A Compositional Petri Net Semantics for SDL*

Hans Fleischhack
Fachbereich Informatik,
Universität Oldenburg,
Postfach 2503, D-26111 Oldenburg,
Fleischhack@informatik.uni-oldenburg.de

Bernd Grahlmann
Institut für Informatik,
Universität Hildesheim,
Marienburger Platz 22, D-31141 Hildesheim,
bernd@informatik.uni-hildesheim.de

Abstract

In this paper a high-level Petri net semantics for SDL (Specification and Description Language) is presented. Emphasis is laid on the modelling of dynamic creation and termination of processes and of procedures - features, which are, for instance, essential for typical client-server systems.

In addition to presenting the main ideas as well as the details of the semantics, we show that we are able to use 'state of the art' verification techniques by basing our approach on M-nets (an algebra of high-level Petri nets). Therefore, we verify our running example, an ARQ (**A**utomatic **R**epeat re**Q**uest) communication protocol, using the verification component of the PEP tool which presently includes partial order based model checking and algorithms based on linear programming as well as interfaces to other verification packages such as INA, SMV and SPIN providing reduction algorithms based on BDDs, on the stubborn set or sleep set method, and on symmetries. Moreover, we give examples how the compositional nature of the M-net semantics can be used to solve the 'state explosion' problem, and how interactive verification may extend the verification possibilities.

Keywords: ARQ protocol, Compositionality, Concurrency, Dynamic Processes, PEP tool, Petri Net Semantics, Procedures, SDL, Verification.

1 Introduction

SDL (**S**pecification and **D**escription **L**anguage [7]) is a real world **F**ormal **D**escription **T**echnique (FDT) not just an academic toy language. Its graphical representation and its expressiveness have been good reasons for the worldwide acceptance. In particular, SDL is well-suited for the design of client-server systems.

Although a lot of effort has been spent to specify the semantics of SDL in the norm Z100 [7], on-going discussions (for instance in the SDL mailing list) show that different interpretations of crucial points (such as the atomicity of SDL transitions) still exist. This is not surprising considering the amount of several hundred pages of technical descriptions contained in the Z100 and its appendices.

*This work has been supported by the DFG (German Research Foundation), the HCM (Human Capital and Mobility) Cooperation Network EXPRESS and the Procope project POEM.

Another drawback is the lack of appropriate tool support for fundamental tasks such as simulation with a graphical interface and verification[1]. Even good professional tools (such as SDT or GEODE) could be improved.

A typical solution for these problems is to give a formal semantics in terms of a formal model to SDL programs. Most of the existing approaches have used Petri nets (PN) because these are executable (with a precise semantics) providing at the same time simulation and verification possibilities.

In order to distinguish our approach from 'similar' approaches we briefly summarise the main criteria which are important for a successful application to client-server systems.

Of course, the relevant parts of SDL must be covered. The abstract data type concept of SDL is nice, but if the main interest is verification it is opportune to support 'only' small data types. On the one hand, large (or even infinite) data types are impossible to handle even with 'state of the art' verification methods due to the well-known 'state explosion' problem. On the other hand, usually, large data types are not necessary for the verification of typical system properties: For instance, the type of data transmitted to a printer in a typical client-server system does not influence the interesting properties.

Procedures and dynamic creation and termination of processes, however, are very important features for the modelling of client-server systems. The usage of procedures may enhance the readability of large SDL specifications by structuring the system description and by obeying the rule 'write things only once'. Dynamic creation and termination of processes is even more essential. Typically, systems are not static. In particular, the number of concurrent client tasks may change dynamically. Moreover, test scenarios, specifying (or reducing the non-determinism of) the dynamic structure of the system explicitly, may be efficient.

Very often, the importance of small differences between different PN models are underestimated. In particular, the best available verification methods should be applicable and (what is quite important) should already be supported by tools. Depending on the individual PN and the property under consideration analysis or model checking algorithms which are based on partial orders [11], BDDs [8], stubborn sets [28] or sleep sets [21], symmetrical reduction [27] or linear programming [25] may be the best choice.

