LARIS 1.0
LAnguage for Railway Interlocking Specifications

Wan Fokkink Jan Friso Groote
CWI
Department of Software Engineering
wan@cwi.nl jfg@cwi.nl

Marco Hollenberg
Philips Research
Department of Software Technology
marco.hollenberg@philips.com

Bas van Vlijmen
Utrecht University
Department of Philosophy
vlijmen@phil.uu.nl

This document has been produced in cooperation
with Holland Railconsult
by order of NS Railinfrabeheer.

Contents

1 Introduction **1**

2 EURIS **3**
2.1 Introduction 3
2.2 Overview of EURIS 4
2.2.1 Flows 5
2.2.2 Ports, telegrams, and variables 6
2.2.3 Logic and sequence charts 9
2.3 Related work on EURIS 11

3 LARIS **11**
3.1 Types and expressions 15
3.1.1 Booleans and integers 16
3.1.2 Enumerated types 17
3.1.3 Components and ports 17
3.1.4 Default values 18
3.1.5 Arrays 19
3.1.6 Clock types 20
3.1.7 Names of variables 23
3.2 Statements 23
3.3 LSCs and behaviours 26
3.4 Systems 28
3.5 LARIS versus EURIS 29
3.6 Translating EURIS into LARIS 32
3.6.1 Toggles 32
3.6.2 Central telegrams 35
3.7 LARIS versus EURIS$_{DTO}$ 40

4 Syntax and static semantics **42**
4.1 Formal syntax in SDF 43
4.2 Notational conventions 48
4.3 Abbreviations 48
4.4 Signatures 49
4.5 Assigning signatures 52
4.6 Types 55
4.7 Type- and declaration-correctness 56
4.8 Expressions 59
4.9 Argument-correctness 60
4.10 Statements 62
4.11 Static semantical correctness 63
4.12 Substitutions 65

5 Operational semantics **67**
5.1 Interpretation of types 67
5.2 Interpretation of expressions 69
5.3 Operational semantics of a component 72
5.3.1 Sets of telegrams 73
5.3.2 States of $\mathcal{M}_C$ 73
5.3.3 Initial state of $\mathcal{M}_C$ 74
5.3.4 Actions of $\mathcal{M}_C$ 74
5.3.5 Transitions of $\mathcal{M}_C$ 75
5.4 Operational semantics of a communication channel 83
5.5 Operational semantics of a specification 84

6 Example: part of Driebergen **87**

7 Conclusions and future work **121**

References **123**

Index **125**

1 Introduction

Already in 1982 it was felt at what is now the company NS Railinfrabeheer that the increasing use of computerized railway control required a change in the ways these were being developed. Suppliers started to develop far more complex control systems, which did not entirely match the Dutch situation. Communicating the Dutch situation unambiguously to these suppliers turned out to be a difficult job, and it was readily recognized that this required a language tailored for the description of control systems. It should be possible to come to a similar situation as providers of infrastructure such as bridges, tunnels, buildings, and of course the railroad itself, who have at their disposal a standardized way of drawing, complemented with descriptions regarding strengths and types of material.

A specification language for control systems was proposed, called EURIS (EUropean Railway Interlocking Specification). This language was inspired by the Prozeß Ablauf Pläne [2] being used at Siemens. EURIS was used to specify the control of several railway yards, and from this point onward it became clear that EURIS was appropriate for its intended use. EURIS has been provided with several simulators [1, 5, 18, 19], and an extensive study into its semantical aspects has been made [4, 7]. A set of standardized railway control components for the Dutch situation has been defined in the form of UniSpec [16], aiming to speed up the design process, and to boost reliability by increasing the level of reuse.

EURIS is not entirely void of problems. Its syntax and semantics have never been appropriately defined. Its standard description document [3] introduces the language in a tutorial-like style, so that it is never crystal clear what the language is exactly. There is no common agreement on what all language constructs are, and there is no common understanding what each language construct is supposed to do. Too often the language must be understood by finding out how the EURIS simulator [18] deals with it, or what the designers of the language have to say about it. As this may provide conflicting answers, it is sometimes impossible to come to a definitive conclusion. Clearly, this situation is undesirable, as EURIS specifications are supposed to equivocally prescribe how railway control ought to work.

EURIS is in essence a graphical language. An advantage of the graphical representation is that specifications are easy to read, and that it is generally easy to see where global variables are changed, or which variables are updated in a flow. However, defining the syntax and semantics, as well as building tools, is considerably easier for textual than for graphical languages. Therefore, the language IDEAL [17, 5, 19] has been defined, as an intermediate between EURIS and its simulator. Furthermore, a graphical language is in general much harder to write than a textual one; it can be unpleasantly complicated to find the optimal layout of a EURIS specification. EURIS also lacks many common constructs from textual languages that facilitate the writing of specifications. These shortcomings lead to the use of 'tricks', which obfuscate the specification.

This is contrary to the intent of specifications, which should be as clear as possible.

As a solution to the shortcomings mentioned above, and to enter a next phase in the use, tooling, and development of EURIS, it has been decided to develop a textual version of it. This textual version is presented in the current document, including a tutorial-style introduction, and a description of its syntax, static semantics and operational semantics. Below we deal with these notions in more detail, and explain their importance.

We have aimed to develop a language that is close to the original EURIS, and in line with contemporary languages. An important design goal was to keep the language concise and expressive, especially for the purpose of safe control systems. We call the language *LARIS 1.0*, *LAnguage for Railway Interlocking Specifications*. The version number 1.0 has been added, as extensive use will undoubtably indicate some shortcomings, which will require changes in the language. In order to avoid confusion, all revisions of LARIS should be accompanied with a version number. Whenever in this document 'LARIS' is mentioned, it should be kept in mind that 'LARIS 1.0' is intended.

This document is set up as follows. Section 2 explains the ideas behind EURIS, and gives a brief overview of its constructs. Section 3 contains an introduction to LARIS, together with its main features, and tells about the similarities and differences between LARIS and EURIS. Section 4 presents the syntax and static semantics of LARIS, while Section 5 gives an operational semantics for LARIS. Section 6 presents an exemplifying LARIS specification of part of the railway yard at the Dutch railway station Driebergen. Finally, Section 7 describes conclusions and further work.

In case of doubt or in case of inconsistencies in this document, the description of LARIS 1.0 in the Sections 4 and 5, which define its syntax and semantics, must be taken as authoritative. The syntax describes which sequences of symbols can be considered as valid LARIS programs. It is written down in SDF (Syntax Definition Format) [14]. On purpose the syntax is defined for plain ascii strings, avoiding the use of different letter types, subscripts, superscripts, and more advanced type settings. Plain strings are one of the few standards that are accepted, and henceforth LARIS specifications can easily be accepted among different computer systems. The static semantics prescribes which LARIS programs are well defined. Typically, it says that all data types that are being used have been declared, that if a variable is used it is declared of the correct type, that types in assignments and procedures are used in the correct way, et cetera. Every static semantical property can be checked at compile time, i.e., before the system is actually being installed. Therefore, checking the static semantics is an effective and cheap way to detect mistakes in the description of control systems. The style of the static semantics is conform current ideas, but the description is considerably shorter than those of comparable languages (cf. SDL [9] or LOTOS [21]). It certainly requires some expertise to read and understand the syntax and static semantics, but luckily this is only required when in doubt about the real meaning of some construct.

The operational semantics as defined in Section 5 explains how a LARIS 1.0 specification is interpreted as a process graph. In this way, it is known which steps a LARIS specification can take. The operational semantics is actually nondeterministic, in the sense that at particular moments many different actions can be taken. So the defined operational semantics ought to be seen as the potential behaviour; a run of the system is obtained by selecting one alternative at each possible moment.

This document has been produced in cooperation with **Holland Railconsult**, by order of **NS Railinfrabeheer**.

Acknowledgements. We thank Peter Middelraad, Daan van der Meij, Henk Scholten, Eric Burgers, Fokko van Dijk, Gea Kolk, Paul van de Ven, Andre van Delft, Richard Hilhorst, Jan Tretmans, and Eric Viertelhauser for helping us understand EURIS, and design LARIS.

2 EURIS

2.1 Introduction

The control and management of a railway system consists of three separate tasks. First, control instructions for the railway yard have to be devised at the *logistic level*. Second, control instructions have to be passed on to the *infrastructure*, which consists of points, signals, level crossings, et cetera. This task is almost always fully automated. Third, it has to be guaranteed that the execution of control instructions does not jeopardise safety; that is, collisions and derailments have to be avoided. This is done by means of an *interlocking*, which is a medium between the infrastructure and the logistic level together with its interfaces. An interlocking logic is the embodiment of safety principles and basic rules, according to which a train moves through a railway yard.

Traditionally, an interlocking logic served as a local solution for a specific railway yard, whereby the logic could be designed to cope with the peculiarities of the railway yard. Modern computer-based railway systems demand a uniform specification method which can be used to formulate interlocking logics for all railway yards. The safety restrictions that are imposed on different railway yards are reasonably consistent, depending mostly on the parameters of autonomous components such as signals and points. Based on this observation, Middelraad *cum suis* from NS Railinfrabeheer evolved a modular specification method EURIS (EUropean Railway Interlocking Specification) [3], to describe fully automated interlocking logics; see [7] for an overview. EURIS assumes an object-oriented architecture, which consists of a collection of generic *components*, representing the building blocks of the infrastructure such as signals and points, and of two clearly separated entities in the outside world, representing the logistic level and the infrastructure. The components, which together make up the interlocking logic, communicate with each other by means of datastructures called telegrams. The components can also exchange telegrams with the logistic level and with the infrastructure.

EURIS not only denotes a specification method, it is also the name for a graphically oriented imperative specification language that is based on this method. A Logic and Sequence Chart (LSC) specifies a component. Each LSC consists of the graphical representation of procedures, which can adapt and test the values of variables, and which can ultimately trigger the transmission of a telegram. Such telegrams can be received by neighbouring components, by the logistic level, and by the infrastructure. Reversely, each component can also receive telegrams from neighbouring components, from the logistic level, and from the infrastructure. The graphical format for LSCs evolved as a compact notation when the Nassi Schneidermann diagrams, which were originally used to express EURIS specifications, became unclear due to deep nestings of if-then-else statements.

A strong advantage of an object-oriented architecture is the possibility to reuse components of a specification. In EURIS, the heart of a specification defines the way that components handle incoming telegrams. When all types of components have been specified in full detail, the specification of a particular railway yard is constructed by simply connecting its separate building blocks in the appropriate manner. A second advantage of the distributed approach is that if the behaviour of say a signal is changed, then this can be taken into account on the level of EURIS by adapting the specification of the corresponding component. A disadvantage can be that in EURIS applications a procedure such as claiming a route is specified implicitly in the designs of several components, so that adapting such a procedure can become non-trivial.

UniSpec [3, 16] is a particular instance of the EURIS method, which has been developed by NS Railinfrabeheer as a complete set of generic components to compose interlocking logics for the Dutch railway system. Holland Railconsult has implemented a simulator for UniSpec [17, 18], which enables one to animate the behaviour of a UniSpec specification. The simulator is part of a tool set named GUIDE, which is currently used by NS Railinfrabeheer to support both the design and validation of UniSpec specifications. After designing a set of LSCs, the user can join instantiations of these LSCs according to the topology of a railway yard. The result is checked for design rule errors and compiled, after which situations at the railway yard can be simulated via a graphical interface. The simulator makes it possible to locate flaws in an interlocking specification at an early stage of the system engineering process. Furthermore, a simulation session gives a detailed insight of the behaviour of the specification, which can be useful in the communication with customers.

2.2 Overview of EURIS

EURIS is a graphically oriented, parallel, event-driven, weakly typed, imperative specification language. A EURIS specification consists of the graphical description of components in the form of *LSCs* (see Section 2.2.3), which are connected by *ports* (see Section 2.2.2). Each LSC consists of procedures with an imperative character, called *flows* (see Section 2.2.1). Components can send data-structures called *telegrams* (see Section 2.2.2) to each other via their ports. Reception of a telegram by a component causes the execution of a flow in the LSC belonging to this component. This flow is determined by the telegram and the channel via which the component received the telegram. Changing the value of a variable may also cause execution of a flow.

EURIS assumes the two standard data types of booleans and integers. The booleans consist of 1, representing *true*, and 0, representing *false*. Three standard functions are defined on integers: addition, subtraction, and multiplication.

2.2.1 Flows

We consider a certain component, with a unique component name. A *flow* for this component is a procedure that is built from the following five basic constructs:

- An *execution condition* formulates under which circumstances the flow is executed. There are two possibilities.

  If the execution condition is of the form `T` ▶ `p`, then the flow is executed if telegram `T` is received at port `p` of the component; see Section 2.2.2.

  If the execution condition consists of a variable name X, then the flow is executed depending on (the change of) the value of X; see Section 2.2.2. A flow is built from the following four constructs.

- A case tests the value of a variable; the returned value influences the subsequent execution of the flow.

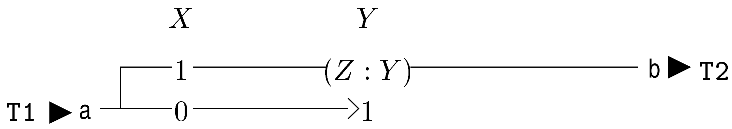
- An assignment adapts the value of a variable.

- A termination symbol marks the end of the execution of the flow.

- A send action `p` ▶ `T` instructs that telegram `T` is sent out via port `p` of the component. A send action is always followed by termination of the flow.

A component is specified by its flows, where the execution conditions of the flows cover all telegrams that can be received by the ports of the component.

Flows are represented graphically, whereby the tests and assignments of the flow are connected with each other by continuous lines. A test whether variable X equals value v is denoted either by placing v in the flow below variable X, or by placing the expression $(X = v)$ in the flow. An assignment of value v to variable X is denoted either by placing $>v$ in the flow below variable X, or by placing the expression $(X : v)$ in the flow. The graphical layout of flows is important for the interpretation of tests and assignments; see the picture below.



The execution condition at the left of this flow expresses that it is executed if telegram `T1` is received via port `a`. First, the flow tests the value of the variable X. If this value is 0, then the flow assigns the value 1 to variable Y, after which it terminates. If the value of X is 1, then the flow assigns the value of Y to Z, after which it sends out telegram `T2` via port `b`. Z is a telegram-field; if this field does not yet exist then it is created, and otherwise its value is adapted.

2.2.2 Ports, telegrams, and variables

A EURIS specification assumes a logistic level and an infrastructure, and specifies the behaviour of a number of components. In particular, it is described how these separate entities send messages called telegrams to each other, and how the components react when they receive a certain telegram from a certain entity. The components and the logistic level and the infrastructure can send telegrams to each other via communication channels, which are constructed by the combination of *ports*. EURIS recognises the following two kinds of ports.

- A component has one or more *route* ports. Each route port `p` of a component `c` is linked with exactly one route port `p'` of another component `c'`, establishing a communication channel between `c` and `c'`. If component `c` sends a telegram into port `p`, then this is received by component `c'` through port `p'`, and vice versa.

- A component may have a port that connects it with the logistic level, and a port that connects it with the infrastructure. Telegrams can travel from the component to the logistic level and to the infrastructure via these ports. Vice versa, for each component, the logistic level, and the infrastructure may have a port via which they can send telegrams to this component.

- A component may have two *central* ports, `left` and `right`. Central telegrams with the direction `left` or `right` are sent out via the corresponding central port. The destination of the telegram is determined by its so-called central list, which consists of a list of component names, including the current component. Central telegrams that are sent out via `left` of `right` are sent to the component to the left or to the right of the current component in the central list, respectively.

A *telegram* has a unique name and carries a *telegram table*, which assigns boolean and integer values to telegram-fields. A telegram may be passed on between a number of components, which all update the information in the table of the telegram. If a flow terminates by sending out a telegram, then it attaches the telegram table that was created, or adapted, during the execution of the flow. Telegram-fields are only meaningful for a component as long as the flow that belongs to their telegram in the component is being executed.

A *route* telegram is sent from one entity to another entity, whereby entities can be components, the logistic level, or the infrastructure. It consists of a telegram name, a telegram table, and a port name from which it is sent out.

A *central* telegram is passed on between the components that are on a so-called *central list*. Each central telegram has a direction, either `left` of `right`, which determines in which order it visits the components in the central list. A central telegram consists of a telegram name, a telegram table, a direction, and a central list.

An *internal* telegram is generated inside a component by special types of variables, depending on the (change of) value of such a variable. An internal telegram consists of a telegram name together with the name of the variable that produced this telegram, and of an empty telegram table.

EURIS distinguishes several types of internal variables, which are local to a component. The initial values of *input* variables, which carry the version symbol '`!`', are latched. The values of internal variables without a version symbol can be adapted without giving rise to the execution of a flow. Finally, internal variables with a version symbol from $\{\texttt{@}, \texttt{\&}, \texttt{\$}, \texttt{\#}, \texttt{?\#}, \texttt{>>\#}\}$ may trigger the execution of a corresponding flow, depending on the (change of) value of such a variable. Before describing these variables in detail, first we say some more about the time domain.

Time plays an important role in the specification of an interlocking logic. It enables to model delays; for example, if a train has passed a section, then for safety reasons this section has to be unoccupied for a certain period of time. We assume a discrete time model, in which time progresses in distinct steps, called time slices. Methods such as Vital Processor Interlocking [13] from the General Railway Signal Company and Prozeß Ablauf Pläne [2] from Siemens, which are used for the implementation of real-life interlockings, and the simulator of EURIS, are based on a discrete time domain. Furthermore, it has been shown in practice that it is technically feasible to synchronise the parallel processes of a EURIS specification on time slices.

One-shot $\texttt{\&}_{\texttt{c},X}(b)$ represents the *one-shot* variable X in component `c`, with boolean value b. This value can be tested, or adapted by an assignment, during the execution of a flow in component `c`.

Intuitively, the variable X emits a telegram exactly once, if the boolean value of X is changed to 1. This is modelled as follows. If the system evolves into a new time slice, then:

- if the boolean value of X is 1, then a telegram is generated, after which the boolean value of X is changed to 0.

The flow for the telegram that is emitted by X in component `c`, if its boolean value is 1, is specified in the flows for `c`.

Toggle $\$_{\mathtt{c},X}(b,b')$ represents the *toggle* variable X in component `c`, where b is the boolean value of X at the end of the last time slice, and b' is the current boolean value of X. Only this last boolean value of X can be tested, or adapted by an assignment, during the execution of a flow in component `c`.

Intuitively, the variable X emits a telegram if its boolean values at the end of the last and the current time slice differ. In other words, if the system evolves into a new time slice, then:

- if b and b' are distinct, then a telegram is generated, and the value b is changed into the value b'.

The flow for the telegram that is emitted by X in component `c`, if b and b' differ, is specified in the flows for `c`.

Timer $\#_{\mathtt{c},X}(n,b)$ represents the *timer* variable X in component `c`, with as clock value the non-negative integer n, and with the boolean value b. The boolean value of X can be tested, or adapted by an assignment, during the execution of a flow in component `c`. The clock value of X can only be tested during the execution of a flow in component `c`.

Intuitively, the timer X is active if the boolean value of X is 1, and otherwise it is inactive with clock value 0. In other words, if the system evolves into a new time slice, then we can distinguish the following possibilities.

- If the boolean value of X is 1, then its clock value is changed to $n+1$.
- If the boolean value of X is 0, then its clock value becomes 0.

Time-out $\mathtt{>>\#}_{\mathtt{c},X}(m,n)$ represents the *time-out* variable X in component `c`, with as period the positive integer m, and with as clock value the non-negative integer n. The clock value of X can be tested during the execution of a flow in component `c`. Furthermore, X can be assigned the boolean value 1, in which case it is 'set', or the boolean value 0, in which case it is 'reset', during the execution of a flow in component `c`. If X is set, then its clock value is changed to m; If X is reset, then its clock value is changed to 0.

Intuitively, the variable X emits a telegram m time units after it has been set. In other words, if the system evolves into a new time slice, then we can distinguish the following possibilities.

- If the clock value n of X is 1, then a telegram is generated, after which the clock value is changed to 0.
- If the clock value n of X is greater than 1, then the clock value is changed to $n-1$.
- If the clock value of X is 0, then the clock value remains 0.

The flow for the telegram that is emitted by X in component `c`, if its clock value is 1, is specified in the flows for `c`.

Cyclic time-out $\texttt{@}_{\texttt{c},X}(m,n,b)$ represents the *cyclic time-out* variable X in component `c`, with as period the positive integer m, with as clock value the non-negative integer n, and with the boolean value b. Only the boolean value of X can be tested, or adapted by an assignment, during the execution of a flow in component `c`. If X is assigned a boolean value b', then the clock value automatically becomes 0, so that the new state of the variable X is $\texttt{@}_{\texttt{c},X}(m,0,b')$.

Intuitively, the variable X emits a telegram every n time units, if the boolean value of X is 1. In other words, if the system evolves into a new time slice, then we can distinguish the following possibilities.

- If the boolean value of X is 1, and its clock value n equals 0, then a telegram is generated, after which the clock value of X is changed to $m-1$.
- If the boolean value of X is 1, and its clock value n is greater than 0, then the clock value of X is changed to $n-1$.
- If the boolean value of X is 0, then the clock value of X becomes 0.

The flow for the telegram that is emitted by X in component `c`, if its boolean value is 1 and its clock value is 0, is specified in the flows for `c`.

Dynamic time-out $\texttt{?\#}_{\texttt{c},X}(n)$ represents the *dynamic time-out* variable X in component `c`, with as clock value the non-negative integer n. The clock value of X can be tested, or adapted by an assignment, during the execution of a flow in component `c`.

Intuitively, the variable X emits a telegram after n time units. In other words, if the system evolves into a new time slice, then we can distinguish the following possibilities.

- If the clock value n of X is 1, then a telegram is generated, after which the clock value is changed to 0.
- If the clock value n of X is greater than 1, and the clock value is changed to $n-1$.
- If the clock value of X is 0, then the clock value remains 0.

The flow for the telegram that is emitted by X in component `c`, if its clock value is 1, is specified in the flows for `c`.

2.2.3 Logic and sequence charts

An LSC consists of a number of graphical representations of flows. An LSC takes as basis a list of internal variables, whereby variables carry their version symbols. Figure 1 presents an example of an LSC. X_1, X_2, and X_3 are internal variables. X_1 is an input variable, which is denoted by the version symbol '`!`',

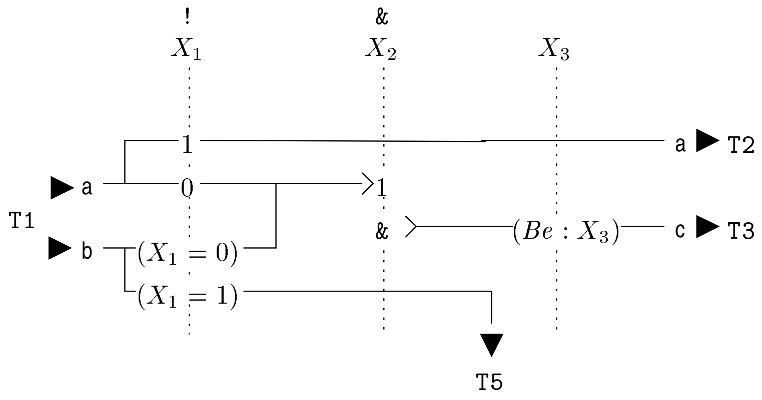
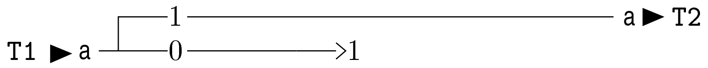
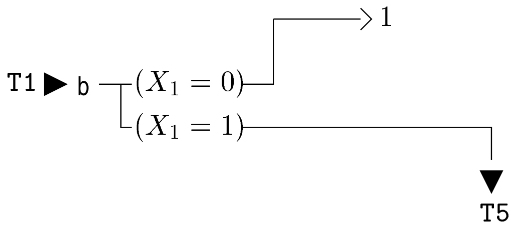


Figure 1: An LSC example.

X_2 is a one-shot variable, which is denoted by the version symbol '&', and X_3 does not carry a version symbol. The graphical representations of the flows that make up the LSC are drawn below this list. Thus, we obtain the picture that is displayed in Figure 1. Below each variable name we have drawn an imaginary vertical dashed line. Each test and assignment in a flow is placed on such a dashed line, in order to relate it to the variable of which the value is tested or adapted. Some of the flows share the same tail, which means that from some point onwards they have the same functionality. However, flows are independent entities. We explain the meaning of each flow.



The flow above is executed if the telegram with name `T1` is received via port `a`. If the input variable X_1 is 0, then this flow assigns the value 1 to the one-shot variable X_2, after which it terminates. If X_1 is 1, then this flow sends out the telegram with name `T2`, and with the unaltered telegram table, via port `a`.



The flow above is executed if the telegram with name `T1` is received via port `b`. If the input variable X_1 is 0, then this flow assigns the value 1 to the one-shot variable X_2, after which it terminates. If X_1 is 1, then this flow sends out the telegram with name `T5` to the infrastructure.

& $\succ$———$(Be : X_3)$—— c ▶ T3

The flow above, generated by one-shot variable X_2, assigns the value of variable X_3 to variable *Be*. Next, the telegram with name `T3` is sent out via port `c`.

2.3 Related work on EURIS

The tool kit for EURIS, developed at Holland Railconsult, uses internally a symbolic representation language called IDEAL [18]. In [5] the syntax and static semantics of IDEAL is specified in ASF+SDF. See [6] for presentation of the ASF+SDF specification environment. The motivating idea behind the ASF+SDF specification of IDEAL was the following. IDEAL could function as an interface between EURIS and other representations which could then aim at, e.g., verification and semantics. Currently this work gains relevance, because an automatic translation of IDEAL to the specification language LARIS proposed in this text is useful practically, and feasible technically. In [1] the semantics of EURIS was studied by some small translation experiments of EURIS into timed coloured Petri nets. The work of the projects mentioned above coalesced in [19] where it is described how ASF+SDF can be used to generate Petri nets from IDEAL.

An intensive study on EURIS was reported on in [4]. This book describes various aspects of EURIS. First, it gives an informal introduction, a reasonably complete description of the syntax, i.e., the syntax found in the UniSpec LSCs used by the EURIS tool kit. Second, a proposal for an operational semantics of EURIS is given in discrete time process algebra, for a language called EURIS_{DTO}. With respect to semantics, EURIS_{DTO} is a precursor of LARIS. Third, the relation of EURIS and Vital Processor Interlocking (VPI) is studied. It turns out that under strong assumptions and restrictions EURIS_{DTO} can be simulated on a VPI, and that this variant of EURIS_{DTO} can be verified in the style of work on VPI [8, 10, 13]. This route seems nevertheless not very attractive due to the restrictions. The translation from EURIS_{DTO} to VPI programs experiments with a symbolic presentation that can be seen as a sketchy syntactic precursor of LARIS. Finally, an incremental approach to the compilation and specification of correctness criteria on interlockings is discussed. In Section 3.7, the relation of LARIS and EURIS_{DTO} are discussed in more detail.

3 LARIS

LARIS (LAnguage for Railway Interlocking Specifications) is a textual language for specifying distributed communicating systems that act as an intermediary between logistic level and infrastructure. The motivating examples are railway interlocking systems. Such systems are given tasks by the logistic level, which are then carried out on the infrastructure level (in this context also known as the track-side level) in a way that respects certain safety criteria. LARIS is meant to be a textual variant of EURIS.

A typical system might look as in Figure 2. The logistic level and the infrastructure level are treated as components, so that there are a total of eighteen components. The sixteen components between the logistic level and the infrastructure together comprise the interlocking. These are the only components that are available to a LARIS specification; the logistic and infrastructure components are treated as the environment.

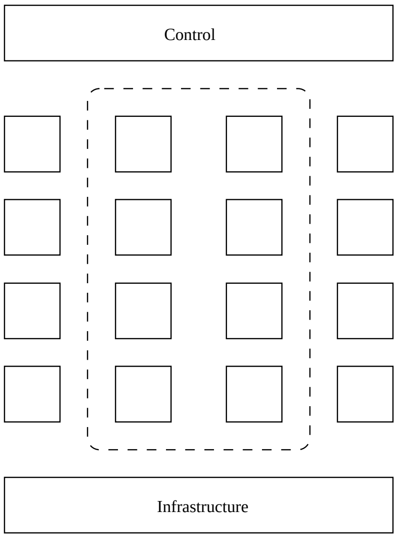


Figure 2: A typical system.

We may only be interested in specifying a portion of an interlocking. For example, we may only be interested in specifying the part within the dashed box in the figure above. LARIS allows such partial specifications. All other components are then treated as part of the environment.

There is always a fixed number of components. Components are not created dynamically on-the-fly, as opposed to languages such as PROMELA [15] and LOTOS [21].

Communication is in principle possible from any component to any other. Communication is taken to be asynchronous. This is modelled by means of two unbounded buffers, FIFO (first-in-first-out) queues, placed between every two components in the specified part, one for each direction of communication (see Figure 3). Modelling communication in an asynchronous way has as a consequence that when component 1 in Figure 4 first sends a telegram a to component 2 and then a telegram b to component 3, which in turn gives rise to a telegram c from component 3 to component 2, the telegram a could be delayed in the communication channel and in fact arrive at component 2 after telegram c arrives.

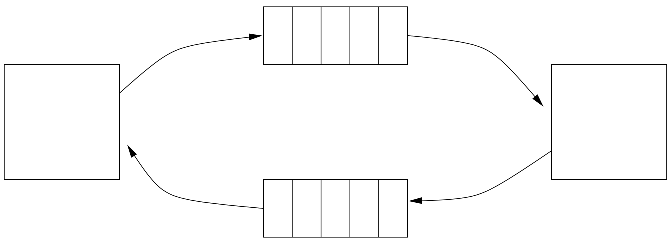


Figure 3: Communication channels between two components.

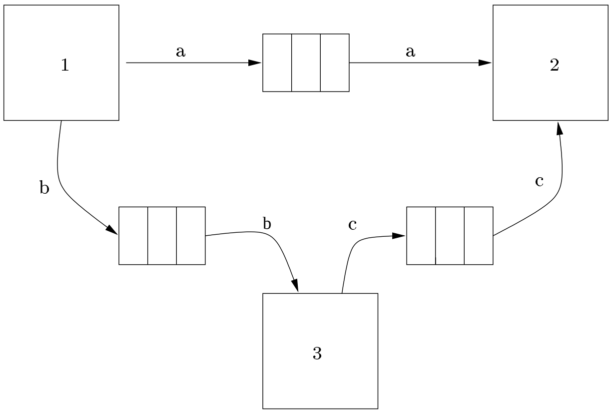


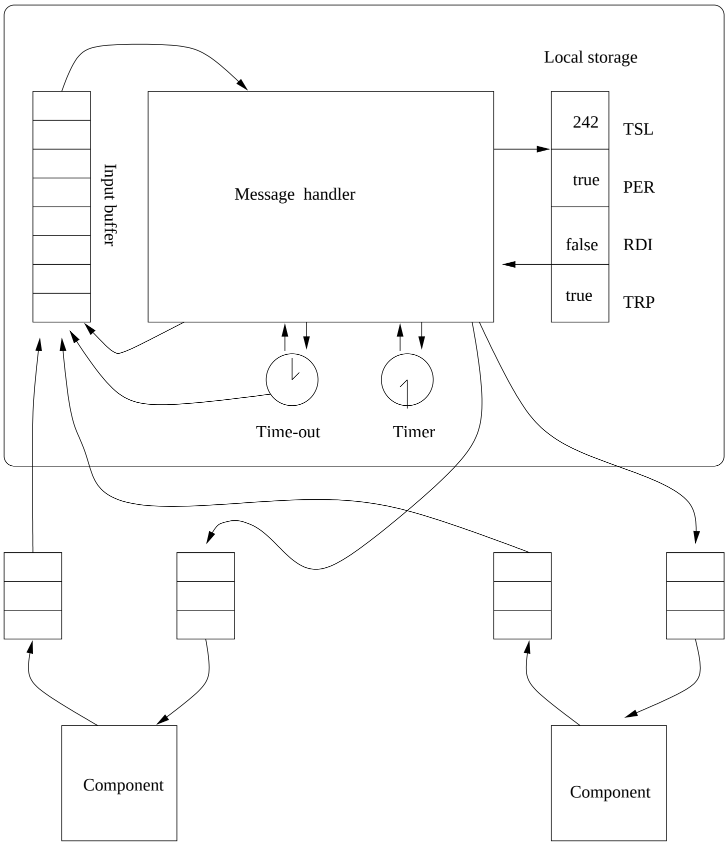
Figure 4: Overtaking of telegram a.

Components themselves have the following internal structure. There is an *input buffer*, where incoming telegrams are stored temporarily, before being processed by the component. Telegrams received from the component itself (internal telegrams) are also stored in this buffer.

There is a *message handler*, which extrudes telegrams from the input buffer and, depending on the type of telegram it finds, performs some operations. These operations could result in putting an internal telegram into its own buffer, sending telegrams along channels to other components, changing the values of locally stored variables, or (de)activating timer processes. The message handler of a component is available to specification in LARIS: it is the programmable part of a component. This part consists of a number of statements, being sequences of instructions, one such sequence for each type of telegram that can be expected to arrive at the component. The basic instructions can be categorized as follows:

- assignments;
- sending telegrams;
- (de)activating timer processes;
- procedure calls.

More is said about such instructions in Section 3.2.



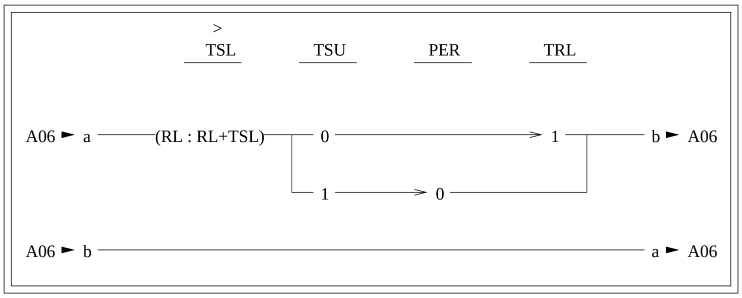
In graphical form, a component has the structure depicted above. This picture is purely for the purpose of demonstration; it is not meant to mean that components always have internal variables `TSL`, nor that components have a single time-out and a single timer.

Summarizing, a system in LARIS is a collection of components and of communication channels between them. Components are the part of the system that are specified in LARIS. However, in general one does not need to specify every component separately. To capture the idea that many components share certain characteristics, and perhaps only differ in a small number of parameters, in EURIS one has the notion of an LSC. LARIS retains this notion. To obtain a specific component from an LSC it suffices to provide a name for the component, and to fill in the parameters that the LSC requires.

