Simulating and Analyzing Railway Interlockings in ExSpect

Twan Basten, Roland Bol, and Marc Voorhoeve

Department of Computing Science, Eindhoven University of Technology, The Netherlands
email: {tbasten,bol,wsinmarc}@win.tue.nl

Abstract

This paper describes a study on simulating and analyzing interlocking specifications in the Interlocking Specification Language (ISL), using the tool ExSpect. ExSpect is a toolkit based on the theory of coloured Petri nets.

An approach to translating ISL to ExSpect is suggested. Experimental results of simulating and analyzing part of an ISL specification in ExSpect are discussed. ExSpect seems to be useful for simulating and analyzing ISL specifications. Furthermore, several interesting topics for future research are identified.

Key words: railway signalling – interlocking – simulation – analysis – (coloured) Petri nets – ExSpect

1 Introduction

This paper reports on a case study on simulating and analyzing railway interlockings using the tool ExSpect. A railway interlocking is any system that is used to guarantee safety of train movements. ExSpect, which is an acronym for Executable Specification tool [1], is a graphical specification and simulation package developed at Eindhoven University of Technology and commercially available from Bakkenist Management Consultants.

The study described in this paper has been done in close cooperation with the Dutch railway company NS, Nederlandse Spoorwegen. To date, railway interlockings are often systems of relays controlled by computer equipment and/or human operators. In the future, the role of electronics and computer equipment will become increasingly important. The most important reason for this is that NS has to guarantee the safety of more passengers, personnel, and goods in an environment that is changing rapidly from a state monopoly to a dynamic, competitive environment. This requires high flexibility at lower cost while maintaining the same high safety requirements as before.

Until recently, interlocking practice was mostly based on experience. The consequences of small changes in, for example, positioning of signals, signalling protocols, or safety margins were only understood by some very experienced engineers. Obviously, there is need for a different approach.

In order to adapt to a dynamic environment without creating dangerous situations, a formalism for describing safety requirements and interlocking behaviour is needed. Such a formalism can be used for simulating interlocking behaviour and, preferably, for formally verifying safety requirements. In addition, if it is possible to simulate or calculate the effects of small changes in signalling protocols or safety margins, the formalism can be used to optimize interlocking behaviour and even lead to adaptation of the infrastructure and train schedules.

Therefore, NS designed a set of specification languages to describe the behaviour of interlockings in a formal way. As in [5], the abbreviation ISL, for Interlocking Specification Language, is used to refer to this set of languages. It is preferred over the name EURIS, which is used by NS and stands for European Railway Interlocking Specification [3, 4]. The name ISL states more clearly that it is meant as a *specification language*. Currently, NS is implementing a simulation package for ISL [9]. Although ISL is an important step towards a more formal approach to building and maintaining interlockings, it is not yet possible to actually prove any safety properties, mainly because ISL lacks a firm mathematical basis.

This paper describes a first step towards simulation and verification in ISL based on a mathematical theory. For this purpose, a small part of an ISL specification is translated into ExSpect. ExSpect is a toolkit based on the theory of Petri nets (see for example [11, 14]). It combines a graphical, easy-to-understand user interface for specifying and simulating many types of information systems with some analysis tools for verifying properties of such systems. Therefore, it overcomes the main shortcomings of ISL. Using the analysis tools of ExSpect, it is shown that the translation into ExSpect maintains an important assumption of ISL specifications.

Note that the approach described in this paper is definitely not the only possibility to provide a mathematical basis for ISL (see [5] for a brief overview of some possibilities). It is also not yet possible to actually verify any safety properties of an interlocking. The most important reason for this is that it is not exactly clear what the safety requirements of an interlocking described in ISL are. Another reason is that the tool ExSpect is not yet powerful enough. However, this is not only a shortcoming of ExSpect. Systems such as railway interlockings are still too complex for formal verification using current technology.

Therefore, this paper should be considered as a preliminary study of interlocking specifications in ExSpect with a two-fold goal. First, it is investigated to what extent ExSpect can be used to improve simulation and verification in ISL. Second, a specification of an interlocking in ISL is an interesting real-world application that can be used to determine the strong and, more importantly, weak points of ExSpect.

The paper is organized as follows. The next section describes the Interlocking Specification Language. It also gives a brief explanation of the current simulation package. Section 3 explains the basics of Petri-net theory and provides an introduction to the tool ExSpect. In Section 4, it is explained how an ISL specification can be translated into ExSpect. This section also discusses the merits and demerits of simulating and verifying ISL specifications in ExSpect. Section 5 discusses some possibilities for future work and Section 6 contains some concluding remarks. Finally, the appendix describes an ExSpect specification of a fragment of an interlocking specification in ISL.

2 The Interlocking Specification Language

The Interlocking Specification Language is actually a set of languages. It consists of four sublanguages: TL for specifying Track Layouts, ECL for Element Connection Layouts, LECL for Logical Element Connection Layouts, and LSC for Logic Sequence Charts. The first three languages are all used to specify railway yards or routes. They only differ in level of abstraction. In essence, all three languages describe how routes are built from generic elements such as signals, points, and track segments. The LSC language is used to describe such elements. Every generic element is specified by one or more Logic Sequence Charts.

Although ISL has languages for four levels of abstraction, only two levels are fundamentally different, namely the level of routes and the level of basic elements. That is, one might be interested in properties of entire routes or single elements. This means that, in the context of simulation and verification, only Logical Element Connection Layouts and Logic Sequence Charts are important. The remainder of this paper, therefore, focusses on these two levels of abstraction in ISL. A full description of these two languages can be found in [3]. The following is a brief introduction that is necessary to understand the remaining sections of this paper.

2.1 Logical Element Connection Layouts

LECL is a simple language that consists of six basic building blocks: signal, point, track, approach-monitoring element, approach-signalling element, and signal-clearance-delay element. The first three elements need no explanation. The latter three elements are related to warning devices. In this paper, only the signal and track elements are used.

A fundamental concept in LECLs (and LSCs) is the concept of *telegrams*. According to [3], telegrams are used for the following two purposes. First, a telegram represents the flow of events (control flow). Second, it is used to exchange information between elements, human operators, control equipment, and trackside devices (data flow). Telegrams are the dynamic component of an ISL specification.

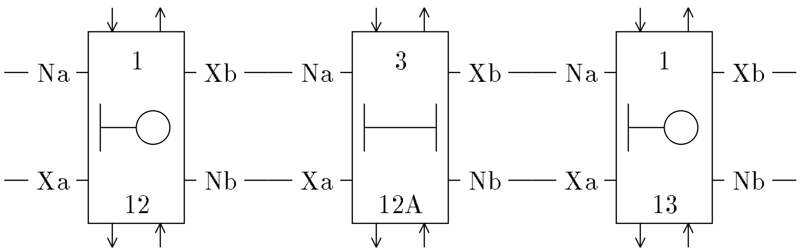

Figure 1: The LECL of a simple route.

Figure 1 shows a very simple route in LECL. From left to right, the route contains an entry signal, one track element, and an exit signal. Each element has its own graphical symbol; each symbol has a unique reference number which is shown at the top of the element. As can be seen in the example, a signal symbol has reference number one and a track symbol number three. Furthermore, every element in a route description has a unique identifier, which is the bottom number in the element. The entry signal, for example, has identifier twelve.

Each element has pins that can be connected to its preceding and succeeding element, the control level, and the trackside devices. Exchange of information via telegrams takes place through these pins. Pins to the control level and trackside devices are depicted by the arrows on the upper and lower side of the element respectively. Pins to which other elements can be connected have an identifier which consists of two characters: an “N” or “X” denoting whether it is an eNtry or an eXit pin, and an “a,” “b,” “c,” or “d” that corresponds to the *side* of an element. Figure 1 shows that an element does not necessarily have pins for all possible pin identifiers. A signal or track does not have sides “c” or “d.” The side from which a telegram enters an element determines the *direction* of the telegram inside the element. The behaviour of a telegram may depend upon its direction.

LECL has rules for connecting elements. Obviously, exit pins of one element must be connected to entry pins of another. There are also some limitations concerning the side identifier of pins, but they are not important in this paper and are, therefore, omitted.

The brief explanation of LECL given here makes clear that it is possible and meaningful to simulate LECLs. Dynamic behaviour can be simulated, investigating the flow of telegrams, timing properties, and properties about the state of individual elements or the entire route.

2.2 Logic Sequence Charts

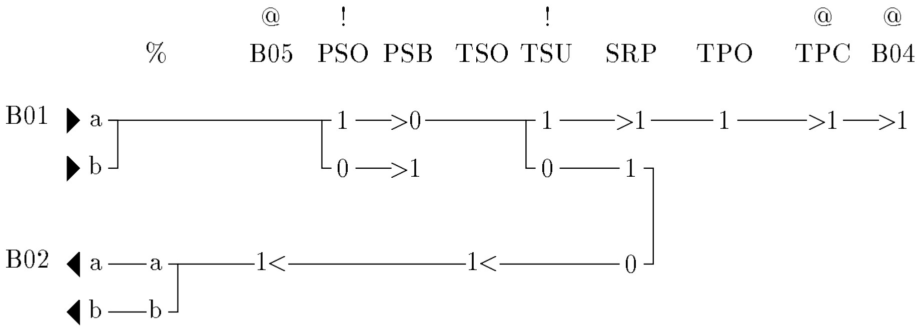

Figure 2: A fragment of an LSC.

Every element in LECL is described by one or more Logic Sequence Charts. Figure 2 shows a detailed fragment of an LSC, which is part of the specification of an ordinary track element. It shows various features of an LSC. The top of an LSC lists all variables and parameters. The left and right side contain entries and exits for telegrams from and to different directions. The LSC fragment in Figure 2, for example, has an entry for telegram B01 from either direction "a" or "b." It has an exit for telegram B02 to either direction. The horizontal and vertical lines in an LSC depict the flow of telegrams. The line of flow can be interrupted by several operations on variables and telegrams; it can branch, and two or more branches can join into a single line.

Most of the variables of an LSC are booleans. Some of the variables are of a special kind. Those marked with an exclamation mark correspond directly to a trackside device. The variable TSU, for example, represents whether or not *Track Section Unoccupied* is true. A variable marked with @ is some kind of timer that, depending on its value, periodically generates telegrams. Both types of variables occur in Figure 2. A kind of variable that does not occur in Figure 2 is a variable marked with [ ], denoting that it is a natural number. The special symbol "%" denotes a logical variable that contains the direction of a telegram.

While a telegram flows through an LSC, several explicit and implicit operations on telegrams and variables can be performed. Operations on variables are always depicted below the variable on which the operation is performed. Operations on telegrams can occur everywhere. The most commonly used operations are assignments and tests. A value v is assigned to telegram data t or variable XXX by "$(t : v)$" and "$> v$" in the column below XXX respectively. A test on telegram data is depicted by inserting "$(t = v)$" in the line of flow; a test on XXX by simply inserting the value v in the column below XXX. Examples of these operations can be seen in Figure 2. Note that the boolean values "true" and "false" are denoted by 1 and 0 respectively. Figure 2 does not contain any operations on telegram data.

Some operations on telegrams or variables may influence the flow of a telegram. Depending on the result of a test, for example, a telegram can proceed in different ways. It is also possible that a telegram terminates. An example of the latter appears in the column below TPO, where the value 1 is inserted. This means that if TPO is false, then the telegram terminates, otherwise it proceeds. A telegram also terminates when the line of flow ends. Termination of telegrams is an implicit operation.

There are two other operations on telegrams that do not appear explicitly in an LSC, namely a change of name or direction of a telegram. The former occurs in Figure 2, where a B01 telegram changes into a B02 telegram, provided that it does not terminate inside the LSC. Similarly, it may happen that a telegram enters an LSC from one particular side and leaves it at another. Such implicit operations are important in the context of simulation where they must be made explicit.

