stackrel{\sigma}{\Longrightarrow} \rho(s_1')$, and*
- *whenever $s_1 \stackrel{\epsilon(d)}{\Longrightarrow} s_1'$, then $s_2 \stackrel{\epsilon(\overline{\rho}(d))}{\Longrightarrow} \rho_d(s_1')$.*

Lemma A.3 *For s_1, s_1' being the states of G_1 and s_2, s_2' being the states of G_2 with $R(s_1, s_2) = R(s_1', s_2')$ always $s_1 \approx s_2$ if and only if $s_1' \approx s_2'$.*

We do not consider the proofs of the proposition and the lemma in detail here, they can be done using a similar schema, as in the PTP case (observing the definition

peculiarities of ATGs).

Observe that for Action Timed Graphs it is not always true that the existence of the transitions $s \xrightarrow{c(d')}$ for all $d' < d$ *implies* the existence of the "limit" transition $s \xrightarrow{c(d)}$, as is was in the PTP case (see Fact 11.13). If, however, for some ATG G all the transition clock predicates are built by positive logical connectives from "upwards closed" elementary predicates (i.e. not admitting the form $t_i < c$ (notice the strong inequality), the form $t_i \leq c$, as well as $t_i > c$, $t_i \geq c$, $t_i \lambda t_j$ are admitted). For such "closed" ATGs the further deciding of both the strong and weak bisimulation equivalences can be done relying on the same symbolic relations, as defined for PTPs (just taking into account the increasing nature of the clocks in ATGs vs. the decreasing nature of the timers in PTPs).

As to the non-closed ATGs, the deciding algorithms also can easily obtained, following similar lines, as in the PTP case, however, the symbolic transition relations for "bisimilar" region detecting must be chosen in slightly different way, e.g. in the case of the strong bisimulation (Definition 11.16), the timed case of the transition relation should be split into two:

- whenever $P \xrightarrow{WT}$, or $Q \xrightarrow{WT}$, then $next_0(X) \in \mathcal{X}$, and
- whenever $P \xrightarrow{WT^Q}$, or $Q \xrightarrow{WT^P}$, then $next_1(X) \in \mathcal{X}$, where $A \xrightarrow{WT^B}$ means that $A \xrightarrow{c(\mu)}$ for $\mu = \mu(A, B)$

(of course, in the case of ATGs the definitions of $\mu(P, Q)$ and $next_i(P, Q)$ must be taken the "dual" ones w.r.t. the corresponding definitions for PTPs).

So, we have given a brief outline of the proof of the following

Theorem A.4 *The strong and weak bisimulation equivalence problems for Action Timed Graphs are decidable.*

One can obtain similar decidability results also for various generalisations of ATGs (which are still Action Timed Graphs in the sense of [NSY91]), allowing more general timer reset functions associated with the graph edges. Since this generalisation does not require any new ideas, and the "generalised" ATGs have not yet been considered in any applications, it is not considered in detail here. Clearly, one cannot hope to obtain the bisimulation deciding algorithms for the models with undecidable vertex reachability problem (see [ACD90] for some notes).

Appendix B

Example of a Real Time System Analysis

We consider an example of how a real time system with both data and quantitative time constraints can be specified and analyzed. For this purpose we choose to use the language SDL [CC89] for writing the specification of the example (the language SDL is widely practically used in specification of real time systems in telecommunication, and other kinds).

We introduce here the subset of the SDL language, needed for the example specification, and outline some possibilities of the SDL program analysis. In particular, the possibility of automatic complete test set generation for the SDL processes is considered.

As to the example itself, we consider the specification of a passenger lift control program which is, first, described as an SDL process, then, modelled in the programming language LTIBA. An initial projectivee for the set of all feasible finite paths in the obtained LTIBA program is also given and a complete test set construction for the SDL lift process is performed (on the basis of the LTIBA program path feasibility graph).

The presented example is not fully practical for both some simplifications made in the specification in comparison with even simple real lifts and the informal style of the presentation. The example just shows some possible role which can be played by the considered algorithms in the r.t.s. analysis. In fact, for the purpose of practical use these algorithms still have to (and can) be rounded by a number of resource-saving tricks (most of them already existing, just not considered in the main part of the theses). One can see [ABBCK91] for this optimization approach carried through in more systematic way. A more practical approach to the SDL program analysis, which is also going to capture both the program dependencies on data and on timers (and involves the consideration of SDL systems which consist of several structural components (processes)) can be found in [BKM91],[Kal91].

B.1 Example Specification Language: SDL

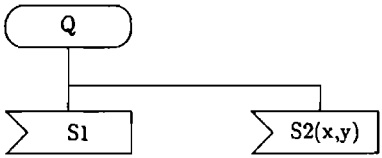
We describe the subset of the language SDL [CC89] used in the example specification below. For the sake of readability most of the permitted language constructions are introduced via examples. We also follow the terminology traditional for SDL in that the "states" for a process are vertexes of special kind in its graph (see below), they are not to be mixed with the states of the labelled transition systems, giving the process semantics.

Only one SDL process is used to describe the real time system. SDL process is a program executing in real time and communicating with the environment by means of signals. SDL process has an input queue into which the environment at certain time moments puts the signals for the process (a time moment can be arbitrary rational, time counting begins at the process start). An input signal for the process may have certain number of integer valued parameters, the signal is recorded in the queue together with all its parameters. The process has also output signals, these signals are sent to the environment at certain time moments according to the process program (process diagram), as a reaction to the input signals.

SDL process is a finite state machine extended by the notions of variable and time and provided with some special statements. More precisely, SDL process is assumed to be able to use finite number of integer-valued internal variables, the process diagram can contain the following constructions (statements).

1. START - the beginning of the process execution. We assume that all process internal variables are initialized to 0 at the execution of START. We depict the statement in the diagram the following way:

2. STATE/INPUT - the complex for awaiting/reading of input signals. It has the following form in the process diagram:

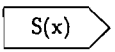


Here Q is a state name, S1 and S2 are the names of signals awaited in this state, x and y are process internal variables to which the values of the parameters conveyed by signal S2 are assigned at consumption (reading) of S2.