A typical LARIS LSC would be:

```
LSC name (Ea, Eb:Component; Pa, Pb:Port; TSL:Int) =
   vars PER, TSU, TRL:Bool;

   initial skip

   mes a? A06 (RL:Int) =
      RL:= RL+TSL;
      if TSU
         then PER:= false
         else TRL:= true;
      Eb |> Pb ! A06(RL)

   mes b? A06 (RL:Int) =
      Ea |> Pa ! A06(RL)

   panic Log |> log ! P01(self)
```

This would have the following analogue in EURIS:



From the above, the reader is expected to get nothing but a flavour of LARIS, not to fully grasp all meaning of the LSC. In the remainder of this section we give a more detailed overview of the language LARIS. We hope to provide some intuitive insights into the language. The inquisitive and mathematically oriented reader is invited to read the formal syntax and semantics provided in Sections 4 and 5.

3.1 Types and expressions

LARIS is a typed language: every expression that is used in a LARIS specification has a certain type. Operations are sensitive to these types. For instance, addition can only be applied to expressions of the type integer. We list the available types and the operations that are applicable to expressions of these types.

3.1.1 Booleans and integers

The first two basic types that LARIS uses are familiar ones: `Bool` for booleans and `Int` for integers.

`Bool` is the type for truth values, of which we have two: `true` and `false`. We have the following standard operations on booleans:

$\wedge$	and
$\sim$	not

`Int` is the type for integers $\ldots, -2, -1, 0, 1, 2, \ldots$. We have the following standard operations on integers:

$+$	addition
$-$	subtraction
$*$	multiplication
`div`	division, rounded down
`mod`	modulo

Furthermore, we have $==$ to express that two integers or booleans are the same, and $<$ to express that one integer is smaller than the other. We use the following abbreviations:

abbreviation	what it abbreviates
$E \mid F$	$\sim(\sim E \wedge \sim F)$
$E =/= F$	$\sim(E == F)$
$E > F$	$F < E$
$E <= F$	$E < F \mid E == F$
$E >= F$	$F <= E$

In EURIS the booleans are represented by the integers 0 and 1. Thus it is possible to add booleans to integers in EURIS. In LARIS overloading of this kind is not allowed. To achieve the same effect, one would first need to convert booleans to integers. Thus, while

```
vars B:Bool; I:Int
I:= I+B
```

is illegal in LARIS, the desired effect may be achieved by

```
vars B:Bool; I:Int
if B then I:= I+1
```

3.1.2 Enumerated types

In LARIS one may define a type by means of a declaration such as:

```
Route = { drive_on_sight,
          normal,
          automatic_normal,
          drive_on_sight_normal }
```

By this means we have defined the type `Route`. Expressions of this type can have four possible values, namely those listed between accolades.

Empty type definitions are not allowed, so the following is forbidden:

```
Nothing = { }
```

3.1.3 Components and ports

`Component` is the type containing names of building blocks such as tracks and signals, which together make up an interlocking. Components may perform internal calculations, communicate with each other and with two special components, `Log` and `Inf`.

`Log` represents the logistic level, while `Inf` represents the infrastructure level. These are the levels between which an interlocking is an intermediary (see Figure 2).

Another important basic type is `Port`. Expressions of this type represent the names of the ports of components on which telegrams may be received. In EURIS, typically, these names are `a`, `b`, `c`, and `d`. In LARIS there is no restriction on the number of port names, nor on the names used. There are a number of presupposed port names.

- `log` is the main port on which `Log` receives telegrams. It is also the name of the port on which interlocking components receive telegrams from the logistic level. This port name corresponds to EURIS execution conditions of the form
$$\begin{array}{c} \mathtt{N} \\ \blacktriangledown \\ | \end{array}$$
- `inf` is the main port on which `Inf` receives telegrams. It is also the port on which interlocking components receive telegrams from the infrastructure level. This port corresponds to EURIS execution conditions of the form
$$\begin{array}{c} | \\ \blacktriangle \\ \mathtt{N} \end{array}$$
- `left` is the port on which left central telegrams are received. This port name corresponds to EURIS execution conditions of the form $- \lhd \mathtt{N}$

- `right` is the port on which right central telegrams are received. This port name corresponds to EURIS execution conditions of the form `N` $\rhd$ $-$

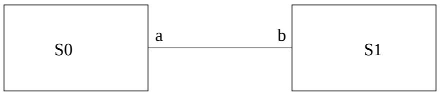
In EURIS, if a component A wishes to send a telegram to a component B, there are two ways to do so. If A has a route port a and B has a route port b which are connected, then A may send the telegram along port a, so that B receives this telegram on port b. This is the main means of communication. Another means is via the use of central telegrams and central lists. If A sends a central telegram and B happens to be on the right spot in its central list, then it is also possible for A to get its telegram across to B.

Central telegrams have such potential that communication from anywhere to anywhere is a definite possibility, although rarely used in such a rogue manner in EURIS specifications. To accommodate the strength of central telegrams, communication in LARIS occurs on a fully connected network. This means that any component is able to send a telegram to any other component. Port names are still important: typically, as in EURIS, different flows are initiated when the same telegram is received on different ports of the component.

Thus, a typical communication of a telegram with name `B01` and contents `true` to a component `S1`, such that `S1` receives this telegram on port `p`, is achieved by the statement

```
S1 |> p ! B01(true)
```

This makes communication in LARIS somewhat different from communication in EURIS. For example, consider the following system layout, EURIS-style:



In EURIS, if `S0` wishes to send a telegram to port `b` of `S1`, then this telegram is sent along the `a`-port of `S0`. The layout of the system ensures that this telegram arrives at the required spot. In LARIS, however, the telegram is sent directly to the `b`-port of `S1`.

Besides being able to handle so-called neighbour-telegrams of the above sort, LARIS-style communication also allows for central telegrams. The beauty of LARIS communication is that all these telegrams use the same means of communication.

3.1.4 Default values

If no specific value has been assigned to a variable of which the type is known, it is assumed to have the *default value* of that type. For the basic types, these default values are listed below.

type	default value
`Bool`	`false`
`Int`	0
`Component`	`Log`
`Port`	`log`
Enumerated type	First value in the list

Note that, as enumerated types must be nonempty, their default value is always defined.

3.1.5 Arrays

LARIS allows the construction of simple arrays. An *array type* is of the form $T[I_0, \ldots, I_n]$, where T is a basic type and each of $I_0, \ldots, I_n$ is an index, that is, either a basic type or a positive integer. Examples are: `Component[Int]`, `Bool[Port,Int]`, and `Bool[100]`. We explain such types by a detailed look at a series of examples.

First consider the type `Component[Int]`. This is the type of arrays with as indices integers and as values names of components. Indices help us to retrieve values out of arrays. If `A` is an array of the type above, then we use the expression `A[3]` to denote the component in `A` at index 3. Note that this expression is then of type `Component`.

We may also have pairs, or even tuples, as indices. An example where this occurs is `Bool[Port,Int]`. Its values are booleans, and its indices are pairs of port names and integers. Thus, if `A` is an array of this type, then `A[a,-3]` is a boolean, namely the one stored at index $(\mathtt{a}, -3)$, assuming of course that `a` is a port name.

Another valid array type is `Int[3]`. This is a list of three integers, which it stores under indices $0, 1, 2$. If `A` is an array of this type, then `A[i]` is undefined for indices i other than $0, 1, 2$.

We construct arrays mainly by means of assignments:

```
vars A:Bool[100]; i:Int
i:= 0; while i < 100 do A[i]:= true
```

This declares an array `A` of type `Bool[100]`, and makes sure that at every index the value `true` is stored. In LARIS, another way to achieve this is by means of the expression

```
{(*,true)}: Bool[100]
```

Values at particular indices can be retrieved from such arrays by traversing the list from left to right, until the index in question matches all but the last item in a tuple. The final item in the tuple then gives the desired value. If no match is found, the default value is given.

$*$ is the *wild-card* character: it matches with anything. Thus in the above case, if one wishes to know the value at index 15, one finds an immediate match with $*$; the produced value is `true`.

Another example is the array that is `false` at 3 and `true` otherwise:

```
{(3,false), (*,true)}: Bool[100]
```

To find the value at 15 here, we traverse the list. The first index we find is 3, with value `false`. 3 does not match 15, so we proceed in the list and we find the same match as before: the produced value is again `true`. Including the type `Bool[100]` is necessary, because we wish to unambiguously know the type of an expression. This is not possible with the notation {`(*,true)`}, which could as easily have been of type `Bool[Port]`.

As a more involved example, with pairs as indices, let `A` be the following array:

```
{(1,2,true), (1,*,false), (*,2,true), (*,*,false)}: Bool[Int,Int]
```

The array is presented as a list of tuples of the form `(a,b,c)`, where the first two represent the index and `c` represents the value at that index. To know the value of `A` at a particular index, one traverses from left to right until either a match is found, or the end of the list is reached and the default value is produced (which in this case is `false`, because the values of `A` are booleans). For instance, `A[1,2]` = `true`, `A[1,3]` = `false`, `A[3,2]` = `true`, and `A[3,3]` = `false`. `A` can be written much shorter, simply as {`(*,2,true)`}`: Bool[Int,Int]`. Order is vitally important in this notation. Let `B` be the result of permuting the first two tuples in the list of `A`:

```
{(1,*,false), (1,2,true), (*,2,true), (*,*,false)}: Bool[Int,Int]
```

Then `B[1,2]` = `false`, and `B` is equivalent to:

```
{(1,2,false), (*,2,true)}: Bool[Int,Int]
```

3.1.6 Clock types

All the previously mentioned types are referred to as *data types*. The next types we introduce are of a more dynamic nature, namely the *clock types*. There are three kinds.

- First there is the type `Timer`. Variables of this type denote simple stop-watches, clocks that can count upwards and may be started and stopped.

  To understand some of the possibilities of variables of this sort, consider the following specification fragment:

  ```
  vars X:Timer; efficient:Bool
  initial start X; ! A01()
  ```

```
mes ? A01() = if active X
                 then {efficient:= (value X < 2); stop X};
              start X; ! A01()
```

This represents part of the specification for an LSC that has a local variable `X` of type `Timer` and a boolean variable `efficient` that is set to `true` if handling of internal telegrams is deemed efficient enough (i.e., if this can be performed within two time units, or to be more precise, before two slice borders have been passed).

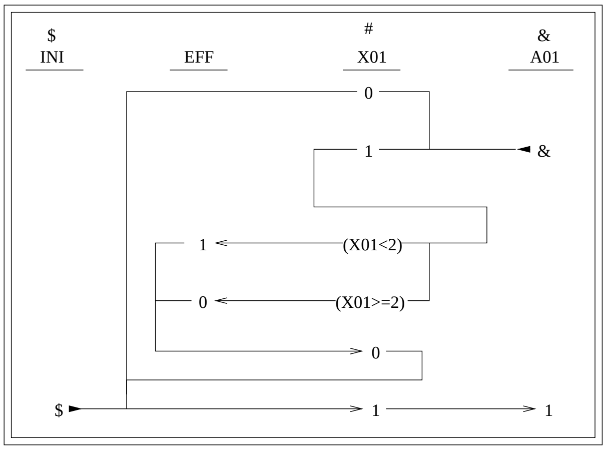
A component's specification consists of a sequence of instructions to be carried out when a telegram of a particular form has been found in its input buffer, denoted by `mes ?`. The clause above gives the instructions that are to be carried out if an internal telegram has been received with name `A01` and no data (hence `()`). The specification also contains some initial instruction to be carried out when the component is started up.

A timer is either active or inactive. The expression `active X` gives a boolean `true` in the first case and `false` otherwise. The only way to activate a timer is by means of the instruction `start X`. This resets the counter of the timer to zero and thereafter increases the counter by one after every time unit. The only way to deactivate a timer is by means of `stop X`, which simultaneously resets the counter to 0. In the specification above, `active X` always produces `true`, as at the point of this test `X` is always active.

Another piece of information that can be extracted from the timer, besides whether its active or not, is the amount of time it has been running. This may be done by means of the expression `value X`. If `X` is active, `value X` gives the number of time units that have passed since its activation. If `X` is not active, then `value X` produces 0.

The above specification describes a component that starts a timer `X` and then sends itself an internal telegram `A01` (by '`! A01()`'). When this telegram is retrieved from the component's buffer and processed, it is verified how long the timer has been running. It is deemed efficient only if the timer has been running less than two time units. After this the timer is restarted (starting at zero again) and the test repeats itself, indefinitely.

A specification in EURIS with the same behaviour is depicted below. Note that a toggle `INI` is needed for the initial behaviour. The `X` and `efficient` have been adapted to EURIS-style and changed to `X01` and `EFF`.



- A second clock type is `Timeout`. Variables of this type represent clocks that count down from a prespecified value, and when their time is up they send a series of internal telegrams. At the sending of these telegrams they are deactivated.

  Time-outs are activated by instructions of the form

  $$\gg\# \;\; \mathtt{X} \;\; n \;\; ! \;\; \mathtt{N}(d_1, \ldots, d_m)$$

  where `X` is the time-out variable, n is the delay, and the telegram to be sent has name `N` and data parameters $d_1, \ldots, d_m$. Time-outs are deactivated either when their time is up, or by means of the instruction `stop X`.

  Time-outs may also be queried as to their status and their value. When `X` is an active time-out, `value X` produces the time to go before the telegram is sent, rounded down. Otherwise it produces 0.

  When a time-out is activated that is already active, the old values are thrown away. Thus, in the following:

  ```
  vars X:Timeout; i:Int
  i:= 1; while i <= 100 do {>># X i ! A01(); i:= i+1}
  ```

  the time-out `X` is activated a hundred times, but eventually, only a single internal telegram is sent, a hundred time units after the while-loop has been abandoned. This does, however, depend on the speed of the while-loop and the length of the time units.

- Finally, we have the clock type `Cycler`. These are similar to time-outs, except that when a cycler variable `X` has been activated by the instruction

  $$@ \;\; \mathtt{X} \;\; n \;\; ! \;\; \mathtt{N}(d_1, \ldots, d_m)$$

then after the time has run out and the telegram has been sent, the cycler is not deactivated, but the delay n is reinstalled. After this delay has passed the telegram is sent again, after which the whole process repeats itself. A cycler `X` can only be deactivated by means of the instruction `stop X`.

3.1.7 Names of variables

For every type above we may declare variables to be of this type. For instance, we may say `X:Int`, declaring the variable `X` to be of type `Int`. This variable may then be used as if it were an integer: addition may be applied to it, but boolean operations may not.

A variable name can be any string starting with an upper- or lowercase character, followed by any string of numbers and characters, interspersed with underscores. A variable name may not end with an underscore.

There are some other restrictions on variable names: the name may not occur as an inhabitant of an enumerated type. For instance, if the type `Route` was defined as above (on page 17), then `normal` could not be used as a variable name.

Furthermore, *keywords* such as `mod` and others to be introduced later, are also not available as names for variables.

3.2 Statements

When a telegram of a particular type is to be processed, a series of appropriate instructions is carried out. This series of instructions is written as a *statement*. Such a statement is built up from basic statements, such as assignments, and control constructs well-known from imperative programming languages, such as if-then-else constructions.

While discussing types and expressions, we have already come across several such statements. In this section we give an exhaustive list of the statements that are allowed in LARIS.

Assignments LARIS has assignments to data-variables. Thus, if `X` is a variable and `E` is an expression of the same data type, then `X:= E` is an assignment. For instance, if `X` is a variable of type `Int[Int]`, then

```
X:= {(0,0), (1,2), (*,1)}
```

is a possible assignment.

It is also possible to change array-variables at particular indices. For instance, let `X` be as above. Then the effect of the previous assignment can be achieved by:

```
X[*]:= 1; X[1]:= 2; X[0]:= 0
```

The first assignment in this series is of interest: we allow the wild-card `*` in these index assignments. If `X` is of type `Int[Int,Int]`, then `X[*,2]:= 4` changes the value of `X` at all indices $(n, 2)$ to 4, where n is any integer.

Telegrams

External `E` $|>$ `p` ! `A01`$(d_1, \ldots, d_m)$ sends the telegram `A01` with data parameters $d_1, \ldots, d_m$ to port `p` of component `E`. This statement is only executable if `E` is not the name of the current component.

Internal ! `A01`$(d_1, \ldots, d_m)$ places the internal telegram with name `A01` and data parameters $d_1, \ldots, d_m$ in the input buffer.

Clock statements

- `start X`, with `X` of type `Timer`, resets `X` to 0 and activates it.
- $\gg$# `X` n ! `A01`$(d_1, \ldots, d_m)$, with `X` of type `Timeout`, activates the time-out `X` and sets the delay to n, where $n > 0$. The message handler carries on, but when the delay has passed, internal telegram `A01` with data parameters $d_1, \ldots, d_m$ is placed in the input buffer, and `X` is deactivated.
- @ `X` n ! `A01`$(d_1, \ldots, d_m)$, with `X` of type `Cycler`, activates the cyclic telegram sender `X`. The length of the cycle is $n > 0$. At the end of each cycle the telegram `A01` with data parameters $d_1, \ldots, d_m$ is placed in the input buffer.
- `stop X` deactivates `X`.

Procedure calls $P(d_1, \ldots, d_m)$ is a procedure call, with P a procedure name and $d_1, \ldots, d_m$ as its arguments.

Parameter passing in procedures is treated *call-by-value*, as opposed to *call-by-reference*. This means that only the values of variables are passed on to a procedure, not the variables themselves. For example, consider the procedure

```
proc set1(X:Int) = X:= 1
```

Then the sequence `X:= 0; set1(X)` would bring us to a state where `X` denotes 0, and not 1, as one might expect under a call-by-reference interpretation of parameter passing.

Skip `skip` is the empty statement: it does nothing.

Control constructions These are familiar from imperative languages:

- `if` E `then` A.
- `if` E `then` A `else` B.

- `while` E `do` A.
- $$\begin{array}{lll} \texttt{case}\,X\;\texttt{in} & & \\ \{\;\; E_0 & : & A_0 \\ \;\;\;\;\vdots & \vdots & \vdots \\ \;\;\;\; E_n & : & A_n \\ \;\;\;\; \texttt{otherwise} & : & A_{n+1}\;\; \} \end{array}$$

  checks whether the value of the data-variable X equals that of E_0. If it does, A_0 is carried out. If not, it is checked whether X equals E_1. It continues in this fashion until either a match is made, or the `otherwise` clause is reached, upon which A_{n+1} is carried out. The `otherwise` clause is obligatory, to specify the action should the cases $E_0, \ldots, E_n$ not cover all possible values of X.
- $A_1; A_2$ is the sequential composition of A_1 and A_2: first carry out A_1, then A_2. The symbol ';' is used as a separator, and not as a terminator. So $A_1; A_2;$ is not syntactically correct in LARIS.

To disambiguate statements we may use curly braces '{' and '}'. For instance, the statement

```
X:= 0; if X == 0 then X:= 1 else X:= 2; X:= X+1
```

is ambiguous. It could be parsed as

```
X:= 0; if X == 0 then X:= 1 else {X:= 2; X:= X+1}
```

which yields the value 1 for `X`. Another parsing is

```
X:= 0; {if X == 0 then X:= 1 else X:= 2};  X:= X+1
```

which assigns `X` the value 2.

Statements are used in LARIS in a variety of places:

- to specify the initial behaviour of a component at startup;
- to specify the response of a component to a telegram;
- for the definition of procedures;
- to specify the behaviour in case of an undesired or unexpected result (panic).

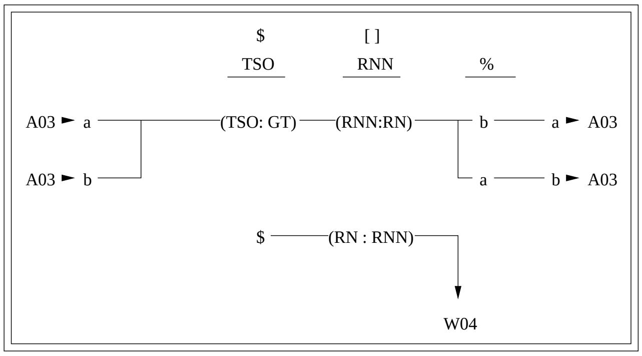
Together, such statements form an *LSC*; these are introduced in the next section, in the course of which the various uses of statements listed above are explained.

3.3 LSCs and behaviours

A typical LSC in LARIS looks as follows:

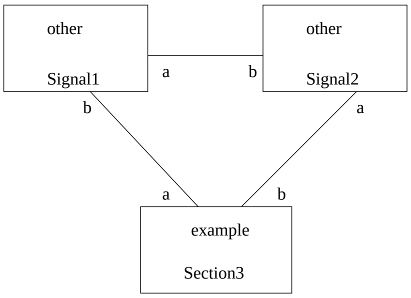
```
LSC example (C_list:Component[Port], P_list:Port[Port]) =
   vars RNN:Int; TSO:Bool

   initial ! TSO()

   mes a ? A03(RN:Int; GT:Bool) = entry(b,RN,GT)
   mes b ? A03(RN:Int; GT:Bool) = entry(a,RN,GT)

   proc entry(P:Port; RN:Int; GT:Bool) =
      vars Old:Bool
      Old:= TSO; TSO:= GT; TSO(Old); RNN:= RN;
      C_list[P] |> P_list[P] ! A03(RN,GT)

   proc TSO(Old:Bool) = if Old =/= TSO then ! TSO()

   mes ? TSO() = Inf |> inf ! W04(RNN)

   panic Log |> log ! P01(self)
```

This could be the translation of the following EURIS LSC (although this is only a possible translation, not the one we actually have in mind, see Section 3.6):



So an LSC begins with the keyword `LSC`, followed by its name, in this case `example`.

The LSC `example` has two parameters, `C_list` and `P_list`, to be supplied when a specific component is desired. An LSC is but a generic component, which can be used to specify many different components. Both `C_list` and `P_list` have as indices the port names of the LSC `example`, being `a` and `b`. Intuitively, `C_list`[`a`] and `P_list`[`a`] represent the component and the port to which the `a`-port of `example` is connected, respectively. Similarly for `C_list`[`b`] and P_list[b]. For instance, suppose we have the following system layout:



If Section3 is the name of the current component, of LSC type example, then the intended parameters would be:

$$\begin{array}{lcl} \texttt{C\_list} & = & \{(\texttt{a},\texttt{Signal1}),\ (\texttt{b},\texttt{Signal2})\} \\ \texttt{P\_list} & = & \{(\texttt{a},\texttt{b}),\ (\texttt{b},\texttt{a})\} \end{array}$$

Let us proceed with the examination of the LSC example. After the list of to-be-supplied parameters we find the symbol '=', followed by a list of LSC variable declarations, preceded by the keyword vars. The values of these variables may change, as opposed to LSC parameters, which remain fixed. At the start, the variables are assumed to have the default values for their types.

After these parameter and variable declarations, the initial behaviour is defined. In this case, all that happens is that an internal telegram with the name TS0 and no data is placed in the input buffer. The EURIS specification has the same initial behaviour: initially, all toggles ($-variables) are assumed to fire. The toggle TS0 is modelled by the boolean variable TS0, the procedure TS0(b:Bool), and the telegram TS0(), which concatenates all the flows of TS0 (if there are more than one) in a certain order.

Now we have reached the heart of an LSC: its list of behaviours. These come in two sorts: responses to telegrams and procedure definitions. The first sort of behaviour we find deals with a certain kind of telegrams found in the input buffer, and defines a response, in the form of a statement. Such behaviours start with the keyword mes ?. The first two behaviours define responses for A03-telegrams found on respectively the a- and b-port. We see that external A03-telegrams have type (Int,Bool), that is, such telegrams carry two pieces of data, the first an integer, the second a boolean. The responses that are defined both first call a procedure entry. The definition of this procedure is the third behaviour. Procedure definitions begin with the keyword proc, followed by the name of the procedure, a list of parameters, the symbol '=', possibly a list of variable declarations, and a statement. The definition of entry contains all these elements.

The definition of the procedure entry involves a local variable Old. LARIS does not allow name clashes between local variables and parameters in procedures and global variables and parameters in LSCs. This is to avoid possible confusion as to which version of a variable an assignment is made. It is perfectly all right, though, for another procedure definition or message handling to use the same names for local variables or parameters. The same convention about the choice of names applies to local variables of telegram responses. In the example none of these have local variables, but they are allowed nevertheless.

After the definitions of the procedures `entry` and `TSO` we find another telegram response, but this time without a port name before the '`?`'. This is the response to the finding of an internal telegram with the name `TSO` (and no data) in the input buffer. As opposed to EURIS, in LARIS such internal telegrams may carry data.

An LSC has a mandatory `initial` clause and a possibly empty sequence of behaviours such as telegram responses and procedure definitions. The final clause of an LSC is the `panic` clause, specifying the behaviour of the component should something unforeseen occur. Examples of such unexpected events are the arrival of a telegram that is not well typed or for which no response has been specified, an attempt to divide by zero, an attempt to send an external telegram to itself, or trying to retrieve a value of an array at an index which is outside of the range of the array. In fact, the panic behaviour could occur at any point: some unforeseen event could have its origin at a source that lies outside the scope of the specification. Of course, manufacturers of interlockings would have to be constrained in when panic behaviour actually may occurs: it should not occur when there is no cause for alarm.

The result of going into panic mode is that the input buffer is emptied and the statement in the panic clause is carried out immediately, terminating whatever it was doing instantly. In the example, we have chosen to send a telegram `P01` to the logistic level, with as datum the name of the component itself (`self`).

3.4 Systems

We have indicated how to specify LSCs in LARIS. To obtain concrete components, constituting a complete control system, such LSCs have to be augmented with a name and a choice of parameters. This is done in the system part of a specification.

Consider again the system layout as depicted on page 27. Suppose we wish to specify the subsystem containing as components just `Signal1` and `Section3`. Suppose also that the LSC with name `other` has similar parameters as `example`, specified in the previous section. The desired subsystem would be specified in LARIS as follows:

```
System system_example =
   External components = {Signal2}
   External ports = {a,b}
   Signal1 other  ({(a,Signal2),(b,Section3)}, {(a,b),(b,a)})
   Section3 example ({(a,Signal1),(b,Signal2)}, {(a,b),(b,a)})
```

Thus, a system begins with the keyword `System` followed by the name of the system. This name is of no real relevance. It is only used to refer to the system. After this name we find again the symbol '`=`', followed by the set of external components and the set of external ports.

The set of external components is used to specify names of components that are not part of the specification, but to which reference is made nevertheless. For instance, the name `Signal2` is used in a parameter of `example`. We need to know, somehow, that this string is of type `Component`. This is now simple; it is a part of the set of external components, so its type can be deduced from this fact.

For similar reasons we must include the set of port names that are not part of the subsystem under scrutiny. In the example, we have placed the port names `a` and `b` in this set, although we could have left the set empty, as `a` and `b` occur as port names in the LSC `example`, so the types of these symbols are already determined.

It is not necessary (in fact, not allowed), to put `Log` and `Inf` in the set of external components. These are assumed to be part of any system: their types are predetermined. Likewise, the set of external ports may not contain the port names `log`, `inf`, `left`, and `right`.

After the definition of these sets we find two component bindings. These consist first of a component name, then the name of the LSC to which it is to be bound, and then a list of parameters that this LSC requires. These parameters are simply expressions of the correct type, but without variables.

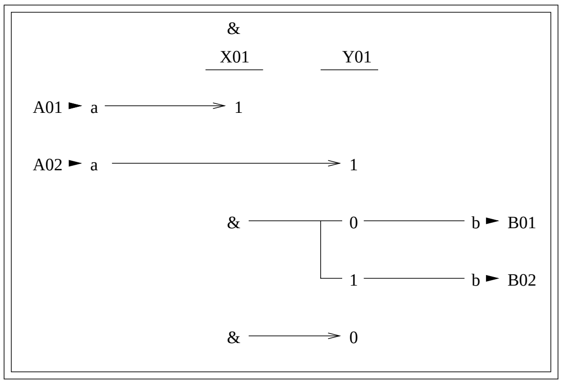
We have now informally described all parts of LARIS. A specification consists of a number of type definitions, a number of LSCs, and a system declaration. The description of LARIS is formalized in Sections 4 and 5.

3.5 LARIS versus EURIS

We look at LARIS from the perspective of EURIS, and list a number of important differences between these two formalisms.

1. EURIS allows more nondeterminism within components than LARIS. This behaviour manifests itself in the interpretation of multiple one-shots and toggles. An example is presented below.

   When the `A01`-telegram is processed and the one-shot `X01` is set, two internal telegrams, corresponding to the two flows of `X01`, are placed in the input buffer, in an arbitrary order. Furthermore, placing these telegrams in the buffer is not an atomic action, as it is possible that the telegram `A02` is placed in the buffer between the two `X01`-telegrams.



In EURIS, events that are not explicitly ordered may occur in any order. LARIS takes issue with this philosophy. It is to the advantage of the size of the state-space if events are ordered as much as possible by the specification. Interactions between components are not available for such an ordering, but activities within a single component definitely are.

This viewpoint has as a consequence that a LARIS translation of the above would only exhibit some of the possible behaviours of the EURIS specification. We present here one possible translation, where we have chosen to execute the top `X01`-flow before the bottom one.

```
LSC ordering (Eb:Component; Pb:Port) =
   vars X01,Y01:Bool

   initial skip

   mes a ? A01() = X01()
   mes a ? A02() = Y01:= true

   proc X01() = if ~X01 then {X01:= true; ! X01()}

   mes ? X01() =
      X01:= false;
      if Y01
         then Eb |> Pb ! B02()
         else Eb |> Pb ! B01();
      Y01:= true

   panic Log |> log ! P01(self)
```

The one-shot is modelled by means of the procedure `X01`, which first tests whether the one-shot is already set. If not, it sets the one-shot and places a single telegram in the buffer, instead of the two found in the EURIS specification. The flow corresponding to this telegram first unsets the one-shot and then simply concatenates the two flows in the EURIS specification.

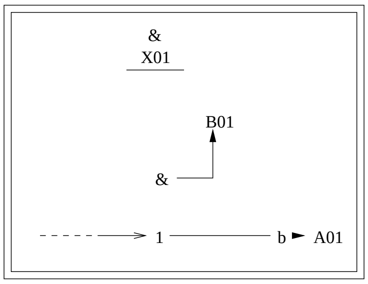
The LSC has two parameters, `Eb` and `Pb`, the first representing the component to which the `b`-port is connected, and the second the port on `Eb` to which the `b`-port is connected. These are needed when one wishes to send a telegram along the `b`-port.

2. LARIS allows the sending of multiple external telegrams within one flow. Thus, the following could be part of a single LARIS flow:

```
Eb |> Pb ! A01(); Log |> log ! B01()
```

This first sends `A01` to port `Pb` of component `Eb` and then a telegram `B01` to the logistic level.

To achieve a similar effect in EURIS, one needs to resort to one-shots:



If the one-shot is invoked purely for reasons of sending two telegrams, it is obvious that the LARIS description is simpler, shorter, and likelier to avoid undesired behaviour, such as interference by external telegrams.

3. In EURIS, route telegrams are sent along route ports, while central telegrams are passed on using central lists. In principle, a central telegram is sent from one component to another, using only the name of the current component, which is recorded in the central list, and the direction of the central telegram.

In LARIS this concept is taken to its logical conclusion: any component can send a telegram to any other component, provided the name of the intended recipient is known. The idea of EURIS that a telegram is received on a certain port, although this information could be incorporated into the telegram itself, is retained in LARIS.

Thus, LARIS allows more freedom in communication than EURIS. For instance, whereas in EURIS only a single central list is ever used, in LARIS one could in principle use more than one such list of components.

Also, in EURIS, telegrams are usually sent along certain ports. This does not happen in LARIS. In LARIS it is possible for a component without ports to send signals to the port of another component.

4. In EURIS it is often not entirely clear which information is carried by a telegram. In principle every telegram-field is present in any telegram, although only a limited few are used, or even relevant, for specific telegrams.

   In LARIS, it is always clear which information is carried by a telegram: it is precisely the information carried in its list of data, in combination with its name.

5. In EURIS internal telegrams carry no data, while in LARIS this is possible.

6. LARIS has the ability to define enumerated finite types. Where these are useful, EURIS often has to resort to using the sort `Int`, which is usually too large a type for the application one has in mind.

7. LARIS has an explicit means for dealing with the arrival of unexpected telegrams and fatal errors discovered during message handling, or for dealing with any other type of calamity, by means of the `panic` procedure. EURIS has no means of specifying the behaviour of a component, should this happen.

3.6 Translating EURIS into LARIS

LARIS is meant to be a textual representation of (an interpretation of) EURIS. We describe a possible translation of some EURIS constructions into LARIS. We have already seen some indications of how such a translation could be achieved. We focus on two EURIS constructions, namely toggle-variables and central telegrams. Hopefully, the reader will be able to conceive of translations of other EURIS constructions by analogy.

3.6.1 Toggles

First, let us see how to deal with 'toggles' or '$'-variables in LARIS. As an example we take a part of UniSpec [16], namely a portion of page 5 of the track specification (July 1, 1997), which contains a number of such variables. The portion of interest is reproduced in Figure 5. The translation of this EURIS LSC into a LARIS LSC would look as follows:

```
LSC track
  (Ea,           % Component connected to the a-port.
   Eb:Component; % Component connected to the b-port.
   Pa,           % Port on Ea connected to the a-port.
   Pb:Port)      % Port on Eb connected to the b-port.
```

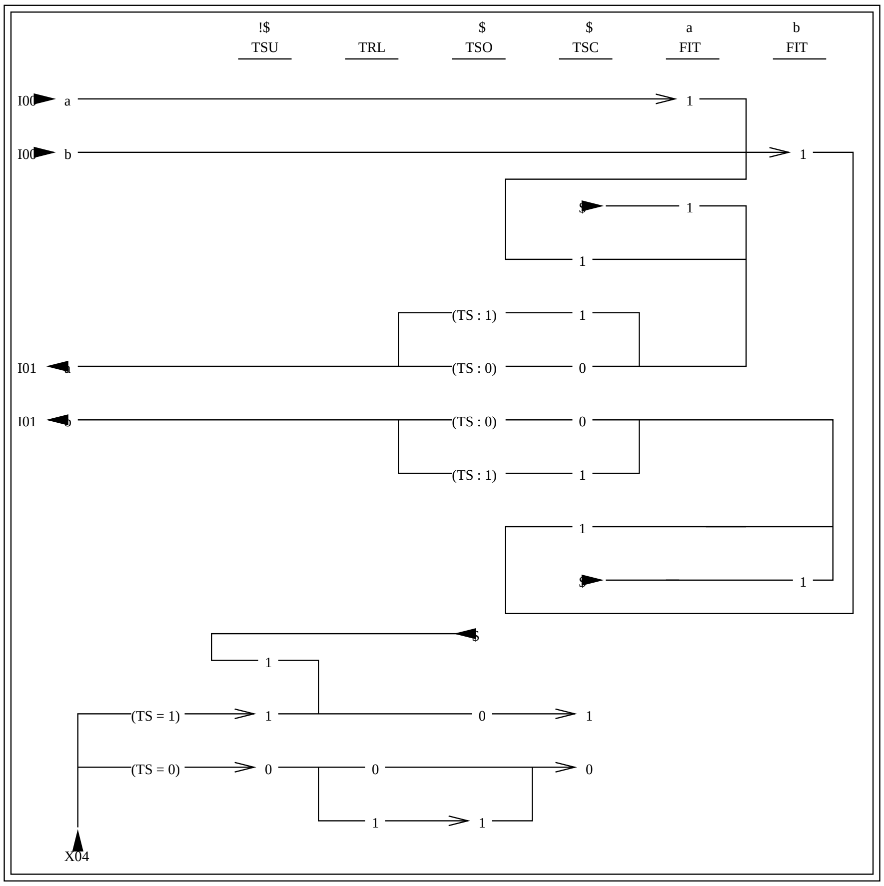


Figure 5: Part of the specification of a track, focusing on toggles.

```
=
vars TSU,TSO,TSC, % Three toggle-variables
     TRL,FITa,FITb:Bool

initial ! TSU(); ! TSO(); ! TSC()
                  % Initially, fire the three toggles.

mes a? I00() = FITa:= true; if TSC then I01a()

proc I01a() =
   vars TS:Bool
   if TSC
```

```
      then TS:= true
      else TS:= false;
   Ea |> Pa ! I01(TS)

mes b? I00() = FITb:= true; if TSC then I01b()

proc I01b() =
   vars TS:Bool
   if TSC
      then TS:= true
      else TS:= false;
   Eb |> Pb ! I01(TS)

mes inf? X04(TS:Bool) =
   if TS
      then {TSU(true); TSOflow()}
      else {TSU(false);
            if TRL then TSO(true);
            TSC(true)}

proc TSU(b:Bool) = if TSU =/= b then {TSU:= b; ! TSU()}

proc TSO(b:Bool) = if TSO =/= b then {TSO:= b; ! TSO()}

proc TSC(b:Bool) = if TSC =/= b then {TSC:= b; ! TSC()}

mes ? TSO() = if TSU then TSOflow()

proc TSOflow() = if ~TSO then TSC(true)

mes ? TSC() = if FITa then I01a();
              if FITb then I01b()
```

Note that the above LARIS LSC is in fact not correct, syntactically. There are two items missing.

