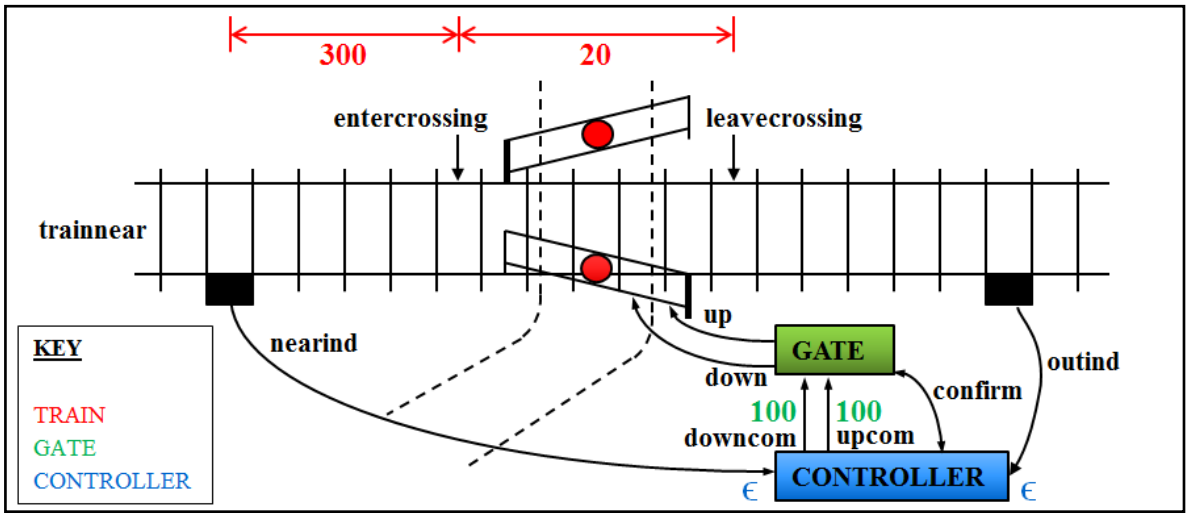
A Timed CSP Simulator for Railway Systems

Marc Dragon

May 2011



Abstract. Using the formal specification of timed CSP outlined in Steve Schneider's 'Concurrent and Real-time Systems: The CSP Approach', we have developed a timed CSP simulator. The simulator is an extension on ProB, an existing untimed CSP-B model checker and animator. We have also demonstrated the applicability of our timed CSP simulator to the railway domain, using CSP models related to railway safety and capability.

Project Dissertation submitted to the Swansea University
in Partial Fulfilment for the Degree of Bachelor of Science

Prifysgol Abertawe
Swansea University

Department of Computer Science
Swansea University

Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

Date: 18/05/2011

Signed:

Statement 1

This dissertation is being submitted in partial fulfilment of the requirements for the degree of a BSc in Pure Mathematics and Computer Science.

Date: 18/05/2011

Signed:

Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

Date: 18/05/2011

Signed:

Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Date: 18/05/2011

Signed:

Acknowledgements

I would like to thank Dr. Markus Roggenbach for providing me with an exciting and challenging project that gave me an opportunity to improve my knowledge in so many different areas of computer science. I also want to thank him for his support, guidance and friendship throughout the duration of the project.

I also want to extend my gratitude to everyone in the Swansea University Processes and Data Research Group. Thank you for the interesting weekly meetings and social events; as well as all the advice throughout the year.

I am grateful to my parents for their continued support and for helping me during the tough times in the year.

I would also like to thank Andy Gimblett for all his help with the Haskell parser and Parsec; and for the engaging, enjoyable meetings.

I want to thank Marc Fontaine and Professor Michael Leuschel of Heinrich-Heine University for their regular correspondence regarding ProB.

Finally I want to thank Erwin R. Catesbeiana (Jr.) for watching over me throughout the year and for inspiring me to go the extra mile.

Contents

PART I
Introduction

I Introduction

The British railway system has seen somewhat of a resurgence in recent years, particularly in areas which have a well-developed public transport system such as London. The railway industry has always been concerned with challenges of a safety-critical nature; ensuring that all operations on the line and the train meet rigorous standards. However, with the growing demand on the industry, there has developed a need to address the challenges of optimisation and efficiency.

These requirements can be described concisely as the challenge of capacity – the science of determining the ideal track plan with which the greatest throughput of trains can be achieved.

1.1 Motivation

To this end, there is a need for a solution that is easy to understand and operate. A timed simulator works towards such a goal; it allows for the optimum throughput of a track plan to be determined quickly and easily. A timed simulator demonstrates the possible executions and allows the user to test various durations within the system to see what works best. Additionally, timed simulation can help solve the safety-critical challenges at the heart of the railway domain.

Furthermore, the need for time in a system is, by its own merits, an important one. Time plays an important role in both non-critical and critical systems. In a non-critical system, for example a system which records mouse clicks, time is necessary to differentiate between two single clicks and a double click. Without time, such a differentiation would not be possible and the system would not function as desired.

More importantly, in a critical system, such as an automatic braking system, time is a vital factor. Without time, the system would not function correctly; with possibly disastrous consequences. The need for time as a factor in many systems is a clear one.

Finally, there is a need for simulation itself. A system starts as an idea in someone's mind, an informal model which can be described through words and diagrams. Before a system can be implemented, this informal model needs to be refined into a formal model. The formal model is ready for implementation and can be model-checked. Simulation allows us to verify that the formal model is correct with respect to the informal model; that it accurately represents the intended function of the informal model.

1.2 Project Aims

In order to address these needs, our project has two main aims:

- To develop a fully-functioning and professional timed CSP simulator.
- To apply the completed implementation to models within the railway domain and take the first steps towards applying it to the safety and capacity challenges.

With regard to the first aim, we chose the timed CSP specification laid out in Steve Schneider's 'Concurrent and Real-time Systems'[1]. The timed CSP process algebra allows us to consider a railway system in the form of a network of communicating and interacting processes, in a real-time setting. Our simulator takes in static CSP specification as input and provides a simulation environment which allows the end-user to dynamically iterate through

the processes. ProB[2] is an existing untimed CSP simulator and to this end we made the decision to extend this software to account for the timed CSP syntax and semantics.

Considering out second aim, we have worked with a simple railway crossing model from Steve Schneider's book in order to demonstrate both our timed CSP simulator's applicability to the railway domain as well as clarifying that our implementation correctly mirrors Steve Schneider's specification. We have also developed a small, primitive model of a London Underground station to take the first steps towards solving the challenge of capability. Capability is the notion of determining the optimum throughput for a specific track plan and can be considered a subset of the capacity challenge. This confirms the viability of our timed CSP simulator for working in the optimisation side of the railway domain.

1.3 Similar Projects

Prior to the development of this project, we considered two other existing projects with similar aims. The first was Uppaal[3], a joint-venture between Uppsala University of Sweden and Aalborg University of Denmark. The project was initially released in 1999 and has undergone continual development ever since. Its main purpose is the modelling, simulation and verification of real-time systems. In Uppaal, systems are represented as networks of timed automata, extended with structured data types. Uppaal is typically applied to areas such as real-time controllers and communication protocols; particularly where time is a critical aspect[4]. Indeed, a railway system is an ideal example[5]. Uppaal is coded in C++ with a Java-based GUI[6]. We decided instead to work with ProB as we wanted to work with timed CSP as a new alternative to timed automata for timed simulation. Furthermore, we decided to work with ProB because of the solid synergy between Prolog and timed CSP due to the programming language's logical nature.

The second alternative was Ben Coombs' third year project. In the academic year 2009/10, Ben Coombs of Swansea University developed a timed CSP simulator in Haskell for his third year dissertation. Its purpose was similar to Uppaal, but was concerned only with the simulation of real-time systems. We considered the possibility of developing this simulator further, which would have involved adding the missing bounded operations and including a parser for proper support of syntactic analysis. Ultimately, we decided instead to base our project on ProB as it already included a parser.

1.4 Outline

This dissertation gives a comprehensive guide to the background research, gives a full analysis of the implementation and explains in detail the models used to demonstrate our project's applicability to the railway domain.

In chapter 2, we cover the key syntax and semantics of untimed CSP as described by Steve Schneider. We discuss the basics of CSP itself; its events and processes. We then look at the various types of operators available in CSP. We start with the most basic operators STOP, SKIP and the prefix operators. We then move on to the other operators necessary for modelling a system; covering compound events, recursion, choice, concurrency, abstraction and control flow. We end the chapter with an EBNF grammar summarising the CSP syntax.

In chapter 3, we expand on our specification of untimed CSP with the timed operators. First we describe the behaviour of untimed CSP operators in a timed context. We then detail the new timed operators; timeout, timed interrupt and timed event prefix. Additionally we look at the syntactic sugar; namely the WAIT and delay operators. We then consider the

semantic laws that apply to our operators. Finally we provide the updated EBNF grammar for timed CSP syntax.

In chapter 4, we take a brief look at the declarative programming language Prolog, which forms the basis for ProB's implementation. We look at the standard layout of Prolog code and consider the how information is segregated within the language; through terms, clauses, programs and queries.

In chapter 5, we discuss the functional programming language Haskell, which is the foundation for ProB's parser. Again we consider the layout and organisation of information within the language. We also briefly look at the Parsec library, which helped ease the process of extending the parser.

In chapter 6, we briefly look at the third and final language used to create the complete ProB package; Tcl/Tk, which is used for the graphical user interface.

In chapter 7, ProB itself is analysed in detail. We first consider the function of a CSP simulator and how operational semantics are carried over to an implementation. This is followed by ProB's software architecture, key features and data flow.

In chapter 8, our first steps towards working with ProB are laid out. In particular, we reference our untimed CSP model based on a simple railway track plan in Kirsten Winter's 'Model Checking Railway Interlocking Systems'.

In chapter 9, we follow on from the previous chapter with an outline of our design decisions throughout the project; from the implementation of syntactic sugar to the organisation of syntax in the abstract syntax tree.

In chapter 10, we give a comprehensive analysis of our Prolog implementation; including our consideration of rational numbers and timed firing rules.

In chapter 11, we detail the rest of our implementation; including the timed CSP syntax in Haskell and the additions to the Tcl/Tk code to allow for extra user input and additional aesthetics.

In chapter 12, we explain the basic models which demonstrate the various timed operators in our implementation. As well as providing a stronger understanding of how each operator behaves, it also allows us to clarify the correctness of our implementation.

In chapter 13, we use the railway crossing model from Steve Schneider's book to demonstrate both the accuracy of our implementation with respect to the book and our simulator's applicability to the railway domain.

In chapter 14, we describe our final model; a simple model of the London Underground. In this chapter, we aim to take the first steps towards a solution to the challenge of capability as a subset of the capacity challenge. To this end, we aim to demonstrate how our timed simulator could be used in the optimisation side of the railway domain.

In chapter 15, we reflect on our original project aims and determine the extent to which our final implementation has met these aims. We also consider the impact our project has had and the scope for future developments.

PART II
Background Research

II Background Research

Part II details the necessary background information required to understand the functionality and context of our timed CSP simulator. First and foremost, it covers the important areas of timed CSP process algebra syntax and semantics. Following on from timed CSP, we give a brief overview of the three main programming languages used in our project; Prolog, Haskell and Tcl/Tk. Finally, we look at the ProB software which has formed the foundation for our implementation; detailing its architecture, key properties and dataflow.

2 Untimed CSP

This section covers the complete specification of untimed CSP. We cover the basics of transitions and static specification. We also look at the key groups of operators; compound events, recursion, choice, concurrency, abstraction and control flow. Finally we summarise the syntax in an EBNF grammar. This section and the timed CSP section are based on Concurrent and Real-time Systems: The CSP Approach. The firing rules in particular are directly taken from the book.

2.1 Basics of Processes and Events

In essence, CSP (Communicating Sequential Processes) is a process algebra which considers a system as a network of processes and the atomic events between them. It is mainly concerned with the external interaction and communication of processes rather than internal events within a process. These processes are described by their interface, a set of all the possible events they can engage in, which is also considered to be the static specification of CSP. Additionally, CSP is compositional; allowing processes to be encapsulated within larger processes.

To understand how a system will work as we iterate through it, we need to determine a dynamic specification. This is achieved in CSP through labelled transitions. These describe an execution of an event and the resultant process for a given state. Given an initial process P1, an external action denoted by μ and an incidental process P2; a basic labelled transition would be as follows:

$$P1 \xrightarrow{\mu} P2$$

All standard external actions can be categorised by the set Σ. We also have the termination action $\checkmark$. While CSP is generally concerned with external actions only, internal actions do play a role and are denoted by the set τ. We can say that all actions are in the set $\sum^{\checkmark,\tau}$ and all external actions (including termination) are in the set $\sum^{\checkmark}$.

2.1.1 STOP and SKIP

The simplest process in CSP is STOP. The interface for this process is the empty set as it has no possible actions at any point during execution. In this sense, it results in a deadlocked system which can no longer run.

We also have the process SKIP which invokes the termination action and leads to a STOP process. This also ends the program but is considered to be an intended (and thus successful) termination of a system.

Its only transition is as follows:

$$\text{SKIP} \xrightarrow{\checkmark} \text{STOP}$$

2.1.2 Event Prefix and Prefix Choice

In order to compose more complex systems, it is necessary to add an operator known as the event prefix. It takes the form: $(a \longrightarrow P)$. This means that the process can initially perform the action a, with an incidental process P. This is confirmed by the simple transition below:

$$\frac{}{(a \longrightarrow P) \xrightarrow{a} P}$$

The previous transition is in the instance where the initial process has only one action available to perform. If a process has multiple actions available, we instead use a convention known as prefix choice denoted by $(x : A \longrightarrow P(x))$ for $A \subseteq \Sigma$. We can say that, given $a \in A$:

$$\frac{}{(x : A \longrightarrow P(x)) \xrightarrow{a} P(a)}$$

In other words, the initial process is willing to perform any process $a \in A$ and the resultant process will be P(a).

2.2 Compound Events

While CSP considers actions to be indivisible, we can create compound events which store information on the nature of an action. One way to consider a compound event is in the form (c.v), where c is a channel and v is a specific value being carried across the channel. An example would be input and output channels for binary. We would have a channel 'in', which carries 0 values and 1 values: $(in.0 \longrightarrow P)$ and $(in.1 \longrightarrow P)$. We would also have a channel 'out', carrying 0 values and 1 values: $(out.0 \longrightarrow P)$ and $(out.1 \longrightarrow P)$.

We can also consider a compound event where there is choice. Given a channel c of type T, the action $\{c.t \mid t \in T\}$ accepts any value across channel c that is classed as type T.

Processes defined with respect to these compound events are slightly different to event prefixes. A specific value $v \in T$ across channel c of type T can be described by the following transition:

$$\frac{}{(c!v \longrightarrow P) \xrightarrow{c.v} P}$$

For any value $v \in T$ across channel c of type T, we have the following:

$$\frac{}{(c?x : T \longrightarrow P(x)) \xrightarrow{c.v} P(v)}$$

The previous transition states that the initial process can take any value x of type T down channel c.

If we then select a compound action c.v where v ∈ T, we are given an incidental process P(v).

2.3 Recursion

A process definition can be represented in the form N = P, where N is the name of the process and P is the body of the definition. If we assume the body P to contain the process name N, then we have a recursive definition. The firing rule proving this states that:

$$\frac{P \xrightarrow{\mu} P'}{N \xrightarrow{\mu} P'}$$

With the side condition that [N = P]. In other words, if P changes, N also changes.

2.4 Choice

We have already analysed the conventions for action choice. CSP also has definitions for process choice. There are two different types of choice, external and internal.

2.4.1 External Choice

External choice involves a selection made by other interacting processes or the user. We can describe this as P □ Q in its most basic form, where P and Q are two initially-available processes. At this point in execution, the actions of both P and Q are available.

When a selection is made by executing an external action a ∈ $\Sigma^{\checkmark}$, only the actions of one process will remain open (specifically, the process P which contains action a). If an internal action is made by a process, the external choice still remains open. This can be described more concisely in the following firing rules:

$$\frac{P \xrightarrow{a} P'}{\begin{array}{l} P \,\Box\, Q \xrightarrow{a} P' \\ Q \,\Box\, P \xrightarrow{a} P' \end{array}} \qquad \frac{P \xrightarrow{\tau} P'}{\begin{array}{l} P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q \\ Q \,\Box\, P \xrightarrow{\tau} Q \,\Box\, P' \end{array}}$$

In the case where the action a appears in both P and Q, we are presented with non-determinism. The external choice still remains intact, but external processes can no longer discern the two options.

2.4.2 Internal Choice

Internal choice is a decision made within a process, described as P ⊓ Q. This choice cannot be influenced by external processes or the user. Instead, the selection is made by the process itself. An example would be a process ROUTER deciding the best channel to send a network signal over based on noise across each channel. The channel is selected automatically by the

process ROUTER without external intervention. The user is not interested in how the router transfers data; only that it does so successfully:

$$\text{ROUTER} = \text{CHANNEL0} \sqcap \text{CHANNEL1} \sqcap \text{CHANNEL2}$$

The firing rule confirms the properties of internal choice as follows:

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P \qquad P \sqcap Q \xrightarrow{\tau} Q}$$

2.5 Concurrency

So far we have dealt with sequential processes; iterating through them, one at a time. To simulate more than the most basic of systems, we need the ability to run multiple processes concurrently. There are three main concurrent operators.

2.5.1 Alphabetized Parallel

The first convention we have to represent concurrency is alphabetized parallel. We can say that for processes P and Q running in parallel with respective interfaces A and B:

$$P \;{}_A||_B\; Q$$

This means that P and Q synchronise for actions $a \in A \cap B$ and run independently for all other actions in A and B. This synchronisation (or handshake synchronisation) requires both processes to initiate the action. This means that if process P is not yet ready to initiate an action $a \in A \cap B$, then action Q will not be able to carry it out, and vice-versa.

With the concept of concurrency, one has to be wary of the possibility for deadlock. This is essentially tantamount to the STOP process, an unwanted termination. A series of processes, each of which require the other to synchronise on an event, may result in deadlock. The most common example of this is known as the Dining Philosopher's problem. There are forks in between each philosopher, but each requires two forks to eat. If every philosopher picks up a fork on their left, they will all be waiting for the right fork indefinitely. No philosopher will retract the action and put the fork back down, nor can they continue until they synchronise with a fork to their right. They are deadlocked.

We can describe the behaviour of alphabetized parallel through the firing rules below.

$$\frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{a} Q'}{P \;{}_A||_B\; Q \xrightarrow{a} P' \;{}_A||_B\; Q'} \qquad [\, a \in A^{\checkmark} \cap B^{\checkmark} \,]$$

$$\frac{P \xrightarrow{\mu} P'}{P \;{}_A||_B\; Q \xrightarrow{\mu} P' \;{}_A||_B\; Q \qquad Q \;{}_A||_B\; P \xrightarrow{\mu} Q \;{}_A||_B\; P'} \qquad [\, \mu \in (A \cup \{\tau\} \setminus B) \,]$$

2.5.2 Interleaving

CSP also has an operator known as interleaving. The implication here is that the processes run independently for all actions, including those which appear in both interfaces. Interleaving is simply described by the notation:

$$P \;|||\; Q$$

The one exception is termination. If a termination event occurs, both processes end simultaneously. These properties can be described concisely by the following firing rules:

$$\frac{P \xrightarrow{\mu} P'}{\begin{array}{l} P \;|||\; Q \xrightarrow{\mu} P' \;|||\; Q \\ Q \;|||\; P \xrightarrow{\mu} Q \;|||\; P' \end{array}} \qquad \frac{P \xrightarrow{\checkmark} P' \qquad Q \xrightarrow{\checkmark} Q'}{P \;|||\; Q \xrightarrow{\checkmark} P' \;|||\; Q'}$$

2.5.3 Interface Parallel

Alphabetized parallel allows processes to synchronise on shared actions $a \in A \cap B$. Interleaving allows multiple processes to concurrently run independent of each other. We can combine these two operators to give interface parallel. Interface parallel not only allows concurrent operation of multiple processes but also synchronises the processes for a specified shared set of actions A. We can describe interface parallel using the following notation:

$$P \underset{A}{\|} Q$$

The following firing rules clarify these notions:

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \underset{A}{\|} Q \xrightarrow{a} P' \underset{A}{\|} Q'} \quad \text{where } a \in A^{\checkmark} \qquad \frac{P \xrightarrow{\mu} P'}{\begin{array}{l} P \underset{A}{\|} Q \xrightarrow{\mu} P' \underset{A}{\|} Q \\ Q \underset{A}{\|} P \xrightarrow{\mu} Q \underset{A}{\|} P' \end{array}} \quad \text{where } \mu \notin A^{\checkmark}$$

2.6 Abstraction

There are three main abstraction operators in CSP; event hiding, forward renaming and backward renaming. However in order to achieve the goals of this project, we are only concerned with event hiding.

2.6.1 Event Hiding

When combining processes to form a larger process, previously external actions should now be considered internal to the new process. In order to reflect this abstraction, CSP provides event hiding. Event hiding is represented by $P \setminus A$ and hides all actions in the set A that have formed part of the interface for new process P. Now, if a previously external action from the set A is performed, it is considered an internal action of $P \setminus A$.

With the inclusion of abstraction, it is important to be aware of the possibility for divergence. Assume there is a system of processes with external actions that refer to each other through a recursive loop. We may decide to event hide these actions. Once encapsulated internally, these actions could form a never-ending loop, inaccessible by external processes or the user. This is worse than the deadlock scenario as not only would the system be stuck, but the internal loop would also be consuming resources.