If the process has reached the state Q during its execution, it is awaiting for the arrival of some signal in the input queue. At the moment when a signal arrives,

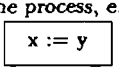
it is consumed (and the corresponding assignments of parameter values to internal variables performed). Further control flow in the diagram depends on the name of the consumed signal (we assume that every signal in the process input queue, which is not awaited in the current state of the process, is not consumed by the process and simply disappears from the queue).

3. OUTPUT - signal sending statement. It has the form

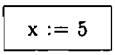


and denotes the sending a signal to the environment at the "current" moment in the process execution (when the instruction in the diagram is reached). Here S is the signal name and x is the process internal variable whose value is assigned to the parameter of the signal.

4. TASK - internal action statement, representing the assignments to the internal variables of the process, e.g.,

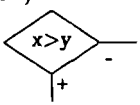


,

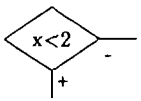


.

5. DECISION - variable comparison statement (in fact, the same, as used in the language LBASE):



,



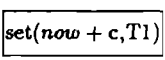
.

We allow also the decision statements with some exits unspecified (i.e. pending) in the process diagram. If the process reaches such a pending exit within some its behaviour, we call this behaviour *incorrect*.

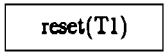
6. SET, RESET statements and timer signals.

Every SDL process has a predefined function *now* at every time moment returning the numeric value of the moment (certain nonnegative rational). A process may have a finite number of timers, each timer being intuitively an "alarm-clock" which can be set up to send a special signal after the expiring of some certain specified time interval (cf. the notion of timer in PTPUs, Section 10.1).

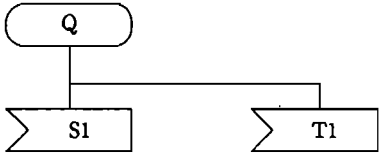
A timer in SDL process can be set up by the instruction (statement)



Here T1 is the timer name and c - an integer constant (a timer is said to be *active* after its setting). The activity of the timer T1 before it "rings" can be disrupted by the following statement:



When the interval of the timer activity expires (i.e. c time units have passed after its setting) and it has not received the *reset* statement, a special signal is put into the process input queue, the signal name being the same as the timer name. This signal can be consumed at process states (a special input branch, corresponding to the timer signal has to be added to the state):



As to the cases of the timer signals not awaited in some states, we assume their disappearance from the input queue, just the same way as ignored are the external input signals to the process.

If some already active timer is set, an automatic reset is executed for the timer before the new setting (so, at every moment of the process execution no more than 1 "copy" of the timers of each name can be active). The timer reset (either explicit, or implicit by the new setting) also removes all the signals with the corresponding name from the process input queue (if there are such).

The execution of the process begins with START statement at the time moment $now = 0$, further processing is performed in accordance with the process diagram. We assume that all internal actions of the process (assignment, comparison, signal sending/consuming, timer setting/resetting) are performed *instantaneously*, so the function *now* may change its value (the time may progress) only if the process waits for some signal (either external one, or a timer signal) in some state.

B.1.1 Histories and Tests for SDL Processes

For the purpose of the SDL program analysis we introduce the notion of the process *history*.

First, a *path* in a SDL process P is every graph-theoretical path in the process diagram.

At every moment of the process execution the process variable value vector, corresponding to this moment is defined in an obvious way. Let $\mathcal{T}(P)$ be the process P timer set, we define the timer value assignment $\delta : \mathcal{T}(P) \to \mathbf{Q}$, corresponding to the process execution moment to assign to an active timer t the value of the absolute time moment when the timer t is going to give its timeout. For an inactive timer t

define $\delta(t) = -1$ (compare with the value definition for PTPUs in Section 10.1, we have just some design changes here).

A history of the process P is any sequence

$$(\langle\langle n_i, \vec{v}_i, \delta_i\rangle, e_i : h_i, \langle n_{i+1}, \vec{v}_{i+1}, \delta_{i+1}\rangle\rangle)_{i<k},$$

where $n_j \in V(P)$ are the process P vertexes (placeholders for the all abovedefined SDL operators, except input), $\vec{v}_j \in \mathbb{Z}^k$ are P variable value vectors (k is the number of P variables) and δ_j are P timer value assignments provided

- $\alpha = n_0 e_0, n_1 e_1, \ldots, n_k$ is an initial path in the process, and
- the process execution along the path α from the process initial variable and timer values, performing the transition along the edge e_i at a the time moment h_i with the target variable value vector $\vec{v}_{i+1}$ and timer value assignment δ_{i+1}, is consistent with the defined process execution semantics;
- no two transitions, corresponding to signal consumption (either external input signals, or the signals from timers) are simultaneous.

For the sake of simplicity we consider only finite initial histories of SDL processes.

If compared with the histories in PTP-like models, considered in Chapter 9 and Chapter 10, the histories of SDL processes differ mostly in that they have the absolute time describing the transition firing moments (i.e. the time relative to the beginning of the process execution) and the timer values are expressed in the terms of the absolute time of the process execution, whilst for PTP-like models every transition in a history contains just the information about how much time has been passed from the *previous* transition firing moment, as well as the timer values are relative to the "current" time moment. The absolute time semantics for SDL processes is naturally inherent in the language constructs (the function *now*), while for PTP-like models the relative time based semantics definition was more suitable for studying the properties like bisimulation equivalence (though there is no principal differences between the two forms for real time system semantics expressing).

Every history ν of a SDL process has associated in a natural way an *input signal sequence*

$$(S_i, h_i, \vec{x}_i), \text{ where}$$

S_i is the ith consumed input signal name (while executing in accordance with ν), h_i is the time moment of the signal input (= the moment of the signal arrival into the queue) and $\vec{x}_i$ is the parameter value vector, conveyed by the signal (note that the timer signals are *not* included in this sequence since they are internal for the process).

For the purpose of the illustrativity let us depict the above considered input signal sequence as

$$(S_1(\vec{x}_1) \text{ at } h_1),\ \ldots\ , (S_k(\vec{x}_k) \text{ at } h_k).$$

One could the same way consider also the output signal sequence for the process, or even a combined one, were there any interest in them.