There is one more important operation, namely *telegram generation*. As mentioned, a variable marked with @ is some kind of timer. More in particular, it is a boolean variable that periodically generates a telegram starting when its value is set to true. It stops when its value is reset to false. A telegram that is generated in this way enters the LSC at a unique position marked "@>" in the column below the variable. From there, it follows the line of flow as an ordinary telegram. In Figure 2, the B01 telegram may start several timers. A telegram that is generated in this way starts in another part of the LSC which is not shown in Figure 2. There are several other types of timers that are, however, not explained here.

The description of LSCs given here makes clear that, in addition to LECLs, LSCs are a second, more detailed level of abstraction that can be simulated in a meaningful way. Simulating and verifying LSCs is useful to determine properties of individual elements instead of entire routes built from those elements. Ideally, it should be possible to switch between both levels of abstraction in one simulation session.

At this point, it should be mentioned that ISL makes one important assumption about the intended dynamic behaviour of systems specified in ISL. It is assumed that, at any point in time, at most one telegram is active in an LSC. In the context of simulation and verification, this assumption is essential for determining the correct behaviour of a system. It also means that some priority or scheduling mechanism is needed for the telegrams entering an LSC from other elements, the control level, and the trackside level, as well as for the telegrams that are generated internally. ISL itself does not give any requirements or restrictions for such a scheduling mechanism.

2.3 The ISL Design and Simulation Package

Figure 3 [9] gives an overview of the current ISL design and simulation package. Both LSCs and LECLs, or routes, are designed using a CAD application package called ACE+. The results are automatically translated to an intermediate language called IDEAL, the Interlocking Design and Application Language. The symbols that are used to design LECLs are automatically generated from the LSC-IDEAL code. That is, the correct number of parameters, pins etcetera is determined from the specification of an element in terms of LSCs.

A compiler has been developed that translates the IDEAL code of LSCs and LECLs to VHDL. For the actual simulation, a VHDL simulator is used extended with a graphical user interface based on the package ACE+. Only LECLs can be simulated. The simulation uses colours to report part of the state of individual elements. One colour denotes that the element is reserved for a route; another colour may indicate that it is released. It is also possible to keep track of the value of individual variables over time. It is, however, not possible to visualize the flow of telegrams nor to simulate individual elements.

In order to guarantee that, at any time, at most one telegram is active in each LSC, a simple queue has been implemented. In this queue, both external and internal telegrams are stored in the order in which they arrive or are generated.

3 Petri Nets and ExSpect

This section explains the basics of Petri-net theory and the tool ExSpect which is based upon this theory. Since Petri nets have unambiguous graphical representations, formal mathematical definitions are omitted. The interested reader is referred to [11, 14]. Instead, a running example of

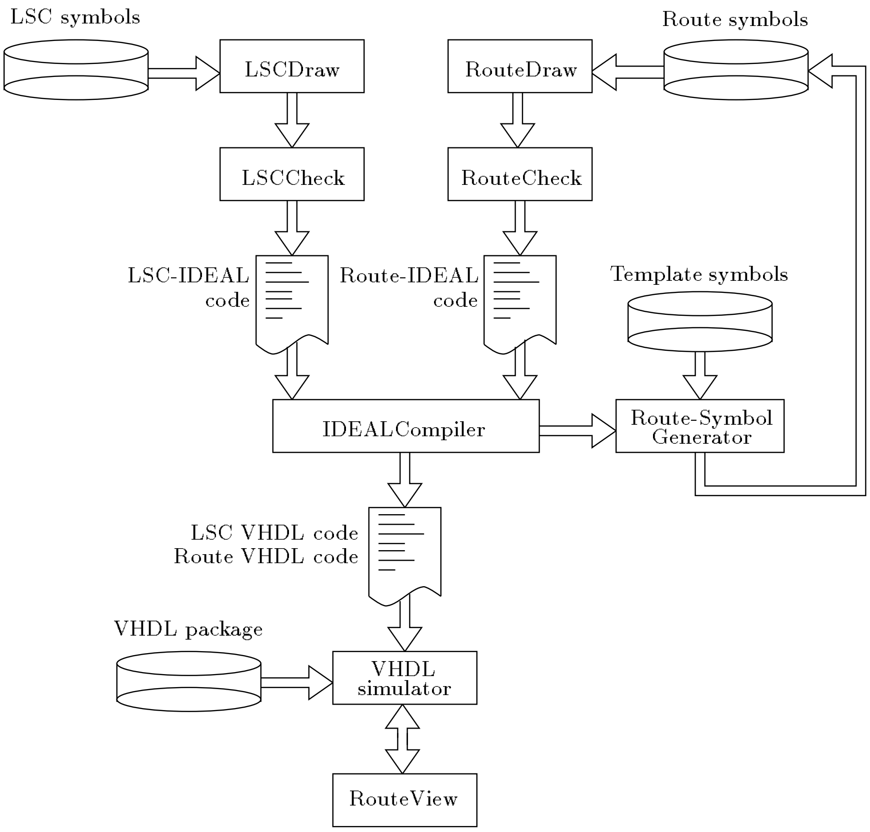

Figure 3: The ISL design and simulation package.

a set of traffic lights at a crossing is used to explain Petri nets and ExSpect. The next subsection gives a brief introduction to the most fundamental form of Petri nets, called Place/Transition- or P/T nets. Section 3.2 discusses the tool ExSpect which is based upon a more general type of Petri nets, so-called high-level Petri nets.

3.1 Place/Transition Nets

Consider a crossing of two streets where each street has a traffic light. The two lights perform their green-yellow-red cycle more or less autonomously. However, before turning green a traffic light needs an incoming synchronization signal from the other light indicating that it turned red. This is to guarantee that always one of the two lights is red. Figure 4 show the P/T net for this crossing.

A P/T net is a directed graph that has two basic structural components: places and transitions. Places are usually depicted by circles and transitions by bars or rectangles. Places and transitions are connected by arrows. Multiple arrows in either direction are allowed between places and transitions. If an arrow connects place p to transition t, then p is called an *input* place of t; if an arrow connects t to p, then p is an *output* place of t. Places and transitions, together with their interconnections, form the static *structure* of a net.

Petri nets also have dynamic components, called *tokens*. Tokens reside in the places of a net. In a graphical representation of nets, tokens are depicted as black dots. All tokens together are the marking of a net, which represents its *state*. The state can change dynamically according to the following *firing rule*:

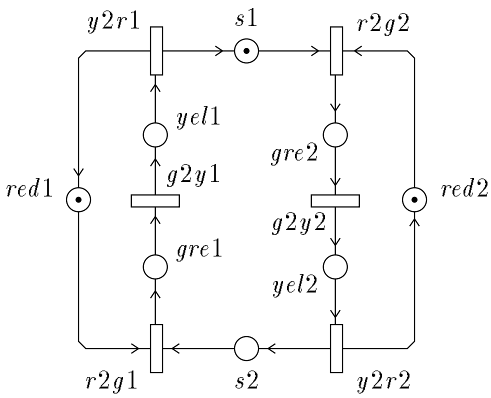

Figure 4: The P/T net of traffic lights at a crossing.

> A transition can fire, or is *enabled*, if each input place has at least as many tokens as there are arrows from this place to the transition. Upon firing a transition, for every arrow from an input place to the transition, one token is taken from the input place. Furthermore, for every arrow towards an output place, one token is added to the output place.

The structure of a P/T net, its initial marking, and the above firing rule uniquely determine the dynamic behaviour of a P/T net. Tokens that are taken from an input place when firing a transition are often called *consumed* tokens; tokens that are added to the output places are called *produced* tokens.

Now, it is easy to understand the behaviour of the net depicted in Figure 4. Initially, there are tokens in the places $red1$ and $red2$, indicating that both traffic lights are red. Since there is also a token in $s1$, the transition $r2g2$, which should be read as "red-to-green-two," is enabled. This means that the second traffic light is allowed to turn green. If the firing rule is applied, the tokens in places $s1$ and $red2$ are consumed and one token is added to place $gre2$. The light has turned green. As a consequence, transition $g2y2$ is enabled. The second traffic light may turn yellow. After applying the firing rule two more times, the second traffic light has turned red again: There is a token in place $red2$. Upon turning red, also a token was produced in place $s2$, thus enabling transition $r2g1$. This means that the first traffic light can now turn green starting its green-yellow-red cycle. The behaviour as described above repeats itself periodically, an unbounded number of times.

Place/Transition nets were introduced as early as 1962 by Petri [13]. Since then, the theory of Petri nets has been extended in many ways and it has been applied to a wide variety of problems. The theory has become so popular for a combination of two reasons: first, the easy-to-understand graphical representation of Petri nets, and, second, the possibilities for formal analysis of Petri nets. In recent years, interest increased even more due to the development of many automated tools based on Petri nets. The remainder of this subsection explains some simple analysis techniques that can be applied to P/T nets. As mentioned, the next subsection describes ExSpect which is one of the tools based on Petri nets.

Since the introduction of Petri nets, many analysis techniques have been developed. A good survey of the available techniques can be found in [11]. The following briefly describes two of the most commonly used analysis techniques.

Place invariants. A place invariant, or S-invariant, is a weighted set of places, such that the weighted sum of tokens in these places is constant. That is, the weighted sum of tokens is independent of any firing.

S-invariants can be used to verify many useful properties of system specifications. The crossing in Figure 4 has, for example, the following S-invariants:

$gre1 + yel1 + red1 = 1$,

$gre2 + yel2 + red2 = 1$, and

$red1 + red2 - s1 - s2 = 1$.

The first and second invariant show that both lights are either green, yellow, or red. Not really a surprising result. The third invariant, however, implies that always at least one of the traffic lights is red. This is a very useful result: It means that the crossing is safe!

An important result in Petri-net theory is that S-invariants can be calculated in a very straightforward way using linear algebra. The details are omitted (see for example [11]). What is important is that it is relatively easy to implement the calculation of S-invariants in an automated tool. Thus, it can be used for automatic verification of system properties.

Transition invariants. A transition invariant, or T-invariant, is a weighted set of transitions, such that all weights are non negative and the marking does not change when all transitions are fired as many times as their weights. That is, the state of the Petri net does not change.

Again, the example of Figure 4 can be used to clarify the notion of T-invariants. The P/T net shown in this figure has

$r2g1 + g2y1 + y2r1 + r2g2 + g2y2 + y2r2$

as a T-invariant. It means that the traffic lights return to the initial state each time both lights have performed their green-yellow-red cycle.

As for S-invariants, T-invariants can be calculated using linear algebra. Therefore, T-invariants are also well suited for automatic verification.

3.2 ExSpect

ExSpect, the Executable Specification tool [1], is a toolkit that is based on high-level Petri nets. High-level Petri nets extend P/T nets to data, time, and hierarchy. The basic features of ExSpect can be best explained using the example of the traffic lights at a crossing. A detailed description of the formal framework behind ExSpect can be found in [7].

ExSpect extends P/T nets to data. This means that tokens have *types* and *values*, often called *colours* in Petri-net literature. In addition, each place in the net also has a type, restricting the type of tokens allowed in that place. When firing a transition, the number of tokens produced and their value may be determined by the value of the consumed tokens. See [7, 8] for a detailed treatment of the theory of coloured Petri nets.

The traffic-light example shows simple use of data in Petri nets. It is no longer necessary to have separate places for each colour that a traffic light may have. Instead, the colour can be stored in the tokens in the net. In ExSpect, one could define the type `colour` and its constants as follows.

```
type colour := string;

green  := 'green'  : colour;
yellow := 'yellow' : colour;
red    := 'red'    : colour;
```

ExSpect provides possibilities to hide the implementation of type `colour`, thus turning it into an *abstract data type*. This means that only the constants `green`, `yellow`, and `red` can be used as elements of type `colour`. Other strings are not allowed.