The firing rules are as follows:

$$\frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \quad [a \in A]$$

$$\frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A} \quad [\mu \notin A]$$

2.7 Control Flow

The final type of untimed CSP operator deals with the transfer of control to other processes. There are two operators, sequential composition and untimed interrupt.

2.7.1 Sequential Composition

Sequential composition is represented by P ; Q. It simply states that on termination of P, Q will be initiated (and the termination is considered an internal action). Only the termination action will have an effect on sequential composition. Any other action will simply change P and leave the composition intact.

This is demonstrated by the following simple firing rules:

$$\frac{P \xrightarrow{\mu} P'}{P \,;\, Q \xrightarrow{\mu} P' \,;\, Q} \; [\mu \neq \checkmark] \qquad \frac{P \xrightarrow{\checkmark} P'}{P \,;\, Q \xrightarrow{\tau} Q}$$

2.7.2 Interrupt

Interrupt is represented by the notation P Δ Q. It allows a process P to be terminated (and incidentally Q to be initiated) at any point, simply by performing the initial action of Q. It is important to note that if P is terminated directly (by performing a termination action), then both P and Q will terminate. Any actions made by P, other than termination, have no effect on the interrupt operator. Furthermore, internal actions made by Q also have no effect.

Interrupt has the following firing rules.

$$\frac{P \xrightarrow{\mu} P'}{P \,\Delta\, Q \xrightarrow{\mu} P' \,\Delta\, Q} \qquad \frac{P \xrightarrow{\checkmark} P'}{P \,\Delta\, Q \xrightarrow{\checkmark} P'} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \,\Delta\, Q \xrightarrow{\tau} P \,\Delta\, Q'} \qquad \frac{Q \xrightarrow{a} Q'}{P \,\Delta\, Q \xrightarrow{a} Q'}$$

2.8 Untimed CSP Grammar

To complete this primer on untimed CSP, the EBNF grammar for the relevant aspects is outlined below.

P, Q ::=	STOP	%% deadlock
	| SKIP	%% successful termination
	| a $\longrightarrow$ P	%% event prefix
	| x : A $\longrightarrow$ P(x)	%% prefix choice
	| c!v $\longrightarrow$ P	%% compound event prefix
	| c?x : T $\longrightarrow$ P(x)	%% compound prefix choice
	| N = P	%% recursion
	| P $\Box$ Q	%% external choice
	| P $\sqcap$ Q	%% internal choice
	| P $_{A}\|\|_{B}$ Q	%% alphabetized parallel
	| P $\|\|\|$ Q	%% interleaving
	| P $\underset{A}{\|\|}$ Q	%% interface parallel
	| P $\setminus$ A	%% event hiding
	| P ; Q	%% sequential composition
	| P Δ Q	%% interrupt

3 Timed CSP

Our existing CSP framework has abstracted away from the concept of time. In order to effectively simulate real-time systems, in particular those with time-sensitive processes; we need to adapt our CSP specification. This comes in two steps. First we need to adjust our notions of existing untimed operators; to fit within a timed environment. The second step will involve adding some further operators to our specification.

3.1 Considering Time

From the outset, we need to make some important assumptions about how processes will interact in a timed environment and the meaning of time in this particular context. We arrive at the timed computational model below.

- **Newtonian Time** – Within our timed system, all processes follow a global Newtonian clock. In other words, the current 'time' for every process in a system is the same. We ignore the notion of relativity and the possibility for processes to evolve through time at different rates.
- **Real-time** – Our notion of time will consider the set $\mathbb{R}^+$ (positive real numbers). Consequentially this means time is assumed to have infinite granularity; intervals can be as large, small and numerous as we choose.
- **Instantaneous Events** – Events are assumed to be instantaneous. In other words we consider them to take no time to execute. We can implement the idea of an event that takes time by making use of evolution transitions, which will be explained in the following section.
- **Maximal Parallelism** – We consider each process to theoretically be running off its own processor. In this respect it is assumed that no matter how many processes are running, there will always be sufficient processing power to cope with the workload. This allows us to ignore the need for scheduling tasks (and incidentally if this was desired, it would need to be explicitly implemented).
- **Maximal Progress** – An event must be executed the instant all participants of the event are ready to engage in it.

In our specification of untimed CSP we only had to deal with event transitions. These are instantaneous transitions which represent an event between processes. In timed CSP, it is necessary to introduce evolution transitions which represent the passing of time. Evolution transitions can be described by $Q \overset{d}{\rightsquigarrow} Q'$, where d is a time value in $\mathbb{R}^+$.

3.2 Untimed Operators in a Timed Context

Evolution transitions are much like internal transitions when we consider untimed operators. They can change a process, but the operator remains intact. For example, both processes in an external choice can evolve over time into new processes. However, the external choice still remains available and has been unaffected by the evolution transition. All the additional timed firing rules for our untimed CSP operators are considered in this section.

3.2.1 STOP and SKIP

STOP and SKIP both have very simple timed firing rules. As time passes, both these operators remain the same, with the only state change being the global time. We can therefore assume that for these processes an infinite duration of time can pass. The firing rules are:

$$\frac{}{\mathrm{STOP} \overset{d}{\rightsquigarrow} \mathrm{STOP}} \qquad \frac{}{\mathrm{SKIP} \overset{d}{\rightsquigarrow} \mathrm{SKIP}}$$

3.2.2 Prefix Operators

We have a similar situation with the prefix operators; event prefix and prefix choice. As time passes, the operators do not change. The only way to differentiate the states is by considering the change in global time. The timed firing rules are:

$$\frac{}{(a \longrightarrow P) \overset{d}{\rightsquigarrow} (a \longrightarrow P)} \qquad \frac{}{(x : A \longrightarrow P(x)) \overset{d}{\rightsquigarrow} (x : A \longrightarrow P(x))}$$

3.2.3 Recursion

Recursion does not implicitly contain delays and any delays need to be explicitly described in the process. Therefore, time can actually change the process body of a recursive operator, which makes it possible for time to end a recursive loop and change the overall operator. The timed firing rule shows this:

$$\frac{P \overset{d}{\rightsquigarrow} P'}{N \overset{d}{\rightsquigarrow} P'}$$

3.2.4 Choice

Unlike prefix choice, the options provided to a user by external choice can change over time, depending on whether any of the choices contain timed operators or not. With this in mind, the timed firing rules for external choice actually involve a state change other than simply global time. On the other hand, internal choice does not have a timed firing rule. This means that time cannot pass when the state has an internal choice to make. In this respect, the internal choice is treated as urgent and once the choice is made, time can pass again. Below is the timed firing rule for external choice.

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad Q \overset{d}{\rightsquigarrow} Q'}{P \,\Box\, Q \overset{d}{\rightsquigarrow} P' \,\Box\, Q'}$$

3.2.5 Concurrency

The concurrency operators all have similar timed firing rules to the external choice. As time passes, both sides of the operator update, which is regarded as an update of the overall operator. Below are the timed firing rules for the three operators:

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad Q \overset{d}{\rightsquigarrow} Q'}{P \,{}_A||_B\, Q \overset{d}{\rightsquigarrow} P' \,{}_A||_B\, Q'} \qquad \frac{P \overset{d}{\rightsquigarrow} P' \quad Q \overset{d}{\rightsquigarrow} Q'}{P \,|||\, Q \overset{d}{\rightsquigarrow} P' \,|||\, Q'} \qquad \frac{P \overset{d}{\rightsquigarrow} P' \quad Q \overset{d}{\rightsquigarrow} Q'}{P \underset{A}{\|} Q \overset{d}{\rightsquigarrow} P' \underset{A}{\|} Q'}$$

3.2.6 Event Hiding

Event hiding, like concurrency and choice, will update as time passes (either through the global time change or also through updating locally if containing a timed CSP operator). However, there is an interesting addition to event hiding. If the current available action is within the hidden set A of P \ A and offered by P, then time cannot pass. In other words, since all hidden actions have become internal to P, they have become urgent and must be executed before time can pass. This is described concisely in the following timed firing rule:

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad \forall a \in A \mid \neg(P \xrightarrow{a})}{P \setminus A \overset{d}{\rightsquigarrow} P' \setminus A}$$

3.2.7 Control Flow

Since sequential composition passes control from a process P to Q after P terminates, time only affects the P process (as the Q process is not yet active). As such, time changes the process P, but Q remains the same. Like the event hiding operator, sequential composition also has conditions where time cannot pass. When the P process is ready to terminate (✓ is available), then it must be performed urgently. At such a state, until P terminates, time cannot pass. The untimed interrupt operator has a similar timed firing rule to the external choice and concurrency operators. The sequential composition and untimed interrupt timed firing rules are as follows:

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad \neg(P \xrightarrow{\checkmark})}{P \; ; A \overset{d}{\rightsquigarrow} P' \; ; A} \qquad \frac{P \overset{d}{\rightsquigarrow} P' \quad Q \overset{d}{\rightsquigarrow} Q'}{P \;\Delta\; Q \overset{d}{\rightsquigarrow} P' \;\Delta\; Q'}$$

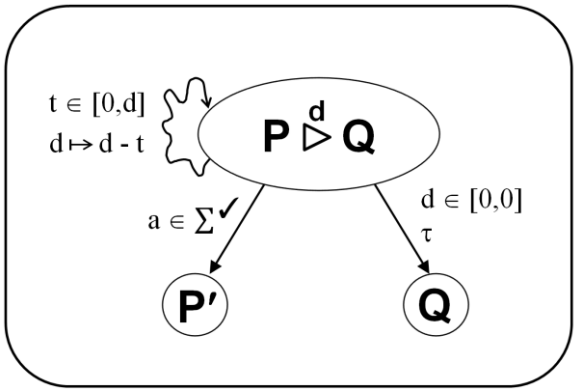
3.3 Timeout

The timeout operator (represented by $P \overset{d}{\triangleright} Q$) is our first new timed CSP operator and is time-sensitive, bound by the time d. If P performs an external event before d reaches 0, then the choice is resolved and we are left with the resultant process of P. If the counter d reaches 0 before an external event is performed by P, then an internal event leads us to continue execution with process Q. As time passes, the counter d decreases. The d can essentially be regarded as the time limit of the operator. This is clear in the firing rules below.

$$\frac{P \xrightarrow{a} P'}{P \overset{d}{\triangleright} Q \xrightarrow{a} P'} \qquad \frac{P \xrightarrow{\tau} P'}{P \overset{d}{\triangleright} Q \xrightarrow{\tau} P' \overset{d}{\triangleright} Q} \qquad \frac{}{P \overset{0}{\triangleright} Q \xrightarrow{\tau} Q}$$

$$\frac{P \overset{d'}{\rightsquigarrow} P'}{P \overset{d}{\triangleright} Q \overset{d'}{\rightsquigarrow} P' \overset{d-d'}{\triangleright} Q} \quad [0 < d' \leq d]$$

Furthermore, we can give a visual representation of the timeout operator's semantics using the following timed transition system:



Here we can see that if an action a is performed, the state does a transition to P′. If a time t in the interval [0,d] passes, then the state updates the timeout's d to d-t. Finally, once d is 0, the state does an internal τ transition to Q.

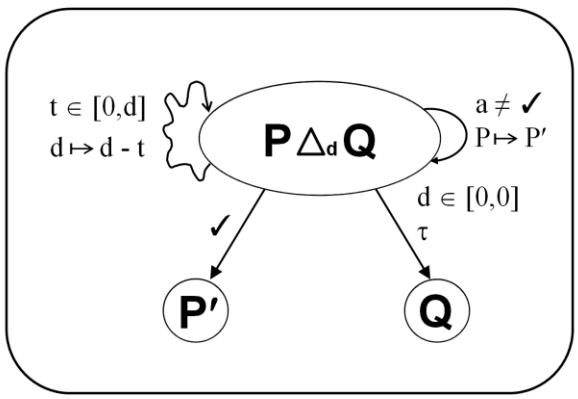
3.4 Timed Interrupt

The timed interrupt operator P Δ_d Q is similar to the interrupt operator from our untimed CSP specification. The difference is that P is interrupted by a time condition, rather than execution by Q. Like timeout, as time passes, the remaining d updates. Once d reaches 0, P is interrupted and a τ transition to Q occurs. Again, d can be described as the time limit of the operator. Unlike timeout, an execution by P does not result in a transition to P (except for termination). The four firing rules are below:

$$\frac{P \xrightarrow{\mu} P'}{P\Delta_d Q \xrightarrow{\mu} P'\Delta_d Q} \quad [\mu \neq \checkmark] \qquad \frac{P \xrightarrow{\checkmark} P'}{P\Delta_d Q \xrightarrow{\checkmark} P'} \qquad \frac{}{P\ \Delta_0\ Q \xrightarrow{\tau} Q}$$

$$\frac{P \overset{d'}{\rightsquigarrow} P'}{P\ \Delta_d\ Q \overset{d'}{\rightsquigarrow} P'\ \Delta_{d-d'}\ Q} \quad [d' \leq d]$$

This operator also forms an important part of our implementation and so we give the following timed transition system:



3.5 Timed Event Prefix

The timed event prefix allows us to store data on the amount of time that has passed between the point an event was made available and the point it was executed. It is denoted by $a@u \longrightarrow Q$. When action a is performed without any delay, we have a resultant process Q with all free occurrences of u replaced by 0. If time passes and the operator evolves, time d is added to all free occurrences of u in the resultant process. We can see this in the firing rules below.

$$\frac{}{(a@u \longrightarrow Q) \longrightarrow Q[0/u]}$$

$$\frac{}{(a@u \longrightarrow Q) \overset{d}{\rightsquigarrow} (a@u \longrightarrow Q[(u+d)/u])}$$

3.6 Syntactic Sugar

There are two instances of syntactic sugar which we will make use of in our timed CSP implementation; delay and WAIT.

3.6.1 Delay

The delay operator utilises both the event prefix and timeout operators. It is an event prefix with an action a, which once executed leads to a timeout instance $STOP \overset{d}{\triangleright} P$. We can write the delay operator as $a \overset{d}{\longrightarrow} P$, where d is the delay time after executing a before actions are available from P.

We say that $a \overset{d}{\longrightarrow} P = a \longrightarrow (STOP \overset{d}{\triangleright} P)$. It is also possible to have a delay which is constrained to an interval, taking the form:

$$a \overset{[d_1,d_2]}{\longrightarrow} P$$

Taking the delay interval [1,3] as an example, after executing a, the option to timeout to P becomes available 1 second later. However time can still pass up until 3 seconds after the execution of a, at which point the timeout must be executed for time to continue passing. For the sake of simplicity, we have only considered delays of a single value rather than an interval for our implementation.

3.6.2 WAIT

The WAIT operator is simply a specific instance of a timeout operator, where process P is a STOP and process Q is a SKIP. We can write the operator as WAIT d, where d is the amount of time before the timeout does a transition to SKIP.

We say that $WAIT\ d = STOP \overset{d}{\triangleright} SKIP$.

3.7 Timed Operator Semantics

Alongside the new timed operators and syntactic sugar, there are also some extra semantics which play a role in our implementation. We list the most important ones here.

Laws for Timeout

$P \overset{d}{\triangleright} (Q \overset{d'}{\triangleright} R) = (P \overset{d}{\triangleright} Q) \overset{d+d'}{\triangleright} R$ ⟨ $\triangleright$-associative ⟩

$(P \sqcap Q) \overset{d}{\triangleright} R = (P \overset{d}{\triangleright} R) \sqcap (Q \overset{d}{\triangleright} R)$ ⟨ $\triangleright$-left-distributive ⟩

$P \overset{d}{\triangleright} (Q \sqcap R) = (P \overset{d}{\triangleright} Q) \sqcap (P \overset{d}{\triangleright} R)$ ⟨ $\triangleright$-right-distributive ⟩

$(P \,\Box\, Q) \overset{d}{\triangleright} R = (P \overset{d}{\triangleright} R) \,\Box\, (Q \overset{d}{\triangleright} R)$ ⟨ $\triangleright$-$\Box$-left-distributive ⟩

$P \overset{d}{\triangleright} (Q \,\Box\, R) = (P \overset{d}{\triangleright} Q) \,\Box\, (P \overset{d}{\triangleright} R)$ ⟨ $\triangleright$-$\Box$-right-distributive ⟩

$STOP \overset{d}{\triangleright} P = WAIT\ d;\ P$ ⟨ $\triangleright$-delay ⟩

$(WAIT\ d;\ P) \overset{d+d'}{\triangleright} Q = WAIT\ d;\ (P \overset{d'}{\triangleright} Q)$ ⟨ $\triangleright$-delay-1 ⟩

$(WAIT(d + d');\ P) \overset{d}{\triangleright} Q = WAIT\ d;\ Q$ ⟨ $\triangleright$-delay-2 ⟩

Laws for Delay

$WAIT\ d;\ WAIT\ d' = WAIT(d + d')$ ⟨ delay-sum1 ⟩

$a \xrightarrow{d} P = a \rightarrow WAIT\ d;\ P$ ⟨ delay-sum2 ⟩

$(WAIT\ d;\ Q) \setminus A = WAIT\ d;\ (Q \setminus A)$ ⟨ hide-delay ⟩

Laws for Timed Interrupt

$P\ \Delta_d\ (Q\ \Delta_{d'}\ R) = (P\ \Delta_d\ Q)\ \Delta_{d+d'}\ R$ ⟨ Δ_d-associative ⟩

$(P \sqcap Q)\ \Delta_d\ R = (P\ \Delta_d\ R) \sqcap (Q\ \Delta_d\ R)$ ⟨ Δ_d-left-distributive ⟩

$P\ \Delta_d\ (Q \sqcap R) = (P\ \Delta_d\ Q) \sqcap (P\ \Delta_d\ R)$ ⟨ Δ_d-right-distributive ⟩

$(P \,\Box\, Q)\ \Delta_d\ R = (P\ \Delta_d\ R) \,\Box\, (Q\ \Delta_d\ R)$ ⟨ Δ_d-$\Box$-left-distributive ⟩

$STOP\ \Delta_d\ P = WAIT\ d;\ P$ ⟨ stop-Δ_d-delay ⟩

$(WAIT\ d;\ P)\ \Delta_{d+d'}\ Q = WAIT\ d;\ (P\ \Delta_{d'}\ Q)$ ⟨ Δ_d-delay-1 ⟩

$(WAIT(d + d');\ P)\ \Delta_d\ Q = WAIT\ d;\ Q$ ⟨ Δ_d-delay-2 ⟩

⟨ $\triangleright$-associative ⟩ and ⟨ Δ_d-associative ⟩ play a particularly important part in our implementation as they imply that a nested timeout or timed interrupt requires all the operators on the left of the expression to be updated. In other words, a recursive updating algorithm was required in our implementation.

3.8 Timed CSP Grammar

We can now expand on our untimed CSP grammar to include the timed CSP operators.

P, Q ::=	STOP	%% deadlock
	| SKIP	%% successful termination
	| $a \longrightarrow P$	%% event prefix
	| $x : A \longrightarrow P(x)$	%% prefix choice
	| $c!v \longrightarrow P$	%% compound event prefix
	| $c?x : T \longrightarrow P(x)$	%% compound prefix choice
	| $N = P$	%% recursion
	| $P \,\Box\, Q$	%% external choice
	| $P \sqcap Q$	%% internal choice
	| $P \,{}_A\|\|_B\, Q$	%% alphabetized parallel
	| $P \,\|\|\|\, Q$	%% interleaving
	| $P \,\underset{A}{\|\|}\, Q$	%% interface parallel
	| $P \,;\, Q$	%% sequential composition
	| $P \,\Delta\, Q$	%% interrupt
	| $P \overset{d}{\rhd} Q$	%% timeout
	| $P \,\Delta_d\, Q$	%% timed interrupt
	| $a@u \longrightarrow Q$	%% timed event prefix
	| $a \overset{d}{\longrightarrow} P$	%% delay
	| $a \overset{[d_1,d_2]}{\longrightarrow} P$	%% interval delay
	| WAIT d	%% wait

4 Prolog

As this project focuses on the extension of ProB, the bulk of which is programmed in SICStus Prolog, a brief overview of Prolog is necessary.

Prolog is a declarative programming language based on first order predicate logic and has incidentally been closely linked with the concept of logic programming. It has a number of applications, from its primary field of artificial intelligence to database systems. Prolog programming has a modular nature and consists of lists of logical rules that the system needs to follow, as well as the goal, represented by queries. Incidentally, this strong use of logical rules will help link Prolog with CSP[7].

Prolog syntax revolves around terms. This data structure consists of atoms (which can be considered strings), numbers (integers and floats for example), variables and compound terms. Compound terms consist of a Prolog atom followed by 1 or more arguments (which can be any term). For example, the add operation add(1,2) could be considered a compound term.

As well as terms we have clauses (facts and rules), programs and queries.

Facts are predicates which are relations explicitly defined as being true. For example, the following could be considered a fact in Prolog:

```
food_is_good.
```

Note that facts end in a full stop.

Rules are the core part of Prolog programming and give meaning and definition to more complex terms. They consist of a head (the term predicate) and a body (further predicates which define the head). Taking our add operation example, we could use the following rule to define it:

```
add(a,b,c) :-
  c is a + b.
```

The body of a rule can consist of multiple facts and terms, which are split by commas. The body, like facts, ends in a full stop to close the clause. Note that a program is simply a sequence of clauses. A complete program will give rules for all terms used until we are left with only facts.