The input signal sequences for SDL processes are important in that they can be considered as *tests* for given process.

From the practical viewpoint it is fair to consider the SDL processes as a *deterministic* specification formalism (the determinism here means no more than one history for every test, we have already got rid of simultaneously incoming signals which could be eventually the only source for some non-determinism in the process behaviour). This means that we can state the problem of exhaustive testing of the processes w.r.t. some process structural element coverage criteria, e.g., one could consider the process test case sets which force the process to execute all feasible (executable on *some* input) branches both of the STATE/INPUT complexes and the DECISION statements, let us call the testing criterion requiring such coverage C_1 for SDL processes (it is a straight analogue of the widely practically used program testing completeness criterion C_1).

In what follows, we demonstrate (both via some general observations and on the basis of an example) how the program analysis methods presented so far allow, given a SDL process, *automatically* generate for it some set of test cases, which is complete w.r.t. the completeness criterion C_1 (in fact, we are using here just very simple results regarding the finite path feasibility in LTIBA programs (Theorem 4.1 and corresponding corollaries in Section 5.2)). We are going also to consider only *correct* tests for the processes, i.e. the tests not leading the process execution to the pending exits in DECISION statements.

B.2 Passenger Lift Specification

As an example to be considered w.r.t. the program analysis and automatic complete test set generation, we describe as a SDL process a control program for some kind of passenger lift. Let us give first some informal explanation of the process behaviour.

The environment for the lift control process consists of lift users and lift hardware.

A lift user can press a call button in every floor thus sending the signal S with one parameter denoting the number of the call floor to the process. Besides that the user can press any button in the lift-cage to pass the request for the lift to go to some floor, so the signal R with the parameter denoting the destination floor number is sent to the process. In some situations the user can also generate signals FU (FloorUp) and FD (FloorDown) by leaving the lift-cage and entering it, respectively (i.e. by the changing the status of the cage floor).

The lift hardware consists of lift driving motor, controlled by the signals MUp, MDown and MStop, lift door motor (controlled by the signals MD1 (open the door),

MD2 (close the door) and MDStop) and some sensors informing the process about the physical state of the lift. The following signals from sensors are considered: $Z(x)$ - floor number x is reached, DOp (Door is Open), DC (Door is Closed).

The behaviour of the lift can be characterized by the following properties:

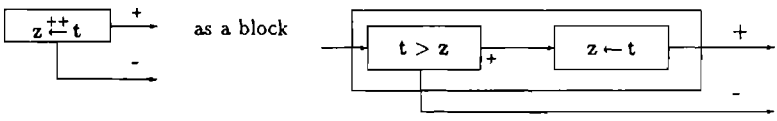
- the lift has no memory for the user requests, signals $S(x)$, $R(x)$ are accepted for the processing only after the previous request has been executed;
- if an empty lift with the open door stays in some floor for more than 20 seconds, the door becomes being closed;
- if the status of the cage floor has changed while the door is closing (i.e. somebody managed to enter or to leave the cage), the closing of the door is interrupted, and the door opens. Besides, if the door was being closed to execute some request to go somewhere, the request is canceled.

The SDL process specification of the lift is given in Figure B.1, Figure B.2 and Figure B.3.

B.3 Modelling of SDL Processes in LTIBA

The modelling of SDL processes as LTIBA programs is done following the same principal lines, as the modelling of PTPs in Section 9.2 and $PTPB_0$s in Section 10.4.

The idea of the SDL process history simulation by a history of the modelling LTIBA program work consists in representing the value of the process function *now* at some point in the process history by the value of the modelling program real time counter z at the "corresponding" point in the program history. Every time when a new signal is read by the process from the process input queue, the time moment of this signal arrival is read in by the modelling program, it is then positively assigned to the program real time counter z, so guaranteeing the non-decreasment of the time between the processing of two input signals. For the sake of brevity in modelling the *strong* advancement of the time we define in the language LTIM (and, so, in LTIBA) a macro-operator



The only difference of the macro-operator $z \overset{++}{\leftarrow} t$ from the positive assignment operator $z \overset{+}{\leftarrow} t$ is that in the case of values of z and t coinciding in the former case the further program execution is passed along the "-" exit, while in the later one