First, at various places an internal telegram `TSU` is sent, but there is no response defined for such telegrams. Such a response is also not present in the partial EURIS specification. If a complete specification were given, we would have to define the appropriate response.

Second, there is no `panic` statement. This is also missing in the EURIS specification (naturally, because EURIS has no means of expressing these). One could choose to add `panic Log |> log ! P01(self)`.

A literal translation of a EURIS specification sometimes yields LARIS statements that could be expressed more succinctly. For instance, consider the body of the `I01a` procedure definition. This whole body could be replaced by the single statement:

```
Ea |> Pa ! I01(TSC)
```

A similar remark holds for the definition of `I01b`.

3.6.2 Central telegrams

We consider the translation of central telegrams into LARIS. There are really two uses of such telegrams in EURIS. First, a central component may wish to set the values of internal variables of certain peripheral components. In this case, the central component is the cause of the activity. Second, a number of components may report certain values to a common central component, which may then act upon the values it receives. We present examples illustrating these two uses, and show how these constructions may be uniformly translated into LARIS.

First, consider the EURIS LSCs as depicted in Figures 6 and 7. The LARIS translation for active centers is the following. It asks for two parameters, a list of components (its central list), and the length of this list. When a central telegram `C01` is to be sent, it sends along this central list, together with a pointer. At the start of the central list the name of the warning device is placed. This ensures the eventual return of the telegram to this device. The pointer indicates where we are in the central list. This information is used when the central telegram initiates another one.

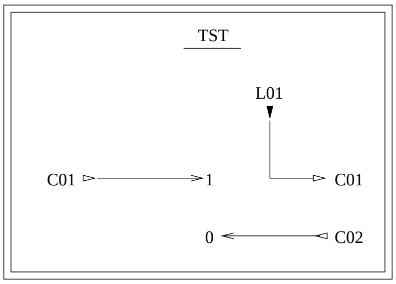


Figure 6: LSC for 'ACTIVE CENTER'.

```
LSC active_center (MyList:Component[Int]; MyLength:Int) =
   vars TST:Bool

   initial skip

   mes right? C01(List:Component[Int]; Length,Pointer:Int) =
      TST:= true
```

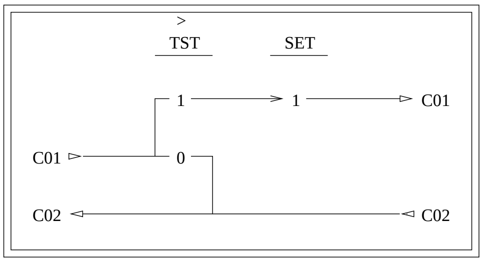


Figure 7: LSC for ‘PASSIVE PERIPHERY’.

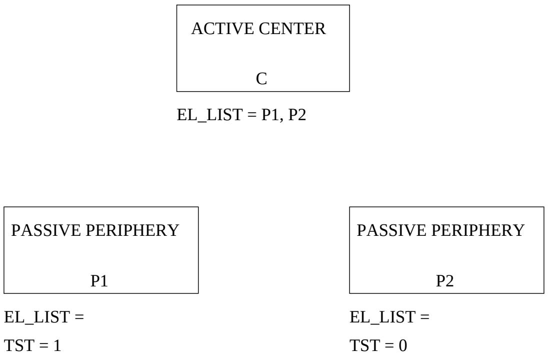
```
   mes left? C02 (List:Component[Int]; Length,Pointer:Int) =
      TST:= false

   mes log? L01() =
      vars List:Component[Int]
      List:= MyList;
      List[0]:= self;
      List[1] |> right ! C01(List,MyLength,1)

   panic Log |> log ! P01(self)
```

The LSC for passive peripheral components does not need a central list as a parameter, as it never actually sends a central telegram of its own accord.

```
LSC passive_periphery (TST:Bool) =
   vars SET:Bool

   initial skip

   mes right? C01(List:Component[Int]; Length,Pointer:Int) =
      if TST
         then SET:= true;
              Pointer:= (Pointer+1) mod (Length+1);
              List[Pointer] |> right ! C01(List,Length,Pointer)
         else sendC02(List,Lenght,Pointer)

   mes left? C02(List:Component[Int]; Length,Pointer:Int) =
      sendC02(List,Lenght,Pointer)

   proc sendC02(List:Component[Int]; Length,Pointer:Int) =
      if Pointer > 0
```

```
         then Pointer:= Pointer-1
         else Pointer:= Length;
      List[Pointer] |> left ! C02(List,Length,Pointer)

   panic Log |> log ! P01(self)
```

A typical system layout using such LSCs is depicted below:



In LARIS this corresponds to the following:

```
System active =
   External components = {}
   External ports = {}
   C  active_center({(1,P1),(2,P2)}:Component[Int],2)
   P1 passive_periphery(true)
   P2 passive_periphery(false)
```

We consider an example where peripheral components pass values to a warning device of their own accord. A concrete example of this may be found in UniSpec [16], namely in the relation between a series of approach monitors and a single warning device. We adopt a simplified version of this concrete example: Figures 8 and 9 contain the EURIS LSCs and Figure 10 contains a potential layout of such components.

The translation into LARIS of such a system yields the following specification. First, we have a type definition. In the LSC above we see that the telegram-field `AM` may take three values: 0, 1, and 2. In EURIS one chooses the type `Int` for `AM`, although `Int` allows many more values. We let `AM` be of type `AMS` (Approach Monitoring Status), a type with three inhabitants: `occupied`, `complete_unoccupied`, and `incomplete_unoccupied`, represented in LARIS by 0, 1, and 2, respectively.

```
AMS = {occupied,
       complete_unoccupied,
       incomplete_unoccupied}
```

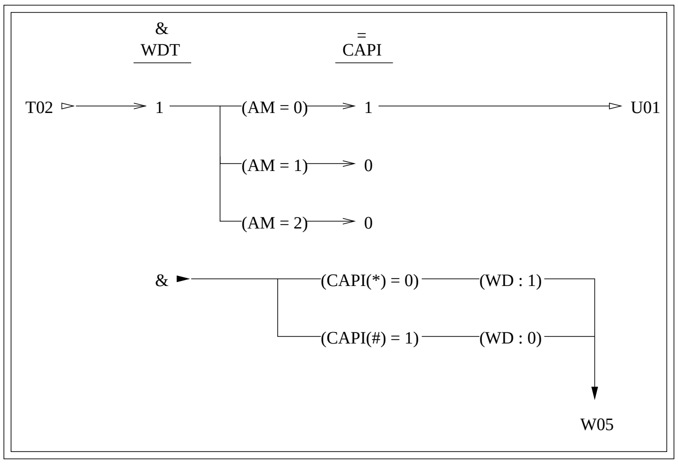


Figure 8: LSC for 'WARNING DEVICE'.

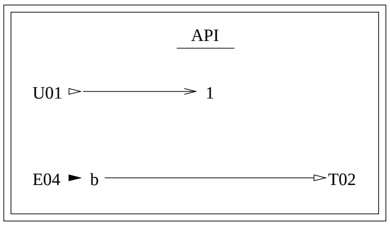


Figure 9: LSC for 'APPROACH MONITOR'.

Next, we give a definition of an LSC for warning devices. A warning device does not need a central list as it does not send central telegrams of its own accord. The presence of a *multivar* `CAPI` in the EURIS LSC for warning devices necessitates a list as a parameter in the LARIS LSC for warning devices, which we call `Monitors`. In EURIS, a multivar keeps track of a number of values, one for each component from which values may be expected. `Monitors` is a list of these components. In the case of the warning device, these are all approach monitors. `Number` is intended to be a positive integer, representing the number of approach monitors to which the warning device is connected. The values of `Monitors` at indices from 1 to `Number` yield the relevant components.

```
LSC warning_device (Monitors:Component[Int]; Number:Int) =
   vars WDT:Bool; CAPI:Bool[Component]
```

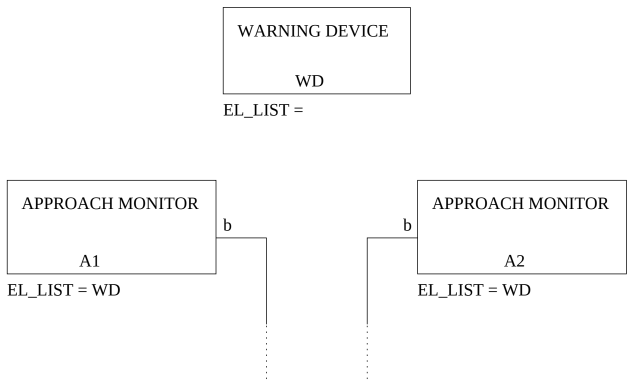


Figure 10: Layout featuring a warning device and two approach monitors.

```
initial skip

mes right? T02(E:Component[Int]; AM:AMS) =
   WDT();
   if AM == occupied
      then {CAPI[E]:= true; E |> right ! U01()}
      else CAPI[E]:= false

proc WDT() =
   if ~WDT then {WDT:= true; ! WDT()}

mes ? WDT() =
   vars i:Int; TESTCAPI, WD:Bool
   TESTCAPI:= false;
   i:= 1;
   while i <= Number do
      TESTCAPI:= TESTCAPI | CAPI[Monitors[i]];
   if TESTCAPI
      then WD:= false
      else WD:= true;
   Inf |> inf ! W05(self, WD)

panic Log |> log ! P01(self)
```

Next, we give a definition of an LSC for approach monitors. From the example on page 35 one might expect a central list as a parameter. This would definitely be a more uniform approach. However, using a list for a situation such as it exists between a warning device and its approach monitors would be overkill. The approach monitor initiates a central telegram `T02`, which it sends to the warning device. For this it only needs to know the name of this warning device. `T02` must carry the name of the approach monitor (using `self`) for the following two reasons. First, a multivar may have to be updated, and then we have to know which version of the multivar to adapt. Second, the telegram `U01` may be sent back to the approach monitor, and the warning device has to know the name of the component to send this telegram to.

```
LSC approach_monitor (E:Component) =
   vars API:Bool

   initial skip

   mes b? E04() = E |> right ! T02(self)

   mes right? U01() = API:= true

   panic Log |> log ! P01(self)
```

Finally, we present the system declaration itself, which consists mainly of filling in the required parameters for the specific warning device and approach monitor in question.

```
System warning_device_and_approach_monitors =
   External components = {}
   External ports      = {}
   WD warning_device   ({(1,A1),(2,A2)}:Component[Int], 2)
   A1 approach_monitor(WD)
   A2 approach_monitor(WD)
```

3.7 LARIS versus EURIS$_{DTO}$

EURIS$_{DTO}$ [4] is an interpretation of EURIS in terms of discrete-time process algebra. We compare EURIS$_{DTO}$ to both EURIS and LARIS. EURIS$_{DTO}$ differs from, or is more explicit than, EURIS in the following respects.

1. EURIS$_{DTO}$ presupposes a discrete time domain and a global clock. All components are synchronized to this clock.

2. EURIS$_{DTO}$ assumes that communication is synchronous.

3. In EURIS$_{DTO}$, a flow can send at most one telegram to one other component, i.e., there are no broadcasts.

4. In EURIS$_{DTO}$, a telegram-generating process can generate only one internal telegram at the time, i.e., expressions like the one depicted below are excluded.

```
        &
       ABC

        &  >——

  ——<   &
```

5. In EURIS$_{DTO}$, internal telegrams are generated at the border of a time slice and its successor. Hence, the telegram-generating processes (variables) `@`, `?#`, and $\gg$`#` count in time slices, and for the generation of telegrams they are synchronized on slice borders. Also, the timer `#` counts in time slices, and the telegram-generating processes `$` and `&` do not generate telegrams on-the-fly, but on slice borders.

6. EURIS$_{DTO}$ does not allow the use of defaults. Especially, reference to a telegram-field that has not been assigned before results in a deadlock.

Furthermore, some simplifications have been excluded from EURIS$_{DTO}$, in order to keep the definitions focused on the core semantical aspects. In the category of these simplifications fall: a variable declaration declares one object, e.g., declarations like

```
a/b          &@
ABC   and   XYZ
```

are excluded; functions and so-called boxes are excluded; and, case statements are excluded.

Distinctions between LARIS and EURIS$_{DTO}$ stem from four sources. First, LARIS is a symbolic syntax that is fully defined (including static semantics) plus a semantics, whereas EURIS$_{DTO}$ is a large subset of EURIS its informal graphical syntax plus a semantics. Second, LARIS is a generalization of EURIS notions, whereas EURIS$_{DTO}$ is a restriction. Third, LARIS and EURIS$_{DTO}$ differ in some semantical choices. Fourth, LARIS and EURIS$_{DTO}$ differ in the style of presentation of the semantics: respectively, a declarative style in standard logical notation and set theory versus a procedural style in algebraic specification and process algebra. In the following paragraphs the latter three sources of differences are expounded on.

From the differences between EURIS$_{DTO}$ and EURIS that were listed just above, and the differences between LARIS and EURIS in Section 3.5, one can infer a number of distinctions between LARIS and EURIS$_{DTO}$. Most notable are the following.

- LARIS offers one large uniform communication space, i.e., one addresses a component, not a port. EURIS$_{DTO}$ has the typical EURIS route ports.

- LARIS is fully and strongly typed and offers a richer set of types and operations. EURIS_{DTO} offers the EURIS types and operations, and is only typed to some extent.
- LARIS offers default values. EURIS_{DTO} generates a deadlock when a non-existing or a non-initialized variable is referred to.
- LARIS facilitates instantiation and binding of components. EURIS_{DTO} does not offer this.
- In LARIS, sending a telegram is just an action. In EURIS_{DTO}, sending a telegram can only be used as the terminal action of a flow.

LARIS and EURIS_{DTO} are both based on discrete global time; the time slice forms the basic unit for timed processes. Distinctions are found in other semantic choices.

- LARIS has asynchronous communication, which is modelled by means of buffers in channels. EURIS_{DTO} has synchronous communication.
- Corruption of telegrams is modelled and operationally catered for by means of a panic procedure. EURIS_{DTO} does not model corruption.
- Generation of telegrams by `$` and `&` variables in LARIS is dynamic, while in EURIS_{DTO} the generation of such telegrams is time-dependent.

Finally, the presentation of the semantics. LARIS and EURIS_{DTO} both have a formal semantics. A formal semantics is an assignment from the well-formed expressions of a language to a collection of mathematical constructions. Essentially, the formal semantics of both languages use the same type of mathematical constructions: process graphs. The difference discussed here, not to be confused with the semantical choices discussed above, is in the presentation. LARIS follows a direct approach in the sense that LARIS expressions are mapped directly to process graphs. In EURIS_{DTO} one level of indirection is used. EURIS expressions are translated into discrete-time process algebra. In turn the standard interpretation of this specification language is based on process graphs.

4 Syntax and static semantics

We define the formal syntax of LARIS specifications. The syntax has two parts. The first part is a context-free grammar, given in the Syntax Definition Formalism (SDF) [14]. The SDF grammar provides us, amongst other things, with a syntactic category `Specification`. The grammar by itself accepts some texts which we in fact do not deem to be well-formed. For instance, the grammar has no means of detecting type clashes. The bare bones of the SDF grammar are therefore augmented with a so-called *static semantics*. This deals precisely with such matters as type clashes, making sure that when a variable is used, it has been declared, and the like.

Section 4.1 provides the SDF grammar. Section 4.2 introduces some notational conventions that allow us to shorten the presentation somewhat in the remainder. The syntax has some superfluencies in the sense that some constructions exist mainly for notational ease, while not adding to the expressivity of the language. Thus, Section 4.3 lists some syntactic constructions that are thereafter viewed as mere abbreviations. Sections 4.4-4.11 deal with the static semantics of LARIS.

4.1 Formal syntax in SDF

We present the context-free syntax of LARIS in SDF. This syntax is augmented later on with a static semantics. The end-result is a notion of 'syntactic' for LARIS constructions. The SDF grammar has a modular structure. Every module exports a list of new *sorts*. These are the syntactic categories that are defined in the grammar. By convention, these are written with a capital.

The first three modules, `Layout`, `Names`, and `Numbers` define some *lexical syntax*. The module `Layout` deals with the layout of the text: tabs (`\t`), spaces (' '), and newlines (`\n`) are not considered part of the LARIS specification, but merely exist for notational convenience. Similarly, anything after a `%` is considered as comment.

The module `Names` indicates which strings are allowed as identifiers. A `Name` is any string consisting of letters and numbers, interspersed with underscores. A `Name` must begin with a letter, and may not end with an underscore. By an SDF convention, keywords are excluded from `Names`. The module `Numbers` indicates which strings are allowed as natural numbers. All modules thereafter provide a context-free syntax to define new sorts. Most constructions in the context-free part are self-explanatory. The symbol `+` stands for one or more and `*` for zero or more occurrences. For instance `Clause+` denotes one or more `Clauses` and { `DataDecl ‘‘;"` }* denotes zero or more `DataDecls` separated by semicolons, but without a trailing semicolon.

{`right`} means that an operator is right-associative, (e.g., for concatenation of statements, `S; S'; S''` denotes `S; (S'; S'')`), while {`left`} means that an operator is left-associative (e.g., for addition of natural numbers, `n+n'+n''` denotes `(n+n')+n''`), and {`non-assoc`} denotes that an operator is not associative (e.g., subtraction on natural numbers is non-associative). The phrase {`bracket`} says that the defined construct is not an operator, but just a way to disambiguate the construction of a syntax tree. The `priorities` section resolves ambiguities in parsing; for example, subtraction '`-`' has highest priority while disjunction '`|`' has lowest priority, when parsing expressions with ambiguous bracketing.

```
module Layout
exports
```

```
  lexical syntax
    [\t \n]+         -> LAYOUT
    "%" ~[\n]* "\n" -> LAYOUT

module Names
exports
  sorts Name NonemptyNameSet NameSet
  lexical syntax
    [a-zA-Z][0-9a-zA-Z_]*[0-9a-zA-Z]* -> Name
  context-free syntax
    "{" {Name ","}+ "}"               -> NonemptyNameSet
    "{" {Name ","}* "}"               -> NameSet

module Numbers
exports
  sorts PosNumeral Numeral
  lexical syntax
    [1-9][0-9]* -> PosNumeral
    "0"         -> Numeral
    PosNumeral  -> Numeral

module Types
imports Layout Names Numbers
exports
  sorts BasicType Index ArrayType DataType ClockType Type
        TypeDef
  context-free syntax
    "Bool"                          -> BasicType
    "Int"                           -> BasicType
    "Component"                     -> BasicType
    "Port"                          -> BasicType
    Name                            -> BasicType
    BasicType                       -> Index
    PosNumeral                      -> Index
    BasicType "[" {Index "," }+ "]" -> ArrayType
    BasicType                       -> DataType
    ArrayType                       -> DataType
    "Timer"                         -> ClockType
    "Timeout"                       -> ClockType
    "Cycler"                        -> ClockType
    DataType                        -> Type
    ClockType                       -> Type
    Name "=" NonemptyNameSet        -> TypeDef

module Declarations
```

```
imports Layout Names Types
exports
  sorts DataDecl DataDecls Decl Decls ParDecls
  context-free syntax
     {Name ","}+ ":" DataType -> DataDecl
     {Name ","}+ ":" Type     -> Decl
     "vars" {DataDecl ";"}*   -> DataDecls
     "vars" {Decl ";"}*       -> Decls
     "(" {DataDecl ";"}* ")"  -> ParDecls

module Expressions
imports Layout Names Numbers Types
exports
  sorts Expr Datum Entry Array Portcons
  context-free syntax
    "log"                         -> Portcons
    "inf"                         -> Portcons
    "left"                        -> Portcons
    "right"                       -> Portcons

    Name                          -> Expr
    "true"                        -> Expr
    "false"                       -> Expr
    Numeral                       -> Expr
    "self"                        -> Expr
    "Log"                         -> Expr
    "Inf"                         -> Expr
    Portcons                      -> Expr
    Expr "[" {Expr ","}+ "]"      -> Expr
    Expr "+" Expr                 -> Expr {left}
    Expr "*" Expr                 -> Expr {left}
    Expr "-" Expr                 -> Expr {non-assoc}
    "-" Expr                      -> Expr
    Expr "mod" Expr               -> Expr {non-assoc}
    Expr "div" Expr               -> Expr {non-assoc}
    Expr "^" Expr                 -> Expr {left}
    Expr "|" Expr                 -> Expr {left}
    "~" Expr                      -> Expr
    Expr "==" Expr                -> Expr {non-assoc}
    Expr "=/=" Expr               -> Expr {non-assoc}
    Expr "<" Expr                 -> Expr {non-assoc}
    Expr ">" Expr                 -> Expr {non-assoc}
    Expr "<=" Expr                -> Expr {non-assoc}
    Expr ">=" Expr                -> Expr {non-assoc}
    "active" Name                 -> Expr
```

```
    "value" Name                    -> Expr
    Array ":" ArrayType             -> Expr
    "(" Expr ")"                    -> Expr {bracket}

    Expr                            -> Datum
    "*"                             -> Datum
    "(" {Datum ","}+ "," Expr ")" -> Entry
    "{" {Entry ","}* "}"            -> Array

  priorities
    Expr "*" Expr -> Expr >
    Expr "div" Expr -> Expr >
    Expr "+" Expr -> Expr >
    Expr "-" Expr -> Expr >
    Expr "mod" Expr -> Expr >
    Expr "==" Expr -> Expr >
    Expr "=/=" Expr -> Expr >
    Expr ">" Expr -> Expr >
    Expr "<" Expr -> Expr >
    Expr "<=" Expr -> Expr >
    Expr ">=" Expr -> Expr >
    "~" Expr -> Expr >
    Expr "^" Expr -> Expr >
    Expr "|" Expr -> Expr

module Statements
imports Layout Names Expressions
exports
  sorts Statement Args Telegram Telegrams Clause
  context-free syntax
    "(" {Expr ","}* ")"                   -> Args
    Name Args                             -> Telegram

    Name ":=" Expr                        -> Statement
    Name "[" {Datum ","}+ "]" ":=" Expr  -> Statement
    Expr "|>" Expr "!" Telegram           -> Statement
    "!" Telegram                          -> Statement
    ">>#" Name Expr "!" Telegram          -> Statement
    "@" Name Expr "!" Telegram            -> Statement
    "start" Name                          -> Statement
    "stop"  Name                          -> Statement
    "skip"                                -> Statement
    Name Args                             -> Statement
    "if" Expr "then" Statement            -> Statement
    "if" Expr "then" Statement
```

```
              "else" Statement           -> Statement
    "while" Expr "do" Statement          -> Statement
    "case" Name "in"
       "{" Clause+
           "otherwise" ":" Statement "}" -> Statement
    Statement ";" Statement              -> Statement {right}
    "{" Statement "}"                    -> Statement {bracket}

    Expr ":" Statement                   -> Clause

module Spec
imports Layout Names Types Declarations Expressions Statements
exports
  sorts Body Behaviour LSC Binding System Specification Port
  context-free syntax
    DataDecls Statement                       -> Body
    Statement                                 -> Body

    Portcons                                  -> Port
    Name                                      -> Port

    "proc" Name ParDecls "=" Body             -> Behaviour
    "mes" Port "?" Name ParDecls "=" Body     -> Behaviour
    "mes" "?" Name ParDecls "=" Body          -> Behaviour

    "LSC" Name ParDecls "="
          Decls
          "initial" Body
          Behaviour*
          "panic" Body                        -> LSC

    Name Name Args                            -> Binding

    "System" Name "="
          "External" "components" "=" NameSet
          "External" "ports" "=" NameSet
          Binding+                            -> System

    TypeDef* LSC+ System                      -> Specification
```

Hereafter the names of the syntactic categories defined above are used for the sets of all syntactic strings of that category. For instance, `Name` denotes the set of all strings of category `Name`. An assumption of SDF we have already alluded to is of relevance here: it is assumed that categories defined in the lexical syntax, such as `Name`, do not contain keywords explicitly used in the context-free syntax:

Definition 4.1 (Keyword) *The* keywords *of LARIS are:*

`Bool`, `Int`, `Component`, `Port`, `Timer`, `Timeout`, `Cycler`, `vars`, `true`, `false`, `self`, `Log`, `Inf`, `log`, `inf`, `left`, `right`, `mod`, `active`, `value`, `start`, `stop`, `skip`, `if`, `then`, `else`, `while`, `do`, `case`, `in`, `proc`, `mes`, `LSC`, `initial`, `panic`, `System`, `External`, `components`, `ports`

□

Similarly, `BasicType` is the union of `Name` and {`Bool`, `Int`, `Component`, `Port`}. Instead of saying '`Bool` ∈ `BasicType`' we may say '`Bool` is a `BasicType`', meaning the same.

4.2 Notational conventions

In the remainder of this paper some mathematical notational conventions are frequently used. These are listed below:

- $a_0, \ldots, a_n$ denotes a list of length $n+1$. If $n = 0$, $a_0, \ldots, a_n$ represents the singleton list a_0, if $n = 1$, it represents a_0, a_1, et cetera.

  If the first index in a list is 1, this indicates the possibility that the list is empty. Thus $a_1, \ldots, a_n$ denotes a list of length n. If $n = 0$, it represents the empty list, if $n = 1$, it is the singleton list a_1, et cetera.

- In the remainder, indices range over the natural numbers. Thus, if it is stated that something is true for every $i \leq k$ (with k some other natural number), this means it is true for all natural numbers from 0 to k.

- Vector notation is sometimes used for a sequence of items. Thus $\vec{d}$ stands for a sequence $d_1, \ldots, d_n$ of some length n.

4.3 Abbreviations

The syntax allows some constructions that in the remainder of this paper are viewed simply as abbreviations for other constructions. We list these abbreviations below:

1. We view a `case-in` statement

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{case}\; X \;\texttt{in}} \\
\{\; E_0 & : & \pi_0 \\
\quad\vdots & \vdots & \vdots \\
\quad E_n & : & \pi_n \\
\quad\texttt{otherwise} & : & \pi_{n+1} \;\}
\end{array}
$$

as an abbreviation for:

$$
\begin{array}{l}
\texttt{if} \quad X == E_0 \\
\quad \texttt{then} \;\; \pi_0 \\
\quad \texttt{else} \\
\qquad\qquad \ddots \\
\qquad\qquad\qquad \texttt{if} \quad X == E_n \\
\qquad\qquad\qquad\quad \texttt{then} \;\; \pi_n \\
\qquad\qquad\qquad\quad \texttt{else} \;\; \pi_{n+1}
\end{array}
$$

This means that while in LARIS we may use the `case-in` statement, in the static and operational semantics we make no further mention of it.

2. An `if-then` statement without an `else` clause is also considered an abbreviation: `if` E `then` π is short for `if` E `then` π `else skip`.

3. A `Decls` or `DataDecls` of the form

$$\texttt{vars}\ X_{0,0}, \ldots, X_{n_0,0} : T_0;\ \ldots\ ;\ X_{0,m}, \ldots, X_{n_m,m} : T_m$$

where for some $j \leq m$ we have $n_j > 0$, is an abbreviation for

$$\texttt{vars}\ X_{0,0} : T_0;\ \ldots\ ;\ X_{n_0,0} : T_0;\ \ldots\ ;\ X_{0,m} : T_m;\ \ldots\ ;\ X_{n_m,m} : T_m$$

4. Similarly, a `ParDecls` of the form

$$(X_{0,0}, \ldots, X_{n_0,0} : T_0;\ \ldots\ ;\ X_{0,m}, \ldots, X_{n_m,m} : T_m)$$

where for some $j \leq m$ we have $n_j > 0$, is an abbreviation for

$$(X_{0,0} : T_0;\ \ldots\ ;\ X_{n_0,0} : T_0;\ \ldots\ ;\ X_{0,m} : T_m;\ \ldots\ ;\ X_{n_m,m} : T_m)$$

5. In the syntax, a `Body` is an optional `DataDecls` followed by a `Statement`. We view `Body`s without a `DataDecls` as an abbreviation. Thus, a `Body` π, where π is a `Statement`, is short for `vars` π. This allows us to refer to any body as:

$$\texttt{vars}\ \mathtt{X}_1 : \mathtt{T}_1;\ \ldots\ ;\ \mathtt{X}_n : \mathtt{T}_n\ \pi$$

without having to consider the special case where there is no variable declaration.

4.4 Signatures

In this section we define so-called *signatures*. Signatures are merely representations of (parts of) LARIS specifications, using set-theoretical notation, with some of the information that is present in this specification already extracted, so that in the future we have easy access to this information. For example, the signature of an LSC has explicitly available, as a set, the names of internal telegrams for which it has a response.

In this section we give a general definition of the notion of a signature, and in the next section we assign specific signatures to LARIS constructs.

Definition 4.2 (Typed `Name`) *A* typed `Name` *is a pair* $\langle N{:}T\rangle$*, where* N *is a* `Name` *and* T *is a* `Type`*.* □

Definition 4.3 (Argument typed `Name`) *An* argument typed `Name` *is a pair* $\langle N{:}S\rangle$*, where* N *is a* `Name` *and* S *is a list of* `DataType`*s.* □

Definition 4.4 (Body signature) *A* body signature *is a pair* (V, π) *where* V *is a set of typed* `Name`*s and* π *is a* `Statement`*. If* B *is such a body-signature, then:*

$$\begin{array}{lcl} B.\mathrm{Vars} & = & V \\ B.\mathrm{Statement} & = & \pi \end{array}$$

□

Definition 4.5 (External telegram signature) *An* external telegram signature *is a triple* (R, P, B) *where:*

- R *is a pair of* `Name`*s;*
- P *is a set of typed* `Name`*s;*
- B *is a body signature.*

If M *is such a signature, then:*

$$\begin{array}{lcl} M.\mathrm{Reaction} & = & R \\ M.\mathrm{Pars} & = & P \\ M.\mathrm{Body} & = & B \end{array}$$

□

Definition 4.6 (Internal signature) *An* internal signature *is a quadruple* (K, N, P, B) *where:*

- $K \in \{\mathtt{mes}, \mathtt{proc}\}$*;*
- N *is a* `Name`*;*
- P *is a list of typed* `Name`*s;*
- B *is a body signature.*

If I *is such a signature, then:*

$$\begin{array}{lcl} I.\mathrm{Kind} & = & K \\ I.\mathrm{Name} & = & N \\ I.\mathrm{Pars} & = & P \\ I.\mathrm{Body} & = & B \end{array}$$

An internal telegram signature *is an internal signature* I *with* $I.\mathrm{Kind} = \mathtt{mes}$. *A* procedure signature *is an internal signature* I *with* $I.Kind = \mathtt{proc}$*.* □

Definition 4.7 (Behaviour signature) *A* behaviour signature *is a* 9*-tuple of the form* $(P, Et, It, R, I, Pt, Me, Mi, Pr)$*, where:*

- P *and* I *are sets of* `Names`*;*
- Et*,* It*, and* Pt *are sets of argument typed* `Names`*;*
- R *is a set of pairs of* `Names`*;*
- Me *is a set of external telegram signatures;*
- Mi *is a set of internal telegram signatures;*
- Pr *is a set of procedure signatures.*

If B *is such a signature, then:*

$$\begin{array}{lcl} B.\text{Ports} & = & P \\ B.\text{ExtTypes} & = & Et \\ B.\text{IntTypes} & = & It \\ B.\text{Reactions} & = & R \\ B.\text{Internal} & = & I \\ B.\text{ProcTypes} & = & Pt \\ B.\text{ExtTelegrams} & = & Me \\ B.\text{IntTelegrams} & = & Mi \\ B.\text{Procedures} & = & Pr \end{array}$$

□

Definition 4.8 (LSC signature) *An* LSC signature *is a* 6*-tuple:*

$$(N, P, V, B, I, Pa)$$

where:

- N *is a* `Name`*;*
- P *is a list of typed* `Names`*;*
- V *is a set of typed* `Names`*;*
- B *is a behaviour signature;*
- I *and* Pa *are body signatures.*

If Λ *is such a signature, then:*

$$\begin{array}{lcl} \Lambda.\text{Name} & = & N \\ \Lambda.\text{Pars} & = & P \\ \Lambda.\text{Vars} & = & V \\ \Lambda.\text{Behaviour} & = & B \\ \Lambda.\text{Initial} & = & I \\ \Lambda.\text{Panic} & = & Pa \end{array}$$

□

Definition 4.9 (Type signature) *A* type signature *is a finite set of triples of the form* (T, S, D) *where* T *and* D *are* `Names`*, and* S *is a finite set of* `Names` *such that* $D \in S$.

If $X = (T, S, D)$ *is an element of a type definition signature then:*

$$\begin{array}{lcl} X.\text{Type} & = & T \\ X.\text{Set} & = & S \\ X.\text{Default} & = & D \end{array}$$

□

Definition 4.10 (System signature) *A* system signature *is a quadruple* (E, Bo, P, Bi)*, where:*

- E, Bo *and* P *are sets of* `Names`*;*
- Bi *is a set of* `Bindings`*.*

If S *is such a signature, then:*

$$\begin{array}{lcl} S.\text{External} & = & E \\ S.\text{Bound} & = & Bo \\ S.\text{Ports} & = & P \\ S.\text{Bindings} & = & Bi \end{array}$$

□

Definition 4.11 (Specification signature) *A triple* (T, L, S) *is a* specification signature *if:*

- T *is a set of type signatures;*
- L *is a set of LSC signatures;*
- S *is a system signature.*

If Σ *is such a signature, then:*

$$\begin{array}{lcl} \Sigma.\text{Types} & = & T \\ \Sigma.\text{LSC} & = & L \\ \Sigma.\text{System} & = & S \end{array}$$

□

4.5 Assigning signatures

Now that we have specified the various types of signatures and how to access their information, we can assign specific signatures to various constructs in LARIS. This is done via the function Sig.