A query has a similar structure to a rule, but is instead tested against the rules programmed. Prolog analyses queries, and checks (according to the pre-defined rules) whether the predicates can all be proved true and then provides a suitable output. Queries generally start with ?- to indicate their nature and this usually appears when running a Prolog instance.[8]

ProB has been specifically programmed in SICStus Prolog[9], primarily for performance reasons. However, this is also important for us as it allows us to work easily within the rational numbers due to SICStus Prolog having infinite-precision integers[10]. We have opted for rational numbers rather than real numbers to help simplify our models within the railway domain. Naturally, there is no real need for irrational numbers in these models, so we have simply decided to abstract away from them.

5 Haskell

Haskell is the language behind ProB's parser. A brief overview of the language is useful for understanding the extensions we have made in our implementation to account for the new timed CSP syntax. We also give a quick summary of the Parsec library which played an important part in making our parser extension an easier process.

5.1 General Haskell

Haskell is a functional programming language which has a number of similarities to Prolog. Like Prolog, it is based on rules and variables cannot be restated during execution. The importance of this is it ensures that there are no side effects in the implementation of variables. It means referential transparency is possible[11]. Also similar is its modular architecture, which allows for definitions to be contained within modules and prevents the possibility of variable clashes[12].

Haskell is a lazy programming language, which means that functions are only executed when absolutely necessary. This leads to a highly-efficient language which avoids performing functions without purpose.

It is also statically typed, which means that types are identified by the compiler. Strings are differentiated from integers, without needing to label the types explicitly; like in languages such as Java[11].

There are two categories that Haskell code can fall under: expressions and definitions.

Expressions are simply statements read by the compiler. These can be arithmetic operations, Boolean operations or more complex tasks such as string concatenation. Due to Haskell's statically-typed nature, these expressions can be read by the compiler and a suitable output provided, without anything defined. Definitions are of the form name = expression[12].

Haskell is a convenient language for the parser as it works well with Prolog due to their similarities. It also allows for use of the Parsec library which makes the process of extending the parser a simpler one.

5.2 Parsec

Parsec is a monadic parser combinator library for Haskell. A parser combinator simplifies the process of designing a parser. Parsers can be generated and combined to form larger parsers. This flexibility also makes extending a parser much simpler.

Since parser combinators are in the same language as the parser itself, there's no need to learn a new language to understand the combinator. Parsec in particular is a useful tool as it has been designed with speed and simplicity in mind; as well as extensive documentation[13].

The use of Parsec in ProB's parser has meant that the code is laid out in an understandable manner and that necessary areas of change were easy to identify. As such, to extend the parser, changes only had to be made in a few, very specific locations. Without the Parsec library, this would have been a much more challenging task.

6 Tcl/Tk

Tcl/Tk is used to implement ProB's GUI. The small modifications we have made to the GUI are more easily understood with a basic knowledge of the Tcl/Tk language. We provide a brief outline here.

Tcl/Tk is a combination of the scripting command language Tcl (Tool Command Language) and the Tk graphical toolkit. Combined, they make an easy-to-use tool for GUI development.

Essentially, Tcl code comprises of a series of commands. These commands handle data and can take various options which determine how they execute.

There are a number of commands in Tcl/Tk directly involved with setting up a GUI. For example; there is a button command for creating buttons and a frame command for setting up frames. This set-up means that working with Tcl/Tk is a very straightforward process[14].

Tcl/Tk is particularly useful for ProB as it is able to call Prolog commands. This means it interacts well with the existing Prolog implementation and makes for easy communication between the GUI and simulator.

7 ProB

We now have a comprehensive specification of CSP and have covered the basics of ProB's three main languages; Prolog, Haskell and Tcl/Tk. We can now begin to look at ProB itself. The following chapter starts by covering the function and method of a CSP simulator. We then look at the key features, software architecture and dataflow of ProB.

7.1 CSP Simulation

Before looking at ProB, it is important to have a precise definition of what a CSP simulator is and how it functions.

Fundamentally, a CSP simulator represents a static CSP specification in a dynamic run-time environment. Using a CSP simulator, we can iterate through transitions and analyse the changes in a system as we move through the execution.

As a user moves through a CSP specification, the simulator offers the actions that can be performed. The user is then able to choose one of these actions and perform it, which leads the simulator to determine the next state in the execution. At this new state, the current available actions are once again offered. This cycle is repeated at every step in the execution.

We can consider the operational semantics of CSP in the form of step laws. A step law is an equational definition consisting of two parts. The first part describes the set of available actions, while the second part gives the resultant process. The second part also has conditional guards which indicate the required actions to result in each process.

As an example, below is the step law for untimed external choice:

$$\text{actions}(P \,\Box\, Q) = \text{actions}(P) \cup \text{actions}(Q)$$

$$\begin{aligned} \text{perform}(x, P \,\Box\, Q) = {} & \text{perform}(x, P) && x \in \text{actions}(P) \setminus \text{actions}(Q) \\ & \text{perform}(x, Q) && x \in \text{actions}(Q) \setminus \text{actions}(P) \\ & \text{perform}(x, P \sqcap Q) && x \in \text{actions}(P) \cap \text{actions}(Q) \end{aligned}$$

The first part describes the available actions for external choice $P \,\Box\, Q$ to be the set actions(P) $\cup$actions(Q); all available actions from P and all available actions from Q.[15]

The second part describes the three possible resultant processes that could come from an action x with the external choice. It also describes the subset action x would need to be in to achieve the corresponding process. We see that if action x is an available action in P, but not Q; then the resultant process is P. Similarly, if action x is an available action in Q, but not P; then the resultant process is Q. Finally, we have the situation where x is an available action for both P and Q. This results in an internal choice of $P \sqcap Q$ and represents the non-deterministic situation described in **2.4.1**.

7.2 An Introduction to ProB

ProB was originally designed as a B-method model checker and simulator. The B-method is a formal method for the specification and development of computer systems. It is primarily used with systems of a critical nature, railway control in particular.

While both B and CSP are formal methods, they are quite different in nature. B is not a process algebra. Incidentally it does not consider a system as a collection of processes and the interactions between them. Rather, it is based on the notion of an abstract machine whereby components are described through sets, relations and functions. Interactions between components are represented by generalised substitutions through weakest predicate transformations.

ProB has been primarily developed in SICStus Prolog, with a Haskell parser and its graphical user interface implemented in the Tcl/Tk GUI toolkit. The tool first translates the original abstract machine notation of the B-language into Prolog term tree representation. This is then processed into a more structured representation by the front-end of ProB, which is then fed to the ProB interpreter. The interpreter iterates through the representation; while making calls to the ProB kernel, which provides the basic datatypes and operations of the B-language.[16]

Although the tool was developed from a previous CSP animator, it initially focused solely on the B-method. Only when the CIA (CSP Interpreter and Animator)[17] tool was combined with ProB did the ability to simulate CSP become an option.

7.2.1 CIA (CSP Interpreter and Animator)

The CSP Interpreter and Animator has become a subset of ProB's modular architecture. The main module we are concerned with is haskell_csp.pl. The file is named this because it utilises a format produced by an existing Haskell compiler fecsp/cspcomp.

This module contains all the operational CSP semantics as a list of firing rules. These firing rules, of the form cspm_trans(e,a,e′,wf), represent an expression e evolving into e′ by performing action a[18]. Note that there is also a fourth predicate, which is for ProB's 'waitflag'; a built in function used in deadlock-checking[19]. This is not an important factor in our implementation; however we have included it in our firing rules to fit within the system.

We can analyse the haskell_csp.pl code directly in order to locate some familiar firing rules from our understanding of CSP.

```
cspm_trans(skip(SrcSpan),tick(SrcSpan),stop(SrcSpan),_).
```

The above code represents the firing rule for the skip process. As we can see, there is an initial process skip, which evolves into the process stop by performing the action ✓ (termination action). This is directly comparable with our understood notation of the firing rule for skip:

$$\mathrm{SKIP} \xrightarrow{\checkmark} \mathrm{STOP}$$

Similarly, we can take the example of internal choice from the code:

```
cspm_trans('|~|'(X,_Y,SrcSpan),tau(int_choice_left(LSpan)),X,_WF) :-
    shift_span_for_left_branch(SrcSpan,LSpan).
cspm_trans('|~|'(_X,Y,SrcSpan),tau(int_choice_right(RSpan)),Y,_WF) :-
    shift_span_for_right_branch(SrcSpan,RSpan).
```

Here, we can see there are two rules. We have one definition for an internal choice where X is selected and a second definition for when Y is selected. The shift_span rules simply update the Span information. In ProB, the Span provides pointers in the execution and is used for error handling.

These two rules correspond directly with our two inference rules for internal choice:

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

7.3 Key Features and Classification of ProB

Simulators and model checkers can be classed in a number of ways. First, we have the standard simulator or model checker which takes in the entire transition system, a static input. In other words, every possible trace that could be made according to the specification of the input code is generated. The issue with this is the state-explosion problem. As the complete transition system tracks every possibility, a small increase in the input code specification can result in an exponential growth of the generated transition system. One way of solving this issue is by creating an 'on-the-fly' simulator and model checker.[20] Here the transition system is generated step-by-step as the user traverses through the actions. It is clear that in such a situation only the visited nodes and the current state's adjacent nodes are generated. ProB is classed as an 'on-the-fly' model checker and simulator. In this respect, implementation changes which apply to all nodes hold less gravity than they would with a regular model checker and simulator (as there wouldn't be the state-space explosion issue). Nevertheless, it is still better that we avoid making changes that affect all nodes in a system, simply because there is likely to be a more efficient alternative.

We can also differentiate between explicit-state and symbolic model checkers and simulators[21]. Given a bounded state (that relies upon a variable) the adjacent nodes can either be represented explicitly or symbolically.

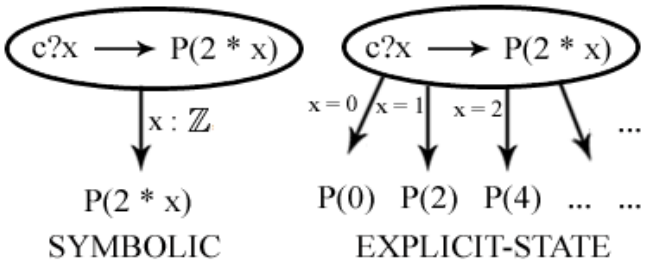


Figure 1. Symbolic and explicit-state representation

Figure 1 uses the example of a CSP channel. In our example, the channel c accepts any x in the set of integers. The output is x multiplied by 2. In explicit-state representation, we say that there are infinitely many possible actions that can be done in this state. If x is 0, we can do the action that produces 0; if x is 1, we can do the action that produces 2, and so on. On the other hand, symbolic representation has just 1 possible action for all x in the set of integers, which produces 2*x. ProB uses explicit-state representation whereas we require time as input; and providing actions for all possible times would be illogical. However, ProB allows for actions leading to a non-ground state (a state with an undeclared variable) to be presented to the user (even though a non-ground state cannot be initialised). We have used this feature to our advantage, substituting the non-ground variable in an action's resultant state with new values based on the explicit time stated by the user before the state is initialised. That way, only one state is generated but substituted with specific values during simulation. This is effectively mimicking the behaviour of a symbolic representation.

There are two main data structures that can be used to represent the process graph; an adjacency matrix and an adjacency list[22]. For a given state or node, these data structures store the data on the adjacent nodes. ProB uses adjacency lists to store such information, as evidenced by Figure 2.

```
:- dynamic current_options/1.
current_options([]).

set_current_options(Options) :-
    retractall( current_options(_) ),
    assert( current_options(Options) ).
```

Figure 2. Dynamic clause for available options in state_space.pl

In Figure 2 we see that the current_options clause, which contains the nodes adjacent to the current one, is a list (as indicated by the empty list type inside the clause).

Finally, we need to make clear that ProB is able to operate completely within the set of rational numbers. We expressed the importance of using rational numbers in the initial document. Essentially we need to clarify that given all inputs are in $\mathbb{Q}$ and all constants (specified in the CSP code) are in $\mathbb{Q}$, then all information will be in $\mathbb{Q}$. In our implementation, the only operations carried out between inputs and constants are two arithmetic operations (namely addition and subtraction) and two Boolean operations (more than and less than). It is known that rational numbers are closed for all the arithmetic operations and the Boolean operations will either output 0 or 1 (both rational numbers). Therefore, we can say that all information in ProB will be in the set of rational numbers. In order to work with rational inputs and provide rational outputs, we have developed a rational number 'framework' which deals directly with conversion and calculations between rational numbers. This 'framework' will be described in detail in part III.

7.4 Software Architecture

As a result of being programmed in Prolog, ProB has a modular architecture. Instead of having all the components merged together through an integrated architecture, each component of ProB is clearly separated into its own module.

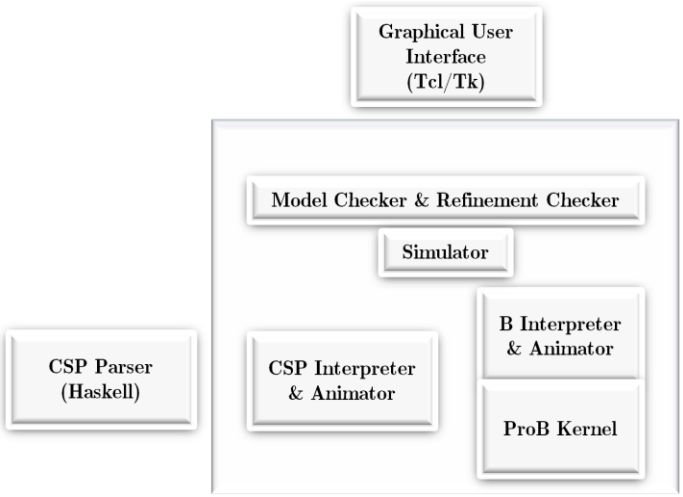


Figure 3. General structure of ProB

The majority of ProB's modules are involved with the B-language interpretation, animation and model-checking. ProB also allows for CSP-B specification which combines both CSP and B code. However, we are more interested in the set of modules that form the CSP Interpreter and Animator. We also work with the simulator, parser and GUI. In Figure 3 we can see where these components fit into the overall software architecture.

While Figure 3 gives us a good grasp of the overall layout of the software; in order to fully understand ProB, we need to look at the how data flows between the various parts of ProB.

7.5 Dataflow

This section considers how data moves through ProB, from the initial CSP code to the GUI output. We also give a more detailed view of the individual files and how they interact with each other.

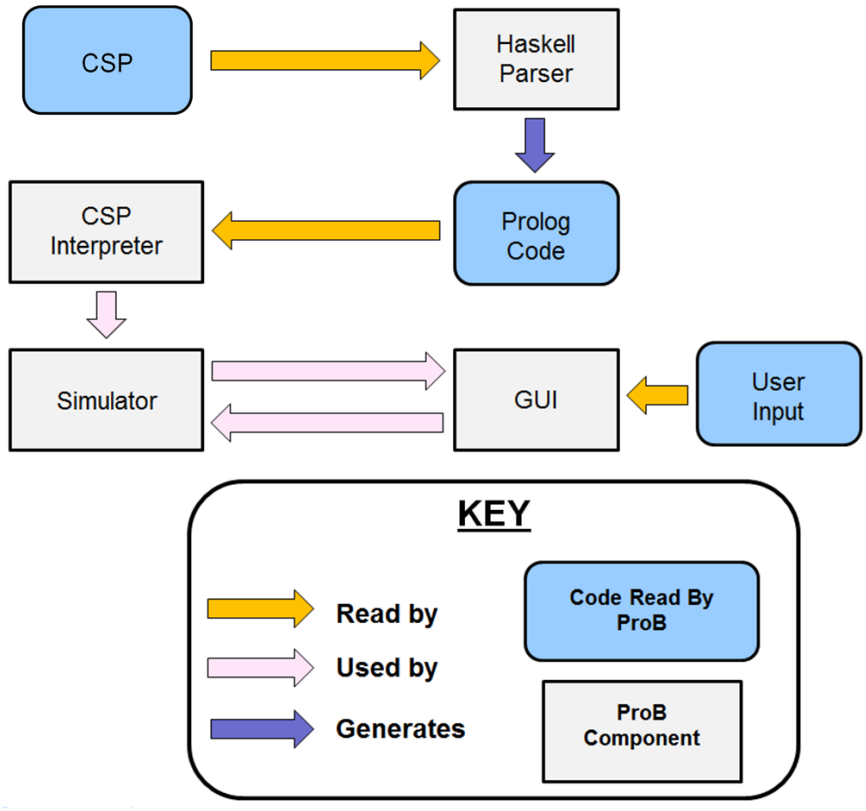


Figure 4. CSP dataflow in ProB

In Figure 4, we can see that we start with CSP code generated by the user. ProB's parser, coded in Haskell, then reads the CSP code and generates its corresponding Prolog code. The CSP Interpreter and Animator then compares the Prolog code against its firing rules and decides its behaviour at the current execution step. This data is then used by the simulator to determine the available actions and the resultant states for each action. The user chooses an option; leading to the next state and new available actions.

Figure 5 gives a concise visual representation of the connection between the relevant files in ProB's implementation.

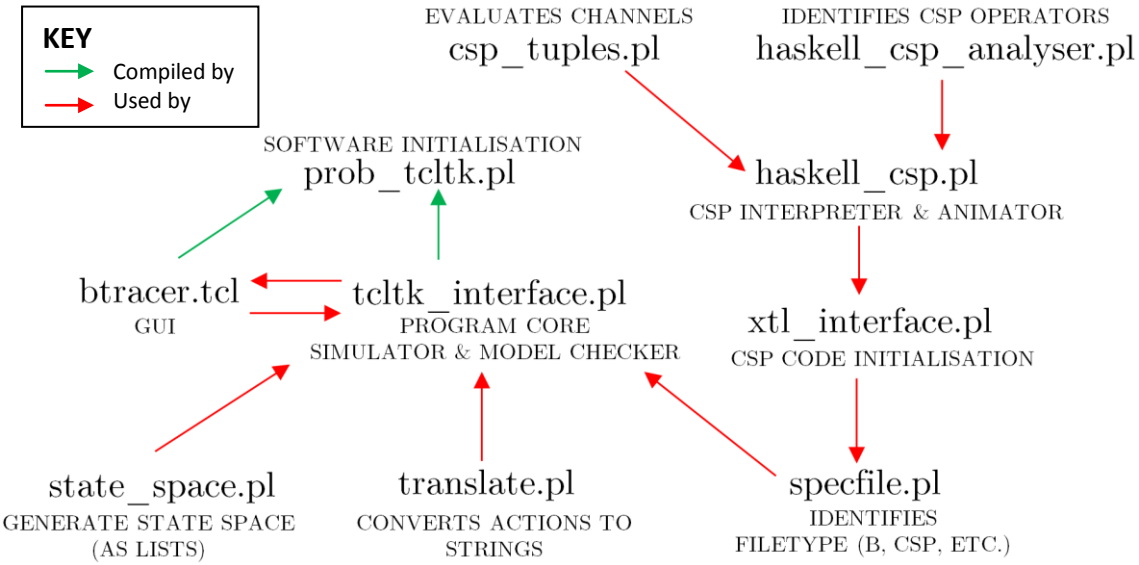


Figure 5. Connections between the key modules of ProB.

In Figure 5, we can see that prob_tcltk.pl deals with the initialisation of ProB. In particular it compiles both btracer.tcl and tcltk_interface.pl. The btracer.tcl file provides the graphical user interface. In terms of our CSP simulation, it takes user input, which is used by the simulator, and also outputs data from the simulator. The heart of ProB is tcltk_interface.pl. This module utilises most of the other modules that make up ProB in order to carry out the simulation and model checking tasks. In our case, the most important modules it uses are state_space.pl and specfile.pl (which calls the transitions from haskell_csp.pl via xtl_interface.pl). The state_space.pl module deals with setting up and maintaining a number of list clauses which are stored in the Prolog database. These list clauses store information such as the available options, current state and history. The haskell csp.pl module is the core of the CIA. It contains all the firing rules and determines transitions based on the state.

Figure 5 also shows us a few auxiliary modules which play an important part in CSP dataflow, but contain little or no extra implementation. Within the CIA, there are two key modules used by haskell_csp.pl; namely csp_tuples.pl and csp_analyzer.pl. The csp_tuples.pl module helps with channel evaluation and CSP I/O. The csp_analyzer.pl module deals with the identification of CSP constructs. We also have xtl_interface.pl which deals with the initialisation of CSP. Namely it sets up the root state and the start_cspm_MAIN action, which are presented on loading a standard CSP file in ProB. It also acts as an interface between the simulator and haskell_csp.pl, via specfile.pl (which identifies the file-type of specific actions). Finally we have translate.pl, which converts the CSP actions and processes into strings prior to being output by the GUI.

In Part II, we have covered the necessary background research. We have looked at the untimed and timed CSP specification in detail. We have also given a brief outline of the three languages involved in our implementation; Prolog, Haskell and Tcl/Tk. Finally we've described the key aspects of ProB, as well as its software architecture and dataflow.

Part III moves on from the background research and considers the implementation itself.

PART III
Implementation

III Implementation

Part III describes the implementation of our timed CSP simulator in full. We first discuss the preparation carried out to familiarise ourselves with ProB. This is followed by the important design decisions made. Finally we give an in-depth analysis of the implemented code in the parser, CIA, simulator and GUI.