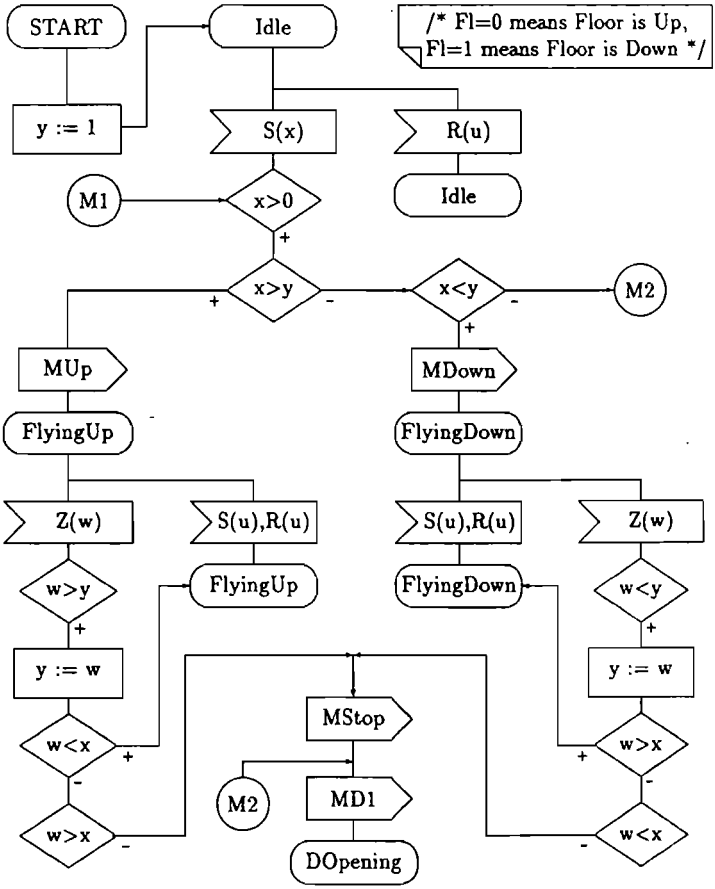


Figure B.1: Lift Process: 1

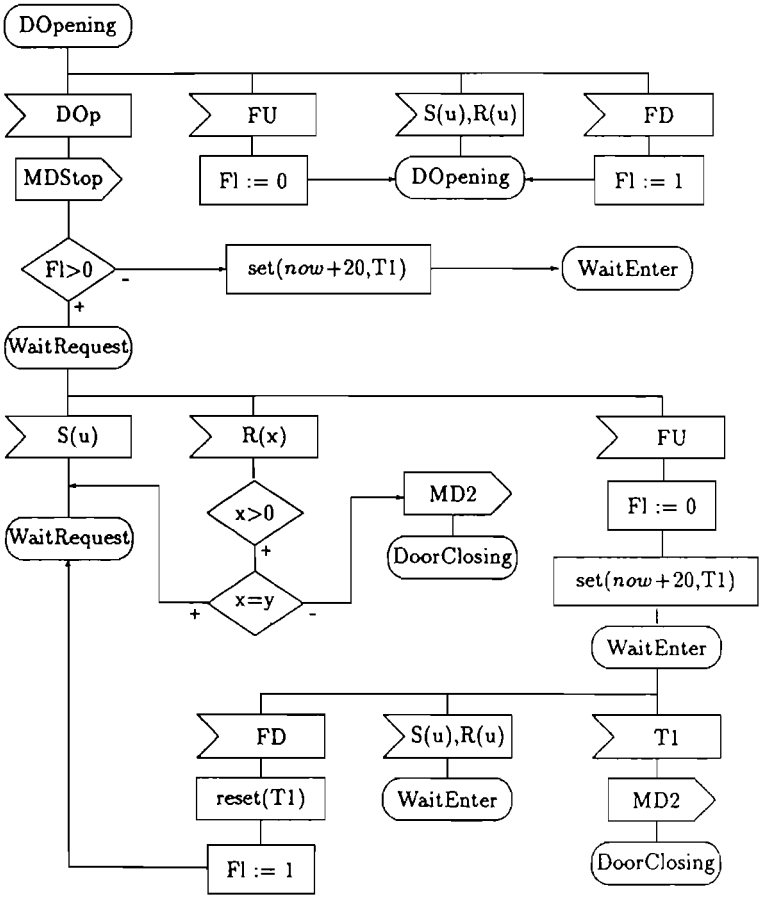


Figure B.2: Lift Process: 2

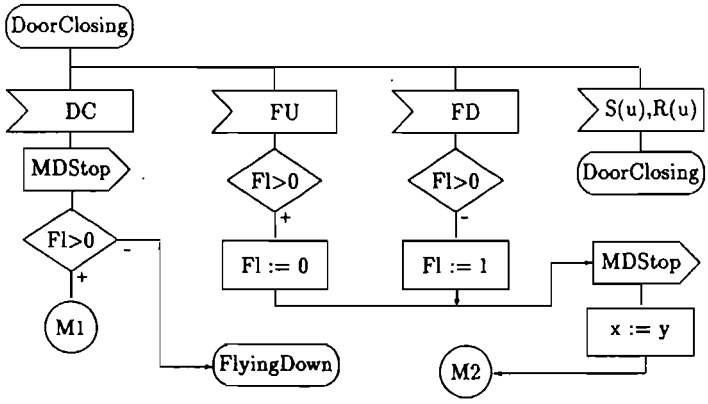
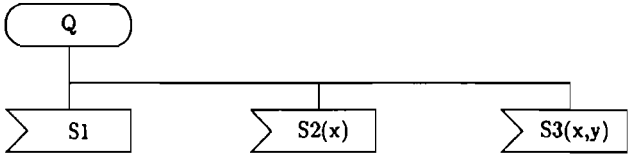


Figure B.3: Lift Process: 3

the corresponding outgoing edge label must be "+". The modelling of SDL process constructs by constructs of the language LTIBA in order to meet the abovementioned correspondence between the process and program histories is done, as follows:

1. The process START statement is transformed into the LTIBA program start label, denoting the program initial vertex.
2. The STATE/INPUT complex



is transformed, as shown in Figure B.4.

As to the LTIM system of commands we again, as in the case of the modelling of PTPs, are satisfied with the $LTIM_0$ subset of them. In the example the variable t is assumed to be the $LTIM_0$ input variable of the modelling program (see Section 3.2 for the notion of the input variable explained). The outgoing edges from the STATE/INPUT complex are corresponding to the outgoing edges from the process state vertex. We put the labels S1, S2, ..., Sk on these program edges to describe which program branch (edge) corresponds to which branch (edge) of the process.

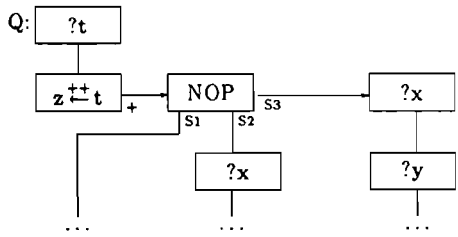


Figure B.4: STATE/INPUT complex modelling

3. The process output signals are not reflected in the modelling program at all for they do not affect the process path feasibility test case generation (tests do consist only of process input signals, the output signal sequence is observable, when the process is executing on some test).

4. The modelling of the process timers is done the following way.

We define a process P timer T *statically active* in a state Q whenever there exists a path α from the process START vertex to Q such that among the timer T treating operators ("set T", "reset T" and consumption of the T signal) the *last* one is "set T" (i.e. the timer T is active "after the path α"). The timer T is *always active* in Q, if T is active after *every* path from the process START vertex to Q and for every such path in all the states *after* the last "set T" operator in the path the signal from T *is awaited* (observe that we do not impose any claim on the *feasibility* of the considered paths, so the statical activity and always activity of given process timer in a given state can be easily checked by some standard graph algorithms).