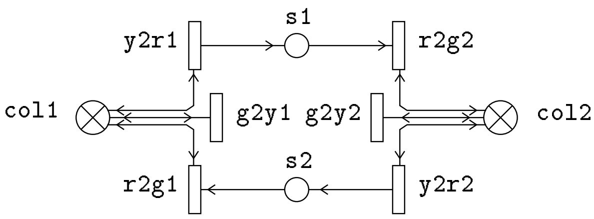

Figure 5: A crossing in ExSpect using data.

The traffic lights at a crossing can now be specified as in Figure 5. It shows two crossed circles, called *stores*. A store is a special kind of place that occurs very often in ExSpect specifications. It always contains exactly one token. Every time a transition consumes this token, it must be replaced by a new token. This requirement is depicted graphically by a bidirectional arrow. Since stores are places, they must have a type. As expected, the two stores in Figure 5 are of type `colour`. A store can be considered a *variable* as it occurs in, for example, C or Pascal programs or LSCs. The two other places are of type `token`, which contains only one value `tokenval`.

As mentioned, in coloured nets, the number of tokens consumed and produced by a transition and their values may depend on the value of the input tokens. Therefore, ExSpect generalizes transitions to so-called *processors*. Processors are transitions whose exact behavior is specified in a functional programming language [1, Chapter 4]. This language has functions to test and modify the value of tokens. It also has a conditional statement which is used to vary behaviour depending on values of tokens. It is not always necessary to specify every individual processor. For example, in Figure 5, the two processors `y2r1` and `y2r2` are instances of the processor `y2r` which is specified as follows:

```
proc y2r
  [store col: colour, out s: token |
   pre col = yellow] :=
col <- red, s <- tokenval
```

The specification of `y2r` states that it is connected to one store `col` of type `colour` and one output place `s` of type `token`. It also has a precondition saying that the value of the token in `col` must be `yellow`. If the precondition is satisfied, the processor may fire, consuming the token in the store. Upon firing, it produces two tokens, one in `col` with value `red`, and one in `s` with value `tokenval`. Informally, the processor may only fire when the traffic light is yellow. As a result from firing the processor, the traffic light turns red, at the same time signalling the other traffic light that it may turn green.

The specification of the other processors is very similar and, therefore, omitted. It is easy to verify that the net in Figure 5 properly models the behaviour of the traffic lights as explained in the introduction to the previous subsection.

Note that the specification of `y2r` does not contain a conditional statement. This means that its behaviour does not depend on the values of the consumed tokens. Consequently, it is very similar to the transitions known from P/T nets. For this reason, a processor whose behaviour is not conditional is often called a transition. The class of transitions is important, because the analysis techniques described in the previous section can be applied to ExSpect specifications consisting only of transitions. ExSpect has a tool to transform any specification into such a specification with the same dynamic behaviour.

The second extension to P/T nets is *time*. In ExSpect, every token is assigned a time stamp, indicating the time that the token becomes available for consumption. A processor is enabled, only if in every input place at least one token is available for consumption. A transition fires as soon as it becomes enabled. The firing rule is said to be *eager*. Upon firing, output tokens to ordinary places can be given a delay using the `delay` statement. Tokens to stores may never be delayed, because it is assumed that there is always exactly one token available in a store.

As an example, the output token in the specification of the processor `y2r` can be delayed.

```
proc y2r
  [store col: colour, out s: token |
   pre col = yellow] :=
col <- red, s <- tokenval delay 2
```

As a result, when the first traffic light turns red, the second light does not turn green immediately, but only after a delay of two time units. Thus, a margin is built in, allowing all cars that are still on the intersection to cross safely. It is also possible to specify the amounts of time that a traffic light must be green, yellow, or red. However, since tokens in a store may never be delayed, it is necessary to add extra places. It is left as an exercise for the interested reader.

Note that the extension to timed nets does not affect the use of place and transition invariants. These invariants only depend on the *causal* relationships between the transitions in the net. They are independent of the specific times at which transitions fire.

Finally, ExSpect adds hierarchy to P/T nets. When designing complex systems, it is essential that a specification language allows for hierarchical design. Starting from a high level of abstraction, the system designer can gradually add more detail to a specification, thus limiting the amount of detail that he or she has to cope with at once.

For example, at a high level of abstraction, a crossing consists of two traffic lights that each have a colour and are mutually synchronized to avoid unsafe situations. The exact implementation of a traffic light is not yet important at that level of abstraction. The ExSpect specification would typically be as in Figure 6. It shows one new element of ExSpect, namely *systems*. Systems are depicted by squares. A system can be considered as a high-level processor that itself must be specified in terms of places, processors, and lower-level systems. In the example, `tl1` and `tl2` are both instances of the system `tl`, which models a traffic light.

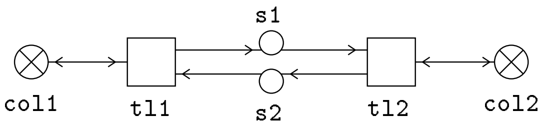

Figure 6: A high-level ExSpect specification of a crossing.

When defining a system such as `tl`, it is important that its interface to the environment is defined. This interface can be specified using so-called *pins*. Pins have types and must be connected to places or stores of the corresponding type when a system is used as part of a specification. There are three types of pins: input pins, output pins, and store pins. They correspond to input places, output places, and stores respectively. Figure 7 shows the specification of the system `tl`, which models a traffic light. It needs no further explanation.

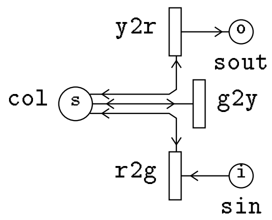

Figure 7: A traffic light in ExSpect.

In order to calculate place and transition invariants, a hierarchical net can be *unfolded*. Each system is replaced by its implementation, yielding a net that consists of only processors. The result can then be transformed to a a net consisting of transitions only, using the tool mentioned earlier. It is easy to see that the result of unfolding the net in Figure 6 is the net shown in Figure 5.

This concludes the introduction to ExSpect. It is sufficient to understand the remainder of this paper. For a more detailed explanation of the tool ExSpect, the reader is referred to [1]. An extensive treatment of the theory behind ExSpect can be found in [7].

4 ISL and ExSpect

The previous two sections introduced ISL and ExSpect respectively. As the observant reader might have noticed, many constructs in ISL map almost directly to ExSpect constructs. This section discusses the details of such a translation from ISL to ExSpect. Since ExSpect is a graphical simulation package based on a formal mathematical theory, a translation from ISL to ExSpect is useful for analyzing ISL specifications of railway interlockings. Thus, signalling practice can be optimized and possible mistakes can be corrected. In this way, the safety of passengers, trains, and goods can be further improved. In addition, higher flexibility in scheduling trains might be achieved. At the end of this section, the results of simulating and analyzing an experimental implementation in ExSpect of a fragment of the ISL specifications from [3] are discussed.

4.1 A Translation from ISL to ExSpect

As explained in Section 2, ISL has a static component, namely the four languages for specifying routes and route elements, and a dynamic component, namely telegrams. Obviously, telegrams correspond to tokens in ExSpect. Furthermore, it seems logical to translate the two fundamentally different hierarchical levels in ISL, namely the level of Logical Element Connection Layouts and the level of Logic Sequence Charts to separate hierarchical levels in ExSpect as well. As we will see shortly, this is a feasible approach provided that an intermediate level is introduced to take care of scheduling all telegrams in an LSC. The latter is necessary to guarantee that, at any time, at most one telegram is active within each LSC.

Figure 8 shows an ExSpect specification of the LECL shown in Figure 1. Each generic element in LECL corresponds to a *system* in ExSpect. Since LECL has six elements, this yields also six ExSpect systems. Two of these systems appear in the example; the two signals are instances of one generic system `signal`. The pins of the LECL elements translate to ExSpect pins. An entry pin Na of an LECL element becomes an input pin `ain` of the corresponding ExSpect system; an exit pin Xb becomes an output pin `bout`. These pins must be specified in the system definitions. In Figure 8, they appear implicitly as places. The place `bs1_at`, for example, connects the output pin `bout` of `signal1` to pin `ain` of `track`.

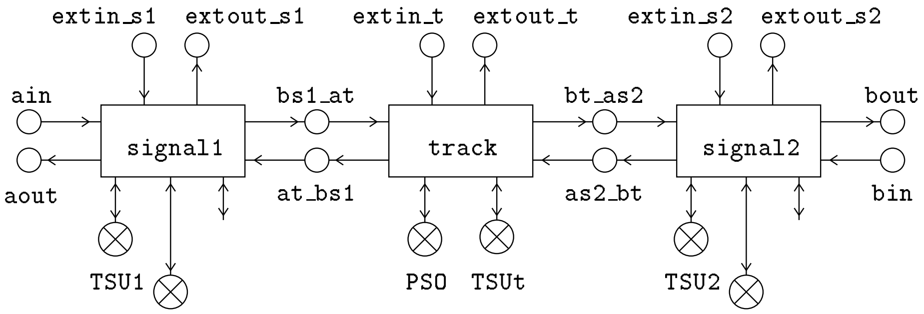

Figure 8: A simple route in ExSpect.

LECL elements also have pins to the environment which consists of the control level and trackside devices. For the sake of simplicity, in Figure 8, no distinction is made between these two parts of the environment. Each ExSpect system has an external input place for receiving telegrams from the environment; it has an external output place for telegrams to the environment. If it is desired, it is straightforward to differentiate between control level and trackside devices.

In addition to pins to the environment, the LSCs that implement an LECL element might have variables that can be changed by the environment. Since these variables are also some form of communication with the environment, they are translated to store pins of the corresponding ExSpect systems. In Figure 8, they are shown at the bottom side of each system. A signal has more store pins than the two pins that are shown; this is depicted by the unconnected bidirectional arrow. Note that, in ISL, no information of variables is known at the LECL level. In ExSpect, variables that can be changed by the environment are made visible at the route level, because in this way all user interaction during a simulation session takes place at the route level.

In LECL, the six generic elements are each implemented by one or more LSCs. Therefore, it seems logical to implement each of the corresponding ExSpect systems by a lower-level net that is a translation of the LSCs. However, ISL imposes an important restriction on the dynamic behaviour of an LECL: At any time, at most one telegram may be active in each of the LSCs. In ExSpect, this restriction must be enforced explicitly. Therefore, an intermediate level as shown in Figure 9 is necessary.

Figure 9 shows the implementation of the system `track`. It consists of two systems, `interface` and `trackLSC`. The latter is a translation of the LSCs specifying the LECL-element track. It is connected to the system `interface` and to two store pins that are exported to the route level. By replacing `trackLSC` by the ExSpect implementation of the LSCs of any of the other five LECL elements, the corresponding ExSpect system is obtained.

System `interface` provides an interface to the environment of the LSC. It is the same for all six elements. The tasks of `interface` are scheduling the telegrams inside `track` and routing incoming and outgoing telegrams. It has, among others, input pins `ain`, `bin`, `cin`, and `din`, one for each possible side of an LECL element. It also has output pins for each side. Since a track element only has sides "a" and "b," the pins for these directions are exported to the higher level; the pins for the other directions are connected to dummy places. System `interface` also has pins for communicating to the environment and to `trackLSC`.