8 Working with ProB

In order to grow accustomed to implementing CSP code and working with ProB, we designed a simple untimed railway in the form of a square with 4 tracks. The design was based on a specification outlined in Kirsten Winter's 'Model Checking Railway Interlocking Systems'[23]. She describes a train in the form of a front and a rear. The rules state that if both the front and rear are on the same track, then the next possible move is the movement of the front to the adjacent track. Furthermore, if the front is already on the next track and the rear on the previous track, the next possible move is the movement of the rear to the adjacent track. A train can only move to the next track if it is clear. To achieve this, we implemented a signal system that synchronises with the trains to prevent them from entering an engaged track. Our implementation utilises mutual recursion to keep the trains running indefinitely.

```
datatype Trains = CR | FS
Tracks = {1..4}

channel Moveff,Mover: Tracks.Tracks
channel GreenSignal, ClearSignal: Tracks

MAIN = Train [| {| GreenSignal,ClearSignal |} |] Signals

Signals = Signal(1) ||| Signal(2) ||| Signal(3) ||| Signal(4)

Train = BehaveTrain0(CR.4.4) ||| BehaveTrain0(FS.2.2)

BehaveTrain0(id.ffront.rear) = Moveff.ffront.((ffront%4)+1) -> BehaveTrain2(id.((ffront%4)+1).rear)

BehaveTrain1(id.ffront.rear) = GreenSignal.((ffront%4)+1) -> Moveff.ffront.((ffront%4)+1) -> BehaveTrain2(id.((ffront%4)+1).rear)

BehaveTrain2(id.ffront.rear) =  Mover.rear.ffront -> ClearSignal.rear -> BehaveTrain1(id.ffront.ffront)

Signal(i) = ClearSignal.i -> GreenSignal.i -> Signal(i)
```

Figure 6. Untimed Railway Model CSP Code.

In Figure 6, we have 4 signals for each track, as well as two trains CR and FS, interleaving. The trains and signals run in parallel for GreenSignal and ClearSignal. Initially, we have both trains following BehaveTrain0. This allows trains to move the front to the next track without a green signal. This is because the trains are initially on opposite ends of the square which means the next track is already clear. After BehaveTrain0, they move onto BehaveTrain2 which is the movement of the rear to the next track, at which point they never return to BehaveTrain0. The trains then alternate between BehaveTrain1 (movement of the front) and BehaveTrain2 (movement of the rear) in a mutual recursive fashion. Movement of the front requires a GreenSignal first, while movement of the rear sends out a ClearSignal after moving off the track. These two actions synchronise with the corresponding Signal(i), which acts as a communicator between the two trains. Also note the use of modular arithmetic (base 4 in this case) in order to determine the next track.

Appendix 1 provides a brief example run of the CSP code in ProB.

9 Design Decisions

We made a number of design decisions throughout our project. This section briefly considers the options that were available to us and justifies the choices we made.

We first needed to decide what parts of timed CSP we wanted to include in our implementation. While the ideal situation would be a complete timed CSP specification, we needed to consider time and resources. The operators we considered were timeout, timed interrupt, timed event prefix, delay, interval delay and wait.

Timeout and timed interrupt were obvious requirements for our implementation. These operators play a big part in timed CSP and most models need to use these operators or syntactic sugar that contains them. The delay was also an important part of our implementation. For our more complex models, including the railway crossing model from Steve Schneider's book, we required the delay operator. Wait was also a useful operator to have and due to its simple syntax, was not a difficult operator to implement.

In the end, we opted to omit timed event prefix and interval delay. The timed event prefix would have required a significant amount of time to implement. The parsing would have also been difficult (the @ symbol is already in use in the parser). Similarly, the interval delay may have been tricky to parse due to its input. A new input type consisting of four parts would also have been needed (the two constraints of the interval and open or closed intervals which may or may not be paired). The amount of code in the implementation would have increased quite significantly. As a result, these two operators were determined to be the best candidates for omission.

Our next important decision was the method of extending the timed CSP syntax. There were two possible options:

- Have one type for untimed and timed process terms (simply add the timed process terms to the existing untimed type).
- Create two types, where timed CSP imports the untimed CSP type.

The first option is less work at the implementation stage; however it also means that we have a flat-type system, which may cause issues in the future. The second option is harder to implement, but also means that changes from CSP are inherited.

We eventually opted for the first option, due to the time and resource constraints. It was also a suitable option for the goals we wanted to achieve.

Finally, we had to decide how we would add the syntactic sugar to our implementation. We could either expand their definitions within the Haskell parser, or define their derived semantic rules in Prolog. The first option has the benefit of keeping semantics small, leading to better correctness. However the syntax may end up large and the representation of terms in the simulator would be different from the user input. The second option has the benefit of keeping the syntax small and properly representing the terms input by the user, but slightly increasing the semantics.

Our final decision was that defining the derived semantic rules in Prolog was the best option as we wanted to correctly represent the intended user-input and it was also the best option from a software engineering perspective.

10 Prolog Implementation

The majority of ProB is programmed in Prolog, including the CIA and simulator. The bulk of our implementation is carried out in both of these. Hence, we start our implementation analysis with the Prolog code. The first part of this chapter will look at the most important module of our implementation: haskell_csp.pl. We will then move on to the modifications made in the simulator: tcltk_interface.pl. Finally, we will cover the minor modifications to the other auxiliary Prolog modules; translate.pl, xtl_interface.pl, haskell_csp_analyser.pl and state_space.pl.

10.1 haskell_csp.pl

In haskell_csp.pl, we have made three major extensions. First, we have added a rational number 'framework' which deals with the conversion, calculation and simplification of rational numbers within ProB. Second, we have implemented auxiliary rules which are called by the firing rules and are used to determine the overall time limit d of the state and calculate the resultant state. The reason we've create auxiliary rules in this manner is to account for nesting in a state, as discussed in **7.2**. Finally, we have provided the new timed firing rules; including untimed operator semantics in a timed context, and the new timed operator semantics.

10.1.1 Rational Number Framework

Our first extension of haskell_csp.pl is the code necessary to convert and simplify rational numbers. We have also added code which makes calculations and Boolean checks with rational numbers. The referenced code is in **Appendix 2**.

The first set of rules (gcd) calculate the greatest common divisor of a fraction. Additionally, lcm determines the lowest common multiple. The simplify rules use the greatest common divisor to simplify fractions (by performing integer division on the numerator and denominator with the GCD). These rules have been designed to be used solely by the other rules in the rational framework (which perform calculations and Boolean checks).

We have implemented rules which add fractions, subtract fractions and do Boolean tests; more than and less than. There are also rational rules which just simplify a fraction and convert an integer to a fraction (for use in the other calculations).

Finally, we have fractoint, which converts a fraction to an integer. This is done after all the calculations, to produce a neater output. An output of 5 is more aesthetically-pleasing to the end-user than 5/1, for example.

10.1.2 Auxiliary Rules

We have developed two basic auxiliary rules which are utilised by the firing rules and simulator in order to achieve our timed CSP implementation.

Calculating the Overall Time Limit – 'find_d'

Our first auxiliary rule is find_d. Its purpose is to determine the overall time limit for a state (including states with nesting). This is particularly important for timeout and timed interrupt as it ensures that the $\langle \triangleright$-associative $\rangle$ and $\langle \Delta_d$-associative $\rangle$ laws from **7.2** are enforced.

As a reminder, they are:

$$P \overset{d}{\triangleright} (Q \overset{d'}{\triangleright} R) = (P \overset{d}{\triangleright} Q) \overset{d+d'}{\triangleright} R$$
$$P \; \Delta_d \, (Q \; \Delta_{d'} \, R) = (P \; \Delta_d \; Q) \; \Delta_{d+d'} \, R$$

These rules imply that a time update on a nested timeout or timed interrupt results in an update of all time limits on the left side of the expression, while the right side remains the same. As such, the overall time limit of a nested timeout or timed interrupt will be the smallest *d* on the left side of the expression. This is reflected in Figure 7.

```
find_d('[d>'(X,D,_Y,_Span),DF) :-
(find_d(X,E) -> rat(D=<E,DF) ; DF=D).

find_d('/d\\'(X,D,_Y,_Span),DF) :-
(find_d(X,E) -> rat(D=<E,DF) ; DF=D).

find_d(wait(D,_Span),DF) :-
(fractoint(D,DF) -> true ; rat(D,DF)).
```

Figure 7. find_d rules for timed operators.

First, note that for the timeout operator we have a state with left side X, time limit D and right side Y (and a Span pointer). This corresponds to the timed CSP syntax $X \overset{d}{\triangleright} Y$. We have a similar situation for the timed interrupt. For WAIT, we simply have a D (corresponding to the *d* in WAIT d) and a Span pointer.

The rules recursively cycle through the left side of the expression and compare the time limits. If the left side X does not have a find_d rule (it is either a SKIP, STOP or event prefix operator) then the rule will fail and the recursive process ends. At this point, the last time limit *d* that was determined to be the smallest is returned as the result.

The WAIT operator does not have nesting, and so its overall time limit is simply defined as the *d* of WAIT d. The value is either converted to an integer (if possible) or reduced to its simplest fraction using fractoint and rat respectively.

The find_d rule is not just useful for the nesting of timeout and timed interrupt; it also deals with the other binary operators.

Appendix 3 provides the find_d rules for choice and concurrency.

Referring to **Appendix 3**, note that aParallel is alphabetized parallel and sharing represents interface parallel. Also, both these operators have corresponding eaParallel and esharing operators. These are simply extended versions of the operators which have their channel expressions expanded. All aParallel and sharing operators call their respective eaParallel and esharing operators in their untimed transitions. Therefore we need to duplicate any rules for these extended versions in order for ProB to function as intended. They are essentially for optimisation within ProB.

For aParallel, we have the left side of the alphabetized parallel operator (X) and the right side (Y). These two sides also have corresponding channel lists CListX and CListY, which are essentially the A and B in $X \; {}_A||_B \; Y$. Finally, we have the SrcSpan pointer.

For sharing, we have the left and right side of the interface parallel operator (X and Y respectively) and the shared channel list CList, which corresponds to A in X $\underset{A}{||}$ Y. Again, we have the SrcSpan pointer.

Finally, for ||| (interleaving) and `[]` (external choice) we have the left and right sides of the operator (X and Y respectively) and the Span pointer.

Also note there is no internal choice in these rules. This is because there are no timed rules for internal choice (due to the choice being urgent) and so without the timed firing rules, we also don't require corresponding auxiliary rules.

Each of these rules attempts to recursively perform find_d on the left and right sides of the binary operators. If a d is found on both sides of the operator, they are compared and the smaller of the two becomes the overall time limit. If only one of the sides returns a d, then that d automatically becomes the overall time limit. If neither side returns a d, the rule fails as no d was found. The d is passed through fractoint or rat before being returned.

Figure 8 shows the next rules, which are for hiding and control flow operators.

```
find_d('\\'(Expr,_CList,_Span),DF) :-
find_d(Expr,DV),(fractoint(DV,DF) -> true ; rat(DV,DF)).

find_d(ehide(Expr,_CList,_Span),DF) :-
find_d(Expr,DV),(fractoint(DV,DF) -> true ; rat(DV,DF)).

find_d(';'(X,_Y,_SeqSpan),DF) :-
find_d(X,DV),(fractoint(DV,DF) -> true ; rat(DV,DF)).

find_d('/\\'(X,Y,_Span),DF) :-
(find_d(X,D),find_d(Y,E) ->  rat(D=<E,DV) ;
(find_d(X,D) -> DV=D ;
(find_d(Y,E) -> DV=E ; fail))),
(fractoint(DV,DF) -> true ; rat(DV,DF)).
```

Figure 8. find_d rules for hiding and control flow operators.

The \\ (hiding) has Expr and CList, which correspond to the process expression Q and list of hidden channels A in the CSP operator Q \ A. We also have the Span pointer. Since there is only one side to this expression, we only need to find d in Expr. Like the aParallel and sharing operators, hiding has a corresponding operator *ehide*, which is its extended version.

The ; (sequential composition) operator has a left side X and a right side Y. While sequential composition is a binary operator, it represents a passing of control and so time only affects the left side of the expression (as the right side doesn't start until the left side has terminated). As such, like hiding, only one expression (the left side P) is analysed and updated.

The untimed interrupt follows the same format as the concurrent and choice operators.

Finally we have agent_call:

```
find_d(agent_call(_Span,F,Par),DF) :-
unfold_function_call_once(F,Par,Value),find_d(Value,DF).
```

Figure 9. find_d rule for agent_call operator.

This construct is specific to ProB and allows process expressions to be encapsulated and called. For example, consider following CSP code:

$$\text{MAIN} = \text{SKIP} \;|||\; \text{OTHER}$$
$$\text{OTHER} = \text{WAIT } 5 \;;\; \text{WAIT } 8$$

The agent_call rule will encapsulate WAIT 5; WAIT 8 in the expression OTHER. This construct is also used in the first state of execution (MAIN). As such, we have included this in the auxiliary rules and timed firing rules to allow ProB to function correctly.

Calculating the Resultant State – 'calculate_res'

Our second auxiliary rule calculates the resultant state of a firing rule. Like find_d, it works recursively, but instead of searching, it actually updates all the necessary time limits d in a nested state. In Figure 10 we have timeout's calculate_res rules.

```
calculate_res('[d>'(X,D,Y,Span),DV,Res) :-
(calculate_res(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res2(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF)),
rat(D-DV,DF),
(fractoint(DF,DFW) -> true ; DFW=DF),
Res='[d>'(X1,DFW,Y1,Span).

calculate_res2('[d>'(X,D,Y,Span),DV,Res) :-
(calculate_res2(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res2(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF)),
rat(D,DF),
(fractoint(DF,DFW) -> true ; DF=DFW),
Res='[d>'(X1,DFW,Y1,Span).
```

Figure 10. calculate_res rules for timeout.

For timeout, calculate_res subtracts a value from every time limit d on the left of the state, using the rational rule for subtraction – here it is called as rat(D-DV,DF). All d on the right side of the state remain the same and are just simplified as necessary (using rat(D.DF) in calculate_res2). If the calculate_res rules cannot be performed for either side of the operator, then the timed transition cspm_ttrans is calculated instead.

The calculate_res rules for timed interrupt are of the same format.

The WAIT operator has the following calculate_res rules:

```
calculate_res(wait(D,Span),DV,Res) :-
rat(D-DV,DF),
(fractoint(DF,DFW) -> true ; DFW=DF),
Res=wait(DFW,Span).

calculate_res2(wait(D,Span),_DV,Res) :-
rat(D,DF),
(fractoint(DF,DFW) -> true ; DFW=DF),
Res=wait(DFW,Span).
```

Figure 11. calculate_res rules for WAIT.

Like with its find_d rule, there is no nesting here and so only one time limit d needs to be updated. The first rule is if the wait operator is on the left side of a state, the second rule is for when it is on the right.

In **Appendix 4**, we have the calculate_res rules for choice and concurrency.

These binary operators simply perform calculate_res (or if it fails, they calculate the timed transition) of both the left expression X and the right expression Y. Again, aParallel and sharing have duplicate rules for their extended versions, eaParallel and esharing.

In **Appendix 5**, we have the other calculate_res rules (for hiding, control flow and ProB's built in agent_call operator).

The \\ (hiding) operator simply calls calculate_res for *Expr* (the Q in Q\A). Failing that, it performs its timed transition. The important addition here is the line:

```
(cspm_trans('\\'(Expr,CList,Span),tau(hide(_ActionX)),_Res ,WF) -> fail
```

This implies that if there is an available action that is hidden (in the set A of Q\A), then the τ transition must be invoked before time can pass, so the result cannot be calculated.

Again, like with find_d, \\ has a duplicate calculate_res rule for its extended version, *ehide*.

The ; (sequential composition) operator has a similar format to \\, except the rule has the following condition:

```
(cspm_trans(';'(P,Q,SeqSpan),tau(tick(_)),_Res,_WF) -> fail ;
```

This means that if P is ready to terminate and there is a ✓ event available, then the state must do a τ transition to Q before time can resume and so calculate_res fails.

Finally we have the untimed interrupt (which follows the same format as the concurrency and choice rules) and agent_call (the built in ProB operator).

10.1.3 Firing Rules

As discussed in **7.2.1**, the CIA already contains all the untimed firing rules. These firing rules are of the form cspm_trans(e,a,e′,wf), where e is the current state, a is the action, e′ is the resultant state and wf is ProB's waitflag predicate. For our timed firing rules, we have simply added a new type of transition; cspm_ttrans(e,a,e′,wf). The rules for cspm_trans and cspm_ttrans are clearly segregated, so there is no risk of confusion despite the similarity in nomenclature.

Before we discuss the firing rules, there are three types of timed event in our implementation which need to be explained.

- Evolution Transition $\lfloor 0,\infty \rfloor$ – This transition has one variable, Span/SrcSpan, which is simply a pointer in the execution. This timed event is presented when the state only contains untimed CSP operators and allows any amount of time to pass. When selected by the end-user, the option to enter any value within the interval $\lfloor 0,\infty \rfloor$ is presented to the user. There is no change to the state itself, but the global time is updated with the additional time. It takes the form time(Span) in the firing rules.

- Evolution Transition DV – This transition has two variables, the overall time limit dv of the state and Span/SrcSpan. This timed event is one of two presented when the state contains timed CSP operators. When this event is selected, time dv passes. This

will cause the time limit DV and all d = dv in the state to change to 0, resulting in a τ transition becoming available (and time cannot pass until this transition is invoked). All other $d' >$ DV will update to d'- DV. It takes the form time(DV,Span) in the firing rules.

- Evolution Transition ⌊0,DV) – This transition has three variables, the overall time limit DV of the state, one time limit E which has been substituted for a non-ground variable in the state and is stored in the transition, and Span/SrcSpan. When this event is selected, the user is presented with the option to enter any value within the interval ⌊0,DV). The non-ground variable will be replaced with the original time limit E (with time subtracted, where time is the value entered by the user). All other time limits d′ in the state will be updated to d′-time. It takes the form time(DV,E,Span) in the firing rules.

Timed Firing Rules for Untimed CSP

Our first timed firing rules are for the untimed CSP operators in a timed context. The three simplest firing rules we discussed in **3.2** were STOP, SKIP and event prefix. These operators are unaffected by a timed transition and the only differentiation of state is through the change in global time (the implementation for global time is within the simulator and is discussed further on in the chapter). The firing rules are below:

$$\frac{}{\text{STOP} \overset{\text{d}}{\rightsquigarrow} \text{STOP}} \qquad \frac{}{\text{SKIP} \overset{\text{d}}{\rightsquigarrow} \text{SKIP}}$$

$$\frac{}{(\text{a} \rightarrow \text{P}) \overset{\text{d}}{\rightsquigarrow} (\text{a} \rightarrow \text{P})}$$

These firing rules correspond directly to our implementation:

```
cspm_ttrans(stop(SrcSpan),time(SrcSpan),stop(SrcSpan),_).

cspm_ttrans(skip(SrcSpan),time(SrcSpan),skip(SrcSpan),_).

cspm_ttrans(prefix(SPAN1,Values,ChannelExpr,CSP,SPAN2),time(SPAN1),
            prefix(SPAN1,Values,ChannelExpr,CSP,SPAN2), _).
```

Figure 12. Timed firing rules for STOP, SKIP and event prefix.

Here we see that stop, skip and event prefix states all have a timed event which leads to the same state. We also have a slot for the waitflag predicate. As these are states consisting of only untimed CSP operators, we have our 1-variable timed event, time(SrcSpan).

The above firing rules are followed by the timed firing rules for the choice operators. Recall from Part II that internal choice has an urgent event, which means it does not have any timed transition. Therefore we have only implemented a timed firing rule for external choice:

```
cspm_ttrans('[]'(X,Y,Span),A,Res,_) :-
(find_d('[]'(X,Y,Span),DV) ->
((DV == 0 ; (calculate_res('[]'(X,Y,Span),DV,Res) -> fail ; true)) -> fail ;
(
        (A=time(DV,Span),calculate_res('[]'(X,Y,Span),DV,Res)) ;

        (A=time(DV,E,Span),(cspm_ttrans(X,time(_D,E,_Span),X1,WF) -> Y1=Y ;
        cspm_ttrans(Y,time(_D,E,_Span),Y1,WF),X1=X),Res='[]'(X1,Y1,Span))
)) ;
cspm_ttrans(X,_AX,X1,WF),cspm_ttrans(Y,_AY,Y1,WF),A=time(Span),Res='[]'(X1,Y1,Span)).
```

Figure 13. Timed firing rule for choice.

Breaking down the above rule, we first have the head. Here we see that the timed transition starts with an external transition state. The action A is defined within the body of the rule as is the resultant state Res.

We start the body with a condition which checks whether or not our auxiliary find_d rule succeeds.

If find_d fails, we can assume that neither side of the choice operator contains any timed operators, which means that any amount of time can pass. As such, we simply compute the timed transitions for the X choice and the Y choice; setting the A as our 1-variable timed transition and the Result as a new choice operator state with an updated X and Y.

If find_d succeeds, we know that the choice operator contains timed operators. First we check whether the overall time limit DV is 0. If it is, that means time cannot pass until a τ transition is carried out and the timed transition fails.

We also test our second auxiliary rule, 'calculate_res'. If the rule fails at this point, it means that one of the expressions in the state is engaged in a τ transition and time cannot pass (for example, an action $a \in A$ is available which is hidden in the operator $Q \setminus A$). While time(DV,Span) would automatically fail if this rule failed, time(DV,E,Span) would still succeed as it doesn't use calculate_res. This is why we need to add this rule to the body as an additional condition.

If both these conditions pass, then time(DV,Span) and time(DV,E,Span) become available.