As to the considered lift process (Figures B.1 , B.2 , B.3), we have in it only one timer, T1. The only process state, in which T1 is active is WaitEnter, one can easily see that T1 is also always active in WaitEnter.

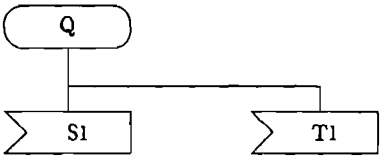
Given a process P we introduce for every its timer Ti a LTIM variable t^i into the modelling program and add for all such variables the assignment operators $t^i \leftarrow (-1)$ in the program graph to be executed at the beginning of the program work (this assignment simply reflects in the modelling program the initial inactivity of all process timers).

The process operators for timer treating are modelled in the program, as follows:

set(*now* + c,Ti) : $t^i \leftarrow z + c$ and

reset(Ti) : $t^i \leftarrow (-1)$

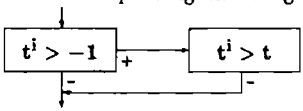
Now let us consider the modelling of the STATE/INPUT complex



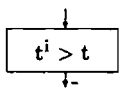
in the presence of statically active timers in the state Q (the states without statically active timers are modelled, as explained in (2.)). Let the set of statically active timers in Q be $\{T1, T2\}$ (we show the modelling translation on the example, covering all the principal possibilities), then the considered STATE/INPUT complex is translated, as demonstrated in Figure B.5 (this modelling situation resembles the PTP edge modelling by LTIM programs, done in Section 9.2: if some transition fires at some moment (here the moment is read from the input into the variable t), this firing will be correct only if no red edge (=no branch, pending on some timer signal) require a transition along it to be performed *earlier*).

Notice the presence of a loop, labelled by the timer T2. The signal from T2 is not awaited in the state Q, however, if the timer *times out* in such a not appropriate moment, its signal is ignored and the timer itself is made "inactive" (actually at this point we are using the modelling of SDL processes by LTIBA programs to explain the semantics of the SDL processes themselves in more detail than it was done in the intuitive description in Section B.1).

If for some state Q some timers Ti active in this state are always active in it, we can the corresponding modelling fragment



replace by a simpler one:



for the value of t^i being greater than -1 *every time* the modelling program reaches this block.

Using the described modelling algorithm we obtain the lift process modelling program in LTIBA, as depicted in Figure B.6 and Figure B.7 (for the sake of brevity we have shortened some names of the states, it is completely straightforward to establish

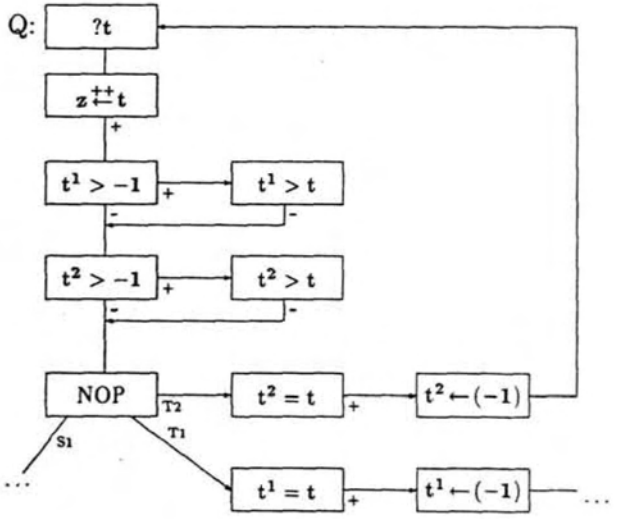


Figure B.5: The Translation of "Timered" States

the correspondence relation between them).

The following brief statements about the abovedefined SDL process as LTIBA program modelling algorithm can be made.

Given a path α in a SDL process P, one can in a natural way associate with it paths α' in its modelling program $M(P)$, which visit the same vertexes in the same order, as the path α goes through in P, and have the edges between these vertexes modelling the corresponding path α edges (see Section 9.2 for an analogic path relation π_P between the paths in PTPs and their modelling LTIM programs), let us call all such paths α' the *codes* of α in $M(P)$.

Let for any process history ν along a path α in P we call modelling program $M(P)$ history ν' along some program path α' coding the process path α the *code* of the history ν provided

- the value of the function *now* at the end of the history ν coincides with $\vec{v}_*^T(\nu').z$ - the value of z at the end of the history ν',
- for every P timer Ti at the end of the histories ν and ν' respectively we have $\delta(\mathrm{Ti}) = \vec{v}^T.t^i$,
- the P variable values at the end of ν coincide with the corresponding modelling program LBASE variable values at the end of ν', and
- for every ν prefix ν_i there exists a ν' prefix ν'_i which is the code of ν_i.

A path α in the process modelling program is *accepting*, if it ends with some vertex, corresponding to a SDL state of the process (equivalently, if the path α is the code of some process path). One can easily obtain the following result characterizing the defined coding relation:

Lemma B.1 *For every SDL process P history α there exists precisely one modelling program history, which is the code of α. Besides that, every modelling program history along an accepting path in the program is a code of some process history, and the coding relation between histories are both ways effective.*

B.4 Analysis of SDL Processes

We begin this section with some theoretical remarks regarding the possibilities of the automated analysis and complete test case generation for SDL processes, then we proceed to the lift process example on the basis of which the ways of LTIBA program path feasibility graph usage in the complete set test generation for SDL processes is explained. Some points regarding the optimizations of the path feasibility graph construction for LTIBA programs (if compared with the algorithm, given in the proof of Theorem 4.1) are also considered.