The details of the implementation of `interface` are not important. It is only important to know that it has a subsystem `scheduler` which takes care of its main task, namely scheduling incoming telegrams and telegrams generated internally such that at most one telegram is active in `trackLSC`. ISL does not specify an algorithm. By implementing the scheduling algorithm in a

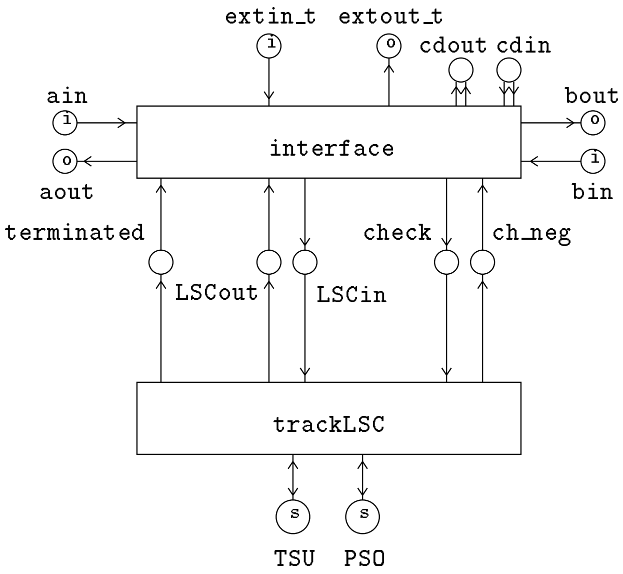

Figure 9: The intermediate level.

separate ExSpect system, it is easy to experiment with different algorithms. See the appendix for more details about system `interface`.

In the experimental implementation that has been made to validate the approach discussed in this paper, the choice was made to give priority to telegrams generated internally and to store incoming telegrams in a FIFO queue. As explained, inside an LSC and thus inside `trackLSC`, telegrams may be generated based on the values of timer variables. The scheduler uses the place `check` to test timer variables periodically. If it has the correct value, a telegram is generated and allowed to proceed immediately until it leaves `trackLSC` via `LSCout` or `terminated`. The latter indicates that the telegram is terminated inside the LSC and does not need to be redirected to the environment. It is important that the scheduler knows when a telegram is terminated, because then another telegram is allowed to enter `trackLSC`. If a timer variable does not have the correct value, `trackLSC` returns a negative acknowledgement via `ch_neg`. If there are no internal telegrams available, the scheduler allows the first external telegram in the queue to enter `trackLSC` via `LSCin`, provided of course that an external telegram is available. Such a telegram proceeds its way through `trackLSC` until it leaves either via `LSCout` or `terminated`.

The third and final part of the translation of ISL to ExSpect is a translation of the LSCs that are used to build the generic elements. This can be done in a very straightforward way. As explained in Section 2, an LSC essentially consists of operations on variables and telegrams. Each one of these operations can be translated to (an instance) of an ExSpect processor. The result of translating the LSC fragment of Figure 2 is shown in Figure 10.

A telegram enters the LSC via the pin `LSCin`. The processor `InSwitch` then tests its name and moves it to the corresponding place. When a processor receives a telegram from its input place, it performs an operation on either the telegram or a variable (store), and then moves the telegram to one of its output places. Since ExSpect often has more than one symbol available for each element of a specification, different symbols have been used for processors that only test values and processors that actually change values. The former are depicted by squares with a filled triangle in the bottom-right corner; the latter are depicted by triangles. If a processor tests the value of a variable to determine in what direction the telegram should proceed, by convention, the telegram

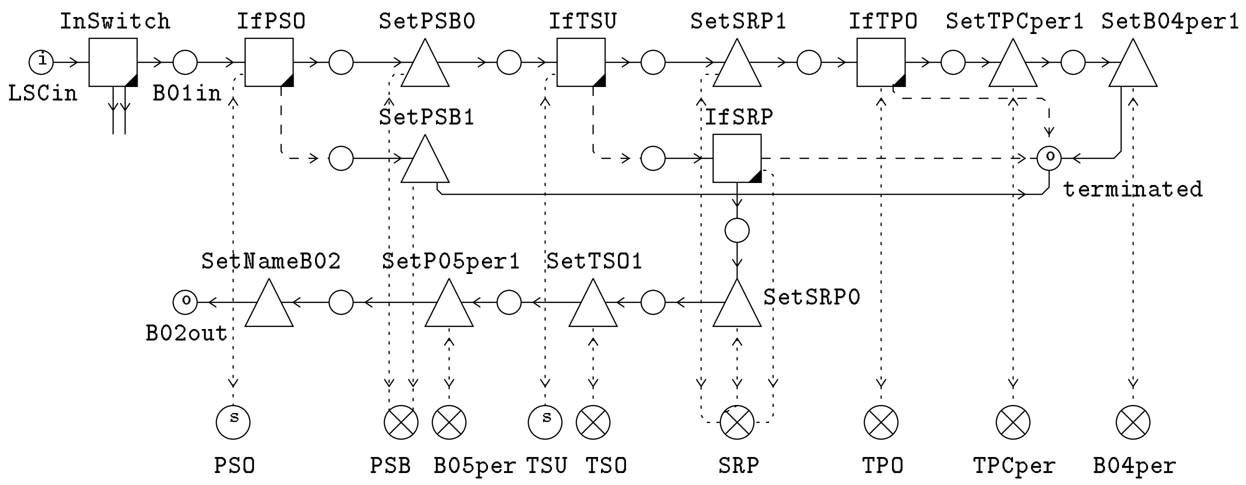

Figure 10: A fragment of an LSC in ExSpect.

follows the solid line if the test yields true, and the dashed line otherwise. A telegram may leave the LSC either via `terminated` or via one of the pins `NAMEout`, where `NAME` is the name of the telegram. All pins of the latter type are connected to the place `LSCout` at the intermediate level.

A few final remarks are in order. First, note that before a B01 telegram leaves the LSC, the name of the telegram is explicitly set to B02. Second, it is in general also necessary to set the current direction of a telegram to the side at which it must leave the LSC. However, in the example, this operation does not appear, because the side at which a B02 telegram leaves the LSC is the same as its current direction. Third, Figure 10 does not show how telegrams are generated in the LSC. Variables that have a name with suffix "per" may, depending on their value, periodically generate telegrams. This can be implemented by defining a subsystem which has pins connected to `check` and `ch_neg` at the intermediate level, the associated variable, and one ordinary output place. If a token is received via `check`, this system tests the value of the associated variable and either generates a telegram or returns a negative acknowledgement via `ch_neg`. The details about the working of these and the other timer variables can be found in [3]; the details about their translation to ExSpect can be found in the appendix.

4.2 Simulation and Analysis in ExSpect

In order to validate the approach pursued in this paper, an experimental translation of a fragment of ISL has been made. This section discusses the results of simulating and analyzing ISL in ExSpect.

Figure 11 shows a snapshot of a simulation session. The window in the center of the screen shows the route which is also shown in Figure 1. There is one token, or telegram in ISL terminology, in place `bs1_at` and there are two tokens in `extout_s1`. Furthermore, each store contains one token. Since most of the communication is between the two signals and the track segment, the history of the places `bs1_at`, `bt_as2`, `as2_bt`, and `at_bs1` as well as the last token that resided in these places are shown in the windows around the center window. The window titled `First of bs1_at`, for example, tells us that the token in `bs1_at` is an A03 telegram, which is represented by the number 3 in the `name` field. Furthermore, it has direction "b," represented by 2, and it has a data set which consists of only one element RT (Route Type) which is equal to 1, indicating that a normal route is being set. The window `bs1_at` shows that this telegram is the only token of the history of `bs1_at`. The time at which it becomes availabe for further processing is 4.4 seconds after the start

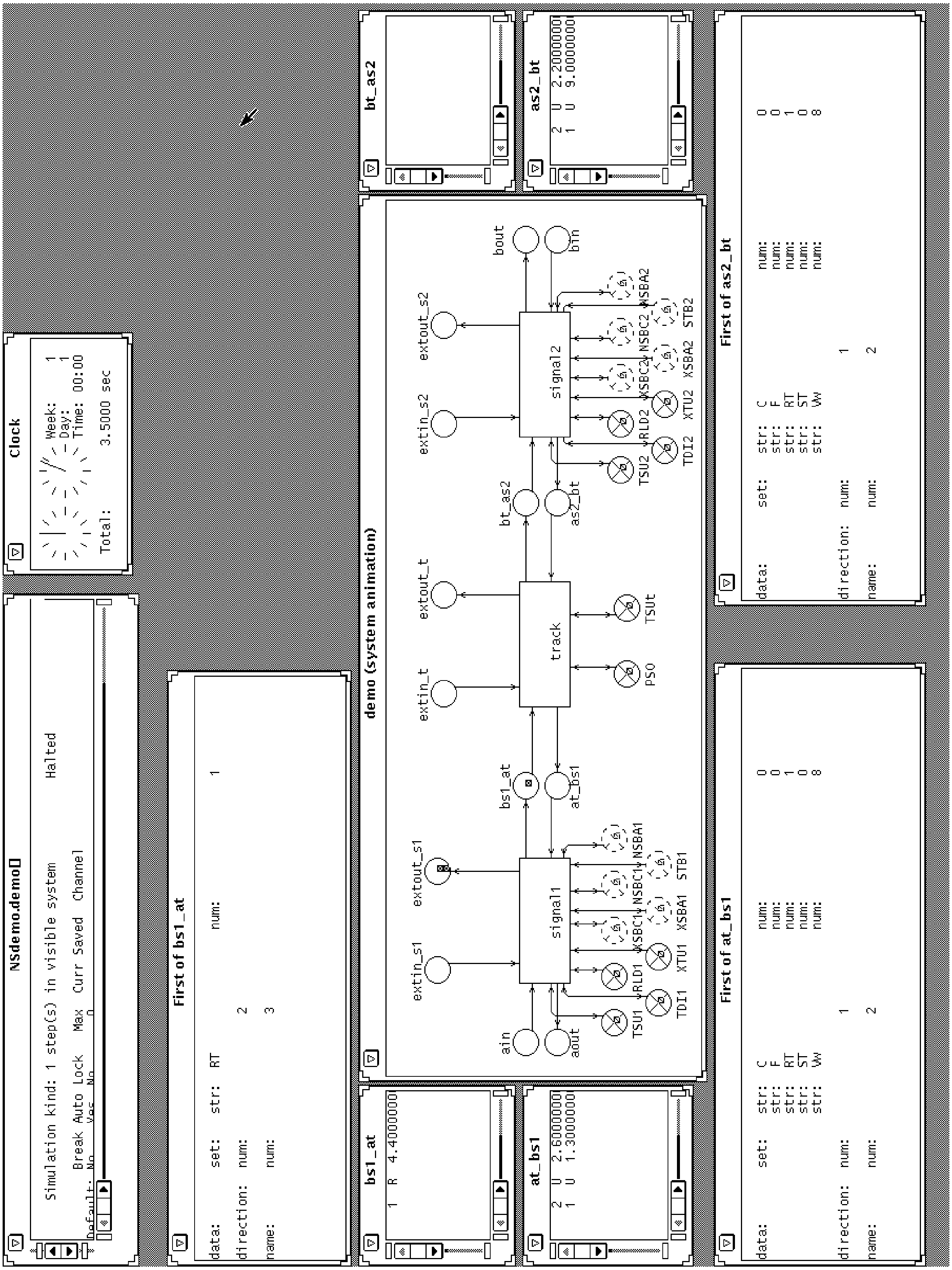

Figure 11: A snapshot of a simulation session.

of the simulation. The clock at the top of the screen shows that the simulation time at the time of the snapshot was 3.5 seconds. The window in the top-left corner of the screen contains all the information of the simulation in a textual format.

As the brief explanation above makes clear, ExSpect allows the designer to visualize the flow of telegrams, their timing behaviour, and their contents. In addition, ExSpect has several features which have not yet been explained but may be very useful for simulating ISL specifications. The contents of an arbitrary token can be shown by just clicking the place in which it resides, provided that it is the last token that entered the place. The entire history of a place can be accessed via mouse driven menus. The contents of each of the tokens in the history can then again be visualized by just clicking the token. It is also possible to add tokens to or delete tokens from places, thus changing the course of the simulation interactively. Although all these features may be very handy, perhaps the most important feature of ExSpect is the possibility to switch between different hierarchical levels at any time during the simulation. By clicking the system `track`, for example, its implementation appears in a separate window. In this way, it is possible to switch between the route level and the LSC level of the simulation any time it is desired.