The time(DV,Span) event has a result Res which is a new state with time d subtracted from all relevant timed limits in the state (left of timed operators, hiding and sequential composition; either side of untimed binary operators). All relevant d = DV will become 0, while all other relevant $d' > d$ in the state will become d' - d. The calculate_res rule takes care of this.

The other timed transition time(DV,E,Span) has a result Res which is a new state with one of the time limits E changed to a non-ground variable. The time limit E is the first limit found in the state, first looking on the left side of the expression, then the right. The actual calculations to determine the new state are not made until after the user has entered a time. The reason for changing one of the time limits to a non-ground variable is to differentiate the state. If the resultant state was the same as the initial state, ProB has some optimisation features which would circumvent the state change process and we wouldn't be able to update the time limits after user-input. The semantics would also be incorrect.

The overall result of our firing rule is that as time passes, both sides of the choice update, meaning the overall choice operator has updated over time:

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad Q \overset{d}{\rightsquigarrow} Q'}{P \,\Box\, Q \overset{d}{\rightsquigarrow} P' \,\Box\, Q'}$$

The concurrency firing rules follow the exact same format as the choice firing rule.

Next, we have the hiding operator. Below is the implemented firing rule:

```
cspm_ttrans('\\'(Expr,CList,Span),A,Res,_) :-
(cspm_trans('\\'(Expr,CList,Span),tau(hide(_ActionX)),_HideRes ,WF) -> fail ;
(find_d('\\'(Expr,CList,Span),DV) ->
((DV == 0 ; (calculate_res('\\'(Expr,CList,Span),DV,Res) -> fail ; true)) -> fail ;
(
        (A=time(DV,Span),calculate_res('\\'(Expr,CList,Span),DV,Res)) ;

        (A=time(DV,E,Span),cspm_ttrans(Expr,time(_D,E,_Span),NExpr,WF),
        Res='\\'(NExpr,CList,Span))
)) ;
cspm_ttrans(Expr,_AExpr,NExpr,WF),A=time(Span),Res='\\'(NExpr,CList,Span))).
```

Figure 14. Timed firing rule for hiding.

As in calculate_res, the body contains a new condition which says that if an untimed transition τ exists which hides an action, the timed transition fails. In other words, if there's an action available which is in the hidden set A of $Q \setminus A$, then time cannot pass.

If there is no such hidden action at this step in the execution, then we test again for find_d, $DV = 0$ and calculate_res.

The other key difference is that hiding is not a binary expression like the concurrency and choice operators and so we only need to analyse and modify Expr. This is also reflected in the calculate_res and find_d rules for hiding. The rule for ehide is the same as for $\backslash\backslash$.

The implementation corresponds directly with the timed hiding firing rules:

$$\frac{P \overset{d}{\rightsquigarrow} P' \qquad \forall a \in A \mid \neg(P \overset{a}{\rightarrow})}{P \setminus A \overset{d}{\rightsquigarrow} P' \setminus A}$$

Below is the firing rule implementation of sequential composition:

```
cspm_ttrans(';'(P,Q,SeqSpan),A,Res,_) :-
(cspm_trans(';'(P,Q,SeqSpan),tau(tick(_)),_Res,WF) -> fail ;
(find_d(P,DV) ->
((DV == 0 ; (calculate_res(';'(P,Q,SeqSpan),DV,Res) -> fail ; true)) -> fail ;
(
        (A=time(DV,SeqSpan), calculate_res(';'(P,Q,SeqSpan),DV,Res)) ;

        (A=time(DV,E,SeqSpan), cspm_ttrans(P,time(_D,E,_Span),P1,WF),Res=';'(P1,Q,SeqSpan))
)) ;
cspm_ttrans(P,_AP,P1,WF),A=time(SeqSpan),Res=';'(P1,Q,SeqSpan))).
```

Figure 15. Timed firing rule for sequential composition.

The sequential composition firing rule is similar to that of hiding. While sequential composition is a binary operator, it represents a passing of control and so time only affects the left side of the expression (as the right side doesn't start until the left side has terminated). As such, like hiding, only the left expression (P) is analysed and updated. Again, this is represented in find_d and calculate_res as well. There is also a condition in the body which states that if the P is ready to terminate, it is urgent, meaning time cannot pass until it has terminated. Incidentally, the timed transition will fail if this condition is met.

Again, this corresponds with the timed firing rule in Steve Schneider's specification:

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad \neg(P \overset{\checkmark}{\rightarrow})}{P \,;\, A \overset{d}{\rightsquigarrow} P' \,;\, A}$$

This is followed by the untimed interrupt firing rule:

```
cspm_ttrans('/\\'(X,Y,Span),A,Res,_) :-
(find_d('/\\'(X,Y,Span),DV) ->
((DV == 0 ; (calculate_res('/\\'(X,Y,Span),DV,Res) -> fail ; true)) -> fail ;

(
        (A=time(DV,Span),calculate_res('/\\'(X,Y,Span),DV,Res)) ;

        (A=time(DV,E,Span),(cspm_ttrans(X,time(_D,E,_Span),X1,WF) -> Y1=Y ;
        cspm_ttrans(Y,time(_D,E,_Span),Y1,WF),X1=X),Res='/\\'(X1,Y1,Span))
)) ;
cspm_ttrans(X,_AX,X1,WF),cspm_ttrans(Y,_AY,Y1,WF),A=time(Span),Res='/\\'(X1,Y1,Span)).
```

Figure 16. Timed firing rule for untimed interrupt

Once again, this follows the same format as concurrency and choice and fits with the timed firing rule from our background research:

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad Q \overset{d}{\rightsquigarrow} Q'}{P \;\Delta\; Q \overset{d}{\rightsquigarrow} P' \;\Delta\; Q'}$$

Finally, we have timed firing rules for agent_call and val_of (like agent_call, val_of is related to ProB's optimisation):

```
cspm_ttrans(val_of(X,Span),A,NewExpr,WF) :- symbol(RenamedX,X,_,_),
   cspm_ttrans(agent_call(Span,RenamedX,[]),A,NewExpr,WF).

cspm_ttrans(agent_call(Span,F,Par),NA,NNewExpr,WF) :- !,
     unfold_function_call_once(F,Par,Value),
         ~~pp_cll(cspm_ttrans(Value,A,NewExpr,WF)),
           full_normalise_csp_process(NewExpr,NNewExpr),
           merge_span_into_event(A,Span,NA).
```

Figure 17. Timed firing rules for val_of and agent_call.

Timed Firing Rules for Timed CSP

For our implementation, we have covered four timed CSP operators. This includes timeout, timed interrupt and the syntactic sugar; delay and WAIT.

Below is the timed firing rule for timeout:

```
cspm_ttrans('[d>'(X,D,Y,Span),A,Res,_) :-
rat(D,RD),
(
    (A=time(DV,Span), find_d('[d>'(X,RD,Y,Span),DI), (fractoint(DI,DV) -> true ; rat(DI,DV)),
    (DV == 0 -> fail ; calculate_res('[d>'(X,RD,Y,Span),DV,Res))) ;

    (A=time(DV,RD,Span), find_d('[d>'(X,RD,Y,Span),DI),(fractoint(DI,DV) -> true ; rat(DI,DV)),
    (DV == 0 -> fail ; Res='[d>'(X,_E,Y,Span)))
).
```

Figure 18. Timed firing rule for timeout.

First, the time limit d is converted into a fraction or simplified if necessary, using our rational number framework. We are then presented with our two timed transitions time(DV,Span) and time(DV,RD,Span), where RD is our d after it is converted into a fraction or simplified. Like the untimed operators, we can use find_d and calculate_res to work with nested operators. As mentioned in **10.1.2**, these rules update only the left side of a nested timeout expression, following the $\langle \triangleright$-associative $\rangle$ law discussed in **7.2**:

$$P \overset{d}{\triangleright} (Q \overset{d'}{\triangleright} R) = (P \overset{d}{\triangleright} Q) \overset{d+d'}{\triangleright} R.$$

The time(DV,Span) event updates all time relevant limits d = DV to 0 and all other relevant d' > DV in the state to d' - DV.

The time(DV,E,Span) event substitutes the d at the center of the expression (the non-nested d) with a non-ground variable and stores the d in E. Once time is determined during simulation, the non-ground d is then replaced by E-time (where time $\in$ [0,DV) is input by the user).

Below we have the the timed firing rule for timed interrupt, which is the same as timeout:

```
cspm_ttrans('/d\\'(X,D,Y,Span),A,Res,_) :-
rat(D,RD),
(
   (A=time(DV,Span), find_d('/d\\'(X,RD,Y,Span),DI), (fractoint(DI,DV) -> true ; rat(DI,DV)),
   (DV == 0 -> fail ; calculate_res('/d\\'(X,RD,Y,Span),DV,Res))) ;
   (A=time(DV,RD,Span), find_d('/d\\>'(X,RD,Y,Span),DI),(fractoint(DI,DV) -> true ; rat(DI,DV)),
   (DV == 0 -> fail ; Res='/d\\'(X,_E,Y,Span)))
).
```

Figure 19. Timed firing rule for timed interrupt.

Alongside our two main timed operators, we have the syntactic sugar; delay and WAIT.

In Figure 20, we see that the timed firing rule for delay is the same as event prefix.

```
cspm_ttrans('-d->'(SPAN1,Values,ChannelExpr,D,CSP,SPAN2),time(SPAN1),
            '-d->'(SPAN1,Values,ChannelExpr,D,CSP,SPAN2), _).
```

Figure 20. Timed firing rule for delay.

As discussed in **3.6.1**, a $\xrightarrow{d}$ P = a $\longrightarrow$ (STOP $\overset{d}{\triangleright}$ P) and so the state is essentially an event prefix (leading to a state using a timeout operator). As such, the delay operator and event prefix share the same timed transition format.

The timed firing rule for WAIT makes use of our existing timeout firing rules:

```
cspm_ttrans(wait(D,SrcSpan),A,Res,_) :-
(D == 0 -> fail ;
(
      (A=time(DV,SrcSpan),
      cspm_ttrans('[d>'(stop(SrcSpan),D,skip(SrcSpan),Span),time(DV,Span),
      '[d>'(stop(SrcSpan),0,skip(SrcSpan),Span),_),
      Res=wait(0,SrcSpan)) ;

      (A=time(DV,RD,SrcSpan),
      cspm_ttrans('[d>'(stop(SrcSpan),D,skip(SrcSpan),Span),time(DV,RD,Span),
      '[d>'(stop(SrcSpan),_E,skip(SrcSpan),Span),_),
      Res=wait(_E,SrcSpan))
)).
```

Figure 21. Timed firing rule for WAIT.

Here, we see that WAIT d = STOP $\overset{d}{\triangleright}$ SKIP is directly implemented. We have the transition time(DV,SrcSpan), which results in WAIT 0 and is determined by performing the timed firing rule for timeout with time(DV,Span), where X is STOP and Y is SKIP. Similarly, we have the transition time(DV,RD,SrcSpan), which results in a WAIT with non-ground variable E (which is replaced by RD-time after time is provided by the user). It is represented by the timed firing rule for timeout with time(DV,RD,Span), where X is STOP and Y is SKIP.

Untimed Firing Rules for Timed CSP

Finally, we have the untimed firing rules for our new timed CSP operators. Below are the untimed firing rules for timeout and timed interrupt:

```
/* Timeout */
cspm_trans('[d>'(X,D,Y,Span),AS,Res,WF) :- cspm_trans(X,A,X1,WF),
 ((A=tau(_), Res='[d>'(X1,D,Y,Span))  ;  (top_level_dif(A,tau(_)), Res=X1)),
 shift_span_for_left_branch(Span,LSpan),
 merge_span_into_event(A,LSpan,AS).
cspm_trans('[d>'(_X,0,Y,Span),tau(timeout(Span)),Y,_WF).

/* Timed Interrupt */
cspm_trans('/d\\'(X,D,Y,Span),AS,Res,WF) :- cspm_trans(X,A,X1,WF),
  (A=tick(_) -> Res=X1 ; Res='/d\\'(X1,D,Y,Span)),
  shift_span_for_left_branch(Span,LSpan),
  merge_span_into_event(A,LSpan,AS).
cspm_trans('/d\\'(_X,0,Y,Span),tau(tInterrupt(Span)),Y,_WF).
```

Figure 22. Untimed firing rules for timeout and timed interrupt.

For timeout, we say that if the executed action is a τ transition, then the X updates, but the timeout state remains intact. Any other type of action made by X results in the entire state doing a transition to X. The shift_span_for_left_branch and merge_span_into_event rules are for updating the Span pointer. Finally, if D = 0, then the state does a τ transition to Y.

Timed interrupt follows a similar format, however only a $\checkmark$ action leads to a transition of the entire state to X. Otherwise, the X updates but the timed interrupt operator remains intact. Again, if D is 0, the state does a τ transition to Y.

The next firing rule is for delay:

```
cspm_trans('-d->'(SPAN1,Values,ChannelExpr,D,CSP,SPAN2), io(EV,Channel,SPAN),
           '[d>'(stop(SPAN2),D,NormCSP,SPAN), WF) :-
   evaluate_channel_outputs(Values,ChannelExpr,EV,Channel,SPAN,WF),
   unify_spans(SPAN1,SPAN2,SPAN),
   full_normalise_csp_process(CSP,NormCSP).
```

Figure 23. Untimed firing rule for delay.

The firing rule for delay has a similar format to the firing rule for event prefix. CSP is equivalent to the P in a $\xrightarrow{d}$ P and NormCSP is the P after performing a. While the result for event prefix would simply be NormCSP, we've embedded this into a timeout expression with STOP as the left side and P as the right. This mirrors the following rule:

$$a \xrightarrow{d} P = a \rightarrow (STOP \overset{d}{\triangleright} P)$$

Finally, we have the untimed firing rule for the WAIT operator:

```
cspm_trans(wait(0,SrcSpan),tau(timeout(SrcSpan)),skip(SrcSpan),_).
```

Figure 24. Untimed firing rule for WAIT.

The only possible untimed action for the WAIT operator is to do a τ timeout transition to a SKIP state when D = 0.

10.1.4 Other Additions

Our other main addition is update_result, which can be found in **Appendix 6**. This rule is called by the simulator once a time has been acquired from the user. It is used to replace the non-ground variable with its original time limit *d* and then it uses calculate_res to subtract the time (input by the user) from all the relevant time limits in the state.

While the rule looks complex, its implication is quite simple. If the state is an untimed operator (other than stop, skip or event prefix) it performs update_result on both the left and right sides of the expression (or only the left if it's the hiding or sequential composition operators). If the state is one of the timed CSP operators it checks to see if its time limit *d* is ground, if it isn't, it substitutes it for the RD stored in the timed transition time(DV,RD,Span). It then performs calculate_res to subtract the time obtained from the user. If the state is stop, skip or event prefix, the state remains unchanged. The entire update state is then produced as the result NewState and returned to the simulator.

10.2 tcltk_interface.pl

The simulator has been updated in a number of ways. First, we've extended the state update to use update result from haskell csp.pl for the timed transitions of type time(DV,RD,Span). We have also included some extra rules which take time data from the GUI, based on user-input, and calculate the global time using a new list clause defined in state_space.pl. We have added to the simulator a rule that enforces the constraints of transitions. So if a user enters a time > DV for Evolution Transition [0,DV), the rule will fail and the user can enter another time that fits within the constraints of the transition. Finally, we have extended the backtrack and forward functions to account for the timed transitions.

10.2.1 State Substitution

In **Appendix 7**, after the next state is determined; we have inserted code that calls update_result from haskell_csp.pl. This grounds and updates the state using the time input from the user (if necessary). This occurs before the next state is called as the current expression. First, we extract the executed action's data and determine that it's a timed event of type time(DV,RD,Span).

If it is, we call update_result from haskell_csp.pl (this rule is imported at the top of the code extract). We then replace the state in the visited expression clause (this is necessary for backtrack to work). If it isn't a timed event of type time(DV,RD,Span), then the state is called as the current state without any modifications.

10.2.2 Global Time & Constraints

Appendix 8 shows the code which calculates the global time and enforces the constraints on user input.

In the original ProB implementation, when an action was selected and performed, tcltk_perform would be called. In our extended implementation, we now call tcltk_timer instead. This fulfills two functions:

- First it ensures the time entered by the user meets the interval constraints (greater than 0 for the Evolution Transition [0,∞) operation and between 0 and d for the Evolution Transition [0,d) operation). It uses the rule tcltk_constraints to achieve this.
- Second, it adds the time entered by the user to a new list clause called timedata (which has been set up in state_space.pl). This list clause stores all the times entered by a user throughout the execution.

Only after tcltk_timer has called both these rules will tcltk_perform be called.

The rule tcltk_get_time is called by the GUI's state window at every step in the execution. As the name suggests, it adds up all the time values in the list clause, timedata (using the rule list_sum) and returns it to the GUI.

The rule tcltk_intialise sets up the timedata clauses at the start of a simulation (otherwise tcltk_timer would be met with an existence error when it tried to call the timedata clause for the first action). It is only called for the first action of the simulation. It also clears anything stored in timedata from a previous simulation.

At the top of the code, we see that the rational number framework is imported from haskell_csp.pl in order to perform the fraction addition for global time.

10.2.3 Backtrack and Forward

Finally, we have added some extra code in backtrack and forward to account for the extension of time. The code extracts for these are in **Appendix 9**.

The additional code in backtrack removes the latest time value and stores it in a second timedata clause (which can be considered the backtrack timedata clause). It also retracts the latest transition.

The additional code in forward does the reverse, removing the latest time value from the second timedata clause (the backtrack timedata clause) and putting it back into the main timedata clause).

10.3 Other Prolog Modules

Alongside our extension of the simulator and haskell_csp.pl we've made some minor additions to the following modules.

10.3.1 haskell_csp_analyser.pl

This module is for identifying the standard operators in the generated Prolog code and we have simply added the new rules for identifying our new timed operators. They are as follows:

```
definite_cspm_process_construct('[d>'(A,D,B,Span),[Span],[D],[A,B]).
definite_cspm_process_construct('/d\\'(A,D,B,Span),[Span],[D],[A,B]).
definite_cspm_process_construct('-d->'(Span,_A,_B,D,C,Span2),[Span,Span2],[_A,_B,D],[C]).
definite_cspm_process_construct(wait(D,Span),[Span],[D],[]).
```

Figure 25. Timed construct identification.

This code simply breaks down the operators into their constituent parts and determines the properties of each part (separating spans, CSP expressions and standard variables (such as rational numbers).

10.3.2 xtl_interface.pl

This module initialises the CSP system and also acts as an interface between haskell_csp.pl and specfile.pl (which is used by the simulator to determine available events).

```
cspm_transition(State,Time,NewState) :- State \= root,
   %print(comp),nl,
   cspm_ttrans_enum(State,Time,NewState).
   %print(new(NewState)),nl.
```

Figure 26. cspm_ttrans_enum call.

There is some simple code in xtl_interface.pl which calls cspm_trans_enum from haskell_csp.pl before it is called by specfile.pl during the listing of available options by the

simulator. Since we added cspm_ttrans_enum to haskell_csp.pl, we have duplicated the code in xtl_interface.pl to call it.

10.3.3 translate.pl

This module converts events and processes into strings for GUI output.

```
translate_event2(time(_Span),['Evolution Transition [0..inf)'|T],T) :-
process_algebra_mode, !.        /* CSP */

translate_event2(time(D,_Span),['Evolution Transition ',D|T],T) :-
process_algebra_mode, !.        /* CSP */

translate_event2(time(D,_,_Span),['Evolution Transition [0..',D,')'|T],T) :-
process_algebra_mode, !.        /* CSP */
```

Figure 27. Translate rules for timed transitions.

We have added rules to translate the 1, 2 and 3-variable time transitions into their respective strings for the available operations window in the GUI. We can see that time(_Span) results in Evolution Transition [0..inf), time(D,_Span) results in Evolution Transition D and time(D,_E,_Span) results in Evolution Transition [0..D).

```
binary_csp_op('[d>'(X,D,Y,_Span),X,D,Y,'[d>').

binary_csp_op('/d\\'(X,D,Y,_Span),X,D,Y,'/d\\').

pp_csp_process(delay(_SPAN1,Values,ChannelExpr,D,CSP,_SPAN2),S,T) :- !,
  pp_csp_value_1([ChannelExpr|Values],'.',S,['-',D,'->'|S2],20),
  pp_csp_process(CSP,S2,T).

pp_csp_process(wait(D,_Span),S,T) :- !, S=['WAIT ',D|T].
```

Figure 28. Translate rules for timed constructs.

We have also added the four timed CSP operators to translate.pl. They can be converted to strings for the GUI state output.

10.3.4 state_space.pl

This module maintains the Prolog database of list clauses that contain important data on the current simulation. This includes history, available options, visited expressions and current state. These clauses are dynamic so that information can be updated and modified (through append and retract methods) throughout the simulation. We have added a new list clause which stores time data. It is used by the simulator to determine the global time (and its latest stored valued is used by update result to update the time limits across the state). The new list clause is set up in state_space.pl as follows:

```
:- dynamic timedata/2.
timedata(v_0,[]).
```

Figure 29. timedata list clause.

11 Parser & GUI Implementation

As well as the extension of the simulator and CIA in Prolog, we have added the timed CSP syntax to the Haskell parser. We have also made additions to the GUI in Tcl/Tk to allow for time input from the user and some general aesthetic changes.

11.1 Haskell Parser

ProB's parser, coded in Haskell, reads in the CSP code provided by the user. It then generates the corresponding Prolog code which is interpreted by the CIA component of ProB. As part of our implementation, we have added the timed CSP syntax to the parser so that it is recognised when included in user input.