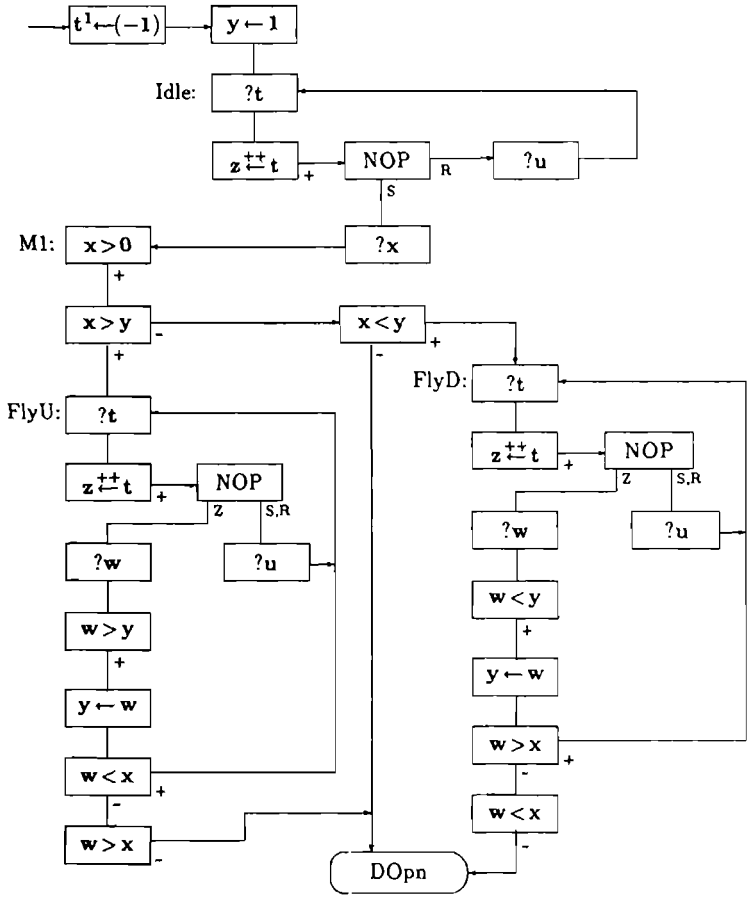


Figure B.6: Lift Process: LTIBA: 1

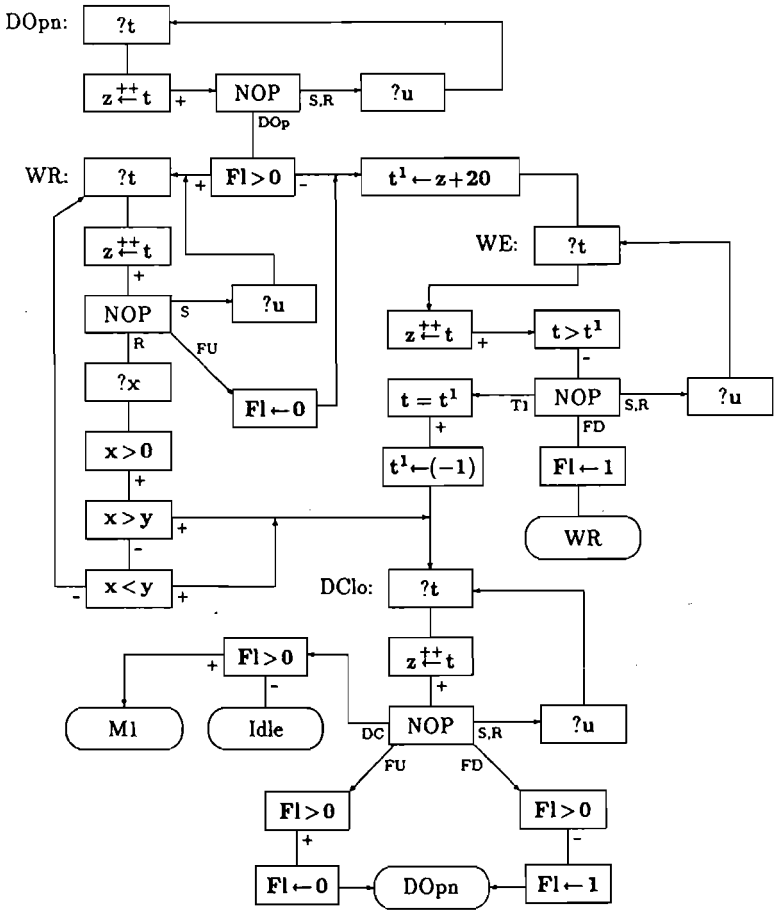


Figure B.7: Lift Process: LTIBA: 2

From Lemma B.1 it follows that a path in a SDL process is feasible if and only if it has a feasible code in the modelling program. Since according to Corollary 5.13 the set of all feasible accepting paths in the modelling LTIBA program is T-projective (see Section 2.2 for the definitions), we obtain a similar T-projectivity result for the set of all finite feasible paths in given SDL process, what allows to solve e.g. the process vertex reachability problem.

Moreover, since we have the result of Corollary 5.16 for LTIBA programs (actually we use a slight its modification), we can for every feasible branch in given program (i.e. the branch, executable within some initial program history) *effectively* exhibit the history on which this branch is covered. So we can, given a LTIBA program, effectively exhibit a finite set of its histories which traverse the all feasible given program branches. If one exhibits such a history set for a LTIBA program which is modelling an SDL process P, Lemma B.1 guarantees that these histories are the codes of the process P histories which also cover all feasible branches in P. Since the step from an SDL process history to its associated input signal sequence is straightforward and all the needed constructions are clearly effective, we have

Theorem B.2 *There exists an algorithm which, given a SDL process, generates for it a complete test set w.r.t. the testing completeness criterion* C_1.

In fact, the result of the theorem can be generalized also to apply to SDL systems which contain several parallel processes, one just needs to constrain the signal queue lengths in the system (for SDL systems consisting of two processes provided with infinite queues and two signals going each way between them, the process vertex reachability problem can be easily shown undecidable).

Now let us consider the CTS generation for the lift process, specified in Figures B.1, B.2, B.3 and presented as a program in LTIBA in Figures B.6 and B.7. We build first a finite path feasibility graph (initial T-projectivee for the set of all finite feasible paths) of the lift program in LTIBA.

B.4.1 Path Feasibility Graphs: Optimizations

According to the definition of the path feasibility graph $G(P)$ for a given program P it may be the case that one path in the program is the projection of more than one path in its path feasibility graph. Also both the LBASE and LTIM program path feasibility graphs $BG(P)$ (and, so, the path feasibility graph for the LTIBA programs), built in the proof of Theorem 4.1 usually have this facility.