Definition 4.12 (Signature of a `Body`) *Let β be a* `Body` *of the form*

$$\texttt{vars } \texttt{X}_1 : \texttt{T}_1; \ldots ;\ \texttt{X}_n : \texttt{T}_n\ \pi$$

where $\texttt{X}_1, \ldots, \texttt{X}_n$ *are* `Names`*,* $\texttt{T}_1, \ldots, \texttt{T}_n$ *are* `DataTypes`*, and π is a* `Statement`*. Then $Sig(\beta)$ is the body signature B with:*

$$\begin{array}{lcl} B.\text{Vars} & = & \{\langle \texttt{X}_1{:}\texttt{T}_1\rangle, \ldots, \langle \texttt{X}_n{:}\texttt{T}_n\rangle\} \\ B.\text{Statement} & = & \pi \end{array}$$

□

Definition 4.13 (Signature of an external flow) *Let B be a* `Behaviour` *of the form*

$$\texttt{mes}\ p\ ?\ N(X_1 : T_1; \ldots ;\ X_n : T_n)\ = \beta$$

where $X_1, \ldots, X_n$, p and N are `Names`*, $T_1, \ldots, T_n$ are* `DataTypes` *and β is a* `Body`*. Then $Sig(B)$ is the behaviour signature S with:*

$$\begin{array}{lcl} S.\text{Ports} & = & \{p\} \\ S.\text{ExtTypes} & = & \{\langle N{:}(T_1, \ldots, T_n)\rangle\} \\ S.\text{ExtTypes} & = & \emptyset \\ S.\text{Reactions} & = & \{\langle p, N\rangle\} \\ S.\text{Internal} & = & \emptyset \\ S.\text{ProcTypes} & = & \emptyset \\ S.\text{ExtTelegrams} & = & \{(\langle p, N\rangle, (\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_n\rangle), Sig(\beta))\} \\ S.\text{IntTelegrams} & = & \emptyset \\ S.\text{Procedures} & = & \emptyset \end{array}$$

□

Definition 4.14 (Signatures of an internal flow) *Let B be a* `Behaviour` *of the form*

$$\texttt{mes}\ ?\ N(X_1 : T_1; \ldots ;\ X_n : T_n)\ = \beta$$

where $X_1, \ldots, X_n$, p and N are `Names`*, $T_1, \ldots, T_n$ are* `DataTypes` *and β is a* `Body`*, then $Sig(B)$ is the behaviour signature S with:*

$$\begin{array}{lcl} S.\text{Ports} & = & \emptyset \\ S.\text{ExtTypes} & = & \emptyset \\ S.\text{IntTypes} & = & \{\langle N{:}(T_1, \ldots, T_n)\rangle\} \\ S.\text{Reactions} & = & \emptyset \\ S.\text{Internal} & = & \{N\} \\ S.\text{ProcTypes} & = & \emptyset \\ S.\text{ExtTelegrams} & = & \emptyset \\ S.\text{IntTelegrams} & = & \{(\texttt{mes}, N, (\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_n\rangle), Sig(\beta))\} \\ S.\text{Procedures} & = & \emptyset \end{array}$$

□

Definition 4.15 (Signature of a procedure definition) *Let B represent a* `Behaviour` *of the form*

$$\texttt{proc}\ N(X_1 : T_1; \ldots\ ;\ X_n : T_n) = \beta$$

where $X_1, \ldots, X_n$ and N are `Name`*s, $T_1, \ldots, T_n$ are* `DataType`*s and β is a* `Body`*. Then $Sig(B)$ is the behaviour signature S with:*

$$\begin{array}{lcl} S.\mathrm{Ports} & = & \emptyset \\ S.\mathrm{ExtTypes} & = & \emptyset \\ S.\mathrm{IntTypes} & = & \emptyset \\ S.\mathrm{Reactions} & = & \emptyset \\ S.\mathrm{Internal} & = & \emptyset \\ S.\mathrm{ProcTypes} & = & \{\langle N{:}(T_1, \ldots, T_n)\rangle\} \\ S.\mathrm{ExtTelegrams} & = & \emptyset \\ S.\mathrm{IntTelegrams} & = & \emptyset \\ S.\mathrm{Procedures} & = & \{(\texttt{proc}, N, (\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_n\rangle), Sig(\beta)\} \end{array}$$

□

Definition 4.16 (Signature of a sequence of `Behaviours`) *If B is a possibly empty sequence $B_1 \ldots B_k$ of* `Behaviour`*s such that*

$$Sig(B_i) = (P_i, Et_i, It_i, R_i, I_i, Pt_i, Me_i, Mi_i, Pr_i)$$

for every $1 \le i \le k$, then:

$$\begin{array}{rl} Sig(B) = & (\bigcup_{1\le i\le k} P_i, \bigcup_{1\le i\le k} Et_i, \bigcup_{1\le i\le k} It_i, \bigcup_{1\le i\le k} R_i, \bigcup_{1\le i\le k} I_i, \\ & \bigcup_{1\le i\le k} Pt_i, \bigcup_{1\le i\le k} Me_i, \bigcup_{1\le i\le k} Mi_i, \bigcup_{1\le i\le k} Pr_i) \end{array}$$

□

Definition 4.17 (Signature of an LSC) *Let L be an LSC of the form*

$$\begin{array}{ll} \texttt{LSC} & N(X_1 : T_1;\ \ldots\ ;\ X_n : T_n) = \\ & \texttt{vars}\ Y_1 : U_1; \ldots\ ;\ Y_m : U_m V \\ & \texttt{initial}\ \alpha \\ & B \\ & \texttt{panic}\ \beta \end{array}$$

where $X_1, \ldots, X_n, Y_1, \ldots, Y_m$ and N are `Name`*s, $T_1, \ldots, T_n$ are* `DataType`*s, $U_1, \ldots, U_m$ are* `Type`*s, α and β are* `Body`*s and B is a (possibly empty) sequence of* `Behaviour`*s. Then $Sig(L)$ is the LSC signature Λ with:*

$$\begin{array}{lcl} \Lambda.\mathit{Name} & = & N \\ \Lambda.\mathit{Pars} & = & (\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_1\rangle) \\ \Lambda.\mathit{Vars} & = & \{\langle Y_1{:}U_1\rangle, \ldots, \langle Y_n{:}U_1\rangle\} \\ \Lambda.\mathit{Behaviour} & = & Sig(B) \\ \Lambda.\mathit{Initial} & = & Sig(\alpha) \\ \Lambda.\mathit{Panic} & = & Sig(\beta) \end{array}$$

□

Definition 4.18 (Signature of a `TypeDef`) *Let T be a* `TypeDef` *of the form*

$$N = \{N_0, \ldots, N_n\}$$

where $N_1, \ldots, N_n$ and N are `Name`*s. Then $Sig(T)$ is the type signature S with:*

$$\begin{array}{lcl} S.\mathrm{Type} & = & N \\ S.\mathrm{Set} & = & \{N_0, \ldots, N_n\} \\ S.\mathrm{Default} & = & N_0 \end{array}$$

□

Definition 4.19 (Signature of a `System`) *Let $\mathcal{S}$ be a* `System` *of the form*

$$\begin{array}{ll} \texttt{System} & N \; = \\ & \texttt{External components} \; = \{E_1, \ldots, E_n\} \\ & \texttt{External ports} \; = \{P_1, \ldots, P_m\} \\ & C_0 \;\; N_0 \; A_0 \\ & \vdots \\ & C_k \;\; N_k \; A_k \end{array}$$

where $E_1, \ldots, E_n$, $P_1, \ldots, P_m$, $C_0, \ldots, C_k$, $N_0, \ldots, N_k$, and N are `Name`*s and $A_0, \ldots, A_k \in$* `Args`*. Then $Sig(\mathcal{S})$ is the system signature S with:*

$$\begin{array}{lcl} S.\mathrm{External} & = & \{E_1, \ldots, E_n\} \\ S.\mathrm{Bound} & = & \{C_0, \ldots, C_k\} \\ S.\mathrm{Ports} & = & \{P_1, \ldots, P_m\} \\ S.\mathrm{Bindings} & = & \{C_i \;\; N_i \; A_i \mid i \leq k\} \end{array}$$

□

Definition 4.20 (Signature of a `Specification`) *If $\mathcal{S}$ is a* `Specification` *of the form $T_1 \ldots T_n \; L_0 \ldots L_m \; S$, where $T_1, \ldots, T_m$ are* `TypeDef`*s, $L_0, \ldots, L_m$ are LSCs, and S is a* `System`*, then $Sig(\mathcal{S})$ is a specification signature Σ with:*

$$\begin{array}{lcl} \Sigma.\mathrm{Types} & = & \{Sig(T_1), \ldots, Sig(T_n)\} \\ \Sigma.\mathrm{LSC} & = & \{Sig(L_0), \ldots, Sig(L_m)\} \\ \Sigma.\mathrm{System} & = & Sig(S) \end{array}$$

□

4.6 Types

Fix some specification signature Σ. Σ contains some definitions of enumerated types, which we may find in Σ.Types. Once the names of these additional basic types are known, we can define the set of Σ-types.

Definition 4.21 (Σ-type)

- *The set of* Σ-basic types *is:*

  $$\mathsf{BasicType}_\Sigma = \{\mathtt{Bool}, \mathtt{Int}, \mathtt{Component}, \mathtt{Port}\} \cup \{T.\mathrm{Type} \mid T \in \Sigma.\mathrm{Types}\}.$$

  Note that $\mathsf{BasicType}_\Sigma$ *is a subset of* `BasicType`. *The latter contains* `Names` *which do not occur as type names in the list of type definitions of* Σ.

- *The set of* Σ-indices *is:*

  $$\mathsf{Index}_\Sigma = \mathsf{BasicType}_\Sigma \cup \mathtt{PosNumeral}.$$

- *The set of* Σ-array types *is:*

  $$\mathsf{ArrayType}_\Sigma = \{B[I_0, \ldots, I_n] \mid \ B \in \mathsf{BasicType}_\Sigma \ \textit{and} \ I_0, \ldots, I_n \in \mathsf{Index}_\Sigma\}.$$

- *The set of* Σ-data types *is:*

  $$\mathsf{DataType}_\Sigma = \mathsf{BasicType}_\Sigma \cup \mathsf{ArrayType}_\Sigma.$$

- *Finally, the set of* Σ-types *is:*

  $$\mathsf{Type}_\Sigma = \mathsf{DataType}_\Sigma \cup \mathtt{ClockType}.$$

□

4.7 Type- and declaration-correctness

This section contains the first part of the static semantics, concerning type- and declaration-correctness.

Type-correctness ensures that every constant has a unique type, i.e., overloading is not allowed. Thus the following sequence of type definitions cannot be part of a type-correct specification:

```
Type1 = { a, b }
Type2 = { b, c }
```

as this leaves ambiguous the type of the constant `b`: it could be either of type `Type1` or of type `Type2`.

As `Component` and `Port` are basic types, we need to know which components and ports are presupposed by a specification:

Definition 4.22 (Component and port) *Let* Σ *be a specification signature. The set* $\mathsf{Component}_\Sigma$ *of components of* Σ *is*

$$\Sigma.\mathrm{System.External} \cup \Sigma.\mathrm{System.Bound} \cup \{\mathtt{Log}, \mathtt{Inf}\}$$

and the set Port_Σ *of ports of* Σ *is*

$$\bigcup_{\Lambda \in \Sigma.\mathrm{LSC}} \Lambda.\mathrm{Behaviour.Ports} \cup \Sigma.\mathrm{System.Ports} \cup \{\mathtt{log}, \mathtt{inf}, \mathtt{left}, \mathtt{right}\}.$$

□

Definition 4.23 (Type-correctness) *A specification-signature Σ is said to be* type-correct *(or* t-correct*) if:*

1. $\mathsf{Component}_\Sigma$ *and* Port_Σ *are disjoint:* $\mathsf{Component}_\Sigma \cap \mathsf{Port}_\Sigma = \emptyset$*;*
2. *If* $T \in \Sigma.\mathrm{Types}$ *then*
   - $T.\mathrm{Set}$ *and* $\mathsf{Component}_\Sigma$ *are disjoint;*
   - $T.\mathrm{Set}$ *and* Port_Σ *are disjoint;*
   - *If* $T' \in \Sigma.\mathrm{Types}\backslash\{T\}$ *then* $T.\mathrm{Name} \neq T'.\mathrm{Name}$ *and* $T.\mathrm{Set}$ *and* $T'.\mathrm{Set}$ *are disjoint.* □

The second main concept of correctness of this section is *declaration-correctness*. This concept deals with two matters. First, declaration-correctness ensures that within a declaration a variable may not be assigned different types. Thus, `X:Bool; X:Int` is not part of any declaration-correct specification. In mathematical terms, we wish declaration to represent functions.

The second matter declaration-correctness deals with is the avoidance of scoping problems. These are avoided by disallowing variables being declared in a procedure, a message handling, or the initial and panic clauses to overlap with the global variables of the LSC.

First we define which `Names` can be variables.

Definition 4.24 (Reserved `Name`) *A* `Name` *is* Σ-reserved *if it is in*

$$\mathsf{Component}_\Sigma \cup \mathsf{Port}_\Sigma \cup \bigcup_{T \in \Sigma.\mathrm{Types}} T.\mathrm{Set}$$

□

Recall that by an SDF convention, keywords (see Definition 4.1) were already excluded from the set of `Names`.

Definition 4.25 (Available `Name`) *An* Σ-available `Name` *is any* `Name` *that is not* Σ*-reserved.* □

Note that there are only finitely many reserved names and infinitely many available ones.

Definition 4.26 (Σ-typed `Name`) *A* Σ-typed `Name` *is a typed* `Name` $\langle X{:}T\rangle$ *such that* X *is a* Σ*-available* `Name` *and* T *is a* Σ*-type.* □

Definition 4.27 (Type function) *A* Σ-type function $\mathcal{T}$ *is a finite set of* Σ*-typed* `Names` *that is* functional, *i.e., if* $\langle X{:}T_1\rangle, \langle X{:}T_2\rangle \in \mathcal{T}$ *then* $T_1 = T_2$*.* □

Definition 4.28 (Type-list) *A* Σ-type list *is a list* $(\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_n\rangle)$ *of* Σ*-typed* `Names` *such that* $X_1, \ldots, X_n$ *are pairwise distinct.* □

Definition 4.29 (Domain function) *The* domain *of a set of typed* `Names` S *is:*

$$\mathsf{dom}(S) = \{X \mid \langle X{:}T\rangle \in S \textit{ for some } T\}.$$

□

Definition 4.30 (From type lists to functions) *If* $(\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_n\rangle)$ *is a list of typed* `Names` *then*

$$\mathsf{Set}(\ (\langle X_1{:}T_1\rangle, \ldots, \langle X_n : T_n\rangle)\ ) \quad = \quad \{\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_n\rangle\}$$

□

If L is a Σ-type list, then obviously $\mathsf{Set}(L)$ is a Σ-type function. Σ-type lists represent two matters: one, the order in which the variables occur; two, a type-function, associating a variable with a type. Order is important when instantiating for instance an LSC, a procedure, or a telegram. The type-function is important for evaluating the type of expressions within the scope of the declaration. Set extracts this type-function from a type-list, abstracting away from the order.

Definition 4.31 (Declaration-correctness of a behaviour) *Suppose* $\Lambda \in \Sigma.\mathrm{LSC}$*, and let*

$$S \in \begin{array}{l}\Lambda.\mathrm{Behaviour.ExtTelegrams} \cup \\ \Lambda.\mathrm{Behaviour.IntTelegrams} \cup \\ \Lambda.\mathrm{Behaviour.Procedures}\end{array}$$

S *is* declaration-correct *(or* d-correct*) with respect to* Σ *and* Λ *if:*

- S.Pars *is a* Σ*-type list;*
- S.Body.Vars *is a* Σ*-type function;*
- *The following sets are pairwise disjoint:*

$$\begin{array}{ll}\mathsf{dom}(\mathsf{Set}(\Lambda.\mathrm{Pars})) & \mathsf{dom}(\Lambda.\mathrm{Vars}) \\ \mathsf{dom}(\mathsf{Set}(S.\mathrm{Pars})) & \mathsf{dom}(S.\mathrm{Body.Vars})\end{array}$$

□

Definition 4.32 (Declaration-correctness of an LSC) *Let* $\Lambda \in \Sigma.\mathrm{LSC}$. Λ *is* declaration-correct *(or* d-correct*) with respect to* Σ *if:*

- Λ.Pars *is a* Σ*-type list;*
- Λ.Vars, Λ.Initial.Vars *and* Λ.Panic.Vars *are* Σ*-type functions;*
- $\mathsf{dom}(\mathsf{Set}(\Lambda.\mathrm{Pars}))$ *and* $\mathsf{dom}(\Lambda.\mathrm{Vars})$ *are disjoint;*
- $\mathsf{dom}(\Lambda.\mathrm{Initial.Vars})$ *and* $\mathsf{dom}(\mathsf{Set}(\Lambda.\mathrm{Pars})) \cup \mathsf{dom}(\Lambda.\mathrm{Vars})$ *are disjoint;*

- $\mathsf{dom}(\Lambda.\text{Panic.Vars})$ *and* $\mathsf{dom}(\mathsf{Set}(\Lambda.\text{Pars})) \cup \mathsf{dom}(\Lambda.\text{Vars})$ *are disjoint;*
- *If*

$$S \in \begin{array}{l} \Lambda.\text{Behaviour.ExtTelegrams} \cup \\ \Lambda.\text{Behaviour.IntTelegrams} \cup \\ \Lambda.\text{Behaviour.Procedures} \end{array}$$

then S *is d-correct with respect to* Σ *and* Λ. □

Definition 4.33 (Declaration-correctness of a specification signature) *A specification signature* Σ *is* declaration-correct *(or* d-correct*) if every* $\Lambda \in \Sigma.\text{LSC}$ *is d-correct with respect to* Σ. □

4.8 Expressions

Fix a t-correct LARIS specification Σ. Given a Σ-type function $\mathcal{T}$, we can construct the set of $(\Sigma, \mathcal{T})$-*expressions*, that is, the set of expressions that may be built using as parameters and variables the names in $\mathcal{T}$. Such expressions can be assigned a definite type (given $\mathcal{T}$).

We define simultaneously by induction:

1. the set $\mathtt{Expr}(\Sigma, \mathcal{T})$ of $(\Sigma, \mathcal{T})$-expressions;
2. a function $\mathsf{type}_{\Sigma,\mathcal{T}} : \mathtt{Expr}(\Sigma, \mathcal{T}) \to \mathsf{Type}_{\Sigma}$.

If Σ is clear from the context, we simply refer to a '$\mathcal{T}$-expression', $\mathtt{Expr}(\mathcal{T})$, and $\mathsf{type}_{\mathcal{T}}$. Also, we circumscribe $\mathsf{type}_{\Sigma,\mathcal{T}}(E) = T$ by 'E has type T', or 'E is of type T', when Σ and $\mathcal{T}$ are clear from the context.

Variables/parameters: If $\langle X{:}T \rangle \in \mathcal{T}$ and $T \in \mathsf{Type}_{\Sigma}$, then X is a $\mathcal{T}$-expression of type T.

Inhabitants of basic types:

- `true` and `false` are $\mathcal{T}$-expressions of type `Bool`.
- If N is a `Numeral`, then N is a $\mathcal{T}$-expression of type `Int`.
- If $E \in \mathsf{Component}_{\Sigma}$, then E is a $\mathcal{T}$-expression of type `Component`.
- If $P \in \mathsf{Port}_{\Sigma}$, then P is a $\mathcal{T}$-expression of type `Port`.
- If $T \in \Sigma.\text{Types}$ and $N \in T.\text{Set}$, then N is a $\mathcal{T}$-expression of type $T.\text{Type}$.

Self: `self` is a $\mathcal{T}$-expression of type `Component`.

Array-positions: Suppose E be a $\mathcal{T}$-expression of type $T[I_0, \ldots, I_n]$, and let $E_0, \ldots, E_n$ be $\mathcal{T}$-expressions such that, for each $i \leq n$:

$$\mathsf{type}_{\mathcal{T}}(E_i) = \begin{cases} I_i & \text{if } I_i \in \mathsf{BasicType}_{\Sigma}; \\ \mathtt{Int} & \text{if } I_i \text{ is a } \mathtt{PosNumeral}. \end{cases}$$

Then $E[E_0, \ldots, E_n]$ is a $\mathcal{T}$-expression of type T.

Arithmetical operations: If E and F are $\mathcal{T}$-expressions of type `Int`, then $E + F$, $E - F$, $E * F$, E `mod` F, and E `div` F are $\mathcal{T}$-expressions of type `Int`.

Boolean operations: If E and F are $\mathcal{T}$-expressions of type `Bool`, then $E \wedge F$, and $\sim E$ are $\mathcal{T}$-expressions of type `Bool`.

Equality: If E and F are $\mathcal{T}$-expressions of the same type $T \in \mathsf{BasicType}_\Sigma$, then $E == F$ is a $\mathcal{T}$-expression of type `Bool`.

Less-than: If E and F are $\mathcal{T}$-expressions of type `Int`, then $E < F$ is a $\mathcal{T}$-expression of type `Bool`.

Active and value: If $\langle X{:}T \rangle \in \mathcal{T}$ with $T \in$ `ClockType`, then:

- `active` X is a $\mathcal{T}$-expression of type `Bool`;
- `value` X is a $\mathcal{T}$-expression of type `Int`.

Arrays: Let T be a Σ-type $B[I_0, \ldots, I_n]$. Let $m \geq 0$, and suppose for each $i \leq n$ and $1 \leq j \leq m$ one of the following holds:

- either $E_{i,j} = *$;
- I_i is a `PosNumeral` and $E_{i,j}$ is a $\mathcal{T}$-expression of type `Int`; or
- $I_i \in \mathsf{BasicType}_\Sigma$ and $E_{i,j}$ is a $\mathcal{T}$-expression of type I_i.

Furthermore, for each $1 \leq j \leq m$, let $E_{n+1,j}$ be a $\mathcal{T}$-expression of type B. Then

$$\{(E_{0,1}, \ldots, E_{n+1,1}), \ldots, (E_{0,m}, \ldots, E_{n+1,m})\} : T$$

is a $\mathcal{T}$-expression of type T.

4.9 Argument-correctness

Not only expressions but also telegrams have specific types in LARIS. The type of a telegram is a sequence of data types. In LARIS there are two types of telegrams. First there are the external telegrams, the ones that are sent between components. In a well-formed specification all external telegrams with the same name have the same type. Thus this a global requirement on the whole specification.

The type-assumptions on internal telegrams are local to the LSC. Within a single LSC, all internal telegrams with the same name have the same type. Different LSCs may use the same telegram names for their internal telegrams, even with differing types. Moreover, it is possible that an external telegram and an internal one share the same name, without sharing their types.

In this section we deal with the types of telegrams, how to extract these, and when a specification has no type clashes in this respect. The corresponding notion of correctness is *argument-correctness*. This notion also deals with the types of defined procedures. The main restriction on these is that procedures with the same name must have the same type. Furthermore, these types must be Σ-types.

Definition 4.34 (Σ-argument typed `Name`) *Let Σ be a specification signature. A* Σ-argument typed `Name` *is an argument typed* `Name` $\langle N{:}(T_1, \ldots, T_n)\rangle$ *where each T_i is a Σ-data type.* □

Definition 4.35 (Σ-argument type function) *Let Σ be a specification signature. A* Σ-argument type function *is a set $\mathfrak{T}$ of Σ-argument typed* `Name`*s such that whenever $\langle N{:}(T_1, \ldots, T_n)\rangle, \langle N{:}(U_1, \ldots, U_m)\rangle \in \mathfrak{T}$ then $(T_1, \ldots, T_n) = (U_1, \ldots, U_m)$.* □

In the following definition we use the term 'complex `Statement`' to refer to any `Statement` of one of the forms:

$$\texttt{if}\ E\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2$$
$$\texttt{while}\ E\ \texttt{do}\ \pi$$
$$\pi_1\, ;\, \pi_2$$

A 'basic `Statement`' is then any `Statement` that is not complex.

Definition 4.36 ($\mathsf{ExtSend}$) *Let Σ be a t-correct specification signature and let $\mathcal{T}$ be a Σ-type function.* $\mathsf{ExtSend}_{\Sigma,\mathcal{T}}$ *is a function defined on all* `Statement`*s as follows.*

If π is a basic `Statement` *then*

- $\mathsf{ExtSend}_{\Sigma,\mathcal{T}}(\pi) = \{\langle N{:}(T_1, \ldots, T_n)\rangle\}$ *if π is a* `Statement` *of the form*

  $$E \mid> p\ !\ N(E_1, \ldots, E_n)$$

  and each E_i is a $\mathcal{T}$-expression of type T_i.

- $\mathsf{ExtSend}_{\Sigma,\mathcal{T}}(\pi) = \emptyset$*, otherwise.*

If π is a complex `Statement` *then*

- $\mathsf{ExtSend}_{\Sigma,\mathcal{T}}(\pi) = \mathsf{ExtSend}_{\Sigma,\mathcal{T}}(\pi')$ *if π is of the form* `while` E `do` π'.

- $\mathsf{ExtSend}_{\Sigma,\mathcal{T}}(\pi) = \mathsf{ExtSend}_{\Sigma,\mathcal{T}}(\pi_1) \cup \mathsf{ExtSend}_{\Sigma,\mathcal{T}}(\pi_2)$ *if π is either of the form*
  `if` E `then` π_1 `else` π_2 *or* $\pi_1 ; \pi_2$. □

Note that $\mathsf{ExtSend}$ solely extracts the type of external telegrams which the statement could possibly send. We needed t-correctness in the definition because otherwise the notion of $(\Sigma, \mathcal{T})$-expression would not be defined.

Definition 4.37 ($\mathsf{ExtType}$) *Let Σ be a t- and d-correct specification signature. Then* $\mathsf{ExtType}_\Sigma$ *is the union of the following sets:*

- Λ.Behaviour.TelTypes, *where* $\Lambda \in \Sigma$.LSC*;*
- $\mathsf{ExtSend}_{\Sigma,\mathcal{T}}(S.\mathrm{Body.Statement})$, *for some* $\Lambda \in \Sigma$.LSC, *where*

$$S \in \begin{array}{l} \Lambda.\mathrm{Behaviour.ExtTelegrams} \cup \\ \Lambda.\mathrm{Behaviour.IntTelegrams} \cup \\ \Lambda.\mathrm{Behaviour.Procedures} \end{array}$$

  and $\mathcal{T} = \mathsf{Set}(\Lambda.\mathrm{Pars}) \cup \Lambda.\mathrm{Vars} \cup \mathsf{Set}(S.\mathrm{Pars}) \cup S.\mathrm{Body.Vars}$*;*
- $\mathsf{ExtSend}_{\Sigma,\mathcal{T}}(S.\mathrm{Statement})$, *where* $S \in \{\Lambda.\mathrm{Initial}, \Lambda.\mathrm{Panic}\}$ *and* $\mathcal{T} = \mathsf{Set}(\Lambda.\mathrm{Pars}) \cup \Lambda.\mathrm{Vars} \cup S.\mathrm{Vars}$. □

By d-correctness $\mathcal{T}$ is a type-function in each case above.

Definition 4.38 (Argument-correctness) *A t- and d-correct specification signature* Σ *is* argument-correct *(or* a-correct*) if:*

- $\mathsf{ExtType}_\Sigma$ *is a* Σ*-argument type function;*
- *For each* $\Lambda \in \Sigma$.LSC, Λ.Behaviour.IntTypes *is a* Σ*-argument type function.*
- *For each* $\Lambda \in \Sigma$.LSC, Λ.Behaviour.ProcTypes *is a* Σ*-argument type function.* □

The requirement that $\mathsf{ExtType}_\Sigma$ is a Σ-argument type function ensures that for each occurrence of an external telegram with name `N`, the cardinality, order, and typing of its fields is exactly the same. The second requirement in the definition contains a similar statement for internal telegrams, this time local to an LSC. The final requirement says that types of procedures are unique to a name, and that only Σ-types are allowed as arguments.

4.10 Statements

Let Σ be a t-, d-, and a-correct specification signature and let $\Lambda \in \Sigma$.LSC. Given two Σ-type functions $\mathcal{T}$ and $\mathcal{V}$ with $\mathcal{V} \subseteq \mathcal{T}$, we define the set of $(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$-statements: $\mathtt{Statement}(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$. $\mathcal{V}$ encompasses the variables, i.e., those `Names` that can occur at left-hand sides of assignments.

Whenever some (but not all) of the parameters in the list $(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$ are clear from the context, we may drop these from the list. Thus, if Σ and Λ are clear from the context, then we may refer to $\mathtt{Statement}(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$ as $\mathtt{Statement}(\mathcal{T}, \mathcal{V})$, and to its inhabitants as '$(\mathcal{T}, \mathcal{V})$-statements'. In the following definition of $\mathtt{Statement}(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$ this convention is used.

Assignment: If $\langle X{:}T\rangle \in \mathcal{V}$, $T \in \mathsf{DataType}_\Sigma$, and E is a $\mathcal{T}$-expression of type T, then $X{:=}\ E$ is a $(\mathcal{T}, \mathcal{V})$-statement.

Entry: If $\langle X{:}T[I_0, \ldots, I_n]\rangle \in \mathcal{V}$, E is a $\mathcal{T}$-expression of type T and for each $i \leq n$ one of the following holds:

- $E_i = *$;
- $I_i \in \mathsf{BasicType}_\Sigma$ and E_i is a $\mathcal{T}$-expression of type I_i;
- I_i is a `PosNumeral` and E_i is a $\mathcal{T}$-expression of type `Int`;

then $X[E_0, \ldots, E_n] := E$ is a $(\mathcal{T}, \mathcal{V})$-statement.

Sending an external telegram: If E is a $\mathcal{T}$-expression of type `Component`, P is a $\mathcal{T}$-expression of type `Port`, $\langle N{:}(T_1, \ldots, T_n)\rangle \in \mathsf{ExtType}_\Sigma$, and for each $1 \leq i \leq n$ we have that E_i is a $\mathcal{T}$-expression of type T_i, then $E \;|{>}\;\; P \;!\; N(E_1, \ldots, E_n)$ is a $(\mathcal{T}, \mathcal{V})$-statement.

Internal telegrams: Let $\langle N{:}(T_1, \ldots, T_n)\rangle \in \Lambda$.Behaviour.IntTypes, and for each $1 \leq i \leq n$, let E_i be a $\mathcal{T}$-expression of type T_i. Furthermore, let E be a $\mathcal{T}$-expression of type `Int`. Then

- $!\; N(E_1, \ldots, E_n)$ is a $(\mathcal{T}, \mathcal{V})$-statement;
- if $\langle X{:}\texttt{Timeout}\rangle \in \mathcal{V}$, then $\gg\# \;\; X \;\; E \;\; ! \;\; N(E_1, \ldots, E_n)$ is a $(\mathcal{T}, \mathcal{V})$-statement;
- if $\langle X{:}\texttt{Cycler}\rangle \in \mathcal{V}$, then $@X\,E!N(E_1, \ldots, E_n)$ is a $(\mathcal{T}, \mathcal{V})$-statement.

Starting a timer: If $\langle X{:}\texttt{Timer}\rangle \in \mathcal{V}$, then `start` X is a $(\mathcal{T}, \mathcal{V})$-statement.

Stopping a timed process: If $\langle X{:}T\rangle \in \mathcal{V}$ for some $T \in$ `ClockType`, then `stop` X is a $(\mathcal{T}, \mathcal{V})$-statement.

Skip: `skip` is a $(\mathcal{T}, \mathcal{V})$-statement.

Procedure call: If $\langle p{:}(T_1, \ldots, T_n)\rangle \in \Lambda$.Behaviour.ProcTypes, and for each $1 \leq i \leq n$ we have that E_i is a $\mathcal{T}$-expression of type T_i, then $p(E_1, \ldots, E_n)$ is a $(\mathcal{T}, \mathcal{V})$-statement.

Control: If E is a $\mathcal{T}$-expression of type `Bool` and π_1 and π_2 are $(\mathcal{T}, \mathcal{V})$-statements, then the following are also $(\mathcal{T}, \mathcal{V})$-statements:

1. `if` E `then` π_1 `else` π_2;
2. `while` E `do` π_1;
3. $\{\pi_1; \pi_2\}$.

4.11 Static semantical correctness

Definition 4.39 (Static semantics of LSC) *Let Σ be a t-, d-, and a-correct specification signature and let $\Lambda \in \Sigma$.LSC. Λ is* Statically Semantically Correct *(SSC) with respect to Σ if the following conditions hold:*

1. *For* S *in the set* Λ.Behaviour.ExtTelegrams, Λ.Behaviour.IntTelegrams, *or* Λ.Behaviour.Procedures, *we have that* S.Body.Statement *is a* $(\mathcal{T}, \mathcal{V})$-*statement, where*

$$\begin{array}{rcl} \mathcal{V} & = & \Lambda.\text{Vars} \cup \mathsf{Set}(S.\text{Pars}) \cup S.\text{Body.Vars} \\ \mathcal{T} & = & \mathcal{V} \cup \mathsf{Set}(\Lambda.\text{Pars}) \end{array}$$

   Note that the variables in S.Pars *are not treated as parameters, despite the name; they are treated as true variables, to which values may be assigned.*

2. *If* $S \in \{\Lambda.\text{Initial}, \Lambda.\text{Panic}\}$, *then* S.Statement *is a* $(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$-*statement, where*

$$\begin{array}{rcl} \mathcal{V} & = & \Lambda.\text{Vars} \cup S.\text{Vars} \\ \mathcal{T} & = & \mathcal{V} \cup \mathsf{Set}(\Lambda.\text{Pars}) \end{array}$$

3. *If* $S_1, S_2 \in \Lambda$.Behaviour.ExtTelegrams *and* S_1.Reaction $= S_2$.Reaction, *then* $S_1 = S_2$.