The previous paragraphs describe simulation of ISL specifications in ExSpect. Although simulation is an important goal, it is not the ultimate goal. The main reason for investigating the possibility to translate ISL into ExSpect is that ExSpect is based on a solid mathematical theory. This may be a basis for formal analysis and verification of railway interlockings. As mentioned earlier, however, this is not yet possible for two reasons. First, systems as railway interlockings are still too complex. Second, safety requirements of railway interlockings have not yet been determined and formalized. Nevertheless, the analysis tools of ExSpect prove to be useful.

In the previous sections, the analysis technique using place invariants has been explained. It has also been explained how this technique is implemented in ExSpect. If this technique is applied to, for example, the system `track`, one of the invariants that is found has the following form:

$$\textit{sum of places in } \texttt{trackLSC} + \textit{some other places} = 1 \ .$$

This means that there is always *exactly one* token in any of the places listed in the sum. Since all the places in the LSC are part of this sum, this means that, at any time, at most one telegram is active in the LSC. This is a requirement that ISL imposes on specifications. ExSpect tools have been used to show that this requirement is maintained by the translation to ExSpect. Although this result may appear somewhat trivial, it is important to note that it is based on a purely mathematical analysis of the ExSpect specification. It may be a basis for further analysis of ISL specifications.

To end this section, we compare ExSpect to the ISL simulation package that is being developed at NS (see Section 2.3). ExSpect has two major advantages over the ISL package. First, in ExSpect, it is possible to switch between different levels. Therefore, an ISL specification can be simulated at both the LECL and the LSC level. Second, ExSpect is based on the theory of Petri nets which provides techniques for formal analysis of ISL specifications. A few minor advantages are that it is possible to actually visualize the flow of telegrams. Furthermore, it is straightforward to replace elements as the scheduler and thus experiment with different scheduling algorithms. Of course, the ISL package also has some advantages. The drawing package is better suited to design LSCs than ExSpect. Furthermore, during the simulation, it uses colours to identify states of elements in a route. Thus, it presents a higher-level view of the state of a route than ExSpect, which is easier to understand for non specialists.

Summarizing, the strengths of ExSpect are the theoretical basis and the possibility to simulate a specification across hierarchical levels; the strength of the ISL package is mainly its customization to designing and simulating interlockings. If ExSpect and the existing ISL package can be integrated, the strengths of both packages can be combined.

5 Future Work

The research described in this paper has raised many interesting questions and problems. In this section, some of them are discussed briefly.

An ISL-to-ExSpect compiler. The first step towards incorporating ExSpect into the ISL design and simulation environment is an ISL-to-ExSpect compiler. Such a compiler is the basis for any future work on using ExSpect for simulating and verifying ISL specifications. In [12], the possibilities for using the ASF+SDF environment [10] to create such a compiler are described. It appears that the translation proposed in this paper can be automated without too many difficulties.

Coloured invariants. An interesting extension to ExSpect would be, the use of colours, or data, in invariants. Place and transition invariants as explained in Section 3 cannot handle data. In [8], invariants are generalized to coloured nets. Design/CPN [6] is a tool that implements coloured invariants. A disadvantage of coloured invariants is that it is no longer possible to calculate all invariants using linear algebra. It is only possible to verify whether a specification satisfies a given invariant.

Hierarchical invariants. As explained, ExSpect unfolds the hierarchy of a specification before calculating invariants. However, this is not always necessary nor desirable. Consider the hierarchical net in Figure 12. It is yet another specification of the traffic lights. At the highest hierarchical level, only the colour red is visible. The colours green and yellow are hidden at a lower level. If one only wants to know whether, at any time, at least one light is red, such a specification is feasible.

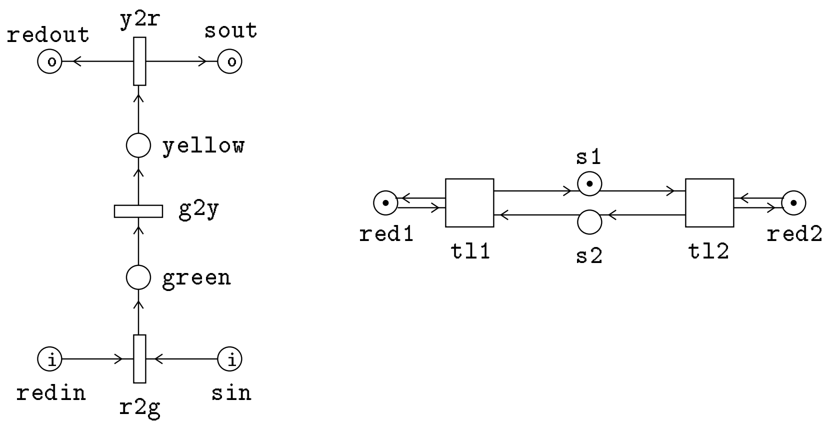

Figure 12: Determining place invariants in a hierarchical way.

It is easy to verify that the traffic light at the left has the following place invariant:

$$\texttt{redin} - \texttt{sin} + \texttt{redout} - \texttt{sout} = c \ , \tag{1}$$

for arbitrary non negative constant c, which is determined by the initial marking of the high-level net. Note that the set of places in this invariant only contains *pins*. The question is: “Can this information be used to determine invariants of the specification at the highest level?" The answer is yes. Consider the following equation:

$$\texttt{red1} + \texttt{red2} - \texttt{s1} - \texttt{s2} = 1 \ , \tag{2}$$

If this equation is *projected* onto the pins of, for example, tl1, we obtain the following equation:

$$\texttt{tl1.redin} + \texttt{tl1.redout} - \texttt{tl1.sin} - \texttt{tl1.sout} = 0 \ ,$$

which is an invariant of the traffic light (see (1)). The same result is obtained for tl2. Therefore, (2) is an invariant of the entire system. We may indeed conclude that, at any time, at least one of the two traffic lights is red.

It does not seem difficult to generalize the example above to arbitrary hierarchical nets. Furthermore, it appears straightforward to implement the strategy outlined here in ExSpect. Since it is a modular approach, it means that place invariants only need to be calculated once for each system definition, thus speeding up the calculation, and simplifying the presentation of invariants to the user.

Unfolding finite colour sets. In ExSpect, data can be used to simplify many specifications. A disadvantage is that data complicates analysis of a specification. However, often it is possible to transform a specification to an equivalent specification using less data or even no data at all. As the following example shows, data types that are *finite* can often be *unfolded*.

Consider again the specification of a crossing with traffic lights in Figure 5. The stores col1 and col2 are of type colour which only contains three values. Therefore, col1 can be unfolded into three separate places, gre1, yel1, and red1; the same can be done for col2. The transitions that are connected to any of the stores must also be adapted. For example, the precondition "col = yellow" of the transition y2r1, whose definition is given on Page 9, translates to a connection from the new place yel1 to y2r1; the output "col <- red" becomes an arrow from y2r1 to place red1. The same transformations can be made for all other transitions, thus making the stores col1 and col2 superfluous and removing all data from the specification. It is easy to verify that the result of these transformations is the P/T net shown in Figure 4.

It is not difficult to do the transformations sketched above automatically. As a result, it is possible to calculate many more invariants of a net. In ExSpect, it is not yet possible to verify formally that the specification in Figure 5 yields a safe crossing. That is, that always one of the traffic lights is red. However, by unfolding the type colour, we have proven this property, because the resulting P/T net has the desired place invariant.

The previous four paragraphs all describe problems that are relatively clear. It seems straightforward to solve any remaining theoretical problems and implement the suggested extensions to the ISL package and ExSpect. The next four paragraphs describe problems that still need a significant amount of theoretical research.

Formalization of safety requirements. It has been mentioned several times before that, to date, the safety requirements of an interlocking are not yet formalized. It is a challenging task to find out what it means in the ISL framework that "no two trains may ever collide."

High-level ExSpect specification of interlockings. Logic Sequence Charts are a very low-level operational specification language for interlockings. This paper shows that it is possible to translate LSCs to ExSpect. However, this does not mean that one automatically gets an ExSpect specification that makes optimal use of the possibilities of ExSpect. In particular for the purpose of formal analysis, a higher-level ExSpect specification seems to be more useful. It appears to be an interesting subject for further research to specify an interlocking directly in ExSpect, in such a way that it is best suited for formal analysis.

Formal verification. The ultimate goal of applying formal techniques to specifying interlockings is formal verification of their behaviour. Of course, Petri-net theory is not the only theory that might be useful. There are many other techniques such as process algebra and temporal logic. Currently, at the department of Computing Science at the Eindhoven University of Technology, a project is under way to integrate process algebra (ACP [2]) and Petri-net theory. Process algebra is known for its powerful verification techniques. Applying the new theory to specifying railway interlockings would be a good opportunity to learn more about the strengths and weaknesses of the theory. The results might be a step towards formally verifying the complete behaviour of interlockings.

Fault-tolerance analysis. Another long term goal is fault-tolerance analysis of railway interlockings. In the current ISL specifications, some fault-tolerance is already built in on an *ad hoc* basis. For example, a track section is only assumed to be unoccupied if the following *two* conditions are satisfied. First, there is no train *physically* detected on the track section. Second, it is also *logically* unoccupied. That is, according to the control logic of the signalling protocol, there is no train on the section. In the future, a formal approach to fault tolerance seems necessary in order to maintain the high safety requirements of railway interlockings.

6 Concluding Remarks

The goal of the research described in this paper was to investigate to what extent ExSpect can be used to improve simulation and analysis of ISL specifications of railway interlockings.

The first few sections explain all relevant parts of both ISL and ExSpect. Furthermore, an introduction to Petri-net theory, which is the theory underlying ExSpect, has been given. Section 4 describes an approach to translating ISL into ExSpect. Part of an ISL specification has been translated to validate this approach. From this, we may conclude that an automated translation from ISL to ExSpect seems possible without too many difficulties. Furthermore, ExSpect appears to have two major advantages over the current ISL simulation package. First, it is possible to simulate both the route level and the LSC level of a specification. Second, since ExSpect is based on a mathematical theory, ExSpect provides a basis for formal analysis of ISL specifications.

The study on translating ISL to ExSpect has raised many questions that deserve to be studied in the near future. Section 5 describes the most interesting ones. The basis for all future work is an ISL-to-ExSpect compiler. Furthermore, several short-term extensions to ExSpect have been suggested, as well as some longer-term research projects on both ISL and ExSpect.

Acknowledgements

The authors would like to thank Bas van Vlijmen for introducing them to the interesting problem of simulating and analyzing railway interlockings. They are also grateful to André Klap and Frits

Makkinga of NS for their willingness to answer many questions and for a demonstration of the ISL software package.

References

1. ASPT. *ExSpect 4.2 User Manual.* Eindhoven University of Technology, Eindhoven, the Netherlands, 1994.

2. J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 1990.

3. J. Berger, P. Middelraad, and A.J. Smith. EURIS, European Railway Interlocking Specification. Technical report, UIC, Committee 7A/16, 1992.

4. J. Berger, P. Middelraad, and A.J. Smith. The European Railway Interlocking Specification. In *Proceedings of The Institution of Railway Signal Engineers*, January 1993.

5. R.N. Bol, J.W.C. Koorn, L.H. Oei, and S.F.M. van Vlijmen. Syntax and Static Semantics of Interlocking Design and Application Language. Technical Report P9422, University of Amsterdam, Programming Research Group, 1994.