11.1.1 AST (Abstract Syntax Tree)

We first added the new operators to the abstract syntax tree. This is shown in **Appendix 10**.

The four timed operators have been added to the AST (marked with comments). The WAIT takes in an expression (the time limit d). Timeout takes an expression (the time limit d) and two processes (X and Y). Timed interrupt is the same. Delay takes in the same data as event prefix, but also takes in an expression (delay d). While the expressions that all these operators take in are of the type LExp (which encompasses the entire AST), their d is actually only allowed to be an integer or a rational number. The code which specifies this is in AstToProlog.hs, which is laid out in **11.1.3**.

11.1.2 Main Parser

Parser.hs contains the main parser, where we have added our timed operators to the precedence table (**Appendix 11**) and created the methods for actually parsing them.

We have placed the WAIT operator at the bottom of the table. Timed Interrupt and Timeout have been placed with their respective untimed operators in order of precedence.

```
{- Replicated Expressions in Prefix form -}

parseProcReplicatedExp :: PT LProc
parseProcReplicatedExp
  = choice
    [
      procRep T_semicolon   ProcRepSequence
    , procRep T_sqcap ProcRepInternalChoice
    , procRep T_box   ProcRepExternalChoice
    , procRep T_interleave ProcRepInterleave
    , procRepAParallel
    , procRepLinkParallel
    , procRepSharing

  , try parseDelayExp -- Delay
  , parsePrefixExp
```

Figure 30. Delay placement.

Event prefix only appears in the method for replicated expressions, so we have added our delay here. A try clause has been added so that if a delay expression is not found, prefix expression is checked for instead.

Each operator has a method in Parser.hs that describes how it is detected. The one exception is the wait operator, which does not need a method as it simply detects the WAIT token and reads in the integer or rational number which follows it. In **Appendix 12** are the methods which parse the other three timed CSP operators.

Timeout looks for two tokens, T_timeOpenBrack and T_timeCloseBrack. These tokens are identified as `[{` and `}>` in another the parser's lexer. If it finds these two tokens, it reads in the expression between them as the time. This data is then sent to the AST. Timed interrupt has a similar process, except that it looks for two tokens `/{` and `}\`.

Delay identifies the channels, then looks for two tokens which make up the delay arrow, namely `-{` and `}->`. It then reads the expression between them as the delay time. This is sent to the AST.

Note that all these methods refer to positions (spos, getNextPos, etc.). This is basically the position of the parsed expression in the CSP code and is used for highlighting columns and rows by the GUI when certain actions are selected.

11.1.3 AstToProlog

This file converts any parsed CSP code into the Prolog code ready to be interpreted by the CIA portion of ProB. Below we have the rules for converting our four operators:

```
    TimedTimeout time p1 p2 -> nTerm "[d>" [te p1, timeexpr time, te p2,plLoc expr]
    TimedInterrupt time p1 p2 -> nTerm "/d\\" [te p1, timeexpr time, te p2,plLoc expr]

DelayExp ch fields time proc ->
nTerm "-d->" [plLoc ch, mkCommFields fields, te ch, timeexpr time, te proc,prefixLoc ]
  where
    prefixLoc = mkSrcLoc $ SrcLoc.srcLocBetween
      (if null fields then srcLoc $ ch else srcLoc $ last fields)
      (srcLoc proc)


        Wait time -> nTerm "wait" [timeexpr time,plLoc expr]
```

Figure 31. AstToProlog codes for timed operators.

The resultant Prolog code corresponds directly with the operator definitions in haskell_csp_analyser.pl which identifies the Prolog versions of the CSP operators.

Note that the value *time* is specified as a *timeexpr* type, which is defined in Figure 32.

```
timeexpr :: LExp -> Term
timeexpr expr = case unLabel expr of
  IntExp i -> term $ atom i
  Fun2 op a b -> case (unBuiltIn op, unLabel a, unLabel b) of
    (F_Div, IntExp i, IntExp j) -> Term ((unAtom (atom i)) <> char '/' <> (unAtom (atom j)))
    _ -> error "Expected a time"
  _ -> error "Expected a time"
```

Figure 32. timeexpr definition.

This ensures that the expressions used to represent d in a timed operator are either integers or rational numbers (or more precisely in this context, a division function with two integer values as input). Otherwise, an error is returned; "Expected a time".

11.1.4 Lexer.x

Lexer.x contains all the token definitions which are used by the parser. We have added the following tokens for our four operators:

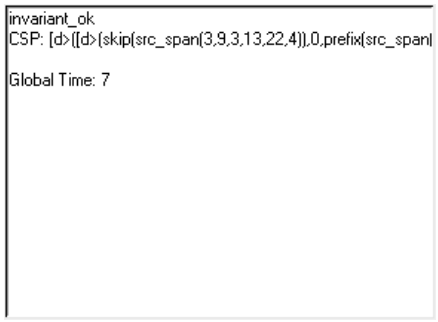
```
<0> "[{" { mkL T_timeOpenBrack }
<0> "}>" { mkL T_timeCloseBrack }
<0> "/{" { mkL T_interruptOpenBrack }
<0> "}\" { mkL T_interruptCloseBrack }

<0> "-{" { mkL T_delayOpenArrow }
<0> "}->" { mkL T_delayCloseArrow }

    <0> "WAIT"    { mkL T_wait }
```

Figure 33. Timed operator tokens.

First we have the timeout tokens, which combined make `[{}>`.
We then have the timed interrupt tokens, which make `/{}\`.
This is followed by the delay tokens, which make `-{}->`.
Finally we have the WAIT token.

11.2 GUI (Graphical User Interface)

The GUI has been updated in two places. The first is the state window, which has been updated to call the tcltk_get_time rule and display a global time at every step in the execution:




```
if [prolog {tcltk_get_time(Time)}] {
    set time $prolog_variables(Time)
    .frmInfo.frmState.list insert end ""
    .frmInfo.frmState.list insert end [concat "Global Time: " $time]
}
```

Figure 34. Global time GUI code and output.

Also, we have added extra code which allows for the user to input time. This is shown in **Appendix 13**.

In **Appendix 13** we have the process which determines what happens when a user double clicks an operation. Originally this simply called the Prolog tcltk_perform rule (which updates the next available options, updates the trace, etc.) with the current option ID as a parameter (where 1 is first in the list, 2 is second, and so on).

A number of additions have been made here to account for the timed transitions. A case for start_cspm_MAIN (the initial action available when a CSP file is first loaded) has been

added, which calls the tcltk_initialise rule instead of tcltk_perform. We have also added cases for the timed transitions. When Evolution Transition [0,d) or Evolution Transition [0,∞) is double clicked, a dialog box pops up (shown in **Appendix 14**) asking for the transition time and maps the user input to a variable called evol. The tcltk_timer rule is then called with the current option ID and evol variable as parameters. When Evolution Transition d is clicked, the value d is mapped to evol and tcltk_timer is called. Note that for all other actions, tcltk timer is called but with the time/evol variable as 0 (as all other actions have no time value).

PART IV
Demonstration

IV Demonstration

Part IV provides the models used to demonstrate our timed CSP simulator's functionality. We first consider simple models using our new timed constructs; demonstrating that they behave as expected. We then use Steve Schneider's railway crossing model which clarifies the correctness of our implementation with respect to the specification laid out in his book as well as showing the applicability to the railway domain. Finally, we provide a simple model of a London Underground station; taking the first steps towards applying the simulator to the capability challenge.

12 Basic Models

We first use some basic models to demonstrate that our timed CSP simulator behaves as expected. These models not only test the operation of the regular timed CSP operators, but also clarify some of the laws laid out in **7.2**.

Our first model tests the WAIT operator and that it works correctly with hiding, event prefix and sequential composition. It tests the three laws for delay:

WAIT d; WAIT d′= WAIT(d + d′)	⟨ delay-sum1 ⟩
$a \xrightarrow{d} P = a \rightarrow$ WAIT d; P	⟨ delay-sum2 ⟩
(WAIT d; Q) \ A = WAIT d; (Q\A)	⟨ hide-delay ⟩

The model is below:

```
channel a, b

MAIN = a -> ((WAIT 5 ; WAIT 3 ; OTHER) \ {b})

OTHER = b -> SKIP
```

Figure 35. WAIT test model.

First, we have WAIT 5 ; WAIT 3. According to delay-sum1 this will be equivalent to WAIT(5+3) = WAIT 8.

We can follow from this and say that we have a -> WAIT(5+3); P, where P is OTHER.

Finally, we have (WAIT(5+3); Q) \ A, where Q is OTHER and A is {b}.

In other words, we can expect this CSP code to behave exactly the same as:

```
channel a, b
MAIN = a -{8}-> (OTHER\{b})
OTHER = b -> SKIP
```

This would perform the action *a*, followed by a timed evolution of 8 units and a timeout. This would be followed by a *b* action (which will be hidden and therefore stop time from passing) and ending with a tick operation.

On execution of our model, we see that precisely this trace is achieved. Not that the first action is on the bottom and the last action is on the top.

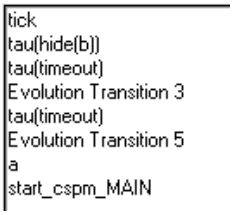


Figure 36. WAIT test trace.

We can test a further model for timeout, which tests the following rule:

$$P \overset{d}{\triangleright} (Q \overset{d'}{\triangleright} R) = (P \overset{d}{\triangleright} Q) \overset{d+d'}{\triangleright} R \qquad \langle \triangleright\text{-associative} \rangle$$

This is the law that played an important part in our implementation and resulted in recursive operations to work with nested states.

The model is below:

```
channel a, b

MAIN = (SKIP [{7}> a -> STOP) [{9}> b -> SKIP
```

Figure 37. Timeout test model.

Here, we expect the model to be equivalent to:

```
SKIP [{7}> (a -> STOP [{2}> b -> SKIP)
```

Purely from the timed transition point of view, this would mean a timed transition of 7, followed by a timeout and a timed transition of 2. We could then do a *b* action and finally a tick. This is clarified below:

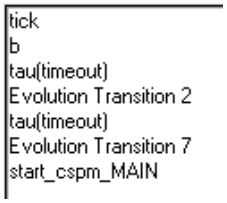


Figure 38. Timeout test trace.

The same can be done for timed interrupt, testing the rule:

$$P \,\Delta_d\, (Q \,\Delta_{d'}\, R) = (P \,\Delta_d\, Q) \,\Delta_{d+d'}\, R \qquad \langle\, \Delta_d\text{-associative} \,\rangle$$

Using the same model as before, but with the timed interrupt operator instead of timeout, we should be able to achieve the same trace.
The model is below:

```
channel a, b

MAIN = (SKIP /{7}\ a -> STOP) /{9}\ b -> SKIP
```

Figure 39. Timed interrupt test model.

Here, we expect the model to be equivalent to:

```
SKIP /{7}\ (a -> STOP /{2}\ b -> SKIP)
```

Again we achieve the expected trace and similar to timeout, but with tInterrupt tau transitions instead of timeout tau transitions:

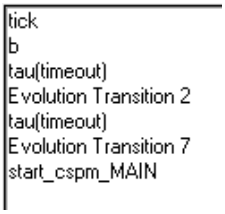


Figure 40. Timed interrupt test trace.

We can also use these models to demonstrate the timed CSP simulator's behaviour for interval transitions. Taking our timeout model again, we start with three available options; tick, Evolution Transition 7 and Evolution Transition [0..7). We also start with a global time of 0. If we pick the interval option and enter a time of 3, the global time should update to 3 and the Evolution Transitions should update to 4. This is clarified below:

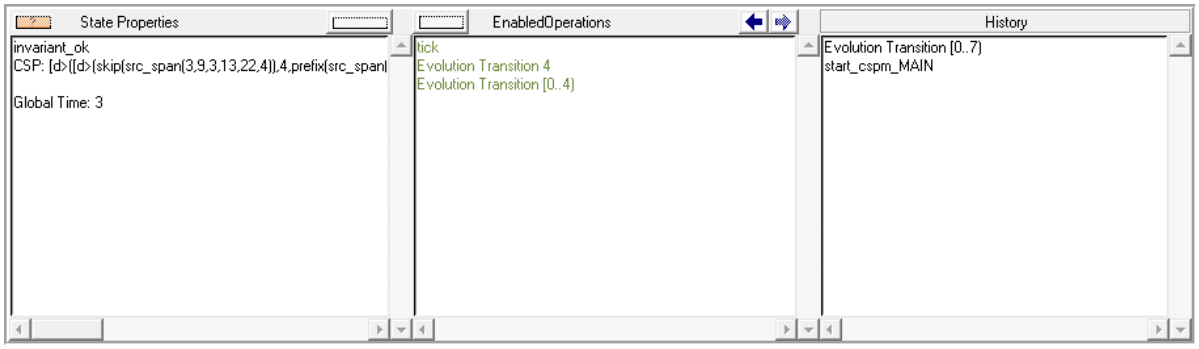


Figure 41. Testing interval transitions.

Also, the state has updated and the interval option has been added to the history. We can also achieve similar results for fractions as time values. Take the same example, but the user enters 1/3 instead. The result is expected to be a global time of 1/3 and updated evolution transitions of 20/3. This is clarified below:

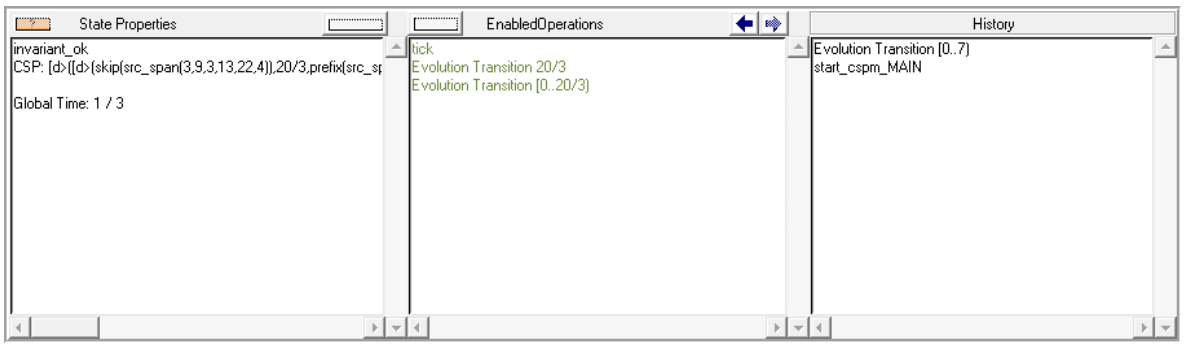


Figure 42. Testing interval transitions with fractions.

13 Railway Crossing

In order to identify our timed CSP simulator's applicability to the railway domain, we have decided to use a railway crossing model. Furthermore, we have adapted this model[24] from Steve Schneider's book, which allows us to also clarify that our implementation of timed CSP is correct with respect to the book's specification.

Below is a visual representation of the railway crossing model:

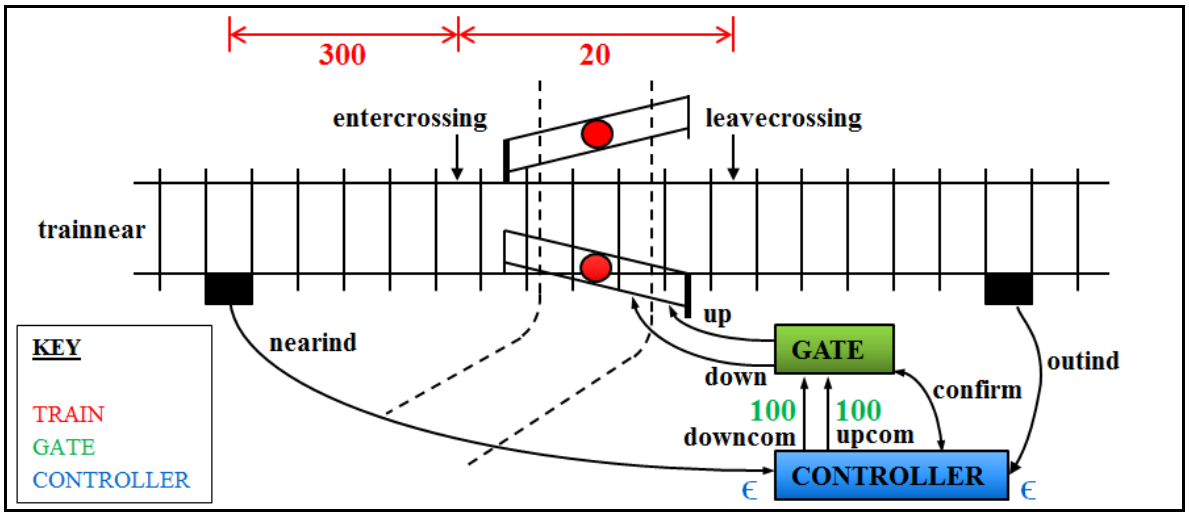


Figure 43. Railway crossing model.

This railway crossing model has 4 safety properties:

- First, if the train enters the crossing, the gate must have gone down more recently than it went up.
- Second, if the train enters the crossing at time t, then no down or up events should have occurred in the previous 10 time units.
- Third, if the gate goes up, the train must have left the crossing more recently than it entered it.
- Finally, the system must be deadlock-free.

The second safety property is of particular interest. Without time, we cannot fulfil this property. Therefore, this property highlights the importance of time in this model.

The model is represented as CSP code in **Appendix 15**. Note that the downcommand, upcommand, up, down and confirm actions have all been hidden. This is because they need to be urgent for the model to function correctly and meet the safety properties.

In order to show that our implementation is correct with respect to this model, we can simply compare the steps in the model and the simulation in **Appendix 16**.

First the train enters the system, represented by trainnear. It reaches the near indicator (signified by the nearind event) and at this point is 300 time units away from entering the crossing. This is received by the controller (in a negligible time ϵ), which initiates the downcommand. 100 time units pass between the downcommand event and the down event (which signifies that the gate is down). This is immediately followed by a confirm event between the gate and controller. As we expect, time has also passed for the train which is

running concurrently to the gate and controller. So there are now 200 time units remaining until the train enters the crossing. After 200 time units, the train enters the crossing. 20 time units pass and the train leaves the crossing. Once the train reaches the out indicator, the controller receives this information (in a negligible time ϵ) and initiates the upcommand. Also, at this point a new train can enter the system. After 100 time units, the gate is up (signified by the up event) and the gate and controller confirm with each other.

We can see in this trial run of the model, that all four of our safety properties were met.

When the train entered the crossing, the gate had gone down more recently than when it went up. Also, the gate had gone down more than 10 time units before the train entered the crossing (it went down at time t-200 to be precise, well within the limits of the safety property). Third, when the gate went up, the train had left the crossing more recently than it entered it. Finally, the system is deadlock-free as already confirmed by Steve Schneider ("the three component timed processes are all non-retracting and have finite interfaces") [24].

14 London Underground

Our final demonstration consists of a simple model of a London Underground station. The purpose of this model is to take the first steps towards applying our timed CSP simulator to the capability challenge as a subset of the capacity challenge.

The CSP code for our model is in **Appendix 17**.

In this model, trains run in parallel to the signals. When a train needs a green light, it waits until the green event for the correct track is available from the signal. Once it's available, the train and signal synchronise on the event and the train is able to continue travelling. The signal then waits for the train to leave a track, at which point they synchronise on a clear event for the relevant track number. The train process is effectively a continuous stream of trains running in interleave. These trains appear in the system every f seconds, where f is the frequency set by the end-user. In the figure, they run every 100 seconds. The entire system also has every action hidden; implying that all actions are urgent once they become available.

The behaviour of the train is partially based on Kirsten Winter's untimed railway model[23], discussed in chapter 8. The train has a front and a rear, the front moves on to a track, followed by the rear. The front of a train cannot enter a track until it is clear (the other train's rear has left the track). Finally, the front of a train cannot move to the next track if the rear of the train has not yet moved to the current track. The difference here is time is now a factor. For the first few steps of BehaveTrain, where delays are 7 seconds each, we represent a train moving at full speed. The delay of 20 seconds represents the deceleration of the train as it comes into the station. We then have a 30 second WAIT, which is the train waiting at the station while passengers board. We have another delay of 20 seconds representing the acceleration of the train as it leaves the station. Finally we have 7 second steps which is the train travelling at full speed again.

This CSP code can essentially be considered as a specific track plan. The end-user can then run the simulation for various frequencies f. If a train has to wait on the first track for a green light, then the trains are running too frequently. If there is a green light to move onto the second track and the next train is not yet waiting for a green light, then the trains are not running frequently enough.

While this model is relatively simple and far from being a real-world example, it is a definite first step towards applying the timed CSP simulator to the challenge of capability. The end-user can determine the optimum throughput for a track plan by performing a few test runs in the simulator, simply adjusting the frequency f each time.

Furthermore, if the simulator can be applied to the capability challenge, there is scope for it to be used to work towards solving the challenge of capacity as well.

PART V
Conclusion

V Conclusion

At the start of this project, we had two aims. The first was to develop a fully-functional and professional timed CSP simulator. The second was to apply this simulator to the railway domain and work towards solving the safety and optimisation challenges within the industry. Reflecting on the work done throughout the project, I believe that we have made significant headway towards achieving these goals.

By extending the existing untimed CSP simulator, ProB, we have ensured that our final product is of a professional standard. We have succeeded in developing many aspects of the software to accommodate for timed CSP; from the additional semantics in Prolog to the aesthetic changes in the GUI. We have also provided a full implementation for the most important timed CSP operators.