4. *Suppose either*

$$S_1, S_2 \in \Lambda.\text{Behaviour.IntTelegrams}$$

   or

$$S_1, S_2 \in \Lambda.\text{Behaviour.Procedures.}$$

   Then S_1.Name $= S_2$.Name *implies* $S_1 = S_2$. □

Given a specification signature Σ, we introduce the following abbreviations:

$$\begin{array}{rcl} \mathsf{Bound}_\Sigma & = & \Sigma.\text{System.Bound} \\ \mathsf{Bindings}_\Sigma & = & \Sigma.\text{System.Bindings} \end{array}$$

□

Definition 4.40 (Static semantics of specification signatures) *A specification signature* Σ *is SSC if the following conditions hold:*

1. Σ *is t-, d-, and a-correct;*
2. *every* $\Lambda \in \Sigma$.LSC *is SSC with respect to* Σ*;*
3. *if* $\Lambda_1, \Lambda_2 \in \Sigma$.LSC *and* Λ_1.Name $= \Lambda_2$.Name *then* $\Lambda_1 = \Lambda_2$*;*
4. *if* $E\, N(E_1, \ldots, E_n) \in \mathsf{Bindings}_\Sigma$, *then for some* $\Lambda \in \Sigma$.LSC*:*
   - Λ.Name $= N$*;*
   - *if* $\Lambda.\text{Pars} = (\langle X_1{:}T_1\rangle, \ldots, \langle X_n{:}T_n\rangle)$, *then* E_i *is a* $(\Sigma, \emptyset)$-*expression of type* T_i, *for* $1 \le i \le n$*;*
5. *if* $E\, N(E_1, \ldots, E_n)$ *and* $E\, M(F_1, \ldots, F_m)$ *are in* $\mathsf{Bindings}_\Sigma$, *then* $N = M$ *and* $(E_1, \ldots, E_n) = (F_1, \ldots, F_m)$. □

In an SSC specification signature Σ, for each $E \in \mathsf{Bound}_\Sigma$ there is exactly one $\Lambda \in \Sigma.\mathrm{LSC}$ such that there is a `Binding` of the form

$$E\ \Lambda.\mathrm{Name}(E_1, \ldots, E_n)$$

in $\mathsf{Bindings}_\Sigma$. Thus we may define a function.

Definition 4.41 (lsc_Σ) *Given an SSC specification signature Σ, we define a function lsc_Σ from Bound_Σ to Σ.LSC as follows. If $E\ N(E_1, \ldots, E_n) \in \mathsf{Binding}_\Sigma$, then $\mathsf{lsc}_\Sigma(E) = \Lambda$, where Λ is the unique element of $\Sigma.\mathrm{LSC}$ with $\Lambda.\mathrm{Name} = N$.* □

4.12 Substitutions

We define the notion of substitution. Although it is used in the definition of the operational semantics, we place it in the part on syntax, because substitution is a purely syntactic notion, dealing with forms, not meaning.

Substitutions replace variables in expressions by expressions, in a type-preserving way. The substitution may result in an expression using other variables than used in the original, so we need two type functions in the definition of substitution, one for the types of variables in the input and one for the output.

Consider an SSC specification signature Σ. Let $\mathcal{T}$ and $\mathcal{T}'$ be Σ-type functions.

Definition 4.42 (Substitution) *A* $(\Sigma, \mathcal{T}, \mathcal{T}')$-substitution *is a function*

$$\rho : \mathsf{dom}(\mathcal{T}) \to \mathtt{Expr}(\Sigma, \mathcal{T}')$$

such that if $\langle X{:}T\rangle \in \mathcal{T}$ then $\mathsf{type}_{\Sigma,\mathcal{T}'}(\rho(X)) = T$. □

Definition 4.43 (Substitution on expressions) *Let ρ be a $(\Sigma, \mathcal{T}, \mathcal{T}')$-substitution. The function*

$$[\rho] : \mathtt{Expr}(\Sigma, \mathcal{T}) \to \mathtt{Expr}(\Sigma, \mathcal{T}')$$

is defined as follows ($[\rho]$ is written postfix).

- *If $\langle X{:}T\rangle \in \mathcal{T}$ and $T \in \mathsf{DataType}_\Sigma$ then $X[\rho] = \rho(X)$.*
- *If N is a constant, i.e., a member of the set*

  $$\{\mathtt{true}, \mathtt{false}, \mathtt{self}\} \cup \mathtt{Numeral} \cup \mathsf{Component}_\Sigma \cup \mathsf{Port}_\Sigma \cup \bigcup_{T \in \Sigma.\mathrm{Types}} T.\mathrm{Set}$$

  then $N[\rho] = N$.

- *If* $E[E_0, \ldots, E_n] \in \texttt{Expr}(\Sigma, \mathcal{T})$, *then*

$$(E[E_0, \ldots, E_n])[\rho] = E[\rho](E_0[\rho], \ldots, E_n[\rho]).$$

- *If* $\Box \in \{+, -, *, \texttt{mod}, \texttt{div}, \wedge, ==, <\}$, *then* $(E \Box F)[\rho] = E[\rho] \Box F[\rho]$.
- *If* $\Box \in \{!, \texttt{active}, \texttt{value}\}$, *then* $(\Box E)[\rho] = \Box(E[\rho])$.
- *For convenience, we extend* $[\rho]$ *to the* `Datum` $*$: $*[\rho] = *$. *Let* E *be the* $(\Sigma, \mathcal{T})$*-expression*

$$\{(E_{0,1}, \ldots, E_{n+1,1}), \ldots, (E_{0,m}, \ldots, E_{n+1,m})\} : T.$$

Then

$$E[\rho] = \{(E_{0,1}[\rho], \ldots, E_{n+1,1}[\rho]), \ldots, (E_{0,m}[\rho], \ldots, E_{n+1,m}[\rho])\} : T.$$

$\Box$

Theorem 4.44 *If* ρ *is a* $(\Sigma, \mathcal{T}, \mathcal{T}')$*-substitution and* E *is a* $(\Sigma, \mathcal{T})$*-expression, then* $\mathsf{type}_{\Sigma,\mathcal{T}}(E) = \mathsf{type}_{\Sigma,\mathcal{T}'}(E[\rho])$.

Definition 4.45 (Substitution on statements) *Let* ρ *be a* $(\Sigma, \mathcal{T}, \mathcal{T}')$*-substitution,* $\Lambda \in \Sigma.\mathrm{LSC}$*, and let* $\mathcal{V} \subseteq \mathcal{T}$ *and* $\mathcal{V}' \subseteq \mathcal{T}'$*. The function*

$$[\rho] : \mathsf{Statement}(\Sigma, \Lambda, \mathcal{T}, \mathcal{V}) \ \rightarrow \mathsf{Statement}(\Sigma, \Lambda, \mathcal{T}', \mathcal{V}')$$

is defined as follows ($[\rho]$ *is written postfix).*

- *If* π *is an assignment of the form* $E{:=}\ F$, *then* $\pi[\rho] = (E[\rho]{:=}\ F[\rho])$.
- *If* π *is of the form* $E \mid> \ p \ ! \ N(E_1, \ldots, E_n)$, *then*

$$\pi[\rho] = \ E[\rho] \ \mid> \ p[\rho] \ ! \ N(E_1[\rho], \ldots, E_n[\rho])$$

- *If* π *is of the form* $! \ N(E_1, \ldots, E_n)$, *then*

$$\pi[\rho] = \ ! \ N(E_1[\rho], \ldots, E_n[\rho])$$

- *If* π *is of the form* $\Box \ X \ E \ ! \ N(E_1, \ldots, E_n)$, *where* $\Box \in \{\gg\#, @\}$, *then*

$$\pi[\rho] = \ \Box \ X[\rho] \ E[\rho] \ ! \ N(E_1[\rho], \ldots, E_n[\rho])$$

- *If* π *is of the form* $\Box \ X$, *where* $\Box \in \{\texttt{start}, \texttt{stop}\}$, *then*

$$\pi[\rho] = \ \Box \ X[\rho]$$

- $\texttt{skip}[\rho] = \texttt{skip}$.

- *If π is a procedure call of the form $p(E_1, \ldots, E_n)$, then*

$$\pi[\rho] = p(E_1[\rho], \ldots, E_n[\rho]).$$

- *If π is of the form* $\texttt{if}\, E \,\texttt{then}\, \pi_1 \;\texttt{else}\, \pi_2$, *then*

$$\pi[\rho] = \texttt{if}\, E[\rho] \,\texttt{then}\, \pi_1[\rho] \;\texttt{else}\, \pi_2[\rho].$$

- *If π is of the form* $\texttt{while}\, E \,\texttt{do}\, \pi'$, *then*

$$\pi[\rho] = \texttt{while}\, E[\rho] \,\texttt{do}\, \pi'[\rho].$$

- *If $\pi = \{\pi_1; \pi_2\}$, then*

$$\pi[\rho] = \{\pi_1[\rho]; \pi_2[\rho]\}.$$

□

5 Operational semantics

We define the operational semantics of a variety of LARIS constructions. First of all, Section 5.1 gives the semantics of types. These are sets of possible values for expressions of the type in question. In Section 5.2 one finds the semantics of these expressions. It does not suffice to interpret components of a system by values or sets of values. Components are dynamic objects: components process information, send telegrams, receive them, change values of local variables, in short, they are active. To fully capture the intended meaning of components, one therefore needs to capture this idea of change. The operational semantics of components can be expressed conveniently using *process graphs*. In such a setting, the meaning of a component is viewed in terms of a set of *states* and the *transitions* that are possible between these states. Inductive proof rules are used to define the set of transitions between states, whereby the validity of a number of transitions may imply the validity of some other transition.

In Section 5.3 one finds the operational semantics of components. Communication channels are provided with an operational semantics in Section 5.4. Finally, the operational semantics of a LARIS specification is viewed as the parallel composition of the operational semantics of its components and its channels. This is described in Section 5.5.

Throughout this section we assume fixed some SSC Σ. The semantics of a specification is the semantics of its signature. We only provide specifications that are SSC with a meaning.

5.1 Interpretation of types

We associate with any Σ-type the set of objects that inhabit this type. This association is achieved by means of a function $[\![\cdot]\!]$ whose domain is the set of Σ-types. At the same time we define a function $\mathsf{default}$ on types that specifies a default value for every Σ-type.

- First the `BasicTypes`.
  - $[\![\texttt{Bool}]\!] = \{\texttt{false}, \texttt{true}\}$.
    $\mathsf{default}(\texttt{Bool}) = \texttt{false}$.
  - $[\![\texttt{Int}]\!] = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the set of integers.
    $\mathsf{default}(\texttt{Int}) = 0$.
  - $[\![\texttt{Component}]\!] = \mathsf{Component}_\Sigma$.
    $\mathsf{default}(\texttt{Component}) = \texttt{Log}$.
  - $[\![\texttt{Port}]\!] = \mathsf{Port}_\Sigma$.
    $\mathsf{default}(\texttt{Port}) = \texttt{log}$.
  - Let $T \in \Sigma$.Types. Then $[\![T.\text{Type}]\!] = T.\text{Set}$.
    Furthermore, $\mathsf{default}(T.\text{Type}) = T.\text{Default}$.
- For convenience we extend the function $[\![\cdot]\!]$ to positive numerals: if N is a `PosNumeral` denoting the natural number n, let $[\![N]\!] = \{0, \ldots, n-1\}$. Then we can define $[\![T[I_0, \ldots, I_n]]\!]$ as the set of functions from $[\![I_0]\!] \times \ldots \times [\![I_n]\!]$ to $[\![T]\!]$:
  $$[\![T[I_0, \ldots, I_n]]\!] = [\![T]\!]^{[\![I_0]\!] \times \ldots \times [\![I_n]\!]}.$$
  $\mathsf{default}(T[I_0, \ldots, I_n])$ is the constant function to $\mathsf{default}(T)$. In other words,
  $$\mathsf{default}(T[I_0, \ldots, I_n])(a_0, \ldots, a_n) = \mathsf{default}(T)$$
  for every $(a_0, \ldots, a_n) \in [\![I_0]\!] \times \ldots \times [\![I_n]\!]$.
- Before defining the interpretation of the types for timed processes, it is useful to define the set of internal telegrams that may occur in the system specified.

  IntTel_Σ is defined as:
  $$\left\{ \{N\} \times [\![T_1]\!] \times \ldots \times [\![T_n]\!] \;\middle|\; \begin{array}{l} \text{there exists a } \Lambda \in \Sigma.\text{LSC with} \\ \langle N{:}(T_1, \ldots, T_n)\rangle \in \Lambda.\text{Behaviour.IntTypes} \end{array} \right\}$$
  This set thus contains tuples of the form $(N, d_1, \ldots, d_n)$, where N is a possible internal telegram name and $d_1, \ldots, d_n$ its data parameters.

  $[\![\texttt{Timer}]\!]$ contains all pairs of the following form:
  - $(\texttt{false}, 0)$.
    This is the value of a timer process that is inactive. It is also the default value for the type `Timer`.
  - $(\texttt{true}, n)$ for $n \in \mathbb{N}$ (where $\mathbb{N}$ denotes the set of natural numbers).
    This is the value of a timer that has been switched on n time units ago. The value n should rise by one after each time unit has passed.

  $[\![\texttt{Timeout}]\!]$ contains all tuples of the following form:

– $(\texttt{false}, 0)$.

  This pair is carried by a passive time-out. It is also the default value for `Timeout`.

– $(\texttt{true}, n, T)$, where $n \in \mathbb{N} \setminus \{0\}$ and $T \in \mathsf{IntTel}_\Sigma$.

  These are the values that an active time-out may take on. After n more time units the telegram T will be put into the input buffer. Once this has been done the time-out will be switched off.

$[\![\texttt{Cycler}]\!]$ contains all tuples of the following form:

– $(\texttt{false}, 0)$.

  This pair is carried by a passive cyclic time-out. It is also the default value for `Cycler`.

– $(\texttt{true}, n, m, T)$, where $0 < n \leq m$ and $T \in \mathsf{IntTel}_\Sigma$.

  `true` indicates that the process is active. n indicates the number of time-steps to go before the next telegram will be sent by the process. m indicates the length of the cycle, so m should be no less than n.

  If n reaches below 1, then the telegram T is put into the input buffer, n is replaced by m, and the cycle is repeated.

5.2 Interpretation of expressions

Definition 5.1 ($\mathcal{T}$-assignment) *Let $\mathcal{T}$ be a Σ-type function. A $\mathcal{T}$-*assignment *σ is a function with domain $\mathsf{dom}(\mathcal{T}) \cup \{\texttt{self}\}$ such that:*

- *whenever $\langle X{:}T\rangle \in \mathcal{T}$, we have $\sigma(X) \in [\![T]\!]$;*
- *$\sigma(\texttt{self}) \in \mathsf{Bound}_\Sigma$.*

Given a $\mathcal{T}$-assignment σ, we define an extension of that function with the same name to the set of all $\mathcal{T}$-expressions, such that if $E \in \texttt{Expr}(\mathcal{T})$, then

$$\sigma(E) \in [\![\mathsf{type}_\mathcal{T}(E)]\!] \cup \{\uparrow\}.$$

If $\sigma(E) = \uparrow$, this means that the computation of the value of E has failed: either an index of an array has run out of range, or division by zero has been attempted.

Variables/parameters: If $\langle X{:}T\rangle \in \mathcal{T}$, then $\sigma(X)$ is already defined.

Inhabitants of finite basic types: If $b \in [\![T]\!]$ for some $T \in \mathsf{BasicType}_\Sigma \setminus \{\texttt{Int}\}$, then $\sigma(b) = b$.

Numerals: If N is a `Numeral` and n is the corresponding natural number, then $\sigma(N) = n$.

Self: $\sigma(\texttt{self})$ was already defined by the definition of $\mathcal{T}$-assignments (see Definition 5.1).

Array-positions: If $E[E_0, \ldots, E_n]$ is a $\mathcal{T}$-expression of type $T[I_0, \ldots, I_n]$, then

$$\sigma(E[E_0, \ldots, E_n]) = \begin{cases} \sigma(E)(\sigma(E_0), \ldots, \sigma(E_n)) & \text{if } \sigma(E_i) \in [\![I_i]\!] \text{ for} \\ & i \leq n, \text{ and } \sigma(E) \neq \uparrow; \\ \uparrow & \text{otherwise.} \end{cases}$$

Arithmetical operations: Let E and F be $\mathcal{T}$-expressions of type `Int`.

If $\sigma(E) = \uparrow$ or $\sigma(F) = \uparrow$, then $\sigma(E + F) = \sigma(E - F) = \sigma(E * F) = \sigma(E \texttt{ mod } F) = \sigma(E \texttt{ div } F) = \uparrow$.

If both $\sigma(E)$ and $\sigma(F)$ are integers, then:

$$\begin{array}{lcl}
\sigma(E+F) & = & \sigma(E) + \sigma(F) \\
\sigma(E-F) & = & \sigma(E) - \sigma(F) \\
\sigma(E*F) & = & \sigma(E \cdot F) \\
\sigma(E \texttt{ mod } F) & = & \begin{cases} \text{the unique } 0 \leq n < \sigma(F) & \text{if } \sigma(E) \geq 0 \\ \text{such that for an } m \in \mathbb{N} & \text{and } \sigma(F) \neq 0; \\ m \cdot \sigma(F) + n = \sigma(E) & \\ & \\ \text{the unique } \sigma(F) < n \leq 0 & \text{if } \sigma(E) < 0 \\ \text{such that for an } m \in \mathbb{N} & \text{and } \sigma(F) \neq 0; \\ m \cdot \sigma(F) + n = \sigma(E) & \\ & \\ \uparrow & \text{if } \sigma(F) = 0. \end{cases} \\
 & & \\
\sigma(E \texttt{ div } F) & = & \begin{cases} \text{the unique } n \geq 0 & \text{if } \sigma(E) \geq 0 \text{ and} \\ \text{such that} & \sigma(F) > 0, \text{ or} \\ n \cdot \sigma(F) \leq \sigma(E) < & \sigma(E) < 0 \text{ and} \\ (n+1) \cdot \sigma(F) & \sigma(F) < 0; \\ & \\ \text{the unique } n \leq 0 & \text{if } \sigma(E) \geq 0 \text{ and} \\ \text{such that} & \sigma(F) < 0, \text{ or} \\ (n+1) \cdot \sigma(F) < \sigma(E) \leq & \sigma(E) < 0 \text{ and} \\ n \cdot \sigma(F) & \sigma(F) > 0; \\ & \\ \uparrow & \text{if } \sigma(F) = 0. \end{cases}
\end{array}$$

Boolean operations: Let E and F be $\mathcal{T}$-expressions of type `Bool`.

If $\uparrow \in \{\sigma(E), \sigma(F)\}$, then $\sigma(E \wedge F) = \uparrow$. Likewise, if $\sigma(E) = \uparrow$, then $\sigma(\sim E) = \uparrow$.

Otherwise, if $\sigma(E), \sigma(F) \in \{\texttt{true}, \texttt{false}\}$, then:

$$\sigma(E \wedge F) = \begin{cases} \texttt{true} & \text{if } \sigma(E) = \sigma(F) = \texttt{true}; \\ \texttt{false} & \text{otherwise.} \end{cases}$$

$$\sigma(\sim E) = \begin{cases} \texttt{false} & \text{if } \sigma(E) = \texttt{true}; \\ \texttt{true} & \text{if } \sigma(E) = \texttt{false}. \end{cases}$$

Equality:

$$\sigma(E \ == \ F) = \begin{cases} \uparrow & \text{if } \sigma(E) = \uparrow \text{ or } \sigma(F) = \uparrow; \\ \texttt{true} & \text{if } \sigma(E) = \sigma(F) \text{ and } \sigma(E) \neq \uparrow; \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Less-than:

$$\sigma(E \ < \ F) = \begin{cases} \uparrow & \text{if } \sigma(E) = \uparrow \text{ or } \sigma(F) = \uparrow; \\ \texttt{true} & \text{if } \sigma(E) < \sigma(F); \\ \texttt{false} & \text{otherwise.} \end{cases}$$

Active: $\sigma(\texttt{active}\, X) = p_0(\sigma(X))$, where for any nonempty tuple $(a_0, \ldots, a_n)$, the projection mapping p_0 produces the first element in the tuple: a_0.

Value: $\sigma(\texttt{value}\, X) = p_1(\sigma(X))$, where for any tuple $(a_0, \ldots, a_n)$ with $n \geq 1$, the projection mapping p_1 produces the second element in the tuple: a_1.

Arrays: σ is defined by induction on arrays by induction on their length.

- $\sigma(\{\ \} : T) = \mathsf{default}(T)$.
- Let $T = B[I_0, \ldots, I_n]$, and let E be the $\mathcal{T}$-expression

  $$\{(D_0, \ldots, D_n, D_{n+1}), E_1, \ldots, E_m\} : T$$

  and let f be $\sigma(\{E_1, \ldots, E_m\} : T)$.
  If $f = \uparrow$ or for some $i \leq n+1$ such that $D_i \neq *$ it is the case that $\sigma(D_i) = \uparrow$, then $\sigma(E) = \uparrow$.
  If for some $i \leq n$ such that I_i is a `PosNumeral` and $D_i \neq *$ it is the case that $\sigma(D_i) \notin [\![I_i]\!]$, then $\sigma(E) = \uparrow$. This case deals with assignments to indices that are out of range.
  Otherwise, if none of the above hold, the function $\sigma(E) : [\![I_0]\!] \times \ldots [\![I_n]\!] \to [\![B]\!]$ is defined as follows:

  $$\sigma(E)(a_0, \ldots, a_n) = \begin{cases} \sigma(D_{n+1}) & \text{if for each } i \leq n \text{ with} \\ & D_i \neq * \text{ we have } a_i = \sigma(D_i); \\ f(a_0, \ldots, a_n) & \text{otherwise.} \end{cases}$$

5.3 Operational semantics of a component

Up to now, we have given meaning to the static parts of a LARIS specification: its types and expressions. The components of a specification require a more dynamic semantics, a semantics where the notion of change is made explicit. We choose to model components, communication channels, and indeed complete systems by means of *process graphs*, which are defined below.

Definition 5.2 (Process graph) *A* process graph *is a tuple*

$$\mathcal{M} = (S, s, A, R)$$

where:

- *S is a nonempty set; the elements of this set are referred to as the* states *of $\mathcal{M}$;*
- *$s \in S$ is the* initial state *of $\mathcal{M}$;*
- *A is the set of* actions *of $\mathcal{M}$;*
- *$R \subseteq S \times A \times S$ is the set of* transitions *of $\mathcal{M}$; if $\mathcal{M}$ is clear from the context, we sometimes write $s_1 \xrightarrow{a} s_2$ instead of $(s_1, a, s_2) \in R$.* □

We associate components of Σ with a process graph semantics. Thus, given a `Binding`

$$C \;\; N(P_1, \ldots, P_k)$$

in Σ.System.Bindings, we define a process graph $\mathcal{M}_C$, the operational semantics of the component with name C. To define $\mathcal{M}_C$, we must provide all ingredients required for a process graph: its set of states, its initial state, its set of actions, and its transitions. These notions are introduced in the following sections. Throughout these sections, we fix the following notations.

- Let Λ be the LSC associated with the `Name` N. Thus, $\Lambda = \mathsf{lsc}_\Sigma(C)$.
- Let Λ.Pars $= (\langle X_1{:}T_1\rangle, \ldots, \langle X_k{:}T_k\rangle)$.
- Let ξ be the $\emptyset$-assignment (which must be defined on `self`, by Definition 5.1) with $\xi(\texttt{self}) = C$. As $P_1, \ldots, P_k$ do not contain variables, ξ is defined on these expressions.

  Next, we define the $\{\langle X_1{:}T_1\rangle, \ldots, \langle X_k{:}T_k\rangle\}$-assignment σ_0:

$$\begin{array}{lcll} \sigma_0(X_i) & = & \xi(P_i) & \text{if } 1 \le i \le k \\ \sigma_0(\texttt{self}) & = & C & \end{array}$$

5.3.1 Sets of telegrams

We first define a few useful sets of telegrams.

- ExtTel_Σ is the set of external telegrams that may appear in the system, not just those that may arrive at, or be sent by, a component specified by Λ. It is defined as:

$$\left\{(N,\vec{d}) \mid \exists T_1,\ldots,T_n \left(\begin{array}{l}\langle N{:}(T_1,\ldots,T_n)\rangle \in \mathsf{ExtType}_\Sigma \text{ and} \\ (\vec{d}) \in [\![T_1]\!] \times \ldots \times [\![T_n]\!]\end{array}\right)\right\}$$

- ExtTel_Λ is the set of telegrams that components specified with Λ may receive, together with the ports on which these may be received:

$$\left\{\langle p,(N,\vec{d})\rangle \mid \begin{array}{l}(N,\vec{d}) \in \mathsf{ExtTel}_\Sigma \text{ and} \\ \langle p,N\rangle \in \Lambda.\text{Behaviour.Reactions}\end{array}\right\}$$

- IntTel_Λ, the set of internal telegrams of Λ, is defined as follows:

$$\left\{(N,\vec{d}) \mid \exists T_1,\ldots,T_n \left(\begin{array}{l}\langle N{:}(T_1,\ldots,T_n)\rangle \in \Lambda.\text{Behaviour.IntTypes} \\ \wedge\ (\vec{d}) \in [\![T_1]\!] \times \ldots \times [\![T_n]\!]\end{array}\right)\right\}$$

- Finally, we define the set Tel_Λ of telegrams that components specified with Λ may receive in their input buffer, consisting of the two previously defined sets, with a tag '$\mathsf{internal}$' attached to internal telegrams:

$$\{(\mathsf{internal},T) \mid T \in \mathsf{IntTel}_\Lambda\} \cup \mathsf{ExtTel}_\Lambda$$

5.3.2 States of $\mathcal{M}_C$

A *state* of $\mathcal{M}_C$ is a quintuple $(\mathcal{T},\mathcal{V},\pi,\sigma,B)$, where:

1. $\mathcal{T}$ is a Σ-type function such that $\mathsf{Set}(\Lambda.\text{Pars}) \cup \Lambda.\text{Vars} \subseteq \mathcal{T}$;

2. $\Lambda.\text{Vars} \subseteq \mathcal{V} \subseteq \mathcal{T} \setminus \mathsf{Set}(\Lambda.\text{Pars})$;

3. π is either a $(\mathcal{T},\mathcal{V})$-`Statement` (thus $\mathcal{V}$ is the set of variables of π to which assignments may be made), or the symbol $\surd$;

   the presence of a `Statement` π in a state indicates that the actions dictated by π should be carried out before another telegram is processed, while the symbol $\surd$ indicates that the component is ready to process a next telegram;

4. σ is a $\mathcal{T}$-assignment that agrees with σ_0 on $\mathsf{dom}(\mathsf{Set}(\Lambda.\text{Pars}) \cup \{\mathtt{self}\})$;

5. B is a queue containing elements of Tel_Λ; we use list notation for queues: $[\,]$ denotes the empty queue and $\cdot$ denotes concatenation of queues.

5.3.3 Initial state of $\mathcal{M}_C$

The *initial state* of $\mathcal{M}_C$ is $s_\iota = (\mathcal{T}_\iota, \mathcal{V}_\iota, \pi_\iota, \sigma_\iota, [])$, where:

1. $\mathcal{V}_\iota = \Lambda.\mathrm{Vars} \cup \Lambda.\mathrm{Initial.Vars}$;
2. $\mathcal{T}_\iota = \mathsf{Set}(\Lambda.\mathrm{Pars}) \cup \mathcal{V}_\iota$;
3. $\pi_\iota = \Lambda.\mathrm{Initial.Statement}$;
4. σ_ι is the following $\mathcal{T}_\iota$-assignment:
   (a) σ_ι agrees with σ_0 on $\mathsf{dom}(\mathsf{Set}(\Lambda.\mathrm{Pars}) \cup \mathtt{self}\})$;
   (b) if $\langle X{:}T\rangle \in \Lambda.\mathrm{Vars} \cup \Lambda.\mathrm{Initial.Vars}$, then $\sigma_\iota(X) = \mathsf{default}(T)$.

5.3.4 Actions of $\mathcal{M}_C$

Definition 5.3 (Act_C) Act_C *is the set containing as elements:*

- $\mathsf{assign}(X, a)$*, where* $\langle X{:}T\rangle \in \Lambda.\mathrm{Vars}$ *for some* $T \in \mathsf{DataType}_\Sigma$ *and* $a \in [\![T]\!]$*;*
- $\mathsf{entry}(X, (a_1, \ldots, a_n, a))$*, where* $\langle X{:}T[I_0, \ldots, I_n]\rangle \in \Lambda.\mathrm{Vars}$ *for some array type* $T[I_0, \ldots, I_n]$*, and for every* $i \leq n$ *we have* $a_i \in [\![I_i]\!] \cup \{*\}$ *and* $a \in [\![T]\!]$*;*
- $\mathsf{send}(E, p, T)$*, where* $E \in \mathsf{Component}_\Sigma \setminus \{C\}$*,* $p \in \mathsf{Port}_\Sigma$*, and* $T \in \mathsf{ExtTel}_\Sigma$*;*
- $\mathsf{receive}(p, T)$*, where* $\langle p, T\rangle \in \mathsf{ExtTel}_\Lambda$*;*
- $\mathsf{in}(T)$*, where* $T \in \mathsf{IntTel}_\Lambda$*;*
- $\mathsf{out}(p, T)$*, where* $\langle p, T\rangle \in \mathsf{ExtTel}_\Lambda$*;*
- $\mathsf{out}(T)$*, where* $T \in \mathsf{IntTel}_\Lambda$*;*
- $\mathsf{timer}(X)$*, where* $\langle X{:}\mathtt{Timer}\rangle \in \Lambda.\mathrm{Vars}$*;*
- $\mathsf{timeout}(X, n, T)$*, where* $\langle X{:}\mathtt{Timeout}\rangle \in \Lambda.\mathrm{Vars}$*,* $n \in \mathbb{N} \setminus \{0\}$*, and* $T \in \mathsf{IntTel}_\Lambda$*;*
- $\mathsf{cycler}(X, n, T)$*, where* $\langle X{:}\mathtt{Cycler}\rangle \in \Lambda.\mathrm{Vars}$*,* $n \in \mathbb{N} \setminus \{0\}$*, and* $T \in \mathsf{IntTel}_\Lambda$*;*
- $\mathsf{stop}(X)$*, where* $\langle X{:}T\rangle \in \Lambda.\mathrm{Vars}$ *for some* $T \in \mathtt{ClockType}$*;*
- panic. □

Now, the actions of $\mathcal{M}_C$ are $\mathsf{Act}_C \cup \{\mathsf{t}, \tau\}$. Here the special action τ denotes the *silent action*, invisible from outside the component (see [11] for a discussion on silent actions and [12] for a treatment of equivalences on process graphs with such actions). t denotes a time-step action: the result of this action is that timed processes are updated.

5.3.5 Transitions of $\mathcal{M}_C$

The transitions of $\mathcal{M}_C$ are defined, not directly, but by means of a *Transition System Specification* [20]. In the following definitions (up to Definition 5.7), we assume fixed a signature Σ and a set A of actions.

Definition 5.4 (Transition rule) *A* transition rule *is of the form*

$$\frac{t_1 \xrightarrow{a_1} t_1' \qquad \ldots \qquad t_n \xrightarrow{a_n} t_n'}{t \xrightarrow{a} t'}$$

where $t, t', t_1, \ldots, t_n, t_1', \ldots, t_n'$ are expressions and $a, a_1, \ldots, a_n$ are actions. □

Definition 5.5 (TSS) *A* Transition System Specification *(or* TSS*) is a set of transition rules.* □

Definition 5.6 (Proof) *A* proof *from a TSS $\mathcal{R}$ of a transition rule*

$$\frac{t_1 \xrightarrow{a_1} t_1' \qquad \ldots \qquad t_n \xrightarrow{a_n} t_n'}{t \xrightarrow{a} t'}$$

consists of an upwardly branching tree in which all upward paths are finite, where the nodes of the tree carry labels of the form $u \xrightarrow{b} u'$, such that:

- *the root has label $t \xrightarrow{a} t'$;*
- *if some node has label $u \xrightarrow{b} u'$, and K is the set of labels of nodes directly above this node, then*
  1. *either $K = \emptyset$ and $u \xrightarrow{b} u'$ equals $t_i \xrightarrow{a_i} t_i'$ for a certain $i \in \{1, \ldots, n\}$,*
  2. *or $\frac{K}{u \xrightarrow{b} u'}$ is a substitution instance of a transition rule in $\mathcal{R}$.*

Definition 5.7 (Process graph determined by TSS) *For a TSS $\mathcal{R}$, we define $\mathcal{M}(\mathcal{R})$ to be the process graph (S, s, A, R), where*

$$R = \{(s_1, a, s_2) \in S \times A \times S \mid \frac{}{s_1 \xrightarrow{a} s_2} \text{ is provable from } \mathcal{R}\}.$$

□

Up to now we have defined the states, the initial state, and the sets of actions of $\mathcal{M}_C$. The above tells us that all that remains is to define a TSS: this yields the induced process graph.

Before defining the required TSS, we introduce an auxiliary notation, which helps to express that transitions involve updating a function.

Definition 5.8 (Updating a function) *Let $f : A \rightarrow B$, let $a_1, \ldots, a_n$ be distinct elements of B and let $b_1, \ldots, b_n$ be (not necessarily distinct) elements of B. Then $f[a_1 := b_1, \ldots, a_n := b_n] : A \rightarrow B$ is defined as follows:*

$$f[a_1 := b_1, \ldots, a_n := b_n](a) = \begin{cases} b_i & \textit{if } 1 \leq i \leq n \textit{ and } a = a_i; \\ f(a) & \textit{otherwise.} \end{cases}$$

□

We are ready to define the TSS $\mathcal{R}$. It contains precisely the following transition rules.

Assignment to variables: In the semantics we make a distinction between assignments to LSC variables and assignments to variables local to a procedure definition or message handling. The first are externally visible, while the second are invisible to the outside world, so that these correspond to τ-steps.

Suppose

- $(\mathcal{T}, \mathcal{V}, X := E, \sigma, B)$ is a state (that is, a state of $\mathcal{M}_C$);
- $\langle X : T \rangle \in \Lambda.\text{Vars}$; and
- $\sigma(E) \neq \uparrow$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, X := E, \sigma, B) \stackrel{\mathsf{assign}(X, \sigma(E))}{\rightarrow} (\mathcal{T}, \mathcal{V}, \surd, \sigma[X := \sigma(E)], B)}$$

is a transition rule in $\mathcal{R}$.

Suppose

- $(\mathcal{T}, \mathcal{V}, X := E, \sigma, B)$ is a state;
- $\langle X : T \rangle \in \mathcal{V} \setminus (\Lambda.\text{Vars})$; and
- $\sigma(E) \neq \uparrow$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, X := E, \sigma, B) \stackrel{\tau}{\rightarrow} (\mathcal{T}, \mathcal{V}, \surd, \sigma[X := \sigma(E)], B)}$$

is a transition rule in $\mathcal{R}$.

Assignment to array-positions: The same distinction between assignments to LSC variables and other variables is made here.