6. Meta Software Corporation. *Design/CPN Manual.* Cambridge, Massachusetts, USA, 1991.

7. K.M. van Hee. *Information Systems Engineering: A Formal Approach.* Cambridge University Press, Cambridge, UK, 1994.

8. K. Jensen. *Coloured Petri Nets. Basic Consepts, Analysis Methods and Practical Use*, volume 28 of *EATCS monographs on Theoretical Computer Science.* Springer–Verlag, Berlin, Germany, 1992.

9. H.A. Klap. Euris-Simulation Tutorial. Technical report, Ingenieursbureau Nederlandse Spoorwegen, 1994. Preliminary version 0.3.

10. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

11. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

12. L.H. Oei. Pruning the Search Tree of Interlocking Design and Application Language Operational Semantics. Technical Report P9418, University of Amsterdam, Programming Research Group, 1994.

13. C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, Germany, 1962.

14. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS monographs on Theoretical Computer Science.* Springer–Verlag, 1985.

A A Fragment of an Interlocking Specification in ExSpect

The appendix describes the experimental translation of part of the ISL specifications in [3] into ExSpect. The elements track and signal taken from [3] are translated as far as it concerns the telegrams A01–A06, B01–B07, and a few telegrams to and from the environment. Section A.1 discusses the ExSpect specification itself. Section A.2 describes a typical simulation session.

A.1 The ExSpect Specification

Figure A-1 shows the ExSpect specification of a simple route. The dashed stores correspond to variables that can be changed by the control level. In ISL, these variables are marked with black diamonds. The other store variables visible at the route level are directly related to trackside devices.

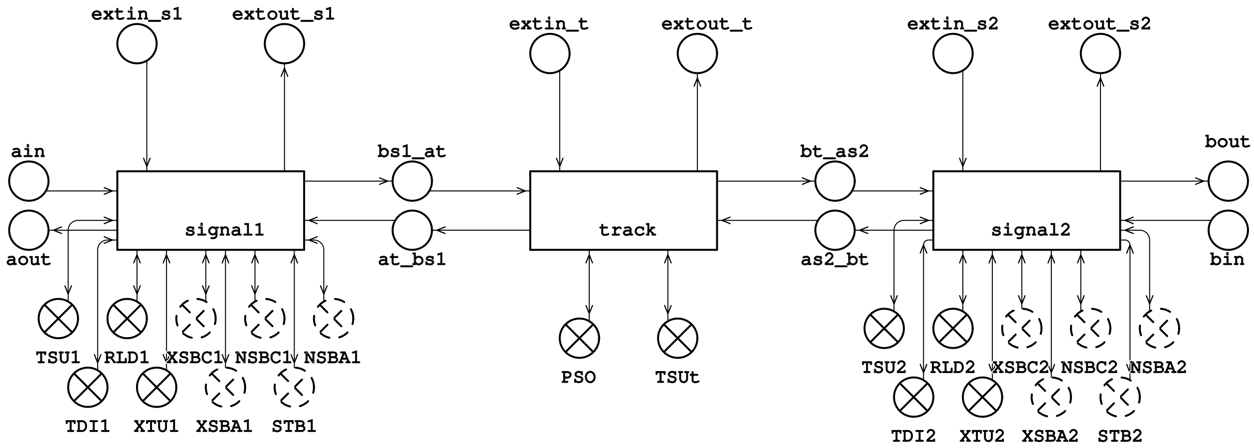

Figure A-1: System `demo`.

Figure A-2 shows the contents of the system `track`. It differs from the implementation suggested in Section 4.1 in several ways. First, the LSC is split into two parts, one for the telegrams A01–A06 and one for the telegrams B01–B07. This is, however, just a technicality to keep the ExSpect systems manageable. Second, the interface between the system `interface` and the systems `tracklscA` and `tracklscB` contains one place for each telegram instead of just the place `LSCin`. The reason for this is that `Interface` already determines the name of an incoming telegram and directs it to a corresponding output pin. The approach explained in Section 4.1 is simpler and yields a uniform system `interface` for each of the six generic elements. Third, all store variables of the track LSC are visible at the intermediate level shown in Figure A-2. As explained, it is more elegant to hide them as much as possible inside the systems `tracklscA` and `tracklscB`.

The system `Interface` is not discussed here. Below, a system `InterfaceProc` is discussed which is used in `signal` and corresponds exactly to the system `interface` that is described in Section 4.1.

Figures A-3 and A-4 show the systems `tracklscA` and `tracklscB`. Except for the processor `InSwitch`, which is missing here, they are exactly as described in Section 4.1. Note that `tracklscB` contains several subsystems `genTimerName`, where `TimerName` is the name of a timer variable. These systems are telegram generators. The details are explained below.

Figures A-3 and A-4 show that the ExSpect specification of an LSC is a fairly complex set of pictures. In order to improve the readability of such pictures, ExSpect provides the possibility

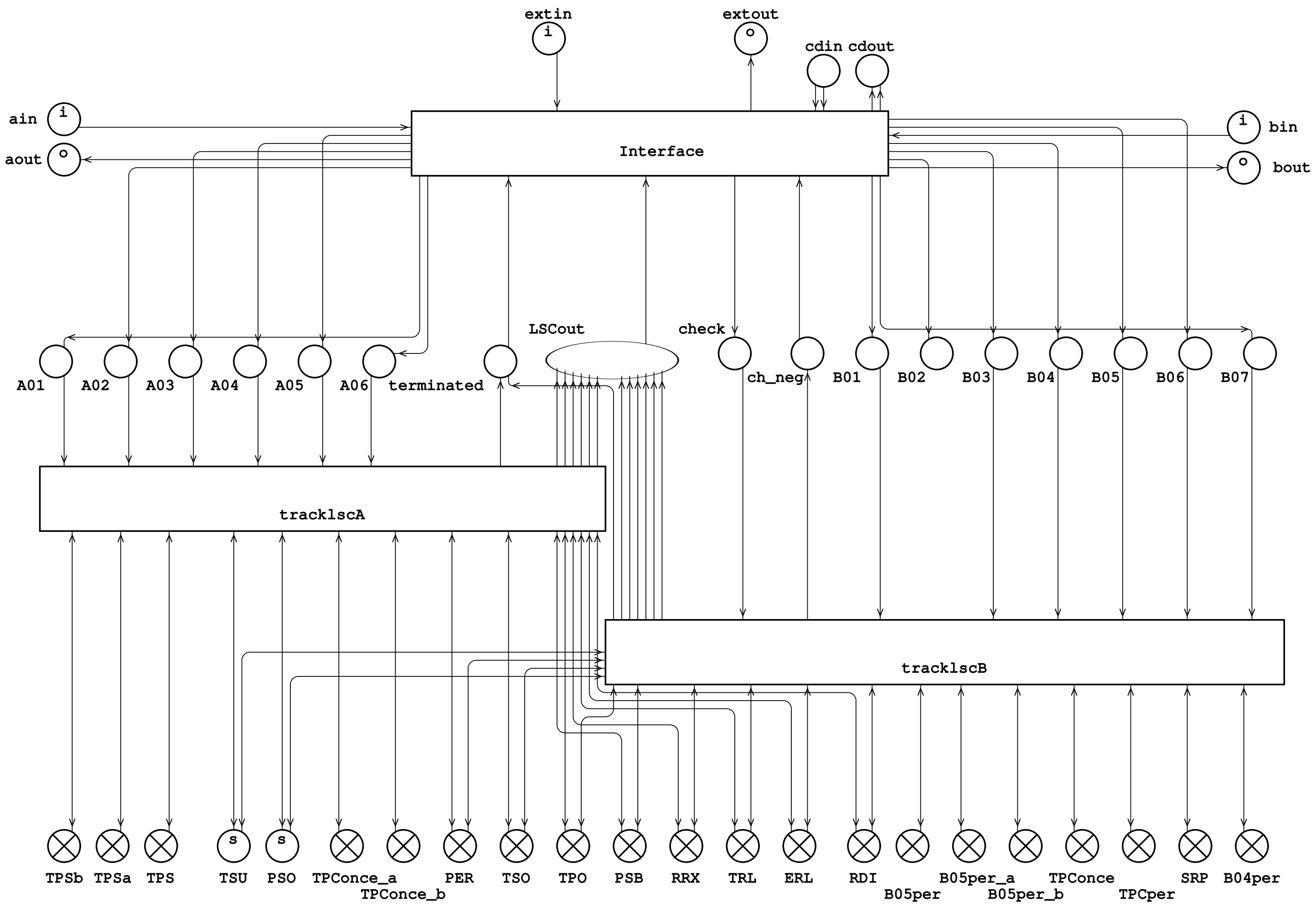

Figure A-2: System `track`.

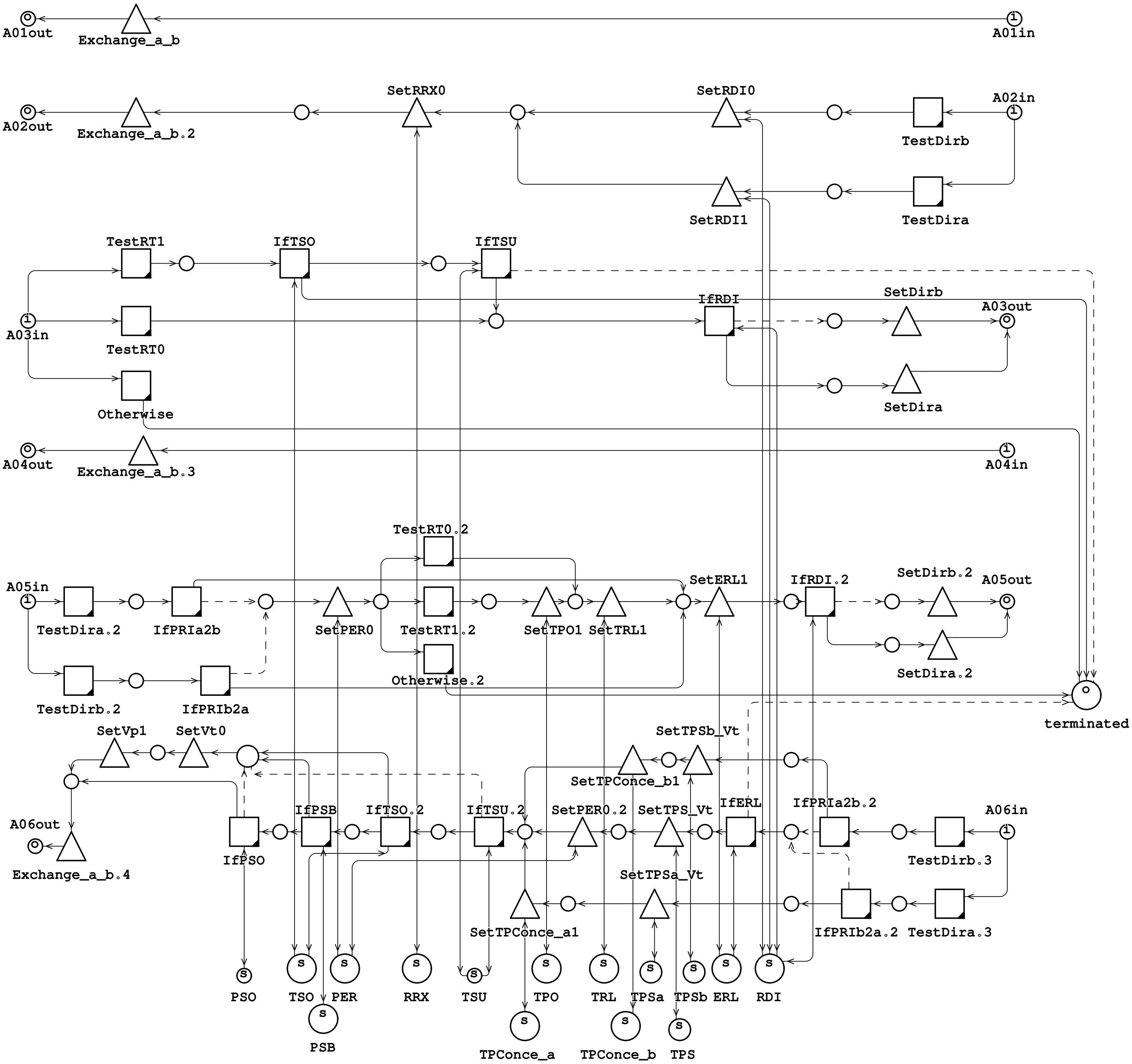

Figure A-3: System `tracklscA`.