We have also been able to demonstrate the functionality of our timed CSP simulator, not just with basic models but also with more advanced systems which address safety and efficiency challenges within the railway domain.

Nevertheless, there is still opportunity for further development within this project. The next logical step is to work towards a well-rounded, complete timed CSP specification that includes the other key operators such as timed event prefix and interval delay. These extra operators would allow us to simulate the more complex systems and achieve a polished end-product.

A complete specification would also allow us to focus on refining our test models. While we managed to take our first steps towards working with the challenge of capability, our test model was highly-simplified. Using real-world data and track plans would allow us to truly ascertain the viability of our timed CSP simulator in the railway domain.

Ultimately, this project has provided a different viewpoint on how to approach timed simulation and has demonstrated that it is an alternative which has merit. There is definite scope to take this project further and develop a viable tool to help solve the pressing challenges within the industry.

PART VI
References & Appendix

VI References & Appendix

15 Works Cited

[1] Schneider, S. (2000). *Concurrent and Real-time Systems: The CSP Approach.* Chichester: John Wiley and Sons.

[2] Leuschel, M. and Butler, M. (2003) The ProB Animator and Model Checker for B. In *Proceedings of 12th International FME Symposium 2003 (FM2003).*

[3] Larsen, K., Yi, W. et al. (1999). *UPPAAL* (4.0.13) [Computer program]. Available at http://www.uppaal.com/ (Accessed 05 October 2010)

[4] Larsen, K., Pettersson, P. and Yi, W. (1997). UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer.* 1 (1+2), p134-152.

[5] Yi, W., Pettersson, P. and Daniels, M. (1994). Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proceedings of the 7th International Conference on Formal Description Techniques*, p234-236.

[6] Behrmann, G., David, A. and Larsen, K. (2004). A Tutorial on Uppaal. *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT 04).* LNCS 3185.

[7] Lucas, P. (2002). *Introduction to Prolog.* Lecture Notes, Aberdeen University. p1-2

[8] Endriss, U. (2007). *An Introduction to Prolog Programming.* Lecture Notes, Amsterdam University. p1, 4-6

[9] Andersson, J., Andersson, S., Boortz, K., Carlsson, M., Nilsson, H., Sjöland, T., and Widen, J. (1993). *SICStus Prolog User's Manual.* Technical Report. UMI Order Number: T93-01., European Research Consortium for Informatics and Mathematics at SICS.

[10] Van Roy, P. (1994). *1983-1993: The Wonder Years of Sequential Prolog Implementation. Journal of Logic Programming*, 19:385–441

[11] Lipovača, M. (2010). *Learn You a Haskell for Great Good.* p6.

[12] Ross, F. et al. (2008). *Introduction To Haskell.* p1-3.

[13] Leijen, D. (2001). *Parsec, a fast combinator parser.* p1.

[14] Peralta, S. J. (2003). *Scripting Graphical Commands with Tcl/Tk Mini-HOWTO.* p2-5.

[15] Roscoe, A.W. (2010). *Understanding Concurrent Systems.* Springer. p350

[16] Leuschel, M. and Butler, M. (2007). ProB: An Automated Analysis Toolset for the B Method. *International Journal on Software Tools for Technology Transfer*, 10 (2). p185, p192.

[17] Leuschel, M., Adhianto, L., Butler, M., Ferreira, C. and Mikhailov, L. (2001). Animation and Model Checking of CSP and B Using Prolog Technology. *In Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic, VCL'2001*, p97-109

[18] Leuschel, M. (2001). Design and Implementation of the High-Level Specification Language CSP(LP) in Prolog. *In Proceedings of PADL 01, Volume 1990 of LNCS*, p18.

[19] Hallerstede, S, Leuschel, M. (2003). *Constraint-Based Deadlock Checking of High-Level Specifications*. p8.

[20] Henzinger, T., Kupferman, O., Vardi, M. (1996). 'A Space-Efficient On-the-fly Algorithm for Real-Time Model Checking' *Concur '96: Concurrency Theory: 7th International Conference*. Springer. p515

[21] Fu, X., Bultan, T., Su, J. (2002). 'Formal Verification of e-Services and Workflows' *CAiSE '02/ WES '02 Revised Papers from the International Workshop on Web Services, E-Business, and the Semantic Web*. London: Springer-Verlag. p192

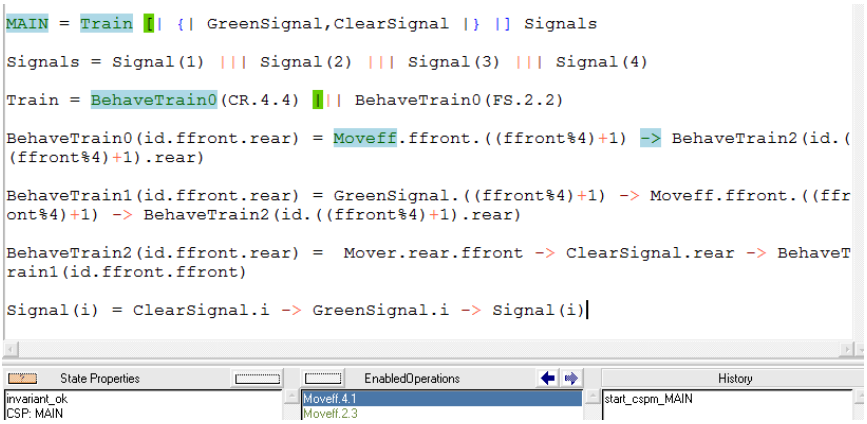
[22] Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2001). *Introduction to Algorithms, Second Edition*. McGraw-Hill. p527-528

[23] Winter, K. (2002). Model Checking Railway Interlocking Systems. *In Proceedings Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia. CRPIT, 4. Oudshoorn, M. J., Ed. ACS. p303-310.

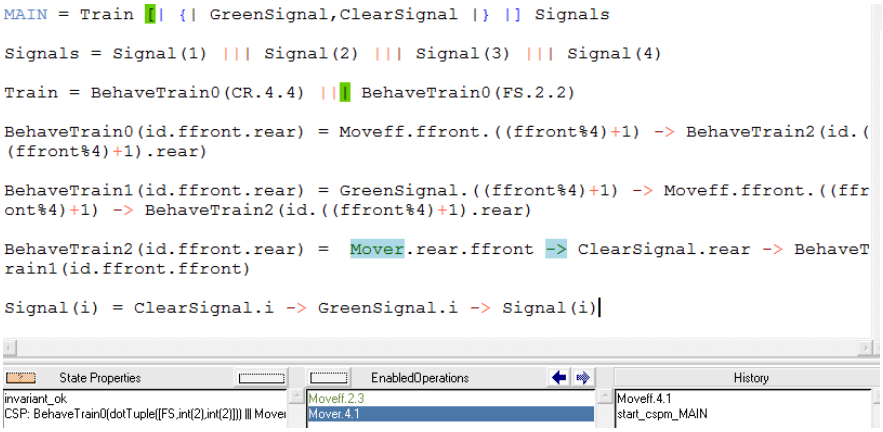
[24] Schneider, S. (2000). *Concurrent and Real-time Systems: The CSP Approach*. Chichester: John Wiley and Sons. p431-436.

16 Appendix

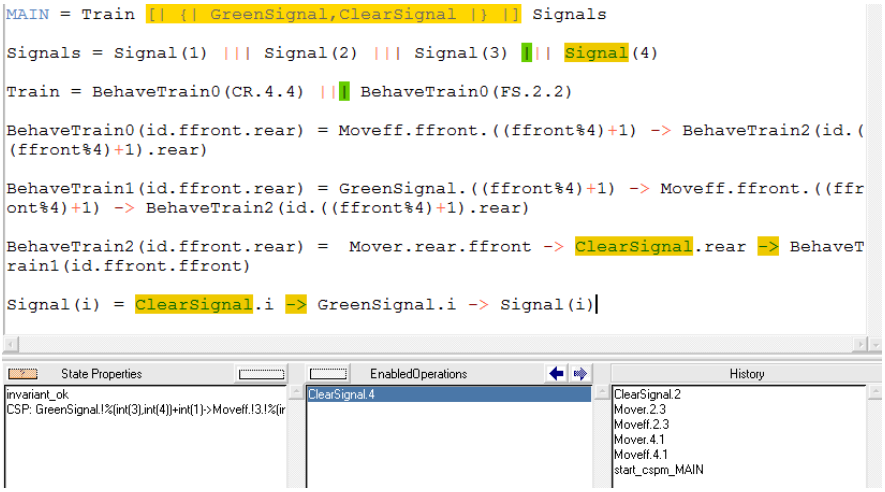
Appendix 1 – Untimed Railway Model Execution



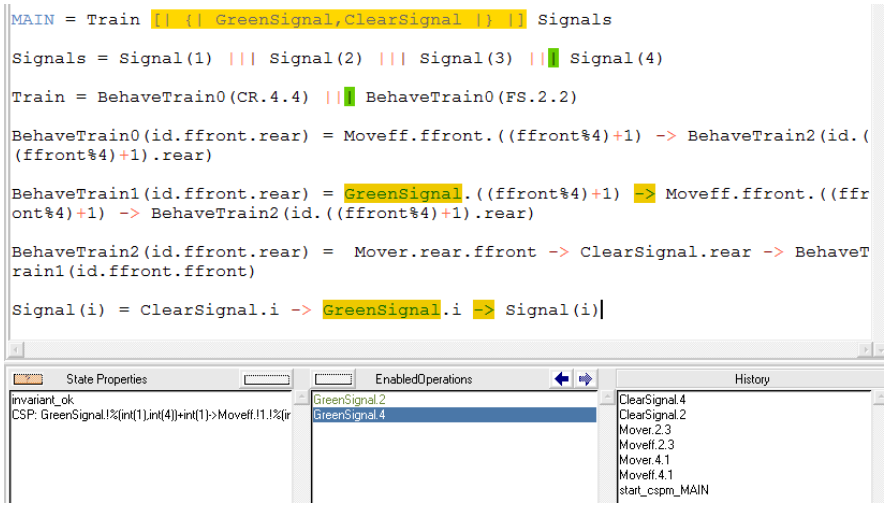
After the main process is activated, we are given the option to move the front of train CR from track 4 to 1 (Moveff.4.1) or to move the front of train FS from track 2 to 3 (Moveff.2.3).



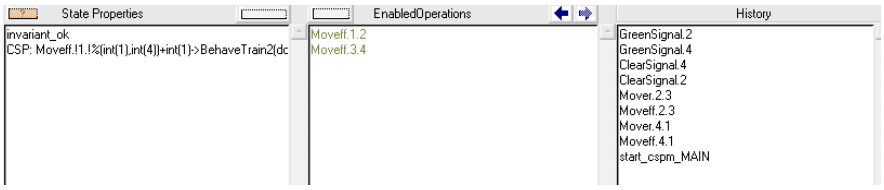
After moving the front of train CR front track 4 to track 1, we now have the option to make the same movement with the rear of train CR. Note that the Moveff event for train FS is still available.



Later on during execution, we can see here an example of the ClearSignal synchronisation. Train FS has no available actions as it requires a GreenSignal to continue. It cannot execute GreenSignal until Signal has executed ClearSignal, which it is about to do with train CR.



After two ClearSignals have executed, Signal(2) and Signal(4) are both ready to synchronise on GreenSignal with the corresponding trains.



Once the two GreenSignals have been executed, both train fronts are free to move onto the next track again, continuing the cycle.

Appendix 2 – Rational Number Framework

```
gcd(X,X,X):- !.
gcd(X,0,X):- !.
gcd(0,X,X):- !.
gcd(_X,1,1):- !.
gcd(1,_X,1):- !.
gcd(X,Y,Z):-
X>Y, !, XY is X mod Y, gcd(XY,Y,Z).
gcd(X,Y,Z):-
X<Y, !, YX is Y mod X, gcd(X,YX,Z).

lcm(X,Y,LCM):-
gcd(X,Y,GCD), LCM is X*Y//GCD.

simplify(A/A,1/1):-!.
simplify(A/1,A/1):-!.
simplify(1/A,1/A):-!.
simplify(A/B,C/D):-
gcd(A,B,E), C is A//E, D is B//E.

rat(A/B+C/D,E/F):- !,
lcm(B,D,Den), N is A*(Den//B) + C*(Den//D), simplify(N/Den,E/F).
rat(X+Y,E/F):-
rat(X,C/D), rat(Y,A/B), !, rat(C/D+A/B,E/F).

rat(A/B-C/D, E/F):- !,
lcm(B,D,Den), N is A*(Den//B) - C*(Den//D), simplify(N/Den,E/F).
rat(X-Y,E/F):-
rat(X,C/D), rat(Y,A/B), !, rat(C/D-A/B,E/F).

rat(A/B=<C/D,E/F):- !,
AB is A/B,CD is C/D,(AB=<CD -> E/F = A/B ; E/F = C/D).
rat(X=<Y,E/F):-
rat(X,C/D), rat(Y,A/B), !, rat(C/D=<A/B,E/F).

rat(A/B>=C/D,E/F):- !,
AB is A/B,CD is C/D,(AB>=CD -> E/F = A/B ; E/F = C/D).
rat(X>=Y,E/F):-
rat(X,C/D), rat(Y,A/B), !, rat(C/D>=A/B,E/F).

rat(A/B, SA/SB):-
CA is A, CB is B, !, simplify(CA/CB,SA/SB).
rat(A, CA/1):-
CA is A.

fractoint(A/1,A).
fractoint(A/A,1).
fractoint(A,A) :- integer(A).
```

Appendix 3 – Choice and Concurrency find_d Rules

```
find_d('[]'(X,Y,_Span),DF) :-
(find_d(X,D),find_d(Y,E) ->  rat(D=<E,DV) ;
(find_d(X,D) -> DV=D ;
(find_d(Y,E) -> DV=E ; fail))),
(fractoint(DV,DF) -> true ; rat(DV,DF)).

find_d(aParallel(_CList,X,_CListY,Y,_SrcSpan),DF) :-
(find_d(X,D),find_d(Y,E) ->  rat(D=<E,DV) ;
(find_d(X,D) -> DV=D ;
(find_d(Y,E) -> DV=E ; fail))),
(fractoint(DV,DF) -> true ; rat(DV,DF)).

find_d(eaParallel(_ECList,X,_EClistY,Y,_SrcSpan),DF) :-
(find_d(X,D),find_d(Y,E) ->  rat(D=<E,DV) ;
(find_d(X,D) -> DV=D ;
(find_d(Y,E) -> DV=E ; fail))),
(fractoint(DV,DF) -> true ; rat(DV,DF)).

find_d('|||'(X,Y,_Span),DF) :-
(find_d(X,D),find_d(Y,E) ->  rat(D=<E,DV) ;
(find_d(X,D) -> DV=D ;
(find_d(Y,E) -> DV=E ; fail))),
(fractoint(DV,DF) -> true ; rat(DV,DF)).

find_d(sharing(_CList,X,Y,_Span),DF) :-
(find_d(X,D),find_d(Y,E) ->  rat(D=<E,DV) ;
(find_d(X,D) -> DV=D ;
(find_d(Y,E) -> DV=E ; fail))),
(fractoint(DV,DF) -> true ; rat(DV,DF)).


find_d(esharing(_CList,X,Y,_Span),DF) :-
(find_d(X,D),find_d(Y,E) ->  rat(D=<E,DV) ;
(find_d(X,D) -> DV=D ;
(find_d(Y,E) -> DV=E ; fail))),
(fractoint(DV,DF) -> true ; rat(DV,DF)).
```

Appendix 4 – Choice and Concurrency calculate_res Rules

```
calculate_res('[]'(X,Y,Span),DV,Res) :-
(calculate_res(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF)),
Res='[]'(X1,Y1,Span).

calculate_res(aParallel(CListX,X,CListY,Y,SrcSpan),DV,Res) :-
(calculate_res(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF))
,Res=aParallel(CListX,X1,CListY,Y1,SrcSpan).

calculate_res(eaParallel(ECListX,X,ECListY,Y,SrcSpan),DV,Res) :-
(calculate_res(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF)),
Res=eaParallel(ECListX,X1,ECListY,Y1,SrcSpan).

calculate_res('|||'(X,Y,Span),DV,Res) :-
(calculate_res(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF)),
Res='|||'(X1,Y1,Span).

calculate_res(sharing(CListX,X,Y,SrcSpan),DV,Res) :-
(calculate_res(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF)),
Res=sharing(CListX,X1,Y1,SrcSpan).

calculate_res(esharing(CListX,X,Y,SrcSpan),DV,Res) :-
(calculate_res(X,DV,X1) -> true ; cspm_ttrans(X,_AX,X1,WF)),
(calculate_res(Y,DV,Y1) -> true ; cspm_ttrans(Y,_AY,Y1,WF)),
Res=esharing(CListX,X1,Y1,SrcSpan).
```

Appendix 5 – Hiding, Control Flow and agent_call calculate_res Rules

```
calculate_res('\\'(Expr,CList,Span),DV,Res) :-
(cspm_trans('\\'(Expr,CList,Span),tau(hide(_ActionX)),_Res ,WF) -> fail ;
(calculate_res(Expr,DV,NExpr) -> true ; cspm_ttrans(Expr,_AX,NExpr,WF)),
Res='\\'(NExpr,CList,Span)).

calculate_res(ehide(Expr,CList,Span),DV,Res) :-
(cspm_trans(ehide(Expr,CList,Span),tau(hide(_ActionX)),_Res ,WF) -> fail ;
(calculate_res(Expr,DV,NExpr) -> true ; cspm_ttrans(Expr,_AX,NExpr,WF)),
Res=ehide(NExpr,CList,Span)).

calculate_res(';'(P,Q,SeqSpan),DV,Res) :-
(cspm_trans(';'(P,Q,SeqSpan),tau(tick(_)),_Res,_WF) -> fail ;
(calculate_res(P,DV,P1) -> true ;
cspm_ttrans(P,_AP,P1,_WF)),Res=';'(P1,Q,SeqSpan)).

calculate_res('/\\'(X,Y,Span),DV,Res) :-
(calculate_res(X,DV,X1) -> true ;
cspm_ttrans(X,_AX,X1,WF)),(calculate_res(Y,DV,Y1) -> true ;
cspm_ttrans(Y,_AY,Y1,WF)),Res='/\\'(X1,Y1,Span).

calculate_res(agent_call(_Span,F,Par),DV,Res) :-
unfold_function_call_once(F,Par,Value),calculate_res(Value,DV,Res).
```

Appendix 6 – update_result Rule

```
update_result(CSPState,Modifier,CurD,NewState) :-
((CSPState =.. [OP,X,Y,Span],((OP == '\\' ; OP == ehide ; OP == ';') ->
        NewState =.. [OP,X1,Y,Span] ; NewState =.. [OP,X1,Y1,Span]) ;
  CSPState =.. [OP,CList,X,Y,Span],(OP == sharing ; OP == esharing),
        NewState =.. [OP,CList,X1,Y1,Span] ;
  CSPState =.. [OP,CListX,X,CListY,Y,Span],
        NewState =.. [OP,CListX,X1,CListY,Y1,Span])
  -> update_result(X,Modifier,CurD,X1),
     ((OP == '\\' ; OP == ehide ; OP == ';') -> true ; update_result(Y,Modifier,CurD,Y1)) ;
((CSPState =.. [OP,X,E,Y,Span] ->
  (ground(E) -> CSPStateD =.. [OP,X,E,Y,Span] ; CSPStateD =.. [OP,X,CurD,Y,Span]) ;
                CSPState =.. [OP,E,Span],
  (ground(E) -> CSPStateD =.. [OP,E,Span] ; CSPStateD =.. [OP,CurD,Span]))
        -> calculate_res(CSPStateD,Modifier,NewState) ; NewState=CSPState)
).
```

Appendix 7 – Non-ground Variable Substitution Code

```
% Uses update_result rule from haskell_csp.pl
:- use_module(haskell_csp,'cia/haskell_csp',[update_result/4]).

tcltk_goto_state(ActionAsTerm,StateID) :-
  ~~mnf(translate:translate_event(ActionAsTerm,ActionAsString)),
  tcltk_goto_state(ActionAsTerm,ActionAsString,StateID).
tcltk_goto_state(ActionAsTerm,ActionAsString,StateID) :-
  (var(StateID) -> print_message(var_state(tcltk_goto_state(ActionAsString,StateID)))  ;true),
   (visited_expression(StateID,StateTempl,StateBody)
    -> true
    ;  (print(state_does_not_exist(tcltk_goto_state(ActionAsString,StateID))),nl,fail)
    ),

    /* ------------------------------------ */
    /* NON-GROUND VARIABLE SUBSTITUTION CODE */
    /* ------------------------------------ */

    % Extracts executed action's data
    timedata(1,[H|_]),ActionAsTerm =.. TermAsList,nth0(0,TermAsList,Check),
    % Checks action is time(DV,RD,Span)
    (Check == time,nth0(3,TermAsList,_) -> nth0(2,TermAsList,D),(fractoint(D,RD) -> true ; rat(D,RD)),
                                           % Calls update_result, substitutes the non-ground variable
                                           % with original D and updates all time limits with time subtracted.
                                           update_result(StateTempl,H,RD,StateTempl2),
                                           % Non-ground state is removed from visited expressions
                                           retract(visited_expression(StateID,_StateTempl,_StateBody)),
                                           % Replaced by the ground state. Necessary step for backtrack to work
                                           assert(visited_expression(StateID,StateTempl2,StateBody))
                                           % If action is not a 3-variable timed event, the state is already ground.
                                           ;   StateTempl2 = StateTempl),
    % The current expression is cleared
   (retract(current_expression(CurID,_CurTempl,_CurBody))->true),
    % The current expression is updated with the new state
   assert(current_expression(StateID,StateTempl2,StateBody)),

   /* ------------------------------------ */
   /*   NON-GROUND SUBSTITUTION CODE ENDS   */
   /* ------------------------------------ */
```