Suppose

- $(\mathcal{T}, \mathcal{V}, X[E_0, \ldots, E_n] := E, \sigma, B)$ is a state;
- $\langle X : T[I_0, \ldots, I_n] \rangle \in \Lambda.\text{Vars}$;

- for each $i \leq n$ such that $E_i \neq *$, we have $\sigma(E_i) \in [\![I_i]\!]$;
- $\sigma(E) \neq\uparrow$;
- σ' is the $\mathcal{T}$-assignment that differs from σ only in its assignment to X:

$$\sigma'(X)(a_0,\ldots,a_n) = \begin{cases} \sigma(E) & \text{if for all } i \leq n \text{ with} \\ & E_i \neq * \text{ we have} \\ & a_i = \sigma(E_i); \\ \sigma(X)(a_0,\ldots,a_n) & \text{otherwise;} \end{cases}$$

- for each $i \leq n$:

$$b_i = \begin{cases} * & \text{if } E_i = *; \\ \sigma(E_i) & \text{otherwise} \end{cases}$$

Then

$$\frac{}{(\mathcal{T},\mathcal{V},X[E_0,\ldots,E_n]{:=}\ E,\sigma,B) \overset{\mathsf{entry}(X,(b_0,\ldots,b_n,\sigma(E)))}{\rightarrow} (\mathcal{T},\mathcal{V},\surd,\sigma',B)}$$

is a transition rule in $\mathcal{R}$.

Suppose

- $(\mathcal{T},\mathcal{V},X[E_0,\ldots,E_n]{:=}\ E,\sigma,B)$ is a state;
- $\langle X{:}T[I_0,\ldots,I_n]\rangle \in \mathcal{V} \setminus (\Lambda.\text{Vars})$;
- for each $i \leq n$ with $E_i \neq *$, we have $\sigma(E_i) \in [\![I_i]\!]$;
- $\sigma(E) \neq \uparrow$; and
- σ' is the $\mathcal{T}$-assignment that differs from σ only in its assignment to X:

$$\sigma'(X)(a_0,\ldots,a_n) = \begin{cases} \sigma(E) & \text{if for all } i \leq n \text{ with} \\ & E_i \neq * \text{ we have} \\ & a_i = \sigma(E_i); \\ \sigma(X)(a_0,\ldots,a_n) & \text{otherwise.} \end{cases}$$

Then

$$\frac{}{(\mathcal{T},\mathcal{V},X[E_0,\ldots,E_n]{:=}\ E,\sigma,B) \overset{\tau}{\rightarrow} (\mathcal{T},\mathcal{V},\surd,\sigma',B)}$$

is a transition rule in $\mathcal{R}$.

Sending an external telegram: Suppose

- π is the `Statement` E |> P ! $N(E_1,\ldots,E_n)$;
- $(\mathcal{T},\mathcal{V},\pi,\sigma,B)$ is a state;
- $\sigma(E) \notin \{C,\uparrow\}$;

- $\sigma(P) \neq \uparrow$; and
- $T = (N, \sigma(E_1), \ldots, \sigma(E_n)) \in \mathsf{ExtTel}_\Sigma$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \pi, \sigma, B) \overset{\mathsf{send}(\sigma(E), \sigma(P), T)}{\to} (\mathcal{T}, \mathcal{V}, \surd, \sigma, B)}$$

is a transition rule in $\mathcal{R}$.

Sending an internal telegram: Suppose

- π is the `Statement` $!\ N(E_1, \ldots, E_n)$;
- $(\mathcal{T}, \mathcal{V}, \pi, \sigma, B)$ is a state; and
- $T = (N, \sigma(E_1), \ldots, \sigma(E_n)) \in \mathsf{IntTel}_\Lambda$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \pi, \sigma, B) \overset{\mathsf{in}(T)}{\to} (\mathcal{T}, \mathcal{V}, \surd, \sigma, B \cdot [(\mathsf{internal}, T)])}$$

is a transition rule in $\mathcal{R}$.

Starting a time-out: Suppose

- π is the `Statement` $\gg\#\ X\ E\ !\ N(E_1, \ldots, E_n)$;
- $(\mathcal{T}, \mathcal{V}, \pi, \sigma, B)$ is a state;
- $\sigma(E) \in \mathbb{N} \setminus \{0\}$; and
- $T = (N, \sigma(E_1), \ldots, \sigma(E_n)) \in \mathsf{IntTel}_\Lambda$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \pi, \sigma, B) \overset{\mathsf{timeout}(X, \sigma(E), T)}{\to} (\mathcal{T}, \mathcal{V}, \surd, \sigma[X := (\mathtt{true}, \sigma(E), T)], B)}$$

is a transition rule in $\mathcal{R}$.

Starting a cyclic time-out: Suppose

- π is the `Statement` $@\ X\ E\ !\ N(E_1, \ldots, E_n)$;
- $(\mathcal{T}, \mathcal{V}, \pi, \sigma, B)$ is a state;
- $\sigma(E) \in \mathbb{N} \setminus \{0\}$; and
- $T = (N, \sigma(E_1), \ldots, \sigma(E_n)) \in \mathsf{IntTel}_\Lambda$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \pi, \sigma, B) \overset{\mathsf{cycler}(X, \sigma(E), T)}{\to} (\mathcal{T}, \mathcal{V}, \surd, \sigma[X := (\mathtt{true}, \sigma(E), \sigma(E), T)], B)}$$

is a transition rule in $\mathcal{R}$.

Starting a timer: If $(\mathcal{T}, \mathcal{V}, \texttt{start}\ X, \sigma, B)$ is a state, then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \texttt{start}\ X, \sigma, B) \xrightarrow{\mathsf{timer}(X)} (\mathcal{T}, \mathcal{V}, \surd, \sigma[X := (\texttt{true}, 0)], B)}$$

is a transition rule in $\mathcal{R}$.

Stopping a timed process: If $(\mathcal{T}, \mathcal{V}, \texttt{stop}\ X, \sigma, B)$ is a state, then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \texttt{stop}\ X, \sigma, B) \xrightarrow{\mathsf{stop}(X)} (\mathcal{T}, \mathcal{V}, \surd, \sigma[X := (\texttt{false}, 0)], B)}$$

is a transition rule in $\mathcal{R}$.

Skip: If $(\mathcal{T}, \mathcal{V}, \texttt{skip}, \sigma, B)$ is a state, then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \texttt{skip}, \sigma, B) \xrightarrow{\tau} (\mathcal{T}, \mathcal{V}, \surd, \sigma, B)}$$

is a transition rule in $\mathcal{R}$.

Procedure call: Suppose $Pr \in \Lambda$.Behaviour.Procedures with

$$\begin{array}{lcl} Pr.\text{Name} & = & p \\ Pr.\text{Pars} & = & (\langle Y_1{:}U_1\rangle, \ldots, \langle Y_n{:}U_n\rangle) \\ Pr.\text{Body.Vars} & = & \{\langle Z_1{:}V_1\rangle, \ldots, \langle Z_m{:}V_m\rangle\} \\ Pr.\text{Body.Statement} & = & \pi. \end{array}$$

Now, suppose

- $(\mathcal{T}, \mathcal{V}, p(E_1, \ldots, E_n), \sigma, B)$ and s are states;
- a is an action (in $\mathcal{M}_C$ of course);
- for all $1 \leq i \leq n$ we have $\sigma(E_i) \neq \uparrow$;
- $N_1, \ldots, N_n$ and $M_1, \ldots, M_m$ are distinct, Σ-available `Names` outside $\mathsf{dom}(\mathcal{T})$;
- $\mathcal{V}' = \mathcal{V} \cup \{\langle N_1{:}U_1\rangle, \ldots, \langle N_n{:}U_n\rangle, \langle M_1{:}V_1\rangle, \ldots, \langle M_m{:}V_m\rangle\}$;
- $\mathcal{T}' = \mathcal{T} \cup \mathcal{V}'$;
- σ' is the extension of σ to the domain

  $$\mathsf{dom}(\mathcal{T}) \cup \{N_1, \ldots, N_n, M_1, \ldots, M_m\}$$

  such that:

  $$\begin{array}{lcll} \sigma'(N_i) & = & \sigma(E_i) & \text{for } 1 \leq i \leq n; \\ \sigma'(M_j) & = & \mathsf{default}(V_j) & \text{for } 1 \leq j \leq m; \end{array}$$

- and ρ is the $(\Sigma, \mathsf{Set}(\Lambda.\mathrm{Pars}) \cup \Lambda.\mathrm{Vars} \cup \mathsf{Set}(Pr.\mathrm{Pars}) \cup Pr.\mathrm{Body.Vars}, : \mathcal{T}')$- substitution defined thus:

$$\rho(X) = \begin{cases} N_i & \text{if } X = Y_i \text{ for some } 1 \leq i \leq n; \\ M_j & \text{if } X = Z_j \text{ for some } 1 \leq j \leq m; \\ X & \text{otherwise.} \end{cases}$$

Then

$$\frac{(\mathcal{T}', \mathcal{V}', \pi[\rho], \sigma', B) \xrightarrow{a} s}{(\mathcal{T}, \mathcal{V}, p(E_1, \ldots, E_n), \sigma, B) \xrightarrow{a} s}$$

is a transition rule in $\mathcal{R}$.

If-then-else: Suppose

- $(\mathcal{T}, \mathcal{V}, \texttt{if } E \texttt{ then } \pi_{\texttt{true}} \texttt{ else } \pi_{\texttt{false}}, \sigma, B)$ is a state; and
- $\sigma(E) \in \{\texttt{true}, \texttt{false}\}$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \texttt{if } E \texttt{ then } \pi_{\texttt{true}} \texttt{ else } \pi_{\texttt{false}}, \sigma, B) \xrightarrow{\tau} (\mathcal{T}, \mathcal{V}, \pi_{\sigma(E)}, \sigma, B)}$$

is a transition rule in $\mathcal{R}$.

While-do: Suppose

- $(\mathcal{T}, \mathcal{V}, \texttt{while } E \texttt{ do } \pi, \sigma, B)$ is a state; and
- $\sigma(E) = \texttt{true}$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \texttt{while } E \texttt{ do } \pi, \sigma, B) \xrightarrow{\tau} (\mathcal{T}, \mathcal{V}, \pi; \texttt{ while } E \texttt{ do } \pi, \sigma, B)}$$

is a transition rule in $\mathcal{R}$.

Suppose

- $(\mathcal{T}, \mathcal{V}, \texttt{while } E \texttt{ do } \pi, \sigma, B)$ is a state; and
- $\sigma(E) = \texttt{false}$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \texttt{while } E \texttt{ do } \pi, \sigma, B) \xrightarrow{\tau} (\mathcal{T}, \mathcal{V}, \surd, \sigma, B)}$$

is a transition rule in $\mathcal{R}$.

Sequential composition: Suppose

- $(\mathcal{T}, \mathcal{V}, \pi; \pi'', \sigma, B)$ and $(\mathcal{T}', \mathcal{V}', \pi', \sigma', B')$ are states;
- $\pi' \neq \surd$; and

- a is an action.

Then

$$\frac{(\mathcal{T},\mathcal{V},\pi,\sigma,B) \xrightarrow{a} (\mathcal{T}',\mathcal{V}',\pi',\sigma',B')}{(\mathcal{T},\mathcal{V},\pi;\pi\prime\},\sigma,B) \xrightarrow{a} (\mathcal{T}',\mathcal{V}',\pi';\pi'',\sigma',B')}$$

is a transition rule in $\mathcal{R}$.

Suppose

- $(\mathcal{T},\mathcal{V},\pi;\pi',\sigma,B)$ and $(\mathcal{T}',\mathcal{V}',\surd,\sigma',B')$ are states; and
- a is an action.

Then

$$\frac{(\mathcal{T},\mathcal{V},\pi,\sigma,B) \xrightarrow{a} (\mathcal{T}',\mathcal{V}',\surd,\sigma',B')}{(\mathcal{T},\mathcal{V},\pi;\pi',\sigma,B) \xrightarrow{a} (\mathcal{T}',\mathcal{V}',\pi',\sigma',B')}$$

is a transition rule in $\mathcal{R}$.

Receiving an external telegram: Suppose

- $(\mathcal{T},\mathcal{V},\pi,\sigma,B)$ is a state; and
- $\langle p,T\rangle \in \mathsf{ExtTel}_\Lambda$.

Then

$$\frac{}{(\mathcal{T},\mathcal{V},\pi,\sigma,B) \xrightarrow{\mathsf{receive}(p,T)} (\mathcal{T},\mathcal{V},\pi,\sigma,B\cdot[(p,T)])}$$

is a transition rule in $\mathcal{R}$.

Starting a flow corresponding to an external telegram: Suppose

- $(p,T) \in \mathsf{ExtTel}_\Lambda$;
- $T = (N, a_1, \ldots, a_n)$;
- $(\mathcal{T},\mathcal{V},\surd,\sigma,[\langle p,T\rangle]\cdot B)$ is a state;
- $Me \in \Lambda$.Behaviour.ExtTelegrams, and

$$\begin{array}{lcl} Me.\text{Reaction.Pars} & = & \langle p,N\rangle \\ Me.\text{Pars} & = & (\langle Y_1{:}U_1\rangle,\ldots,\langle Y_n{:}U_n\rangle) \\ Me.\text{Body.Vars} & = & \{\langle Z_1{:}V_1\rangle,\ldots,\langle Z_m{:}V_m\rangle\} \\ Me.\text{Body.Statement} & = & \pi; \end{array}$$

- $\mathcal{V}' = \Lambda.\text{Vars} \cup \mathsf{Set}(Me.\text{Pars}) \cup Me.\text{Body.Vars}$;
- $\mathcal{T}' = \mathcal{V}' \cup \mathsf{Set}(\Lambda.\text{Pars})$;
- σ' is the $\mathcal{T}'$-assignment defined as follows:

$$\sigma'(X) = \begin{cases} \sigma(X) & \text{if } \langle X{:}T\rangle \in \mathsf{Set}(\Lambda.\text{Pars}) \cup \Lambda.\text{Vars}; \\ a_i & \text{if } X = Y_i \ (1 \le i \le n); \\ \mathsf{default}(V_j) & \text{if } X = Z_j \ (1 \le j \le m). \end{cases}$$

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \surd, \sigma, [\langle p, T\rangle] \cdot B) \overset{\mathsf{out}(p,T)}{\to} (\mathcal{T}', \mathcal{V}', \pi, \sigma', B)}$$

is a transition rule in $\mathcal{R}$.

Starting a flow corresponding to an internal telegram: Suppose

- $T = (N, a_1, \ldots, a_n) \in \mathsf{IntTel}_\Lambda$;
- $(\mathcal{T}, \mathcal{V}, \surd, \sigma, [\langle \mathsf{internal}, T\rangle] \cdot B)$ is a state;
- $Mi \in \Lambda$.Behaviour.IntTelegrams, and

$$\begin{array}{lcl} Mi.\text{Name.Pars} & = & N \\ Mi.\text{Pars} & = & (\langle Y_1{:}U_1\rangle, \ldots, \langle Y_n{:}U_n\rangle) \\ Mi.\text{Body.Vars} & = & \{\langle Z_1{:}V_1\rangle, \ldots, \langle Z_m{:}V_m\rangle\} \\ Mi.\text{Body.Statement} & = & \pi; \end{array}$$

- $\mathcal{V}' = \Lambda.\text{Vars} \cup \mathsf{Set}(Mi.\text{Pars}) \cup Mi.\text{Body.Vars}$;
- $\mathcal{T}' = \mathcal{V}' \cup \mathsf{Set}(\Lambda.\text{Pars})$;
- σ' is the $\mathcal{T}'$-assignment defined as follows:

$$\sigma'(X) = \begin{cases} \sigma(X) & \text{if } \langle X{:}T\rangle \in \mathsf{Set}(\Lambda.\text{Pars}) \cup \Lambda.\text{Vars}; \\ a_i & \text{if } X = Y_i \ (1 \leq i \leq n); \\ \mathsf{default}(V_j) & \text{if } X = Z_j \ (1 \leq j \leq m). \end{cases}$$

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \surd, \sigma, [\langle \mathsf{internal}, T\rangle] \cdot B) \overset{\mathsf{out}(T)}{\to} (\mathcal{T}', \mathcal{V}', \pi, \sigma', B)}$$

is a transition rule in $\mathcal{R}$.

Panic: Suppose

- $(\mathcal{T}, \mathcal{V}, \pi, \sigma, B)$ is a state;
- $\mathcal{V}' = \Lambda.\text{Vars} \cup \Lambda.\text{Panic.Vars}$;
- $\mathcal{T}' = \mathcal{V}' \cup \mathsf{Set}(\Lambda.\text{Pars})$; and
- σ' is the $\mathcal{T}'$-assignment defined as follows:

$$\sigma'(X) = \begin{cases} \sigma(X) & \text{if } \langle X{:}T\rangle \in \mathsf{Set}(\Lambda.\text{Pars}) \cup \Lambda.\text{Vars}; \\ \mathsf{default}(T_i) & \text{if } \langle X{:}T\rangle \in \Lambda.\text{Panic.Vars}. \end{cases}$$

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \pi, \sigma, B) \overset{\mathsf{panic}}{\to} (\mathcal{T}', \mathcal{V}', \Lambda.\text{Panic.Statement}, \sigma', [])}$$

is a transition rule in $\mathcal{R}$.

Time: Suppose

- $(\mathcal{T}, \mathcal{V}, \pi, \sigma, B)$ is a state;
- σ' is the $\mathcal{T}$-assignment defined as follows:

$$\sigma'(X) = \begin{cases} (\texttt{true}, n+1) & \text{if } \langle X{:}\texttt{Timer}\rangle \in \mathcal{V} \text{ and} \\ & \sigma(X) = (\texttt{true}, n); \\ (\texttt{true}, n-1, T) & \text{if } \langle X{:}\texttt{Timeout}\rangle \in \mathcal{V}, \\ & \sigma(X) = (\texttt{true}, n, T), \\ & \text{and } n \geq 2; \\ (\texttt{false}, 0) & \text{if } \langle X{:}\texttt{Timeout}\rangle \in \mathcal{V}, \text{ and} \\ & \text{for some } T \in \mathsf{IntTel}_\Lambda \text{ we have} \\ & \sigma(X) = (\texttt{true}, 1, T); \\ (\texttt{true}, n-1, m, T) & \text{if } \langle X{:}\texttt{Cycler}\rangle \in \mathcal{V}, \\ & \sigma(X) = (\texttt{true}, n, m, T), \\ & \text{and } n \geq 2; \\ (\texttt{true}, m, m, T) & \text{if } \langle X{:}\texttt{Cycler}\rangle \in \mathcal{V} \text{ and} \\ & \sigma(X) = (\texttt{true}, 1, m, T); \\ \sigma(X) & \text{otherwise;} \end{cases}$$

- $$\begin{array}{rl} & \{(X,T) \mid \langle X{:}\texttt{Timeout}\rangle \in \mathcal{V} \text{ and } \sigma(X) = (\texttt{true}, 1, T)\} \\ \mathsf{Send} = & \cup \\ & \{(X,T) \mid \langle X{:}\texttt{Cycler}\rangle \in \mathcal{V} \text{ and } \sigma(X) = (\texttt{true}, 1, m, T);\} \end{array}$$

- $(X_1, T_1), \ldots, (X_n, T_n)$ is an enumeration of Send (and thus not containing any doubles);
- $Q = [\langle \mathsf{internal}, T_1\rangle, \ldots, \langle \mathsf{internal}, T_n\rangle]$.

Then

$$\frac{}{(\mathcal{T}, \mathcal{V}, \pi, \sigma, B) \xrightarrow{\mathsf{t}} (\mathcal{T}, \mathcal{V}, \pi, \sigma', B \cdot Q)}$$

is a transition rule in $\mathcal{R}$.

5.4 Operational semantics of a communication channel

We define another process graph: $\mathsf{Channel}$, the operational semantics of a communication channel.

States: The states of $\mathsf{Channel}$ consists of all queues containing elements $\langle p, T\rangle$, where $p \in \mathsf{Port}_\Sigma$ and $T \in \mathsf{ExtTel}_\Sigma$.

Initial state: The initial state of $\mathsf{Channel}$ is the empty queue $[\,]$.

Actions: We define the set of transitions of $\mathsf{Channel}$. First define ChAct as the set containing all elements that are of one of the following forms:

1. $\mathsf{read}(p, T)$, where $p \in \mathsf{Port}_\Sigma$ and $T \in \mathsf{ExtTel}_\Sigma$;
2. $\mathsf{transmit}(p, T)$, where $p \in \mathsf{Port}_\Sigma$ and $T \in \mathsf{ExtTel}_\Sigma$.

Then the set of actions of $\mathsf{Channel}$ is $\mathsf{ChAct} \cup \{\mathsf{t}\}$. Thus, there are no silent actions in a communication channel.

Transitions: The transitions that exist in $\mathsf{Channel}$ are the following.

Receiving a telegram: If B is a state of $\mathsf{Channel}$, $p \in \mathsf{Port}_\Sigma$, and $T \in \mathsf{ExtTel}_\Sigma$, then $B \stackrel{\mathsf{read}(p,T)}{\rightarrow} B \cdot [(p, T)]$ is a transition in $\mathsf{Channel}$.

Transmitting a telegram: If B is a state of $\mathsf{Channel}$, $p \in \mathsf{Port}_\Sigma$, and $T \in \mathsf{ExtTel}_\Sigma$, then $[(p, T)] \cdot B \stackrel{\mathsf{transmit}(p,T)}{\rightarrow} B$ is a transition in $\mathsf{Channel}$.

Time: If B is a state of $\mathsf{Channel}$, then $B \stackrel{\mathsf{t}}{\rightarrow} B$ is a transition in $\mathsf{Channel}$.

5.5 Operational semantics of a specification

We give an operational semantics for the SSC specification signature Σ, using the operational semantics already given for components and communication channels.

Definition 5.9 (Σ-action) *The set of* Σ-actions *is defined as:*

$$\mathsf{Act}_\Sigma = \begin{array}{l}
\{\mathsf{Act}_C \times \{C\} \mid C \in \mathsf{Bound}_\Sigma\} \\
\cup \\
\{\mathsf{ChAct} \times \{(C, D)\} \mid C, D \in \mathsf{Bound}_\Sigma, C \neq D\} \\
\cup \\
\left\{ \mathsf{inchannel}(\langle p, T\rangle, (C, D)) \mid \begin{array}{l} p \in \mathsf{Port}_\Sigma, T \in \mathsf{ExtTel}_\Sigma, \\ C, D \in \mathsf{Bound}_\Sigma, C \neq D \end{array} \right\} \\
\cup \\
\left\{ \mathsf{outchannel}(\langle p, T\rangle, (C, D)) \mid \begin{array}{l} \langle p, T\rangle \in \mathsf{ExtTel}_{\mathsf{lsc}_\Sigma(D)}, \\ C, D \in \mathsf{Bound}_\Sigma, C \neq D \end{array} \right\} \\
\cup \\
\{\mathsf{unexpected}(C) \mid C \in \mathsf{Bound}_\Sigma\} \\
\cup \\
\{\mathsf{t}, \tau\}.
\end{array}$$

□

Definition 5.10 **($\mathbb{T}_C$)** *If $\mathcal{M}_C = (S, s, \mathsf{Act}_C \cup \{\mathsf{t}, \tau\}, R)$, then $\mathbb{T}_C$ is the process graph $(S, s, \mathsf{Act}_\Sigma, R')$, where*

$$R' = \begin{array}{l} \{(s_1, (a, C), s_2) \mid a \in \mathsf{Act}_C \textit{ and } (s_1, a, s_2) \in R\} \\ \cup \\ \{(s_1, a, s_2) \mid a \in \{\mathsf{t}, \tau\} \textit{ and } (s_1, a, s_2) \in R\}. \end{array}$$

□

Definition 5.11 **($\mathbb{C}_{(C,D)}$)** *Suppose $C, D \in \mathsf{Bound}_\Sigma$ and $C \neq D$. Let $l = (C, D)$ and $\mathsf{Channel} = (S, s, \mathsf{ChAct} \cup \{\mathsf{t}\}, R)$. Then $\mathbb{C}_l = (S, s, \mathsf{Act}_\Sigma, R')$, where*

$$R' = \begin{array}{l} \{(s_1, (a, l), s_2) \mid a \in \mathsf{ChAct} \textit{ and } (s_1, a, s_2) \in R\} \\ \cup \\ \{(s_1, \mathsf{t}, s_2) \mid (s_1, \mathsf{t}, s_2) \in R\} \end{array}$$

□

Definition 5.12 (Communication function:) *We assume a partially defined* communication function $\gamma : \mathsf{Act}_\Sigma \times \mathsf{Act}_\Sigma \to \mathsf{Act}_\Sigma$. *$\gamma(a, b)$ is defined and equal to c if and only if one of the following holds:*

- $\{a, b\} = \{(\mathsf{send}(D, p, T), C), (\mathsf{read}(p, T), (C, D))\}$ *and*

  $c = \mathsf{inchannel}(\langle p, T\rangle, (C, D))$

  where $C, D \in \mathsf{Bound}_\Sigma$, $C \neq D$, $p \in \mathsf{Port}_\Sigma$, and $T \in \mathsf{ExtTel}_\Sigma$.

  send *is an action of a component and* read *of a channel. Their communication results in placing a telegram in the input buffer of the channel.*

- $\{a, b\} = \{(\mathsf{receive}(p, T), D), (\mathsf{transmit}(p, T), (C, D))\}$ *and*

  $c = \mathsf{outchannel}(\langle p, T\rangle, (C, D))$

  where $C, D \in \mathsf{Bound}_\Sigma$, $C \neq D$, and $\langle p, T\rangle \in \mathsf{ExtTel}_{\mathsf{lsc}_\Sigma(D)}$.

  transmit *is an action of a channel and* receive *that of a component. When they communicate in the above manner, a telegram is moved from the channel to the input buffer of the component.*

- $\{a, b\} = \{(\mathsf{transmit}(p, T), (C, D)), (\mathsf{panic}, D)\}$ *and*

  $c = \mathsf{unexpected}(D)$

  where $C, D \in \mathsf{Bound}_\Sigma$, $C \neq D$, and $\langle p, T\rangle \in \mathsf{ExtTel}_\Sigma \setminus \mathsf{ExtTel}_{\mathsf{lsc}_\Sigma(D)}$.

  This models the receipt of a telegram that cannot be processed by D, for it has no appropriate response to it.

□

The above definition ensures that γ is commutative, meaning that $\gamma(a, b)$ is isomorphic to $\gamma(b, a)$ for all $a, b \in \mathsf{Act}_\Sigma$. It also ensures that γ is associative, as both $\gamma(a, \gamma(b, c))$ and $\gamma(\gamma(a, b), c)$ are always undefined.

Definition 5.13 (Parallel composition) *Suppose* $\mathsf{t} \in A$*, and for each* $i \in \{1,2\}$ *let* $\mathcal{M}_i = (S_i, s_i, A, R_i)$ *be a process graph. Let* $\gamma : A \times A \to A$ *be a partially defined commutative and associative communication function. The* parallel composition *of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *is*

$$\mathcal{M}_1 \parallel \mathcal{M}_2 = (S_1 \times S_2, (s_1, s_2), A, R)$$

where R *is*

$$\begin{array}{l}
\{((t_1, u), a, (t_2, u)) \mid a \in A \setminus \{\mathsf{t}\}, (t_1, a, t_2) \in R_1, u \in S_2\} \\
\cup \\
\{((t, u_1), a, (t, u_2)) \mid a \in A \setminus \{\mathsf{t}\}, t \in S_1, (u_1, a, u_2) \in R_2\} \\
\cup \\
\{((t_1, u_1), \gamma(a, b), (t_2, u_2)) \mid \gamma(a, b) \text{ defined}, (t_1, a, t_2) \in R_1, (u_1, b, u_2) \in R_2\} \\
\cup \\
\{((t_1, u_1), \mathsf{t}, (t_2, u_2)) \mid (t_1, \mathsf{t}, t_2) \in R_1, (u_1, \mathsf{t}, u_2) \in R_2\}.
\end{array}$$

□

$\parallel$ is a commutative and associative operation. That is, if $\mathcal{M}_1, \mathcal{M}_2$, and $\mathcal{M}_3$ are process graphs over the set of actions A, and γ is a partially defined commutative and associative communication function on A, then

$$\begin{array}{lll}
\mathcal{M}_1 \parallel \mathcal{M}_2 & \text{is isomorphic to} & \mathcal{M}_2 \parallel \mathcal{M}_1; \\
(\mathcal{M}_1 \parallel \mathcal{M}_2) \parallel \mathcal{M}_3 & \text{is isomorphic to} & \mathcal{M}_1 \parallel (\mathcal{M}_2 \parallel \mathcal{M}_3).
\end{array}$$

Due to associativity, the notation $\mathcal{M}_0 \parallel \ldots \parallel \mathcal{M}_n$ is unambiguous. If $n = 0$, this denotes simply $\mathcal{M}_0$; otherwise an arbitrary bracketing gives us its meaning, using Definition 5.13.

Enumerate Bound_Σ as $C_0, \ldots, C_n$ and $\{(C, D) \mid C, D \in \mathsf{Bound}_\Sigma, C \neq D\}$ as $l_1, \ldots, l_m$. Then we may define the following.

Definition 5.14 ($\mathcal{M}_\Sigma$) $\mathcal{M}_\Sigma$ *is the process graph*

$$\mathbb{T}_{C_1} \parallel \ldots \parallel \mathbb{T}_{C_n} \parallel \mathbb{C}_{l_1} \parallel \ldots \parallel \mathbb{C}_{l_m}.$$

□

Definition 5.15 (Encapsulation) *Let* $\mathcal{M} = (S, s, A, R)$ *be a process graph and let* $H \subseteq A$*. The* encapsulation operator ∂_H *is defined on* $\mathcal{M}$ *as*

$$\partial_H(\mathcal{M}) = (S, s, A, R \cap (S \times H \times S))$$

□

Finally, we are ready to give our intended semantics for Σ.

Definition 5.16 ($\mathbb{T}_\Sigma$) *Let* H *contain all* Σ*-actions of the following forms:*

- $(\mathsf{send}(D,p,T),C)$, *where* $C,D \in \mathsf{Bound}_\Sigma$, $C \neq D$, $p \in \mathsf{Port}_\Sigma$, *and* $T \in \mathsf{ExtTel}_\Sigma$*;*
- $(\mathsf{read}(p,T),(C,D))$, *where* $C,D \in \mathsf{Bound}_\Sigma$, $C \neq D$, $p \in \mathsf{Port}_\Sigma$, *and* $T \in \mathsf{ExtTel}_\Sigma$*;*
- $(\mathsf{transmit}(p,T),(C,D))$, *where* $C,D \in \mathsf{Bound}_\Sigma$, $C \neq D$, $p \in \mathsf{Port}_\Sigma$, *and* $T \in \mathsf{ExtTel}_\Sigma$*.*

Then $\mathbb{T}_\Sigma = \partial_H(\mathcal{M}_\Sigma)$. □

Note that the above does not encapsulate the receive actions. This is because it is not assumed that we are in a closed system. Any component in our system may receive telegrams from the environment, be it the logistic or infrastructure level, or components of the interlocking outside the specification. In specific cases we may wish to encapsulate some receive actions, if knowledge about the information flow tells us that particular telegrams may only arrive from components within the specification.

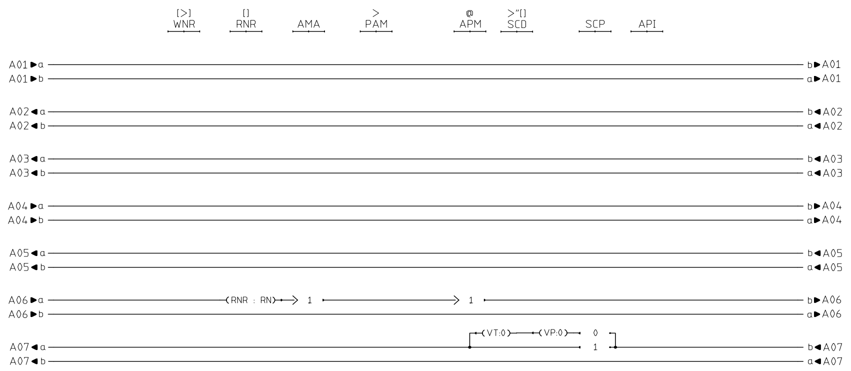
6 Example: part of Driebergen

We work out a concrete exemplifying LARIS specification. The example is a translation into LARIS of the specification in EURIS of the railway yard at Driebergen. We specify only a relatively small part of it, concerning three tracks surrounded by six approach monitors which report to a warning device. The specifications for the tracks, approach monitors, and warning device are taken from UniSpec [16]. The approach monitor and warning device specifications are from UniSpec version September 13, 1997, and the track specification from UniSpec version September 30, 1997. In the translation we have ignored the simulation pages, although a complete specification should not ignore these. The present example, though, is only meant as a demonstration of what a UniSpec specification looks like when it is translated into LARIS. Just before each LSC specification we have placed their UniSpec originals.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                             %
%      LARIS-specification of a part of Driebergen            %
%                                                             %
%      Contents:                                              %
%      1. Type definition                                     %
%      2. Approach Monitor LSC                                %
%      3. Warning Device LSC                                  %
%      4. Track LSC                                           %
%      5. System layout                                       %
%                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%% 1. TYPE DEFINITION(S) %%%%%%%%%%%%%%%%%%%%
%
% The use of enumerated types is heavily encouraged. Their use
% makes much clearer the actual range of values than using
% large types such as Int.
%
% We have two enumerated types, for which in EURIS the type Int
% is used: AMS (Approach Monitoring Status) and Route (for Route
% Type).