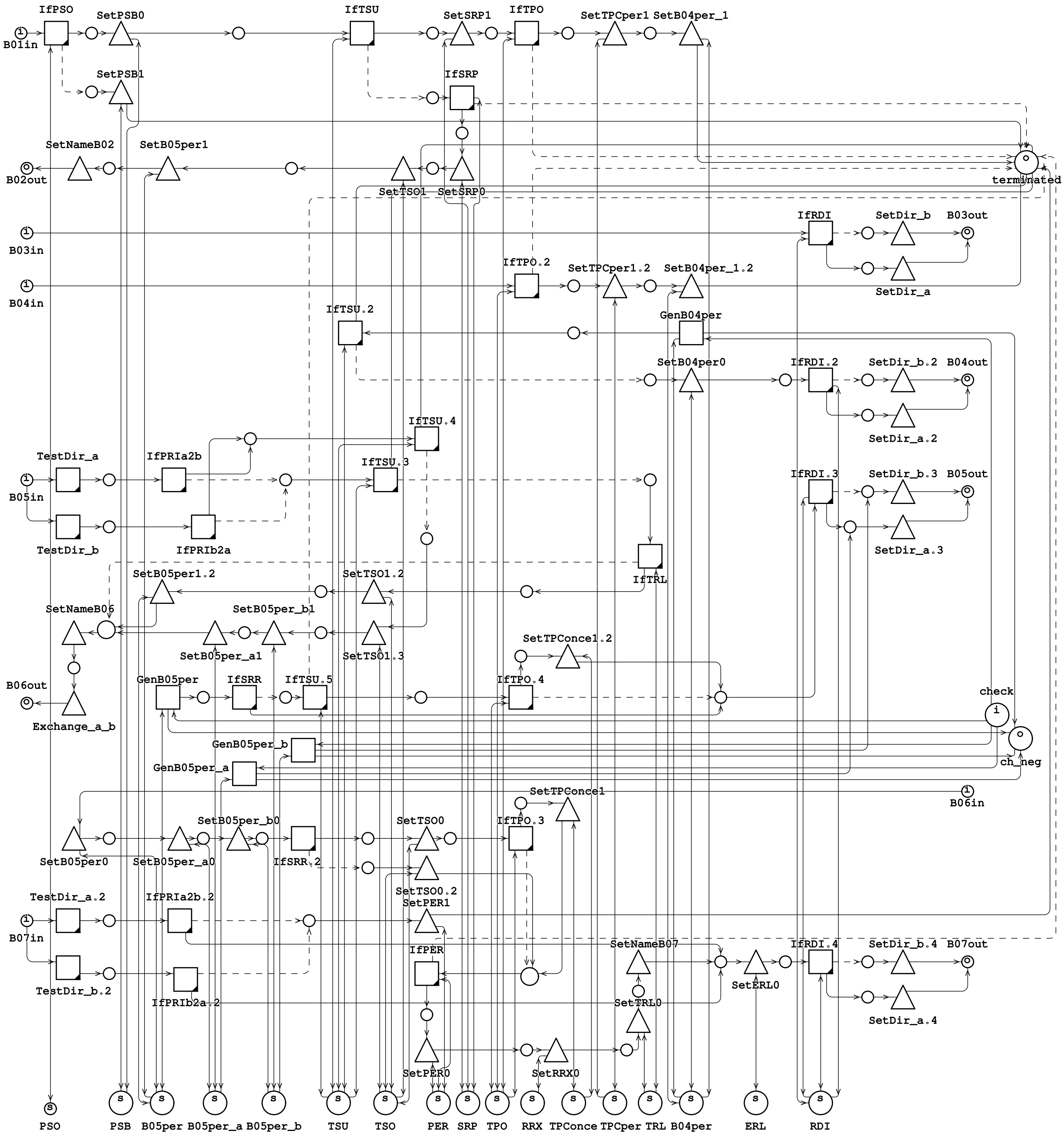

Figure A-4: System tracklscB.

to decompose pictures into several layers. Figures A-3 and A-4, for example, each consist of four layers: one layer containing the control logic (the pins `check`, `ch_neg`, etc.), another layer containing all store pins (variables), a third layer containing all ordinary places and pins, and finally a layer containing all processors and systems. By hiding the appropriate layers, it is possible to focus on specific aspects of a specification. If one, for example, hides the control logic and all the variables in the ExSpect specification of an LSC, the result is a picture that strongly resembles the corresponding ISL specifications in [3].

Figure A-5 shows the contents of system `signal`. It is almost as described in Section 4.1. As explained, `InterfaceProc` is just the system `interface` in Section 4.1. The details are explained below. Again, all store variables of the system `signalLSC` are visible at the intermediate level; they should be hidden inside `signalLSC`.

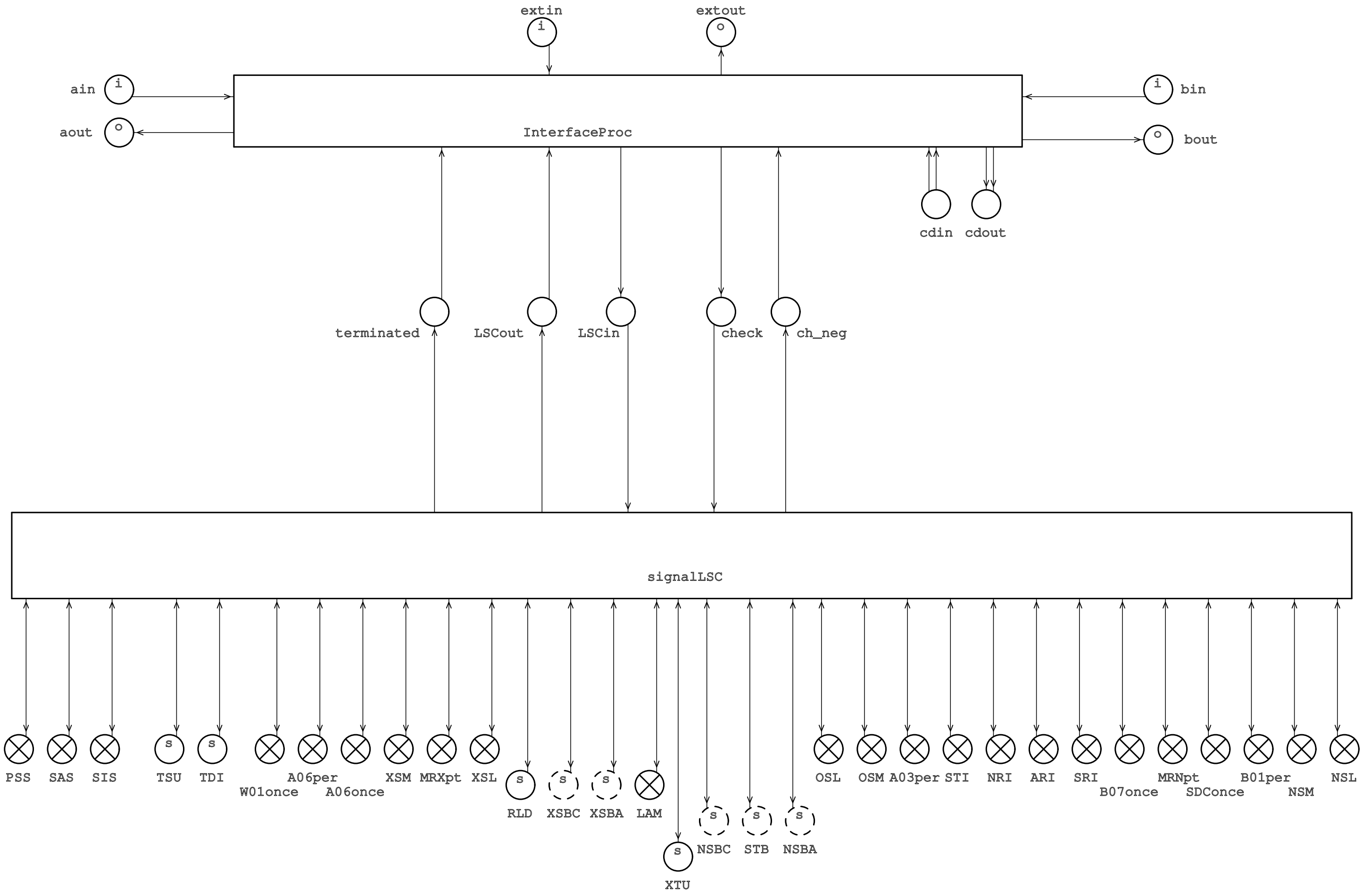

Figure A-5: System `signal`.

Figure A-6 shows the implementation of the system `InterfaceProc`. It corresponds to the system `interface` as described in Section 4.1. System `InterfaceProc` initializes the direction of all incoming telegrams. Incoming telegrams are then stored in `tin`, waiting for the scheduler. The scheduler is triggered by a place `LSCfree`. The details of the scheduling algorithm are explained below. Telegrams that leave the LSC are directed to the correct output pin by the processor `LSCoutSwitch`.

Figure A-7 shows the implementation of the system `signalLSC`. It is completely different from

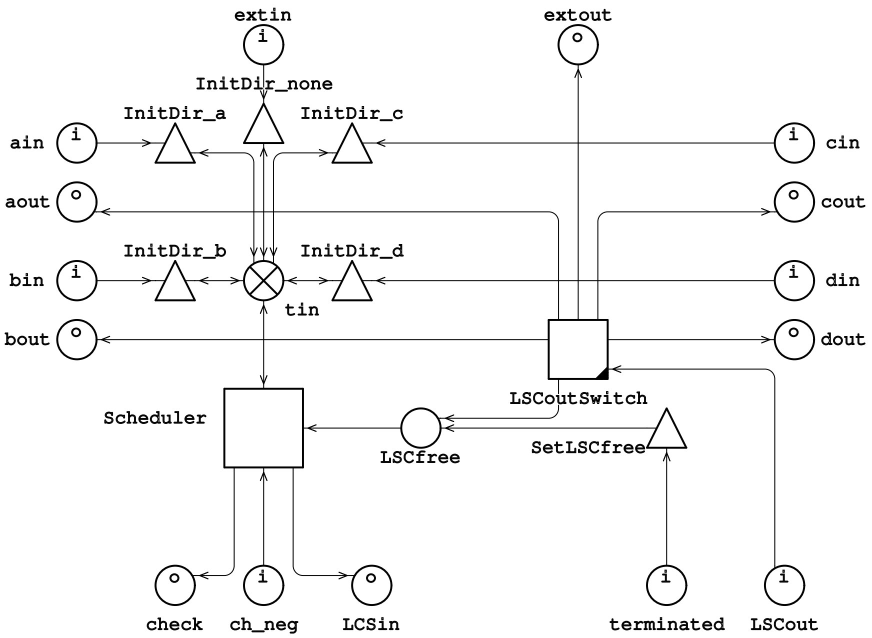

Figure A-6: System `InterfaceProc`.

the implementation suggested in Section 4.1. External telegrams as well as telegrams that are generated internally by the telegram generators on the right, are copied into place `intern_tel`. This place is the input place to a *processor* that implements the signal LSCs. Basically, this processor is a large conditional statement, which defines the effect of all possible telegrams on the state of the LSC. This implementation was chosen for two reasons. First, it served as a test to find out whether it is possible to implement an entire LSC by a single ExSpect processor, using the expressive power of the functional language of ExSpect. This seems possible, although it appears difficult to automate such a translation (see [12]). Second, it saved time, which is a purely pragmatic reason. A disadvantage of this approach is that it is no longer possible to simulate ISL specifications at the LSC level. An advantage is that simulation of complex routes is faster when compared to the other approach.