Appendix 8 – Global Time & Constraint Rules

```
:- use_module(haskell_csp,'csp/haskell_csp',[rat/2,fractoint/2]).

tcltk_initialise(Nr) :-
   retractall(timedata(_,_)),
   assert(timedata(1,[0])),
   assert(timedata(2,[0])),
   tcltk_perform(Nr).

tcltk_get_time(Toutput) :-
   timedata(1,Y),
   list_sum(Y,Sum),
   (fractoint(Sum,Time) -> Toutput = Time ; rat(Sum,A/B),Toutput = [A,'/',B]).

tcltk_constraints(Nr,Evol) :-
   (Evol >= 0 -> true ; fail),
   current_options(Options),
   nth1(Nr,Options,(_Id,_Action,ActionAsTerm,_NewID)),
   ActionAsTerm =.. TermAsList,
   nth0(0,TermAsList,Check),
   (Check == time,nth0(3,TermAsList,_) -> nth0(1,TermAsList,ULimit),
   (Evol < ULimit -> true ; fail) ; true).

list_sum([], 0).
list_sum([Head | Tail], TotalSum) :-
   list_sum(Tail, Sum1),
   rat(Head+Sum1,TotalSum).

tcltk_timer(Nr,Evol) :-
   (tcltk_constraints(Nr,Evol) ->
   timedata(1,X),
   retract(timedata(1,_)),
   assert(timedata(1,[Evol|X])),
   tcltk_perform(Nr) ; fail).

tcltk_perform(Nr) :-
   current_options(Options),
   tcltk_perform2(Nr,Options).
```

Appendix 9 – Backtrack and Forward

```
tcltk_can_backtrack :- history([_|_]).
tcltk_backtrack :- \+ tcltk_can_backtrack,!,
   print_message('Cannot backtrack'),fail.
tcltk_backtrack :-

                 /* added code */
                timedata(1,[H|T]),
                retract(timedata(1,_)),
                assert(timedata(1,T)),
                timedata(2,X),
                retract(timedata(2,_)),
                assert(timedata(2,[H|X])),
                /* added code */

                retract(history(History)),
                retract(trace(Trace)),
                retract(current_expression(CurID,_,_)),!,
                History= [LastID|EHist],
                assert(history(EHist)),
                visited_expression(LastID,LastExpr,LastBody),
                assert(current_expression(LastID,LastExpr,LastBody)),

                /* added code */
                transition(LastID,ActionAsTerm,_ActID,CurID),
                (ActionAsTerm =.. TermAsList,nth0(3,TermAsList,_) ->
                retractall(transition(LastID,_ActionAsTerm,_ActID,CurID)) ; true),
                /* added code */

                Trace= [LastTraceH|NewT],
                assert(trace(NewT)),
                (retract(forward_history(FwdHist)) -> true ; FwdHist=[]),
                assert(forward_history([forward(CurID,LastTraceH)|FwdHist])).
tcltk_can_forward :- forward_history([_|_]).
tcltk_forward :- \+ tcltk_can_forward,!,
   print_message('Cannot go forward'),fail.
tcltk_forward :-

                /* added code */
                timedata(2,[H|T]),
                retract(timedata(2,_)),
                assert(timedata(2,T)),
                timedata(1,X),
                retract(timedata(1,_)),
                assert(timedata(1,[H|X])),
                /* added code */

                retract(forward_history([forward(FwdID,LastTraceH)|FwdHist])),
                retract(history(EHist)),
                retract(trace(Trace)),
                retract(current_expression(CurID,_,_)),!,
                assert(forward_history(FwdHist)),
                assert(history([CurID|EHist])),
                assert(trace([LastTraceH|Trace])),
                visited_expression(FwdID,FwdExpr,FwdBody),
                assert(current_expression(FwdID,FwdExpr,FwdBody)).
```

Appendix 10 – Abstract Syntax Tree

```
-- expressions
type LExp = Labeled Exp

data Exp
  = Var LIdent
  | IntExp Integer
  | SetExp LRange (Maybe [LCompGen])
  | ListExp LRange (Maybe [LCompGen])
  | ClosureComprehension ([LExp],[LCompGen])
  | Let [LDecl] LExp
  | Ifte LExp LExp LExp
  | CallFunction LExp [[LExp]]
  | CallBuiltIn LBuiltIn [[LExp]]
  | Lambda [LPattern] LExp
  | Stop
  | Skip
  | CTrue
  | CFalse
  | Events
  | BoolSet
  | IntSet
  | ProcSet
  | Wait LExp -- WAIT d
  | TupleExp [LExp]
  | Parens LExp
  | AndExp LExp LExp
  | OrExp LExp LExp
  | NotExp LExp
  | NegExp LExp
  | Fun1 LBuiltIn LExp
  | Fun2 LBuiltIn LExp LExp
  | DotTuple [LExp]
  | Closure [LExp]
  | ProcSharing LExp LProc LProc
  | TimedTimeout LExp LProc LProc -- Timed Timeout [{d}>
  | TimedInterrupt LExp LProc LProc -- Timed Interrupt /{d}\
  | ProcAParallel LExp LExp LProc LProc
  | ProcLinkParallel LLinkList LProc LProc
  | ProcRenaming [LRename] (Maybe LCompGenList) LProc
  | ProcException LExp LProc LProc
  | ProcRepSequence LCompGenList LProc
  | ProcRepInternalChoice LCompGenList LProc
  | ProcRepExternalChoice LCompGenList LProc
  | ProcRepInterleave LCompGenList LProc
  | ProcRepAParallel LCompGenList LExp LProc
  | ProcRepLinkParallel LCompGenList LLinkList LProc
  | ProcRepSharing LCompGenList LExp LProc--
  | PrefixExp LExp [LCommField] LProc--
  | DelayExp LExp [LCommField] LExp LProc-- -- Delay -{d}->
```

Appendix 11 – Precedence Table

```
baseTable :: OpTable
procTable :: OpTable
(baseTable, procTable) = (
   [
--    [ infixM ( cspSym "." >> binOp mkDotPair) AssocRight ]
--    ,
-- dot.expression moved to a seperate Step
-- ToDo : fix funApply and procRenaming
    [ postfixM funApplyImplicit ]
   ,[ postfixM procRenaming ]
   ,[ infixM (nfun2 T_hat     F_Concat ) AssocLeft,
     prefixM (nfun1 T_hash    F_Len2 ) -- different from Roscoe Book
    ]
   ,[ infixM (nfun2 T_times   F_Mult ) AssocLeft
    ,infixM (nfun2 T_slash   F_Div ) AssocLeft
    ,infixM (nfun2 T_percent F_Mod  ) AssocLeft
    ]
   ,[ infixM (nfun2 T_plus    F_Add ) AssocLeft,
      infixM (nfun2 T_minus   F_Sub ) AssocLeft
    ]
   ,[ infixM (nfun2 T_eq      F_Eq ) AssocLeft
    ,infixM (nfun2 T_neq     F_NEq) AssocLeft
    ,infixM (nfun2 T_ge      F_GE ) AssocLeft
    ,infixM (nfun2 T_le      F_LE ) AssocLeft
    ,infixM (nfun2 T_lt      F_LT ) AssocLeft
    ,infixM (do
       s <- getNextPos
       gtSym
       e <- getLastPos
       op <- mkLabeledNode (mkSrcSpan s e) (BuiltIn F_GT)
       return $ (\a b-> mkLabeledNode (posFromTo a b) $ Fun2 op a b)
     ) AssocLeft
    ]

   ,[ prefixM ( token T_not >> unOp NotExp )]
   ,[ infixM ( token T_and >> binOp AndExp) AssocLeft ]
   ,[ infixM ( token T_or >> binOp OrExp) AssocLeft ]
   ],
   [[ infixM proc_op_aparallel AssocLeft ]
   ,[ infixM proc_op_lparallel AssocLeft ]

   ,[infixM procOpSharing AssocLeft ]

   ,[infixM (nfun2 T_backslash  F_Hiding     ) AssocLeft]
   ,[infixM (nfun2 T_amp        F_Guard      ) AssocLeft]
   ,[infixM (nfun2 T_semicolon  F_Sequential ) AssocLeft]
   ,[infixM timedInterrupt AssocLeft ] -- Timed Interrupt
   ,[infixM (nfun2 T_triangle   F_Interrupt  ) AssocLeft]
   ,[infixM (nfun2 T_box        F_ExtChoice  ) AssocLeft]
   ,[infixM timedTimeout AssocLeft ] -- Timeout
   ,[infixM (nfun2 T_rhd        F_Timeout    ) AssocLeft]
   ,[infixM (nfun2 T_sqcap      F_IntChoice  ) AssocLeft]
   ,[infixM procOpException AssocLeft]
   ,[infixM (nfun2 T_interleave F_Interleave ) AssocLeft]
  ]
  ,[ prefixM ( token T_wait >> unOp Wait )] -- Wait
  )
```

Appendix 12 – Parse Methods

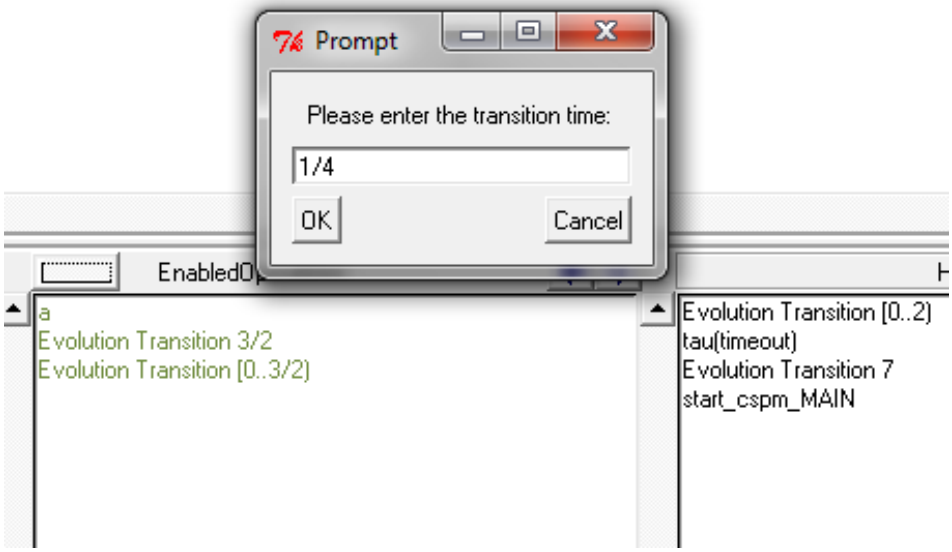
```
timedTimeout :: PT (LProc -> LProc -> PT LProc)
timedTimeout = try $ do
  spos <- getNextPos
  time <- between ( token T_timeOpenBrack) (token T_timeCloseBrack) parseExp
  epos <- getLastPos
  return $ (\a b  -> mkLabeledNode (mkSrcSpan spos epos) $ TimedTimeout time a b)

timedInterrupt :: PT (LProc -> LProc -> PT LProc)
timedInterrupt = try $ do
  spos <- getNextPos
  time <- between ( token T_interruptOpenBrack) (token T_interruptCloseBrack) parseExp
  epos <- getLastPos
  return $ (\a b  -> mkLabeledNode (mkSrcSpan spos epos) $ TimedInterrupt time a b)
```

```
parseDelayExp :: PT LExp
parseDelayExp = do
  spos <- getNextPos
  start <- parseExp_noProc -- channel or just an expression
  rest <- parsePrefix
  epos <- getLastPos
  case rest of
    Nothing -> fail "Expected rhs of delay expression"
    Just (comm,time,body) -> mkLabeledNode (mkSrcSpan spos epos) $
                               DelayExp start comm time body
  where
  parsePrefix :: PT (Maybe ([LCommField],LExp,LExp))
  parsePrefix = optionMaybe $ do
    commfields <- many parseCommField
    time <- between ( token T_delayOpenArrow) (token T_delayCloseArrow) parseExp
    exp <- parseProcReplicatedExp <?> "rhs of prefix operation"
    return (commfields,time,exp)
```

Appendix 13 – Time Input Codes for GUI

```
proc procPerformOption {} {
    set iOption [.frmInfo.frmPerform.list curselection]
    if {$iOption != ""} {
        incr iOption
        set action [.frmInfo.frmPerform.list get [expr $iOption - 1]]
        if {[.frmInfo.frmPerform.list get [expr $iOption - 1]] == "start_cspm_MAIN"} {
            if [prolog tcltk_initialise($iOption)] {
                procInsertHistoryOptionsState
            }
        } elseif {[string first "Evolution Transition \[0.." $action] == 0} {
            set evol [Dialog_Prompt "Please enter the transition time: "]
            if {$evol != "" && [prolog tcltk_timer($iOption,$evol)]} {
                procInsertHistoryOptionsState
            }
        } elseif {[string first "Evolution Transition" $action] == 0} {
            set evol [string trim $action "Evolution Transition "]
            if [prolog tcltk_timer($iOption,$evol)] {
                procInsertHistoryOptionsState
            }
        } else  {
            if [prolog tcltk_timer($iOption,0)] {
                procInsertHistoryOptionsState
            }
        }
    }
}
```

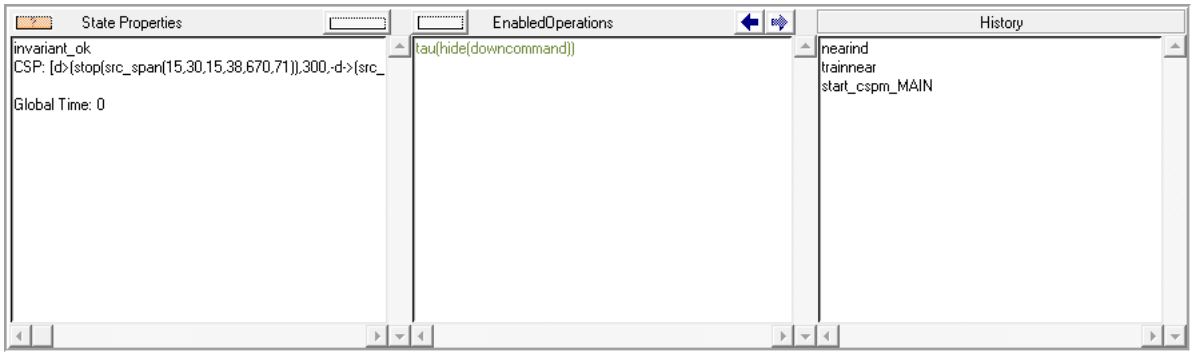
Appendix 14 – Screenshot of Time Input Pop-up

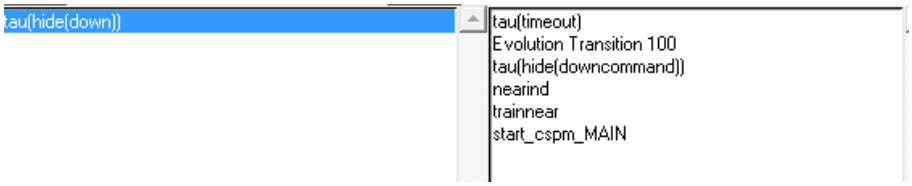


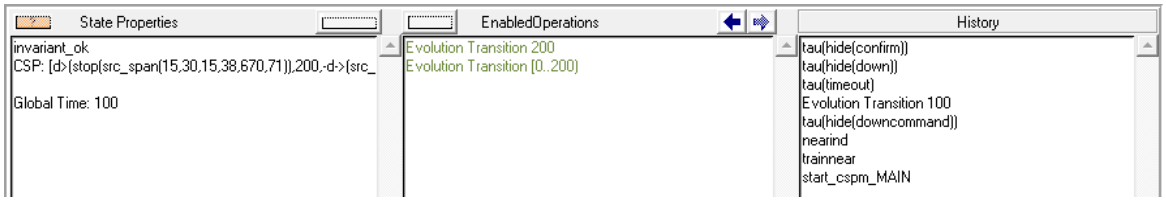
Appendix 15 – CSP Code for Railway Crossing Model

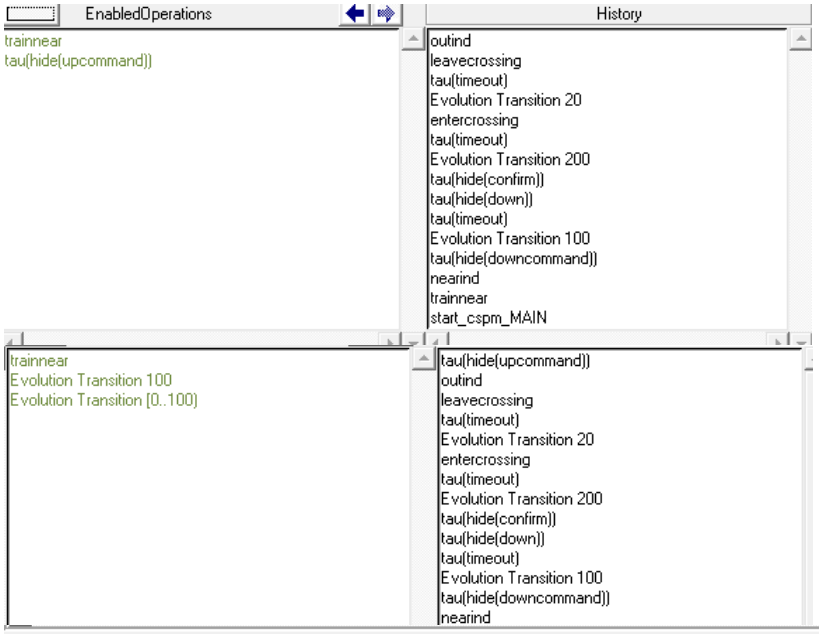
```
channel trainnear, nearind, entercrossing, leavecrossing,
        outind, downcommand, upcommand, down, up, confirm

MAIN = TRAIN [ {trainnear, nearind, entercrossing, leavecrossing, outind} ||
               {nearind, outind, downcommand, upcommand, down, up, confirm} ]
       CROSSING

CROSSING = CONTROLLER
           [ {nearind, outind, downcommand, upcommand, confirm} ||
             {upcommand, downcommand, up, down, confirm} ]
           GATE \ {up,down,upcommand,downcommand,confirm}

CONTROLLER = nearind -> downcommand -> confirm -> CONTROLLER []
             outind -> upcommand -> confirm -> CONTROLLER

GATE = downcommand -{100}-> down -> confirm -> GATE []
       upcommand -{100}-> up -> confirm -> GATE

TRAIN = trainnear -> nearind -{300}-> entercrossing -{20}->
        leavecrossing -> outind -> TRAIN
```

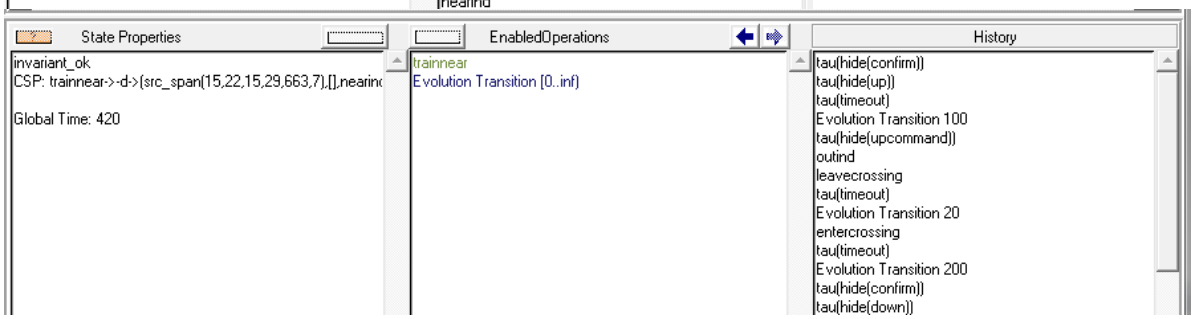
Appendix 16 – Test Run of Railway Crossing Simulation











Appendix 17 – CSP Code for London Underground Model

```
datatype Trains = LU1 | LU2 | LU3
datatype Signals = s1 | s2
Tracks = {0..3}

channel Moveff,Mover: Trains.Tracks
channel Green, Clear: Signals

MAIN = Train [| {| Green, Clear |} |] Signal \
{Moveff.x.y, Mover.x.y, Green.z, Clear.z | x <- Trains, y <- Tracks, z <- Signals}

Train = BehaveTrain(LU1) ||| WAIT 100 ; BehaveTrain(LU2) |||
        WAIT 200 ; BehaveTrain(LU3) ||| WAIT 300 ; Train

Signal = BehaveSignal(s1) ||| BehaveSignal(s2)

BehaveTrain(tID) = Moveff.tID.1 -{7}-> Mover.tID.1 -> Green.s1 -{7}->
                   Moveff.tID.2 -{7}-> Mover.tID.2 -{20}-> WAIT 30 ;
                   Green.s2 -{20}-> Moveff.tID.3 -{7}-> Mover.tID.3 ->
                   Clear.s1 -{7}-> Moveff.tID.0 -{7}-> Mover.tID.0 ->
                   Clear.s2 -> STOP

BehaveSignal(sID) = Green.sID -> Clear.sID -> BehaveSignal(sID)
```