AMS   = {occupied, complete_unoccupied,
         incomplete_unoccupied}

Route = {drive_on_sight, normal, automatic_normal,
         drive_on_sight_normal}
```

```
%%%%%%%%%%%%%%%%%%% EXTERNAL TELEGRAM TYPES %%%%%%%%%%%%%%%%%%
%                                                            %
% We list here the types of external telegrams used throughout %
% the system, together with mnemonic names for their         %
% positions, as used in the EURIS-specification.             %
% The fields are extracted from the EURIS-specifications for %
% approach monitors, warning-devices and tracks. It is likely %
% that more fields must be added when other LSCs are studied. %
% This would imply that the current specification would also %
% have to be altered.                                        %
%                                                            %
% A01:()           B01:(TO:Bool) C01:()       D01:(AR:Bool)  %
% A02:()           B02:()        C02:(RL:Int) D02:(AR:Bool)  %
% A03:()           B03:()        C03:(RL:Int) D03:()         %
% A04:(RT:Route)   B04:()        C04:()                      %
% A05:()                         C05:()                      %
% A06:(RN,RL:Int)                C06:()                      %
% A07:(VP,VT:Int)                                            %
%                                                            %
% E01:(WN:Component; RW,CD:Int)       F01:(TS,DL:Bool)       %
% E02:(WN:Component; AM:AMS; RW,CD:Int)                      %
% E03:()                                                     %
% E04:(TS:Bool)                                              %
%                                                            %
% I00:()                                                     %
% I01:(TS:Bool) J01:()                                       %
% I02:()        J02:()                                       %
% I03:()                                                     %
% T01:(E:Component; AM:AMS; WT:Bool; CD:Int)                 %
% T02:(E:Component; TS:Bool)                                 %
%  (TS:Track Section status, false = occupied,               %
%                           true = unoccupied)               %
% U01:()                                                     %
% W02(E:Component; VA:Int)                                   %
% W03(E:Component; VA:Int)                                   %
% W05:(E:Component; WD:Bool)                                 %
%  (WD:Warning Device control, false = warning,              %
%                             true = no warning)             %
% X04(TS:Bool)                                               %
%                                                            %
% P01:(E:Component)                                          %
%  (Not present in the EURIS specification, but used for     %
%   reporting panic to the logistic level. )                 %
%                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```



this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d. Meij	13-sep-1996
checked		
approved		

simulation	0

NV NEDERLANDSE SPOORWEGEN

service :
dep :

apprmonitor

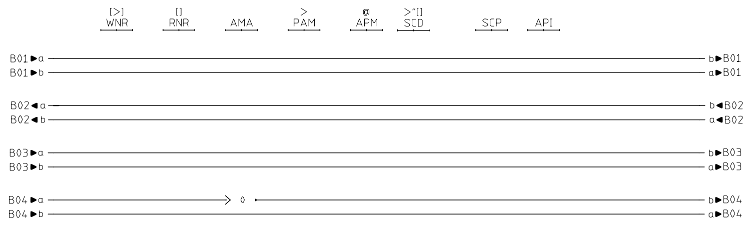
11

scale AM EU

format A4R

release

1



this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d. Meij	13-sep-1996
checked		
approved		

simulation	0

NV NEDERLANDSE SPOORWEGEN

service :

dep :

apprmonitor

11

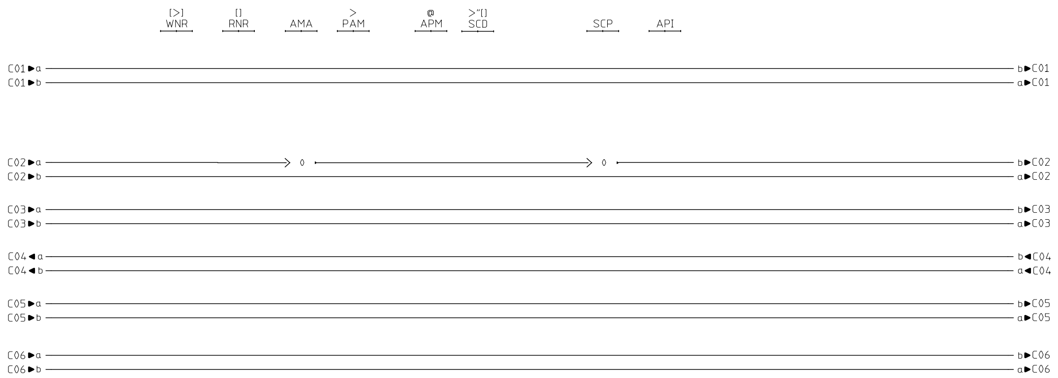
scale AM EU

format A4R

release

2

/tmp_mnt/home/euris/guide_top/unispec/lscdestop/apprmonitor/apprmonitor.sbk(02;19) Mon Dec 1 10:49:07 1997



simulation	0	

this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d. Meij	13-sep-1996
checked		
approved		

NV NEDERLANDSE SPOORWEGEN

service :
dep :

apprmonitor

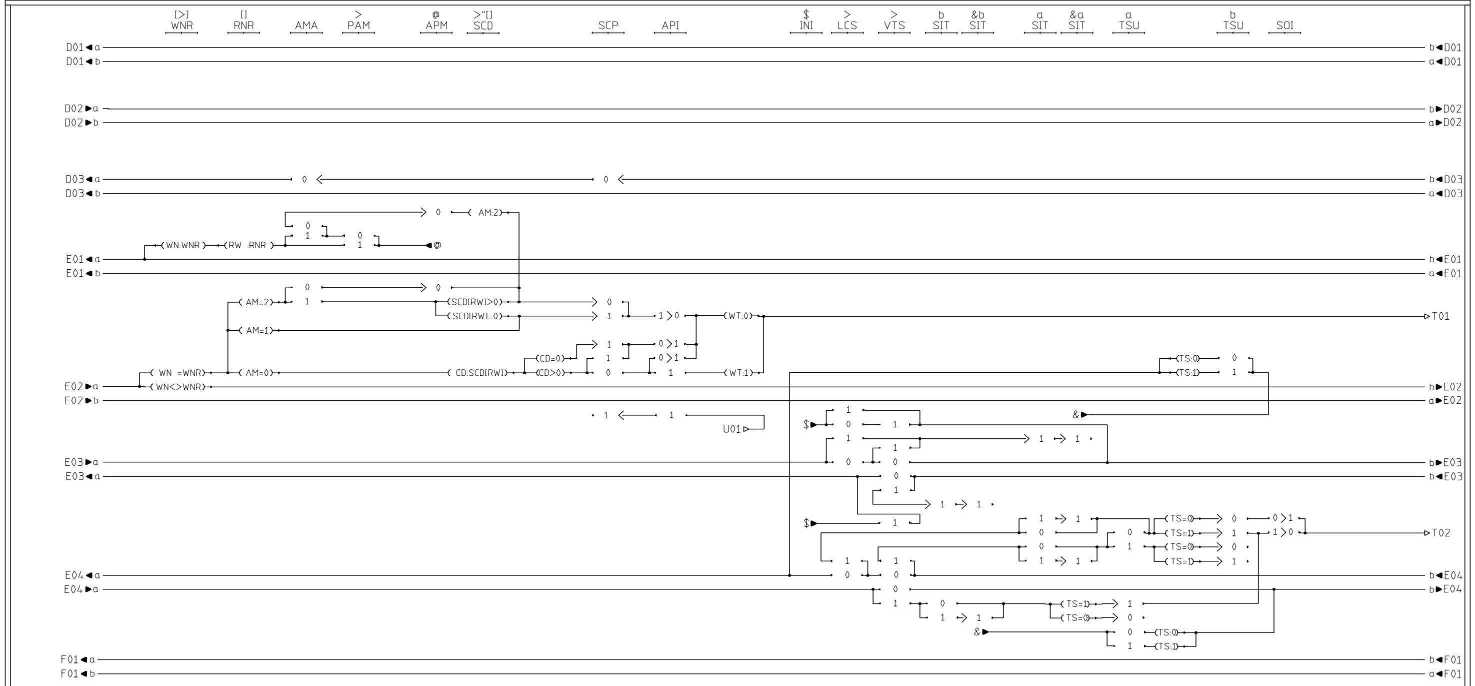
11

scale AM EU

format A4R

release

3



this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d. Meij	17-sep-1996
checked		
approved		

simulation 0

apprmonitor

scale AM EU

format A4R

release

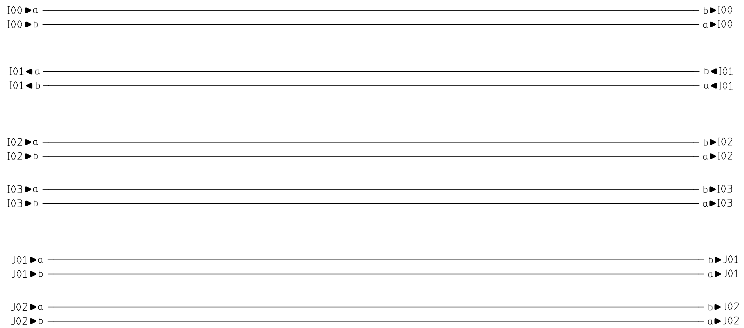
NV NEDERLANDSE SPOORWEGEN

service :
dep :

11

4

/tmp_mnt/home/euris/guide_top/unispec/lscdestop/apprmonitor/apprmonitor.sbk(04;19) Mon Dec 1 10:49:21 1997



simulation	0

this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d.Meij	9-jul-1996
checked		
approved		

NV NEDERLANDSE SPOORWEGEN

service :
dep :

apprmonitor

11

scale AM EU

format A4R

release

5

```
%%%%%%%%%%%%%%%%%%%% 2. APPROACH MONITOR LSC %%%%%%%%%%%%%%%%%%

LSC
apprmonitor     % Name of the LSC

% Parameters of the LSC, and their types:

 (Ea,           % Name of the component connected to the a-port;
  Eb,           % Name of the component connected to the b-port;
  C:Component;  % Name of the warning device to which the current
                % component reports;
  Pa,           % Name of the port of Ea that is connected to
                % port a of the current component.
  Pb:Port;      % Name of the port of Eb that is connected to
                % port b of the current component.
  PAM,          % Passive Approach Monitor
  LCS,          % Level Crossing Section
  VTS:Bool;     % Virtual Track Section
  SCD:Int[Int]) % Signal Clearance Delay: has a value for each
                % route-number.

% Note that the parameter WNR (Warning device NumbeR) has
% disappeared. As we have the type 'Component' we need not
% use a number here, but may actually use the name of the
% waring device in question, here: C.

% C also functions as the 'central list' of the approach-monitor.
% It is not a list, but in this particular use of central
% telegrams no list is needed. All we need is the name of the
% warning device. Central telegrams to this warning device
% (T01 and T02) carry their own names. This name is used
% to update the correct version of multivars in the warning
% device and may also be used to return the telegram U01.
% The use of a list of components would clearly be 'overkill'
% in this instance.

=
vars       % Beginning of the LSC variable declaration.

AMA,       % Approach Monitoring Active
SCP,       % Signal Clearance Permission
API,       % APproach Indicator
SITa,      % Indicator for track Section Transmission at side a
SITa1,     % Oneshot-variable for SITa
SITb,      % Indicator for track Section Transmission at side b
```

```
SITb1,     % Oneshot-variable for SITb
TSUa,      % Track Section Unoccupied at side a
TSUb,      % Track Section Unoccupied at side b
SOI:Bool;  % Track Section Occupation Indicator
RNR:Int;   % Route NumbeR
APM:Cycler % Approach Monitoring telegram initiator

% Next we must define the behaviour of the system at startup.
% This behaviour is achieved by means of the INI-telegrams
% in the EURIS-specification (toggles always fire at startup).
% This is simulated in LARIS by sending the INI-telegram
% (with no data, indicated by '()') in the 'initial' statement.
% There is only a single INI-telegram in our specification: the
% two INI-flows of the EURIS-specification have been concatenated
% in the flow corresponding to INI in our specification.
% This means that a cetain order on these flows has been chosen.

% A certain order on the INI-telegrams has been chosen, where
% this order was not present in the original.

initial ! INI()

% The next route setting telegrams are simply passed from one
% port to the next.
% We comment on the first one: the others have an analogous
% meaning. If an A01-telegram is received on the a-port,
% send it to the component Eb, on port Pb of that component
% (i.e., send it along port b).

mes a? A01() = Eb |> Pb ! A01()
mes b? A01() = Ea |> Pa ! A01()
mes a? A02() = Eb |> Pb ! A02()
mes b? A02() = Ea |> Pa ! A02()
mes a? A03() = Eb |> Pb ! A03()
mes b? A03() = Ea |> Pa ! A03()
mes a? A04(RT:Route) = Eb |> Pb ! A04(RT)
mes b? A04(RT:Route) = Ea |> Pa ! A04(RT)
mes a? A05() = Eb |> Pb ! A05()
mes b? A05() = Ea |> Pa ! A05()

% The route locking telegram A06 is passed on from the a-port to
% the b-port. The telegram must carry an integer value RN,
% representing the Route Number and an integer RL, representing
% the Route Length.
```

```
% Among other things, below  a cyclic telegram APM is activated
% with cycle one. The name APM is overloaded here. It serves
% both as a variable of type Cycler, and as the name of the
% internal telegram.

mes a? A06(RN,RL:Int) =
   RNR:= RN;
   AMA:= true;
   @ APM 1 ! APM();
   Eb |> Pb ! A06(RN,RL)

% If the A06-telegram is received on the b-port, it is simply
% passed on.

mes b? A06(RN:Bool; RL:Int) = Ea |> Pa ! A06(RN,RL)

% A07 (route monitoring) carries two values: VP (Passing speed)
% and VT (Target speed), both of them integers.
% A07 is carried along to the opposite port.

mes a? A07(VP,VT:Int) = Eb |> Pb ! A07(VP,VT)

% However, when received on the b-port it is checked whether we
% have Signal Clearance Permission. If not, then VP and VT are
% set accordingly, before being passed along.

mes b? A07(VP,VT:Int) =
   if ~SCP then {VP:= 0; VT:= 0};
   Ea |> Pa ! A07(VP,VT)

mes a? B01(TO:Bool) = Eb |> Pb ! B01(TO)
mes b? B01(TO:Bool) = Ea |> Pa ! B01(TO)
mes a? B02() = Eb |> Pb ! B02()
mes b? B02() = Ea |> Pa ! B02()
mes a? B03() = Eb |> Pb ! B03()
mes b? B03() = Ea |> Pa ! B03()
mes a? B04() = AMA:= false; Eb |> Pb ! B03()
mes b? B04() = Ea |> Pa ! B04()
mes a? C01() = Eb |> Pb ! C01()
mes b? C01() = Ea |> Pa ! C01()
mes a? C02(RL:Int) =
   AMA:= false; SCP:= false; Eb |> Pb ! C02(RL)
mes b? C02(RL:Int) = Ea |> Pa ! C02(RL)
mes a? C03(RL:Int) = Eb |> Pb ! C03(RL)
mes b? C03(RL:Int) = Ea |> Pa ! C03(RL)
```

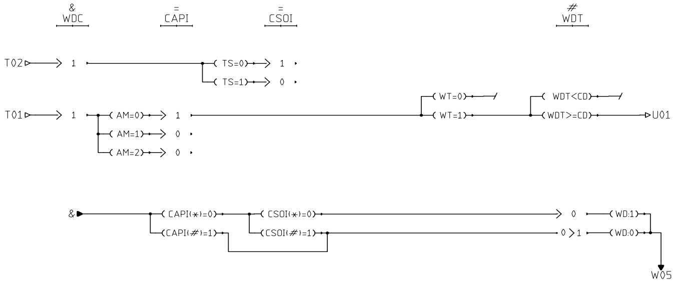
```
mes a? C04() = Eb |> Pb ! C04()
mes b? C04() = Ea |> Pa ! C04()
mes a? C05() = Eb |> Pb ! C05()
mes b? C05() = Ea |> Pa ! C05()
mes a? C06() = Eb |> Pb ! C06()
mes b? C06() = Ea |> Pa ! C06()
mes a? D01(AR:Bool) = Eb |> Pb ! D01(AR)
mes b? D01(AR:Bool) = Ea |> Pa ! D01(AR)
mes a? D02(AR:Bool) = Eb |> Pb ! D02(AR)
mes b? D02(AR:Bool) = Ea |> Pa ! D02(AR)
mes a? D03() = Eb |> Pb ! D03()
mes b? D03() =  SCP:= false; AMA:= false; Ea |> Pa ! D03()

% WN stands for Warning Device Number. Since we have a type for
% components, E01 may carry the name of a warning device,
% instead of a number.

mes a? E01(WN:Component; RW,CD:Int) = Eb |> Pb ! E01(WN,RW,CD)
mes b? E01(WN:Component; RW,CD:Int) = Ea |> Pa ! E01(WN,RW,CD)

% In the following we see a case-in statement, with the mandatory
% 'otherwise' clause. This clause is redundant, as AM can have
% only the listed values, due to its type.

mes a? E02(WN:Component; AM:AMS; RW,CD:Int) =
   if WN == C
      then case AM in
           {occupied             : Occupied(RW,CD)
            complete_unoccupied  : Complete(RW,CD)
            incomplete_unoccupied: Incomplete(RW,CD)
            otherwise            : stall()}
      else Eb |> Pb ! E02(WN,AM,RW,CD)

% For each case above, we have defined a procedure. These are
% listed below.

% The main if-then-else clause of the 'Occupied' procedure
% tests the value of CD: if it is below zero, the process is
% stalled by means of the procedure 'stall', defined elsewhere.
% Ideally, it should be verified that SCD[RW] never has a
% negative value.

% The penultimate line of the following statement sends
% a central telegram. This is treated as all other
% telegrams. The component it is sent to is C, and the port
```

```
% on which it arrives is ‘right’.

% The telegram T01 carries the datum ‘self’. This denotes the
% name of the current component.

proc Occupied(RW,CD:Int) =
   CD:= SCD[RW];
   if CD >= 0
      then if C == 0
              then {SCP:= true; T01(false,occupied,0)}
              else if SCP | ~API
                      then T01(false,occupied,CD)
                      else C |> right !
                           T01(self,occupied,true,CD)
      else stall()

proc Complete(RW,CD:Int) =
   SCP:= true; T01(true,complete_unoccupied,CD)

% Within the following procedure we find again a test
% whether a value is above zero: if not, we stall.
% Also, we find a statement ‘stop APM’, which deactivates the
% cyclic time-out APM.

proc Incomplete(RW,CD:Int) =
   if AMA
      then if SCP >= 0
              then SCP:= (SCD[RW] == 0)
              else stall()
      else {stop APM; SCP:= false};
   T01(true,incomplete_unoccupied,CD)

% The following procedure is meant to handle uniformly certain
% flows. The first datum is to be matched with API. If there is
% such a match, API is flipped and a central telegram T01 is
% sent.

proc T01(b:Bool; AM:AMS; CD:Int) =
   if API == b then {API:= ~b;
                    C |> right ! T01(self,AM,false,CD)}

mes b? E02(WN:Component; AM:AMS; RW,CD:Int) =
   Ea |> Pa ! E02(WN,AM,RW,CD)

% The following flows (under certain conditions) execute
```

```
% the one-shot procedures SITa1 or SITb1. These put an
% internal telegram into the buffer if the corresponding
% boolean variables SITa1 or SITb1 are not already set.

mes a? E03() =
   if LCS | VTS
      then {SITa:= true; SITa1()}
      else Eb |> Pb ! E03()

mes b? E03() =
   if VTS
      then {SITb:= true; SITb1()}
      else Ea |> Pa ! E03()

mes a ? E04(TS:Bool) =
   if VTS
      then {{if SITb then SITb1()};
            if TS
               then TSUa:= true; T02(true)
               else TSUa:= false}
      else Eb |> Pb ! E04(TS)

mes b? E04(TS:Bool) =
   if VTS
      then {{if SITa then SITa1()};
            TSUb:= TS;
            {if ~TSUa then T02(TS)}}
      else if LCS
              then {{if SITa then SITa1()}; TSUb:= TS; T02(TS)}
              else Ea |> Pa ! E04(TS)

proc T02(TS:Bool) =
   if SOI == TS then {SOI:= ~TS; C |> right ! T02(self,TS)}

mes a? F01(TS,DL:Bool) = Eb |> Pb ! F01(TS,DL)
mes b? F01(TS,DL:Bool) = Ea |> Pa ! F01(TS,DL)

% The following denotes receival of a central telegram from the
% right. The port ‘right’ is treated as just another port.

mes right? U01() = if API then SCP:= true

% The following is an internal telegram, which we can see from
% the absence of a port name before the ‘?’. It is related to
% the cyclic time-out APM, although the name of the telegram
```

```
% does not guarantee this link.

mes ? APM() =
   if PAM | AMA
      then Ea |> Pa ! E01(C,RNR,0)
      else {stop APM;
            SCP:= false;
            T01(true,incomplete_unoccupied,0)}

% The next internal telegram is purely intented for the initial
% behaviour. It is called in the ‘initial’ clause. Note how the
% two INI-flows in the EURIS-specification are processed
% sequentially here.

mes ? INI() =
   if LCS | VTS then Eb |> Pb ! E03();
   if VTS then Ea |> Pa ! E03()

% One-shots are simulated as follows. For every one-shot
% variable X there is a boolean variable X, a procedure
% X() and a telegram X(). The variable X records whether
% the one-shot is set. The procedure only sets X if it is
% not already set, in which case it sends the internal
% telegram X. When this is processed, it unsets X and carries
% out the flow.
% Below we choose the names SITa1 and SITb1, because
% SITa and SITb are already used as ordinary boolean variables.

proc SITa1() =
   if ~SITa1 then {SITa1:= true; ! SITa1()}

mes ? SITa1() = SITa1:= false; Ea |> Pa ! E04(TSUb)

proc SITb1() =
   if ~SITb1 then {SITb1:= true; ! SITb1()}

mes ? SITb1() = SITb1:= false; Eb |> Pb ! E04(TSUa)

mes a? I00() = Eb |> Pb ! I00()
mes b? I00() = Ea |> Pa ! I00()
mes a? I01() = Eb |> Pb ! I01()
mes b? I01() = Ea |> Pa ! I01()
mes a? I02() = Eb |> Pb ! I02()
mes b? I02() = Ea |> Pa ! I02()
mes a? I03() = Eb |> Pb ! I03()
```

```
mes b? I03() = Ea |> Pa ! I03()
mes a? J01() = Eb |> Pb ! J01()
mes b? J01() = Ea |> Pa ! J01()
mes a? J02() = Eb |> Pb ! J02()
mes b? J02() = Ea |> Pa ! J02()

% The procedure ‘stall’ simply makes sure that no action is
% taken. This ensures that the panic clause is reached.

proc stall() = if 0 == (0 div 0) then skip

% The following describes the action to be undertaken should
% something unforeseen occur. We have chosen to send a telegram
% to the logistic level, but one may choose any other type of
% action. We use ‘Log’ to denote the logistic level, which is
% treated as just another component. ‘log’ is the only port
% that we assume present at this level. The panic-feature is
% strictly outside the scope of EURIS.

panic Log |> log ! P01(self)
```



this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d. Meij	13-sep-1996
checked		
approved		

simulation	0

NV NEDERLANDSE SPOORWEGEN

service :
dep :

warningdevice

13

scale AM EU

format A4R

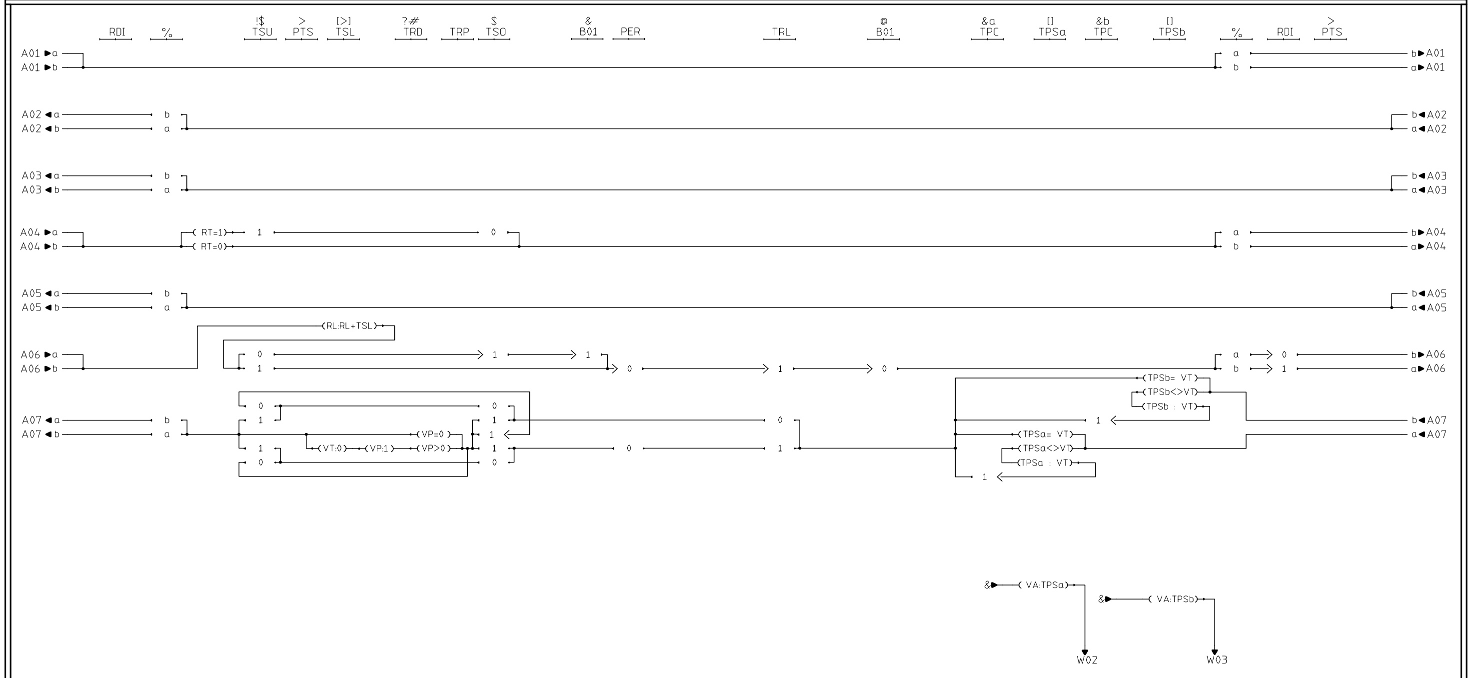
release

1

/tmp_mnt/home/euris/guide_top/unispec/lscdestop/warningdevice/warningdevice.sbk(01;21) Mon Dec 1 10:51:25 1997

```
%%%%%%%%%%%%%%%%%%%% 3. WARNING DEVICE LSC %%%%%%%%%%%%%%%%%%%%

LSC warning device

% List is the list of approach monitors that report to this
% warning device. We do not know in advance how many monitors
% to expect, so the type of the list is Component[Int]. As this
% makes Component[Int] into an infinite list, we also need to
% specify the part that we are interested in. For this we use
% the parameter 'Length'. For instance, if there are two approach
% monitors a1 and a2 (in that order in the central list), then we
% set these parameters as List = {(1,a1), (2,a2)} and Length = 2.

(List:Component[Int]; Length:Int) =

% The local variables: two multivars, represented as lists of
% booleans with components as indices, a timer and a boolean,
% used for a one-shot.

vars CAPI,CSOI:Bool[Component]; WDT:Timer; WDC:Bool

% There is no initial behaviour:

initial skip

% A warning device only has one port, namely 'right': on this it
% may receive telegrams T01 and T02.

% Value WDT returns the time that the timer has been active.
% If it is passive then 0 is returned.

mes right? T01(E:Component; AM:AMS; WT:Bool; CD:Int) =
   WDC();
   if AM == occupied
      then {CAPI[E]:= true;
            if {WT | (value WDT) >= CD} then E |> right ! U01()}
      else CAPI[E]:= false

mes right? T02(E:Component; TS:Bool) =
   WDC(); CSOI[E]:= ~TS

% The one-shot WDC is again split up into a procedure that
% decides whether to send the telegram and the message handling
% itself.
```

```
proc WDC() = if ~WDC then {WDC:= true; ! WDC()}

% The flow corresponding to the WDC one-shot involves testing
% the multivars. For this we have to proceed through List,
% from the components at indices from 1 to Length, and see
% whether one of the corresponding CAPI or CSOI-values has
% been set to true or not. To proceed through the list we
% use the local variable i. Variables TESTCAPI and TESTCSOI
% are used to store the disjunction (‘logical or’) of CAPI
% and CSOI respectively. TESTCAPI and TESTCSOI are explicitly
% set to their default values, according with good programming
% practice, even though it is not strictly necessary to do so.

mes ? WDC() =
   vars TESTCAPI,TESTCSOI:Bool; i:Int
   WDC:= false;
   i:= 1; TESTCAPI:= false; TESTCSOI:= false;
   while i <= Length do
      {TESTCAPI:= CAPI[List[i]] | TESTCAPI;
       TESTCSOI:= CSOI[List[i]] | TESTCSOI;
       i:= i+1};
   if TESTCAPI | TESTCSOI
      then {{if ~(active WDT) then start WDT};
            Inf |> inf ! W05(self,false)}
      else {stop WDT; Inf |> inf ! W05(self,true)}

% The panic clause: same as for approach monitors.

panic Log |> log ! P01(self)
```



this drawing replaces :		this drawing has been replaced with :	
	name	date	
drawn	D.A. v.d. Meij	3-jul-1996	
checked			
approved			

Track

scale: AM EU

format: A4R

release

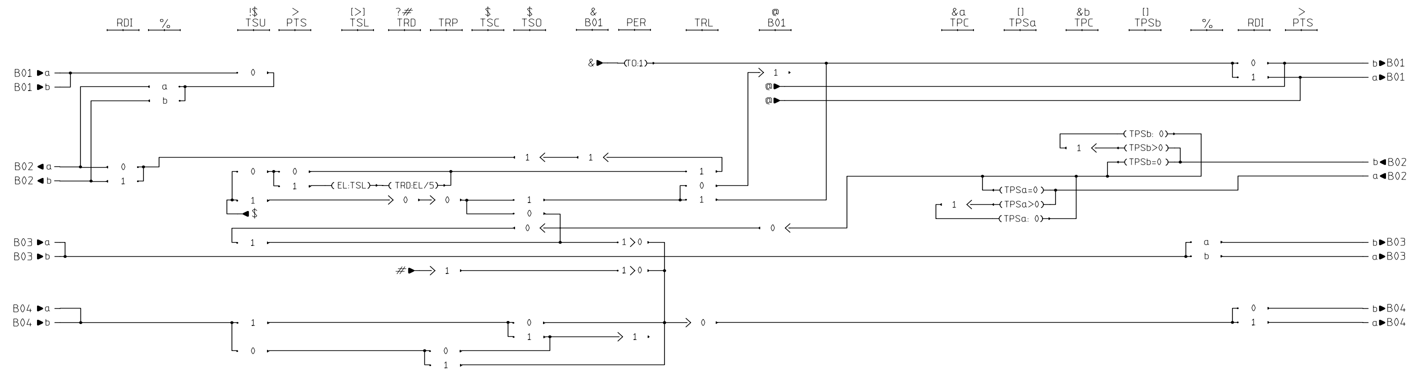
NV NEDERLANDSE SPOORWEGEN

service :
dep :

2

1

simulation	0



simulation	0

this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d. Meij	3-jul-1996
checked		
approved		

NV NEDERLANDSE SPOORWEGEN

service :
dep :

Track

2

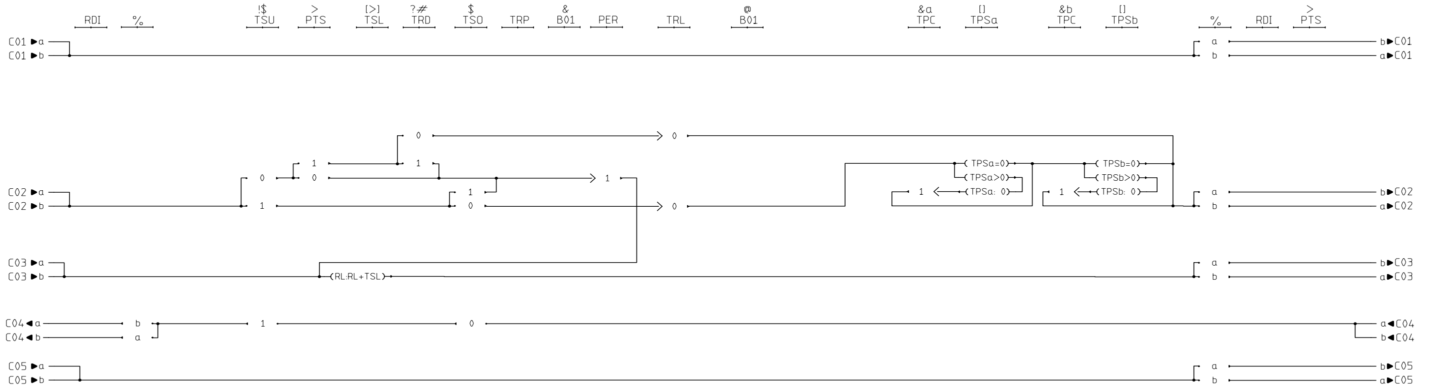
scale AM EU

format A4R

release

2

/tmp_mnt/home/euris/guide_top/unispec/lscdestop/track/track.sbk(02;112) Mon Dec 1 10:51:56 1997



this drawing replaces :		this drawing has been replaced with :
	name	date
drawn	D.A. v.d. Meij	26-mrt-1996
checked		
approved		

simulation	0

NV NEDERLANDSE SPOORWEGEN

service :

dep :

Track

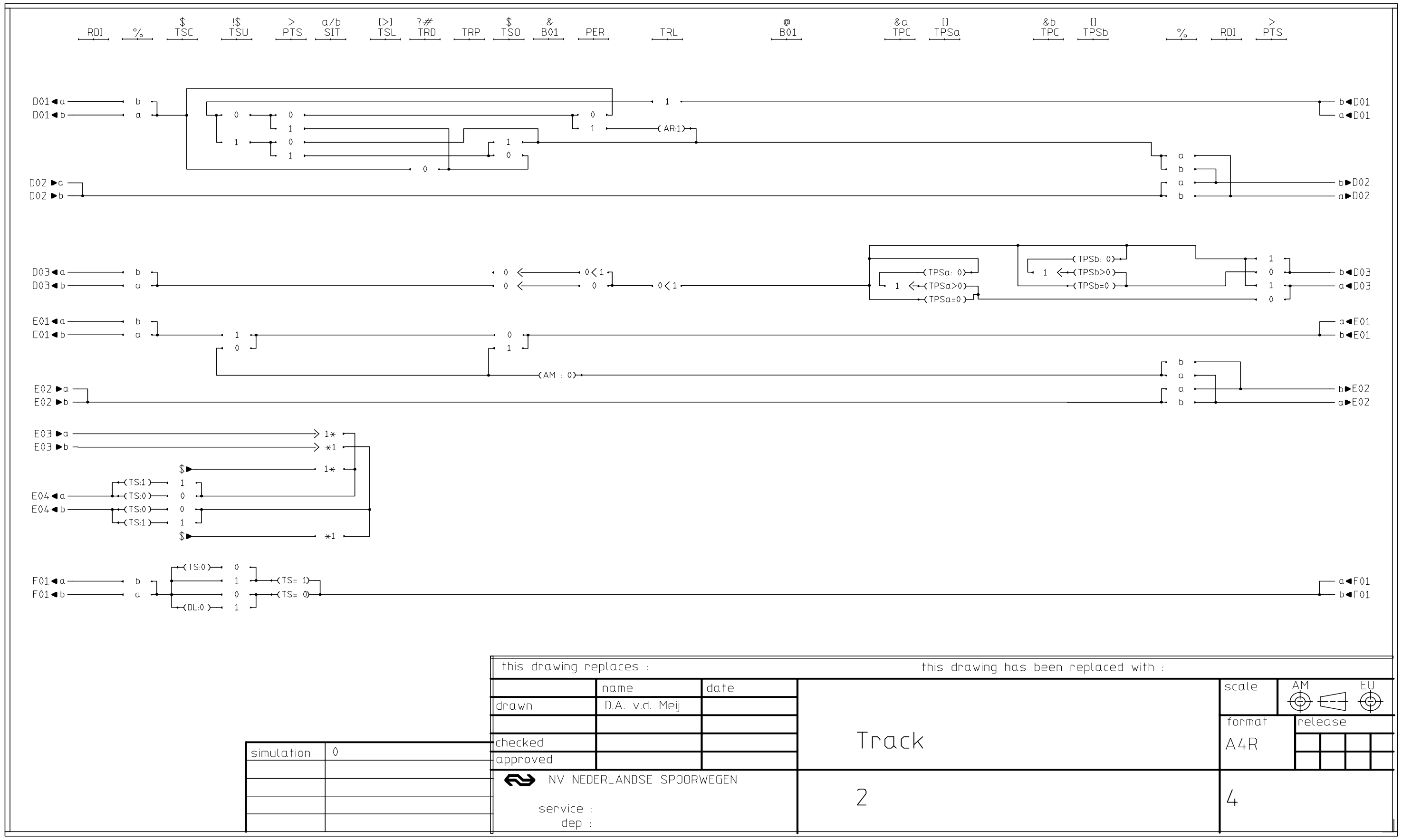
2

scale AM EU

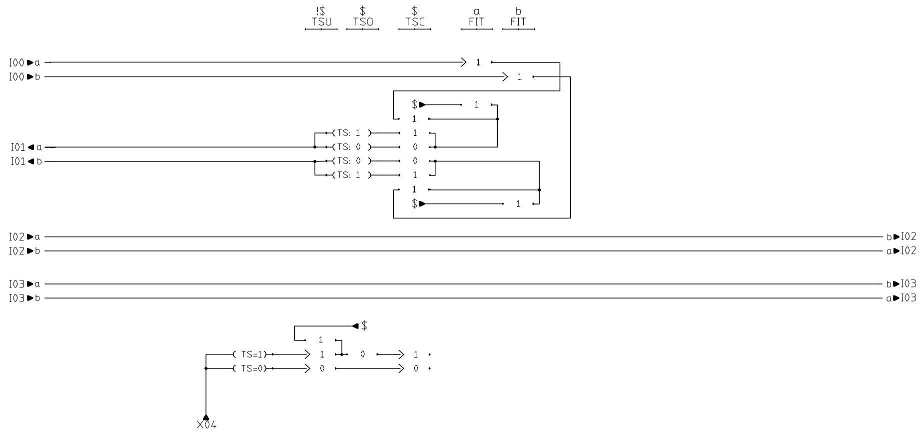
format A4R

release

3



/tmp_mnt/home/euris/guide_top/unispec/lscdestop/track/track.sbk(04;73) Mon Dec 1 10:52:32 1997



simulation	0	

this drawing replaces :

this drawing has been replaced with :

	name	date
drawn	D.A. v.d. Meij	18-jun-1996
checked		
approved		

track

scale AM EU

format A4R

release

NV NEDERLANDSE SPOORWEGEN

service :

dep :