One can notice that for P being either a LBASE, or LTIM program, every vertex in the path feasibility graph $BG(P)$ is a pair (n, C) for n being a program P vertex and $C \in C_P$ being a program variable configuration. Let for an arbitrary path α in the program P the context $S(\alpha)$ be defined as the set of all configurations, which can

be reached in $BG(P)$ along the path α, more precisely, let

$$S_P(\alpha) = \{C \in C_P | \exists \alpha' : \alpha = proj(\alpha') \ \& \ C_*(\alpha') = C\}.$$

Clearly, one can, given a LBASE or LTIM program P, compute for it a finite path feasibility graph with the vertexes $\langle n, S\rangle$ for $S = S(\alpha)$ for some P path α with $n_*(\alpha) = n$, one just has to draw the edges between these vertexes in a way that an edge, labelled by a program P edge from $\langle n, S\rangle$ to $\langle n', S'\rangle$ is drawn if and only if

for every $C' \in S'$ there exists $C \in S$ such that
there exists an e-labelled edge in $BG(P)$ from $\langle n, C\rangle$ to $\langle n', C'\rangle$.

This approach of the program path feasibility graph construction for LBASE programs was developed in [BBK74, BBK77] (see also [ABBCK91] for a survey) and it leads in practical examples to the path feasibility graphs of considerably smaller size than $BG(P)$s considered above. The path context approach becomes especially attractive since one can manage to provide the configuration set $S(\alpha)$ with a comparably short and effective characteristics by the means of *inequality systems* (just as the variable configurations for both LBASE and LTIM programs were characterized above in Section 4.1) w.r.t. the "current" variable values after the execution of the path α. The state-space saving effect arises since the variable configurations for LBASE programs contain the information about the mutual relations of *all* pairs of variables, while the path contexts $S(\alpha)$ do not require the determination of the unimportant (in some sense) ordering of the variable pair values. The reason of the choosing the variable configuration approach in this work was the intention to use the graph $BG(P)$ also for *infinite* path feasibility analysis, where, at least for our approach the full information about the variable value comparisons after every operator execution was necessary (see e.g. the definition of an accomplished loop in Section 6.1).

As to the case of LTIM programs, the path contexts are defined the same way, as for LBASE programs, however, it may be not the case that a particular context $S(\alpha)$ can be characterized by a simple inequality system w.r.t. the "current" variable values after the execution of the path α, in general, some more complex data structures are needed. One possible solution to this problem is given in [ABBCK91], where every of the path contexts $S(\alpha)$ is again partitioned into several subsets (each of them being a configuration set, reachable along the path α). This way both the obtained graph can be shown to be a path feasibility graph for the given program, and every its vertex becomes a pair $\langle n, S\rangle$ for n being a program vertex, and S - some set of program variable configurations with their union being a solution set of some simple linear inequality system.

However, given a LTIM_0 program P, for every path α in it the defined context $S(\alpha)$ obeys the property of being a solution set of some simple inequality system, so the further its splitting is not necessary. Just for the sake of brevity in the path

feasibility graph for the lift program two such splittings were made (see Figure B.9), we save this way just about 9 reachability graph vertexes.

It could be at least theoretically an interesting problem to find the algorithms looking for the "optimal" finite path feasibility graphs in given programs (these algorithms also themselves should run in a reasonable time), however, it is not touched in the thesis.

There are two more simplifications which are taken into account when building the path feasibility graph for the LTIBA lift program.

First, we put in the graph only the vertexes, corresponding to the program vertexes which are *essentially located* [ABBCK91], namely, a vertex class containing at least one vertex for every loop in the program (we require also all accepting lift program vertexes to be present).

Second, some of the path contexts (inequality systems) are simplified by the excluding variables whose values may become to be used in the future only *after* some prior assignment (i.e. the "current" values of these variables are "irrelevant" for the further behaviour of the program), see [ABBCK91] for a more detailed explanation (just note that only program structure analysis techniques are used in this step to make the decision about the "inessentiality" of some variables).

B.4.2 A Path Feasibility Graph for LTIBA Lift

The vertexes of the path feasibility graph are pairs $\langle n, S\rangle$ where n is an essentially located program vertex and S is an inequality system characterizing the program variable values (both LBASE and LTIM variables (base points) are taken into account) after the program paths leading to this vertex (more precisely, the program paths which are the projections of the graph paths leading to $\langle n, S\rangle$).

The label of a given edge in the graph describes the path between the corresponding essentially located vertexes in the program, which corresponds to this edge. For the full description of such a path it suffices to denote the exits of the all program operators (vertexes), where a real decision is being made (i.e. which have *more than one* exit), see the graph.

In order to make the graph more readable we choose to denote all cyclic edges around the vertexes by these edge labels put in the parentheses in the vertex description itself, e.g.,

WR,S6(S,R−−)

means that there are loops around the vertex WR, S6 which are labelled by S and R − −. If the asterisks (**) are put in the parentheses for some graph vertex, look for another vertex with the same name, where all outgoing edges from it are specified.

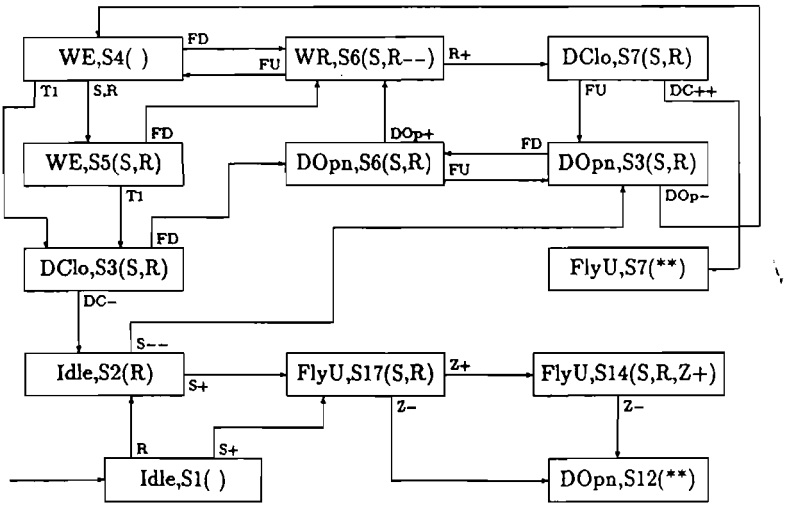


Figure B.8: Path Feasibility Graph: 1

The path feasibility graph for LTIBA lift program is presented in Figure B.8 and Figure B.9 and the explanation of the "context" components of the graph vertexes is given in Figure B.10.