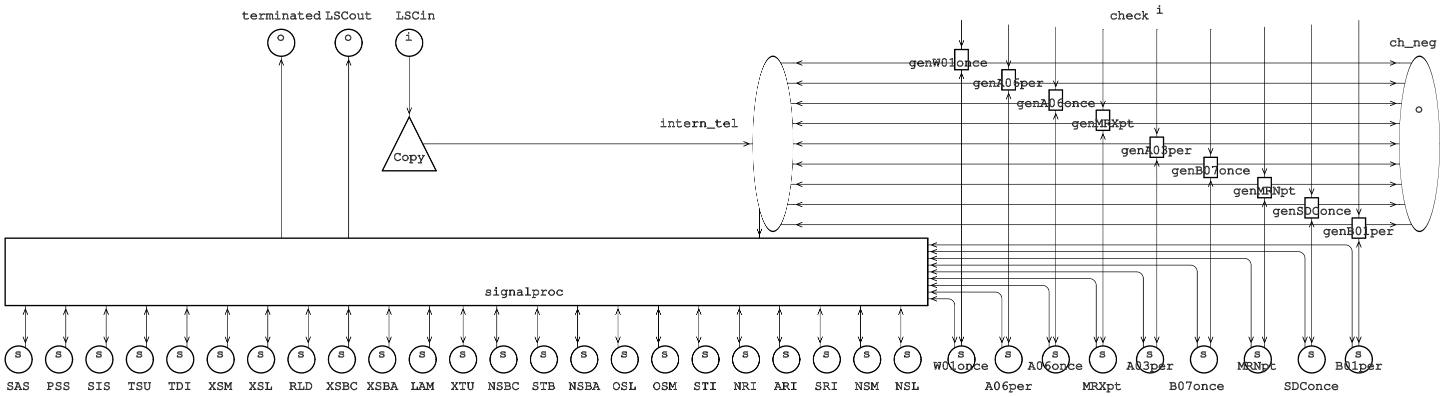

Figure A-7: System `signalLSC`.

Figure A-8 shows an implementation of system `Scheduler`. The processor `NextTelegram` can fire when a token is available in `enabled`. Tokens that are received via either `ch_neg` or `LSCfree` are simply copied to this place. The reason for differentiating between `ch_neg` and `LSCfree` is that other scheduling algorithms might use this information.

Processor `NextTelegram` determines the next telegram that may enter the LSC as follows. First, it checks the store `toCheck`, which contains identifiers of telegram generators in the LSC that still need to be checked for the availability of telegrams. If this store still contains identifiers, `NextTelegram` checks the corresponding telegram generators one by one via the pins `check` and `ch_neg` (see also Section 4.1). After checking a generator, its identifier is removed from `toCheck`. If it does not contain any identifier anymore, `NextTelegram` picks the first telegram from `tin`, provided of course that one is available.

Processor `ResetCheck` periodically produces a token for `ToCheck` that contains the set of all identifiers of telegram generators in the LSC. The time period is a parameter of the system that is specified in ISL.

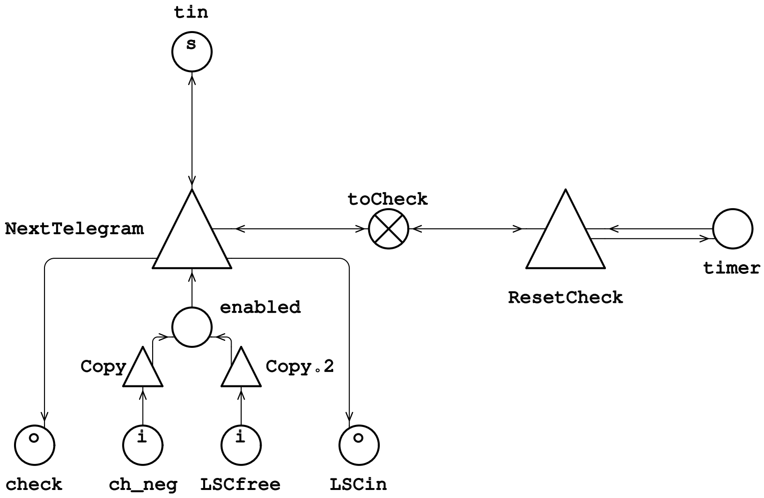

Figure A-8: System `Scheduler`.

The only systems that are not yet discussed are the telegram generators. There are three such systems, namely `gen_tel_once`, `gen_tel_per`, and `gen_tel_pt`. The first one corresponds to some kind of timer which generates a single telegram when the associated variable is true. After generating a telegram, it resets the associated variable. System `gen_tel_per` is the periodic timer that we have seen several times before. System `gen_tel_pt` corresponds to the so-called projected timer, which is an ordinary timer that is set when its associated variable is set to true and generates a telegram upon timing out.

Figures A-9 and A-10 show the systems `gen_tel_once` and `gen_tel_per` respectively. Their implementation is almost identical. Upon receiving a token with the correct identifier via `check`, depending on the value of the variable `ass_var`, either a telegram is generated that enters the LSC via `tout` or a negative acknowledgement is returned to the scheduler via `ch_neg`. The processor `pgen_tel_once` in system `gen_tel_once` resets the variable `ass_var` each time a telegram is generated. Processor `pgen_tel_per` in the periodic timer does not reset the associated variable.

Although several instances of the system `gen_tel_pt` occur in `signalLSC`, they never need to generate telegrams in the part of the LSC that has been implelemented. Therefore, they are not completely implemented. A picture of its current implementation is omitted.

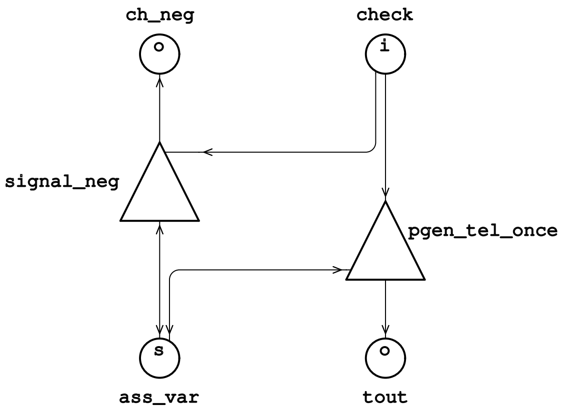

Figure A-9: System gen_tel_once.

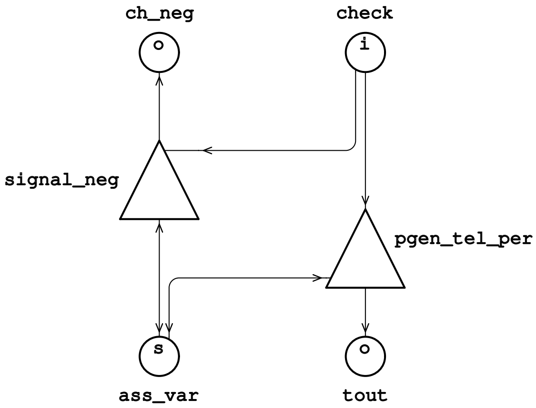

Figure A-10: System gen_tel_per.

A.2 A Typical Simulation Session

This section describes a typical simulation session of the ExSpect specification given in the previous section. It corresponds to the example described in [3, Sections 4.4.1 – 4.5.1]. It is recommended to use [3] for a better understanding of the session described below.

After starting the simulation of the system `demo` and loading the default configuration file, the following steps must be taken.

Route proving

- Add token `(data: {(RT,1),(ST=0),(C=0),(F=0)}, direction: 1, name: 121)` to place `extin_s2`. The numbers in the `direction` and `name` field are numerical representations of "a" and "M01" respectively.
- Open `signal2`.
- Open `InterfaceProc`; open `Scheduler`.
- Simulate until the telegram arrives in the place `LSCin` in `signal2`.
- Open `signalLSC`.

- Simulate until the telegram appears in `LSCout`.
- View the token in LSCout. The `name` field now contains a one, indicating that it is an A01 telegram.
- Close `signal2`.
- Simulate one step; the telegram appears in `as2_bt` in system `demo`.
- Open `track`.
- Simulate one step; a token appears in place `check`.
- Open `tracklscB`.
- Hide planes 2 and 3 respectively, and see what happens.
- Hide plane 1 and show plane 2.
- Show planes 1 and 3.
- Simulate until a token appears in place `A01` in `track`; close tracklscB; open `tracklscA`.
- Simulate one step; the token passes through `tracklscA` and appears in `LSCout`; close `track`.
- Simulate three steps; the token passes `at_bs1` and ends in `extout_s1`.
- View the token in `extout_s1`; its name indicates that it is a P01 telegram.

Route marking

- Add an M02 token to place `extin_s2` with the same specifications as the M01 telegram above, except for the name which is 122.
- Simulate until a token appears in `as2_bt`; the window `First of as2_bt` shows that it is an A02 telegram.
- Simulate until the telegram leaves `signal1` and appears in `extout_s1`.
- View the first token of `extout_s1`; it is a P03 telegram.

Proving for route locking

- Simulate one more step; an A03 telegram appears in `bs1_at`. This telegram was generated by a periodic timer which was activated by the A02 telegram.
- Simulate until the telegram appears in `as2_bt`; it has become an A04 telegram.

Route locking

- Simulate until the telegram disappears in `signal1`. Another A03 telegram appears in `bs1_at`.
- Simulate one more step; the A04 telegram changed into an A05 telegram, in the mean while, setting a periodic timer inside `signal1` that is going to generate B01 telegrams.
- Continue the simulation; the A03 telegram proceeds and becomes an A04 telegram. Furthermore, the A05 telegram continues until it terminates inside `signal2`, setting another periodic timer.

Route monitoring

- Continue the simulation. After a while, an A06 telegram appears in `as2_bt`.
- Simulate until the A06 telegram terminates inside `signal1` and a token appears in `extout_s2`. This token is a W01 telegram, showing the colour and speed indication at the signal.

At this point, the simulation reaches a stable point. If the simulation is continued, A06 telegrams will appear periodically. Nothing else will happen, until a train arrives at the entry signal.

Train operated route release

- Change the token in `TSU1` to false, indicating that a train has arrived at the entry signal.
- Continue the simulation. Another A06 telegram appears, and after a while, also a B01 telegram.
- Continue the simulation until the B01 telegram terminates in `track`.

At this point, the simulation reaches again a stable point until the train moves onto the track segment.

- Change the token in `TSUt` to false, indicating that the train has moved onto the track segment.
- Continue the simulation. The next B01 telegram turns inside `track` and becomes a B02 telegram. The B02 telegram becomes a B03 telegram in `signal1`.
- Continue the simulation until a B04 telegram appears in `bout`.

The specification of the signal element in [4] does not specify the effect of a B04 telegram that enters the signal from side "a." However, in the simulation it does happen. We have found a mistake in the specifications in [3]. This mistake has already been corrected by NS in a more recent specification of the signal element.

- Change `TSU1` to true; the train has passed the entry signal.
- Continue the simulation until a B07 telegram terminates in `signal2`. No more A06 telegrams will appear.
- Change `TSU2` to false and `TSUt` to true; the train moves on.
- Continue the simulation. A B05 telegram appears which becomes a B06 telegram in `signal2`. After continuing the simulation, the B06 telegram terminates and nothing will happen anymore.