2

5

```
%%%%%%%%%%%%%%%%%%%%% 4. TRACK LSC %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

LSC track

(Ea,           % Component connected to the a-port
 Eb:Component; % Component connected to the b-port
 Pa,           % Port on Ea connected to the a-port
 Pb:Port;      % Port on Eb connected to the b-port
 PTS:Bool;     % Passive Track Section
 TSL:Int)      % Track Section Length

=
vars
RDI,           % Route Direction Indicator
TSU,           % Track Section Unoccupied (toggle)
TSC,           % Track Section Checked unoccupied (toggle)
TRP,           % Track section route Release Permitted
TSO,           % Track Section logically Occupied (toggle)
B01,           % B01 telegram initiator (one-shot)
PER,           % Preceding Element Released
TRL,           % Track section Route Locking
               % Telegram initiator for Train Protection Control
               % (one-shot):
TPCa,          % (side a)
TPCb,          % (side b)
               % Indicator for track Section Information
               % Transmission:
SITa,          % (side a)
SITb,          % (side b)
               % Indicator for Fouling Information Transmission:
FITa,          % (side a)
FITb:Bool;     % (side b)
               % Train Protection Speed:
TPSa,          % (side a)
TPSb:Int;      % (side b)
TRD:Timeout;   % Track section route Release Delay timer
B01C:Cycler    % B01 telegram initiator (cyclic time-out)

% Initially, internal telegrams related to toggles ($-variables)
% are sent. An order was chosen here. It is possible that an
% external telegram is received before these are all put into
% the input buffer. If this does not happen, none of these
% telegrams have much of an effect, because they all test for
% the truth of some boolean LSC variables which by defalt are
% all false. Note that a certain order of putting the telegrams
```

```
% into the input buffer has been chosen. LARIS does not impose
% any particular order here.

initial ! TSU(); ! TSC(); ! TSO()

mes a? A01() = Eb |> Pb ! A01()
mes b? A01() = Ea |> Pa ! A01()
mes a? A02() = Eb |> Pb ! A02()
mes b? A02() = Ea |> Pa ! A02()
mes a? A03() = Eb |> Pb ! A03()
mes b? A03() = Ea |> Pa ! A03()
mes a? A04(RT:Route) = A04(Eb,Pb,RT)
mes b? A04(RT:Route) = A04(Ea,Pa,RT)

% The following case-in statement has a useful otherwise clause.
% If RT is either automatic_normal or drive_on_sight_normal, an
% error should occur: this is forced by the stall procedure.

proc A04(E:Component; p:Port; RT:Route) =
   case RT in
      {drive_on_sight: TSU(true); TSO(false);
                       E |> p ! A04(RT)
       normal        : E |> p ! A04(RT)
       otherwise     : stall()}

mes a? A05() = Eb |> Pb ! A05()
mes b? A05() = Ea |> Pa ! A05()

mes a? A06(RN:Bool; RL:Int) = A06(a,RN,RL)
mes b? A06(RN:Bool; RL:Int) = A06(b,RN,RL)

proc A06(p:Port; RN:Bool; RL:Int) =
   RL:= RL+TSL;
   {if ~TSU then {TSO(true); B01()}};
   PER:= false;
   TRL:= true;
   stop B01C;
   if p == a
      then {RDI:= false; Eb |> Pb ! A06(RN,RL)}
      else {RDI:= true;  Ea |> Pa ! A06(RN,RL)}

mes a? A07(VP,VT:Int) =
   {if TPSa =/= VT then {TPSa:= VT; TPCa()}};
   A07(Eb,Pb,VP,VT)
```

```
mes b? A07(VP,VT:Bool) =
   {if TPSb =/= VT then {TPSb:= VT; TPCb()}};
   A07(Ea,Pa,VP,VT)

proc A07(E:Component; p:Port; VP,VT:Int) =
   if ~TRL | ~PER
      then if TSO | ~TSU
              then {if TRL then TSO:= true};
                   {if VP >= 0
                       then if VP > 0
                               then E |> p ! A07(1,0)
                               else E |> p ! A07(VP,VT)
                       else stall()}
              else E |> p ! A07(VP,VT)}

mes a? B01(TO:Bool) = if ~TSU then Ea |> Pa ! B01(TO)
mes b? B01(TO:Bool) = if ~TSU then Eb |> Pb ! B01(TO)

mes a? B02() =
   if TPSa >= 0
      then {if TPSa > 0 then {TPCa(); TPSa:= 0}};
           B02()
      else stall()

mes b? B02() =
   if TPSb >= 0
      then {if TPSb > 0 then {TPCb(); TPSb:= 0}};
           B02()
      else stall()

proc B02() = stop B01C; TSO(false); if TSU then PER()

mes a? B03() = Eb |> Pb ! B03()
mes b? B03() = Ea |> Pa ! B03()

mes a? B04() = B04()
mes b? B04() = B04()

proc B04() =
   if (TSU ^ TSO) | (~TSU ^ ~TRP)
      then PER:= true
      else {TRL:= false;
            if RDI
               then Ea |> Pa ! B04()
               else Eb |> Pb ! B04()}
```

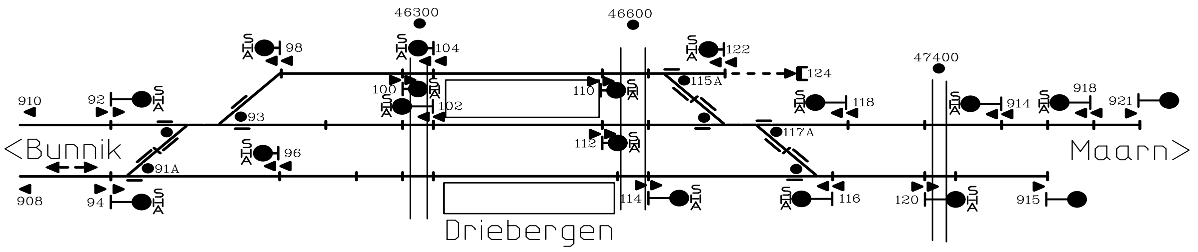
```
mes a? C01() = Eb |> Pb ! C01()
mes b? C01() = Ea |> Pa ! C01()

mes a? C02(RL:Int) = C02(Eb,Pb,RL)
mes b? C02(RL:Int) = C02(Ea,Pa,RL)

% The following procedure's first two parameters tell us
% where to send a telegram, should this be called for. We
% see above that if C02 is recieved on the a-port the
% telegram should be sent along the b-port, and vice-versa.

proc C02(E:Component; p:Port; RL:Int) =
   if (TSU ^ ~TSO) | (~TSU ^ PTS ^ ~(active TRD))
      then TRL:= false;
           {if TPSa >= 0 ^ TSU
               then {if TPSa > 0 then {TPSa:= 0; TPCa()}}
               else stall()};
           {if TPSb >= 0 ^ TSU
               then {if TPSb > 0 then {TPSb:= 0; TPCb()}}
               else stall()};
           E |> p ! C02(RL)
      else {{if TSO then RL:= RL+TSL};
            E |> p ! C03(RL)}

mes a? C03(RL:Int) = C03(Eb,Pb,RL)
mes b? C03(RL:Int) = C03(Ea,Pa,RL)

% The same trick as used in the C02 procedure is used here.

proc C03(E:Component; p:Port; RL:Int) =
   if RL >= 0
      then {{if RL>0 | (TSO ^ (~PTS | active TRD))
                then RL:= RL+TSL};
           E |> p ! C02(RL)}
      else stall()

mes a? C04() = if TSU then Eb |> Pb ! C04()
mes b? C04() = if TSU then Ea |> Pa ! C04()
mes a? C05() = Eb |> Pb ! C05()
mes b? C05() = Ea |> Pa ! C05()

mes a? D01(AR:Bool) = D01(Eb,Ea,Pb,Pa,AR)
mes b? D01(AR:Bool) = D01(Ea,Eb,Pa,Pb,AR)
```

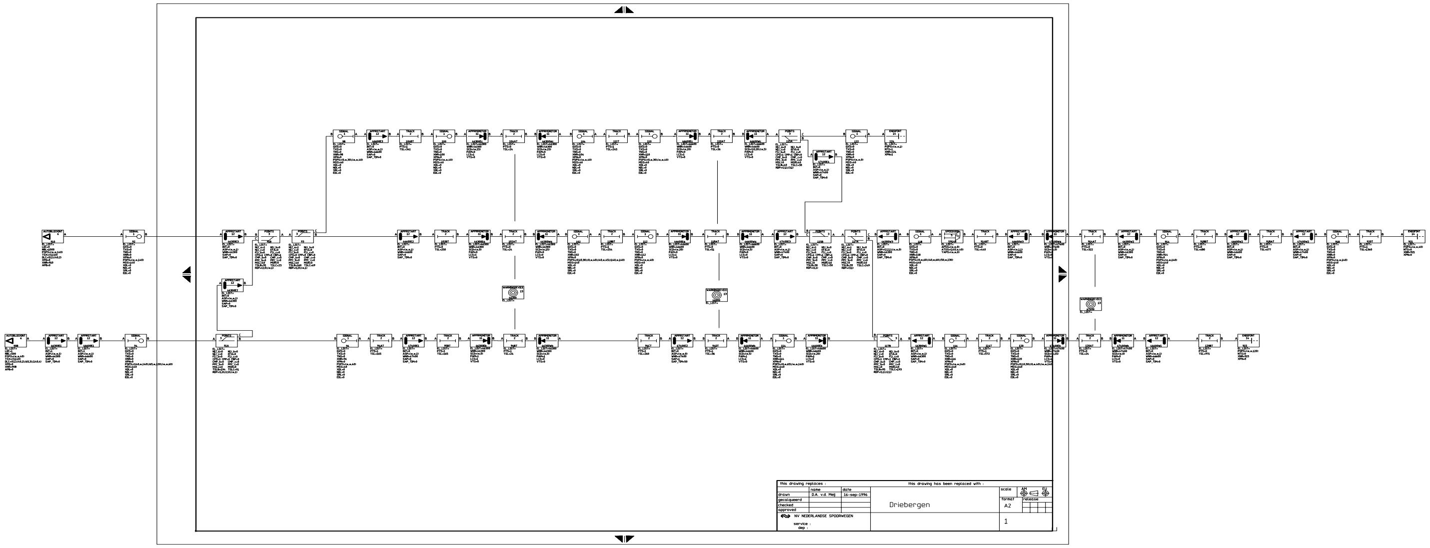
```
% The following procedure carries two component names and two
% port names as parameters. If a telegram is to be sent along
% (say from the a- to the b-port) then it is sent to the
% ‘Forward’ component, which receives it on port ‘forward’.
% Otherwise, if the telegram is to be returned, it is sent to
% ‘Back’, which receives it on the port ‘back’.

% The nested if-the-else structure of the D01-flow in the EURIS
% specification has been simplified somewhat by the use of
% boolean expressions.

proc D01(Forward,Back:Component; forward,back:Port; AR:Bool) =
   if TRL
      then {if (TSU ^ (~PTS | TSO)) | (~PTS ^ PER)
               then {if ~TSU then AR:= true};
                    Back |> back ! D02(AR)
               else if ~PTS | (TSU ^ TSO) | ~(active TRD)
                       then Forward |> forward ! D01(AR)}

mes a? D02(AR:Bool) = Eb |> Pb ! D02(AR)
mes b? D02(AR:Bool) = Ea |> Pa ! D02(AR)

mes a? D03() =
   if ~PTS then {if TPSa >= 0
                    then {{if TPSa > 0 then TPCa()}; TPSa:= 0}
                    else stall()};
   D03(Eb,Pb)

mes b? D03() =
   if ~PTS then {if TPSb >= 0
                    then {{if TPSb > 0 then TPCb()}; TPSb:= 0}
                    else stall()};
   D03(Ea,Pa)

proc D03(E:Component; p:Port) =
   if TRL
   then {TRL:= false; TSO(false);
         if PER
            then PER:= false
            else E |> p ! D03()}

mes a? E01(WN:Component; RW,CD:Int) =
   E01(Eb,Ea,Pb,Pa,WN,RW,CD)
mes b? E01(WN:Component; RW,CD:Int) =
   E01(Ea,Eb,Pa,Pb,WN,RW,CD)
```

```
proc E01(Forward,Back:Component; forward,back:Port;
         WN:Component; RW,CD:Int) =
   if TSO | ~TSU
      then Back |> back ! E02(WN,occupied,RW,CD)
      else Forward |> forward ! E01(WN,RW,CD)

mes a? E02(WN:Component; AM:AMS; RW,CD:Int) =
   Eb |> Pb ! E02(WN,AM,RW,CD)
mes b? E02(WN:Component; AM:AMS; RW,CD:Int) =
   Ea |> Pa ! E02(WN,AM,RW,CD)
mes a? E03() = SITa:= true; Ea |> Pa ! E04(TSC)
mes b? E03() = SITb:= true; Eb |> Pb ! E04(TSC)
mes a? F01(TS,DL:Bool) = F01(Eb,Pb,TS,DL)
mes a? F01(TS,DL:Bool) = F01(Ea,Pa,TS,DL)

proc F01(E:Component; p:Port; TS,DL:Bool) =
   if TS
      then TS:= TSU
      else if TSU then DL:= false;
   E |> p ! F01(TS,DL)

mes a? I01(TS:Bool) =
   FITa:= true;
   if TSC then Ea |> Pa ! I01(true)

mes b? I01(TS:Bool) =
   FITb:= true;
   if TSC then Eb |> Pb ! I01(true)

mes a? I02() = Eb |> Pb ! I02()
mes b? I02() = Ea |> Pa ! I02()
mes a? I03() = Eb |> Pb ! I03()
mes b? I03() = Ea |> Pa ! I03()

% The following flow deals with a telegram X04 received from
% the infrastructure level: such telegrams are received on
% port ‘inf’.

mes inf? X04(TS:Bool) =
   TSU(TS);
   if TS
      then {if TSO then TSC(true)}
      else {{if TRL then TSO(true)}; TSC(true)}
```

```
% One-shots:

proc TPCa() = if ~TPCa then {TPCa:= true; ! TPCa()}
proc TPCb() = if ~TPCb then {TPCb:= true; ! TPCb()}
proc B01()  = if ~B01  then {B01:= true; ! B01()}

mes ? TPCa() = Inf |> inf ! W02(self,TPSa)
mes ? TPCb() = Inf |> inf ! W03(self,TPSb)

mes ? B01() = if RDI
                 then Ea |> Pa ! B01(true)
                 else Eb |> Pb ! B01(true)

% A toggle X is modelled by a boolean variable X,
% a procedure X(b:Bool), and a telegram X(), which
% concatenates all the flows of X (if there is more
% than one) in a specific order. Again, random orders
% of such flows are not available in LARIS.

% The toggle X is set by calling X(true) and reset by
% calling X(false). The procedure checks whether this
% means that the value has changed. If it has, then
% the X-telegram is put into the input buffer.

proc TSU(b:Bool) = if TSU =/= b then {TSU:= b; ! TSU()}
proc TSC(b:Bool) = if TSC =/= b then {TSC:= b; ! TSC()}
proc TSO(b:Bool) = if TSO =/= b then {TSO:= b; ! TSO()}

% Toggle-flows:

% In the first flow, in case ~TSU ^ PTS, we see that a
% time-out is activated, with delay TSL div 5. In the EURIS-
% specification, a telegram-field EL was used to‘carry’
% the value of TSL to TRD, but this is unnecessary here.

mes ? TSU() =
   if TSU
      then stop TRD; TRP:= false;
           if TSO
              then if TRL
                      then if RDI
                              then Ea |> Pa ! B01(false)
                              else Eb |> Pb ! B01(false)
                      else stop B01C
              else PER()
```

```
      else {{if PTS then >># TRD (TSL div 5) ! TRD()};
            if TRL
               then {B01();
                     if RDI
                        then Eb |> Pb ! B02()
                        else Ea |> Pa ! B01()}}

proc PER() =
   if PER then {PER:= false; TRL:= false;
                if RDI
                   then Ea |> Pa ! B04()
                   else Eb |> Pb ! B04()}

% The following flow concatenates the four flows of TSC.

mes ? TSC() =
   if SITa then Ea |> Pa ! E04(TSC);
   if SITb then Eb |> Pb ! E04(TSC);
   if FITa then Ea |> Pa ! I01(TSC);
   if FITb then Eb |> Pb ! I01(TSC)

mes ? TSO() = if TSU ^ ~TSO then TSC(true)

% Time-out telegram:

mes ? TRD() = TRP:= true; PER()

% Cyclic time-out telegram:

mes ? B01C() =
   Ea |> Pa ! B01(false); Eb |> Pb ! B01(false)

% Stalling procedure:

proc stall() = if 0 == (0 div 0) then skip

% Panic clause:

panic Log |> log ! P01(self)
```





/tmp_mnt/home/euris/guide_top/unispec/lscvaltop/driebergen/driebergen.sbk(01;129)   Mon Dec 1 15:20:43 1997

```
%%%%%%%%%%%%%% 5. System Declaration %%%%%%%%%%%%%%%%%%%%%

System
Part_of_Driebergen % The name of the system
=

% First, we list the components that are not in the subsystem
% of Driebergen being modelled here, but to which reference is
% made nevertheless, for instance in the parameters of the
% components which we do specify.  If it is not known which
% component, for instance, is connected to the a-port of a
% component in our system, we must give a name to this component
% nonetheless.

% It is not necessary (and not allowed) to mention the names Inf
% and Log of the infrastructure and logistic level components
% explicitly in this set: it is assumed that these are always
% present.

   External components = {T94C, T95B, T102B,
                          S100, S102, S104}

% Next, we list the port names on these external components to
% which we may send telegrams. In this particular case, we could
% have left these extra port names out of the specification,
% because they also exist as port names on approach monitors.

% It is unnecessary (and not allowed) to mention the port names
% left, right, inf, and log. These are assumed to be present in
% any system.

   External ports = {a,b}

% Next we have a list of bindings:

   apprmonitor Am46300Ea(T102B, T102A, Wd46300, b, a,
                         false, true, false, {}:Int[Int])

% The last argument of the approach monitor {(*,15)}:Int[Int]}
% represents an array (of type Int[Int]) with all values equal
% to 15.

   apprmonitor Am46300Wa(S102, T102A, Wd46300, b, b,
                         false, false, false, {(*,15)}:Int[Int])
   apprmonitor Am46300Eb(T95B, T94B, Wd46300, b, a,
```

```
                         true, true, false, {}:Int[Int])
   apprmonitor Am46300Wb(T94C, T94B, Wd46300, a, b,
                         true, false, false, {}:Int[Int])
   apprmonitor Am46300Ec(S100, T104A, Wd46300, b, a,
                         false, true, false, {(*,15)}:Int[Int])
   apprmonitor Am46300Wc(S104, T104A, Wd46300, b, b,
                         false, false, false, {(*,15)}:Int[Int])

% A Warning Device needs two arguments: a list of components,
% and an integer (representing the length of the list).

   warningdevice Wd46300(
      {(1,Am46300Ea),(2,Am46300Wa),(3,Am46300Eb),
       (4,Am46300Wb),(5,Am46300Ec),(6,Am46300Wc)}:Component[Int],
       6)

% Three tracks:

   track T104A(Am46300Ec, Am46300Wc, b, b, false, 24)
   track T102A(Am46300Ea, Am46300Wa, b, b, false, 24)
   track T94B(Am46300Eb, Am46300Wb, b, b, true, 24)
```

7 Conclusions and future work

EURIS is a specification method for interlocking logics. The object-oriented approach of EURIS is a strong point in its favour, and its graphical format allows for compact specifications. However, this compactness can obstruct the clarity of EURIS specifications, especially because in practice EURIS is often treated more like a programming language than as a specification language.

In this document we have provided EURIS with a symbolic variant, LARIS 1.0 (referred to as plain 'LARIS' in most of the document). LARIS 1.0 may be viewed as an intermediate representation between IDEAL [17] and established specification languages for which tools such as simulators and model-checkers exist, such as PROMELA [15], LOTOS [21], and SDL [9]. Alternatively, and this is the view that the authors take, LARIS 1.0 may be used as the prime specification language. EURIS is then simply one way to write LARIS specifications.

This situation is depicted in Figure 11. Specifications (which initially exist only as ideas) can be written either in EURIS (preferably using the simulator, which forces rigorous syntax) or in LARIS. If a specification is given in EURIS using the simulator, then this specification outputs IDEAL code. Future work could be on a compiler from IDEAL to LARIS. Sections 3.5 and 3.6 contain ideas about how this could be achieved.

Ideally, tool support would have to be developed for LARIS 1.0, for example to check its static semantics, to allow simulations, or to perform model-checking. Instead of developing such tools directly, it would seem wise to instead translate LARIS 1.0 into languages for which such tools already exists. The translations should be such that any behaviour of the translation is also one of the original, and if some behaviour of the original does not exist in the translation, then this should be motivated explicitly.

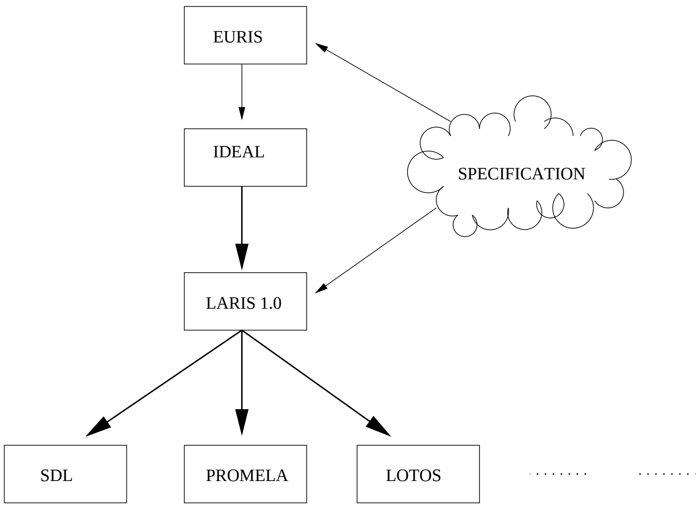


Figure 11: Future development with LARIS 1.0.

Excluding certain traces (sequences of actions) from the system, which the semantics we have defined allows, seems necessary in any case. In the semantics we defined, it is perfectly possible (for instance) for traces to consist purely of time-steps. This would correspond to a totally inactive system. Another, related, possibility is that one component is starved, that is, not allowed any action. In the latter case, it could still be that other components act normally. Fairness constraints are needed to rule out such traces. Preferably at the moment of constructing the state-space, or else one could state the formal requirements in a conditional way: 'if such-and-such conditions hold, then these other requirements also hold'. These extra conditions would then of course become part of the requirements of an implementation.

A problem of specifications written in LARIS 1.0 is the size of the generated state-space. One could argue that this is due to the requirement that EURIS specifications, which themselves generate huge state-spaces, should be expressible in LARIS 1.0. To reduce the state-space, it seems necessary to place a severe bound on the size of the buffers used, both those internal to components, and those of channels. Work of relevance here may be found in [4], where a number of bounds are explored.

When doing verifications, one should be aware that LARIS does not automatically assume that the system is closed. Thus external telegrams may arrive at a component, even when it is not specified where these come from. If one knows that such telegrams can only come from components inside the specification, one should restrict the occurrence of such lone receive actions, by encapsulating these.

We believe the language, as presented, is rich enough to express all EURIS constructs. However, the translation is sometimes cumbersome. For instance, toggles are not represented by variables declared as `X : $`, but are simulated, as demonstrated in Section 3.6. One could allow to define a macro for toggles. A translation to proper LARIS 1.0 is then required. A similar remark holds for one-shots.

The choice of defining procedures as call-by-value can feel as a real restriction. An example is in the use of procedures when translating EURIS boxes into LARIS. Such boxes are modules, where telegram-fields may be altered. To capture such alterations in the corresponding LARIS 1.0 procedure, one would need to declare a global LSC variable and change that variable. This is unsatisfactory. So perhaps call-by-reference is desired. This would lead to a redefinition of the static and operational semantics, and would also make these slightly more complex.

References

[1] T. Basten, R. Bol, and M. Voorhoeve. Simulating and analyzing railway interlockings in ExSpect. *IEEE Parallel & Distributed Technology, Systems & Applications*, 3(3):50–62, 1995.

[2] J.W.F.M. Bejaars. Prozeß Ablauf Pläne: introductie tot een specificatiemethode. Technical Report 1992/JBe/3, IB ETS-T&K, 1992. In Dutch.

[3] J. Berger, P. Middelraad, and A.J. Smith. EURIS, European Railway Interlocking Specification. UIC, Commission 7A/16, May 1992.

[4] J.A. Bergstra, W.J. Fokkink, W.M.T. Mennen, and S.F.M. van Vlijmen. *Spoorweglogica via EURIS*, volume 22 of *Questiones Infinitae*. Utrecht University, 1998. In Dutch.

[5] R.N. Bol, J.W.C. Koorn, L.H. Oei, and S.F.M. van Vlijmen. Syntax and static semantics of the interlocking design and application language. Technical Report P9422, University of Amsterdam, 1994.

[6] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping, an Algebraic Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[7] F.J. van Dijk, W.J. Fokkink, G.P. Kolk, P.H.J. van de Ven, and S.F.M. van Vlijmen. EURIS, a specification method for distributed interlockings. In

Proceedings of the 17th Conference on Computer Safety, Reliability and Security (SAFECOMP'98), volume 1516 of *Lecture Notes in Computer Science*, pages 296–305. Springer-Verlag, 1998.

[8] C. Eisner. Using symbolic model checking to verify the railway stations of Hoorn–Kersenboogerd and Heerhugowaard. In *Proceedings of the 10th IFIP Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 1999.

[9] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL – Formal Object-Oriented Language for Communicating Systems.* Prentice Hall, 1997.

[10] W.J. Fokkink. Safety criteria for the vital processor interlocking at Hoorn–Kersenboogerd. In *Proceedings of the 5th Conference on Computers in Railways (COMPRAIL'96)*, pages 101–110. Computational Mechanics Publications, 1996.

[11] W.J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science, an EATCS Series. Springer-Verlag, 1999.

[12] R.J. van Glabbeek. The linear time - branching time spectrum II: the semantics of sequential processes with silent moves. In E. Best, editor, *Proceedings of the 4th Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1993.

[13] J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn–Kersenboogerd. In *Proceedings of the 10th IEEE Conference on Computer Assurance (COMPASS'95)*, pages 57–68. IEEE, 1995.

[14] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[15] G.J. Holzmann. *Design and Verification of Protocols.* Prentice Hall, Englewood Cliffs, NJ 07632, 1990.

[16] NS Rail Infrabeheer. UNISPEC. Version November 1997.

[17] F. Makkinga. IDEAL: Interlocking Design and Application Language Guide and Reference. Version 0.3, January 1993.

[18] F. Makkinga and F. van Dijk. Euris-simulation tutorial reference. Technical report, Holland Railconsult, 1995. Versie 3.0.

[19] L.H. Oei. Pruning the search tree of interlocking design and application language operational semantics. Technical Report P9418, University of Amsterdam, 1994.

[20] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[21] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. Elsevier Science Publishers B.V., 1989.

Index

&, 7
$, 8
#, 8
>>#, 8, 22
@, 9, 23
?#, 9
$\wedge$, 16
$\sim$, 16
$|$, 16
$+$, 16
$-$, 16
*, 16, 20
$==$, 16
$=/=$, 16
$<$, 16
$>$, 16
$<=$, 16
$>=$, 16
;, 25
$\vec{d}$, 48
$[\![\cdot]\!]$, 67
$\mathbb{N}$, 68
$\uparrow$, 69
ξ, 72
$\cdot$, 73
[ ], 73
π_ι, 74
σ_0, 72
σ_ι, 74
$\surd$, 73
τ, 74
γ, 85
$\|$, 86
∂_H, 86

Act_C, 74
action, 72
  Σ-, 84
  silent, 74
active, 21
ambiguity, 25
approach monitor, 37
argument-correct, 62
assignment, 23
  $\mathcal{T}$-, 69
    extension of, 69
associativity, 43
  left-, 43
  right-, 43

behaviour, 27
  initial, 21
Bindings, 64
Bool, 16, 59
boolean, 16
Bound, 64
brace, 25
broadcast, 40

$\mathbb{C}_{(C,D)}$, 85
call-by-reference, 24
call-by-value, 24
case-in, 24
case, 25
central list, 6, 7, 18
Channel, 83
communication, 12, 18
  asynchronous, 12
  synchronous, 40
commutativity, 85
Component, 17
component, 3
  external, 29
$\mathsf{Component}_\Sigma$, 56
corruption, 42
Cycler, 22, 24
cycler, 22

declaration-correct, 57
  behaviour, 58
  LSC, 58
  specification signature, 59
default, 67
default value, 18
div, 16

dom, 58
domain, 58

encapsulation, 86
EURIS, 3
EURIS_{DTO}, 11
execution condition, 5
expression, 16
    $(\Sigma, \mathcal{T})$-, 59
    $\mathcal{T}$-, 59
$\texttt{Expr}(\Sigma, \mathcal{T})$, 59
$\texttt{Expr}(\mathcal{T})$, 59
ExtSend, 61
ExtTel_Λ, 73
ExtTel_Σ, 73
ExtType, 61

fairness constraint, 122
false, 16, 59
flow, 5
function
    communication, 85

IDEAL, 1
if-then-else, 24
if-then, 24
if, 24
index, 19
    Σ-, 56
Inf, 17
inf, 17
infrastructure, 3
initial, 28
input buffer, 13
Int, 16
integer, 16
interlocking, 3
internal, 73
IntTel_Λ, 73

keyword, 23, 48

LARIS, 11
left, 6, 17
level, 17
Log, 17, 29
log, 17
logistic level, 3
LSC, 9, 15
lsc_Σ, 65

$\mathcal{M}_C$, 72
    actions of, 74
    state of, 73
        initial, 74
    TSS, 76
$\mathcal{M}_\Sigma$, 86
mes ?, 27
message handler, 13
mod, 16
multivar, 38

Name
    Σ-argument typed, 61
    Σ-available, 57
    Σ-reserved, 57
    Σ-typed, 57
    typed, 50
        argument, 50

one-shot, 7, 29
otherwise, 25

panic, 25, 28
parallel composition, 86
parameter
    in a procedure, 27
    LSC, 26
Port, 17
port, 6
    central, 6
    external, 29
    route, 6
Port_Σ, 56
procedure call, 24
procedure definition, 27
process graph, 67, 72

queue, 73

right, 6, 18

s_ι, 74

`self`, 28
semantics
  operational, 67
  static, 42, 63
separator, 25
sequential composition, 25
Set, 58
Sig, 52
  `Behaviour`, 54
  `Body`, 53
  flow
    external, 53
    internal, 53
  `LSC`, 54
  procedure definition, 54
  `Specification`, 55
  `System`, 55
  `TypeDef`, 55
signature, 49
  behaviour, 51
  body, 50
  internal, 50
  LSC, 51
  procedure, 50
  specification, 52
  system, 52
  telegram
    external, 50
    internal, 50
  type, 52
`skip`, 24
sort, 43
`start`, 21, 24
state, 72
  initial, 72
`Statement`
  basic, 61
  complex, 61
statement, 13, 23
  $(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$-, 62
  $(\mathcal{T}, \mathcal{V})$-, 62
`Statement`$(\Sigma, \Lambda, \mathcal{T}, \mathcal{V})$, 62
`Statement`$(\mathcal{T}, \mathcal{V})$, 62
Static Semantical Correctness, 63, 64
`stop`, 21, 24
substitution
  on expressions, 65
  on statements, 66
  $(\Sigma, \mathcal{T}, \mathcal{T}')$-, 65
syntax, 42
  context-free, 43
  lexical, 43
Syntax Definition Formalism, 42
`System`, 29

$\mathcal{T}_\iota$, 74
$\mathbb{T}_C$, 85
$\mathbb{T}_\Sigma$, 86
`t`, 74
telegram, 6
  -field, 6
  central, 7, 18
    left, 17
    right, 18
  external, 24
  internal, 7, 13, 24
  neighbour, 18
  route, 7
  table, 6
Tel_Λ, 73
terminator, 25
time slice, 7, 41
time-out, 8, 22
  cyclic, 9, 22
  dynamic, 9
`Timeout`, 22, 24
`Timer`, 20, 24
timer, 8, 21
toggle, 8, 32
trace, 122
transition, 72
transition rule, 75
  proof of a, 75
Transition System Specification, 75
`true`, 16, 59
type, 16
  array, 19
  clock, 20
  empty, 17

enumerated, 17
Σ-, 56
function, 57
list, 57
Σ-array, 56
Σ-basic, 56
Σ-data, 56
type-correct, 56, 57
$\mathsf{type}_{\Sigma,\mathcal{T}}$, 59
$\mathsf{type}_{\mathcal{T}}$, 59

UniSpec, 4

$\mathcal{V}_\iota$, 74
`value`, 21
variable
in a procedure, 27
input, 7
internal, 7
LSC, 27
name, 23
`vars`, 27
version symbol, 7

warning device, 37
`while-do`, 25
wild-card, 20