B.4.3 A Complete Test Set for Lift Process

Given the path feasibility graph for the LTIBA lift program, we can look for every given branch b of the program, whether it is a projection of some branch in the graph. If such a graph branch is found, we can consider an initial path α in the graph, containing this branch and find some program history ν along this path (in the case, if there is no such branch in the graph, the corresponding program branch b is *infeasible*). The input signal sequence of the process history, coded by ν, yields a test for the SDL lift process, covering the process branch, which corresponds to b. Choosing some order of the program branch consideration, one can obtain, for instance, the

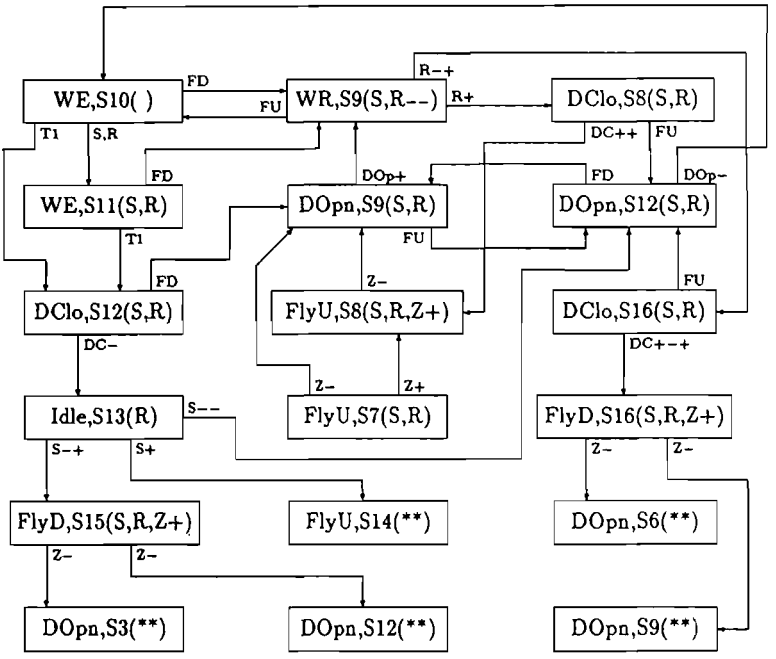


Figure B.9: Path Feasibility Graph: 2

S1:	$-1 = t^1 < 0 = z,$	$y = 1,$	$Fl = 0$
S2:	$-1 = t^1 < 0 < z,$	$y = 1,$	$Fl = 0$
S3:	$-1 = t^1 < 0 < z,$	$x = y = 1,$	$Fl = 0$
S4:	$0 < z < z + 20 = t^1,$	$x = y = 1,$	$Fl = 0$
S5:	$0 < z < t^1 < z + 20,$	$x = y = 1,$	$Fl = 0$
S6:	$-1 = t^1 < 0 < z,$	$x = y = 1,$	$Fl = 1$
S7:	$-1 = t^1 < 0 < z,$	$x > y = 1,$	$Fl = 1$
S8:	$-1 = t^1 < 0 < z,$	$x > y > 1,$	$Fl = 1$
S9:	$-1 = t^1 < 0 < z,$	$x = y > 1,$	$Fl = 1$
S10:	$0 < z < z + 20 = t^1,$	$x = y > 1,$	$Fl = 0$
S11:	$0 < z < t^1 < z + 20,$	$x = y > 1,$	$Fl = 0$
S12:	$-1 = t^1 < 0 < z,$	$x = y > 1,$	$Fl = 0$
S13:	$-1 = t^1 < 0 < z,$	$y > 1,$	$Fl = 0$
S14:	$-1 = t^1 < 0 < z,$	$x > y > 1,$	$Fl = 0$
S15:	$-1 = t^1 < 0 < z,$	$0 < x < y,$	$Fl = 0$
S16:	$-1 = t^1 < 0 < z,$	$0 < x < y,$	$Fl = 1$
S17:	$-1 = t^1 < 0 < z,$	$x > y = 1,$	$Fl = 0.$

Figure B.10: Path Feasibility Graph: Contexts

Test N1. (R(0) at 1).

Test N2. (S(3) at 1), (S(0) at 2), (R(0) at 3), (Z(2) at 4).

Test N3. (S(2) at 1), (Z(2) at 2), (S(0) at 3), (R(0) at 4), (FD at 5), (FU at 6).

Test N4. (S(1) at 1), (DOp at 2), (S(0) at 3), (R(0) at 4) (observe that this test invokes the timer signal T1 to come at the moment 22).

Test N5. (S(1) at 1), (DOp at 2), (FD at 3), (S(0) at 4), (R(1) at 5).

Test N6. (S(1) at 1), (DOp at 2), (FD at 3), (R(2) at 4), (S(0) at 5), (R(0) at 6).

Test N7. (S(1) at 1), (DOp at 2), (FD at 3), (R(2) at 4), (DC at 5), (Z(2) at 6), (DOp at 7).

Test N8. (S(3) at 1), (Z(3) at 2), (DOp at 3), (DC at 24), (S(1) at 25), (S(0) at 26), (R(0) at 27), (Z(2) at 28), (Z(1) at 29).

Test N9. (S(1) at 1), (DOp at 2), (FD at 23).

Test N10. (S(1) at 1), (DOp at 2), (FD at 3), (R(2) at 4), (FU at 5).

Test N11. (S(2) at 1), (Z(2) at 2), (DOp at 3), (FD at 4), (R(1) at 5).

Figure B.11: A Complete Test Set for Lift Process

collection of the tests for the lift process, which form the needed Complete Test Set (according to C_1), as shown in Figure B.11