Unfortunately, the complexity of SDL programs can grow beyond the limits of each of these (fully automatic) verification methods. User interaction or user guidance seem to be the only solution and should therefore be supported.

One way to reduce the state space is restricting the behaviour of the environment by considering different test scenarios (one by one). This should preferably be supported on the top most level (i.e. the SDL level).

Neglecting the influence of some parts of an SDL system is an orthogonal solution. For instance, the user may decide that a certain property does not depend on some of the SDL variables (e.g., x). In this case, a condition '$x = 0$' has the same effect as 'true'. This kind of abstraction is particularly interesting for partial order methods.

In contrast to these two methods, compositional verification is able to handle large

[1]If we use the word 'verification' we do not speak about 'simple' syntax checks or type checking for, e.g., variables, channels or signal routes. Instead, we mean full verification of SDL properties such as 'Is it always possible to reach a state (of the SDL system) in which process π_1 is in state a and the local variable x of process π_2 has the value b and the input queue of process π_3 is empty?'.

systems by analysing the different components (which are relatively small and thus easy to verify) individually and by deriving properties of the whole system from properties of its components. This presumes that the semantics of an SDL system is given in a completely compositional way.

We have chosen to translate SDL specifications into an algebra of modular high-level (HL) Petri nets, called M-nets (for modular multilabelled Petri nets) [5]. Due to the rich set of composition and communication operators we are able to define a semantic operator on M-nets for each syntactic operator of the SDL language. Thus, a fully compositional syntax-directed – and hence transparent – semantic mapping is defined.

Using this approach a property of an SDL system is checked as follows (cf. Fig. 1):

1. The M-net semantics of the SDL system is calculated.
2. The M-net is unfolded into a Petri box (a special low-level net) [3].
3. The SDL property is transformed into a net property.
4. The net property is checked against the Petri box.
5. The result is transformed back to the SDL level.

This approach has already been implemented in the **MOBY** tool (**M**odelling **o**f Dis-tri**b**uted S**y**stems) [1] using parts of the verification component of the **PEP** tool (**P**rogramming **E**nvironment based on **P**etri nets) [6, 19] which itself offers all the mentioned verification methods.

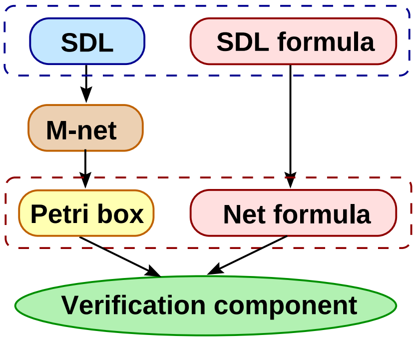


Fig. 1: Overview of our approach.

The paper is organised as follows. The relevant part of SDL is briefly introduced in section 2 by considering our running example, a small client-server system. Section 3 presents the semantic model of M-nets. The core of the paper is the definition of the M-net semantics of SDL specifications (covering dynamic process creation and – also recursive – procedures) in section 4. In section 5 we discuss the different verification approaches and give some results which we achieved. After comparing our approach to related work, we conclude in section 7.

2 SDL by an example

Over many years SDL has emerged as a quasi-standard language for the specification of distributed systems, especially in the area of telecommunication. The fact that SDL specifications may also be *represented graphically* has contributed to the wide acceptance of SDL. We will not give a detailed description of the syntax and the semantics of SDL, but we will try to introduce the most relevant parts very briefly. In particular, we will sketch the standardised graphical representation (GR). We use the GR to enhance readability, although the formal semantics is based on the textual *phrase* representation (PR), because the PR is unique and can be generated automatically from the GR.

An SDL specification describes a *system* consisting of *processes*. Processes are extended communicating finite state machines. Process declarations are labelled by two parameters (e.g., Client(1,2)) denoting the number of instances created initially and the maximum number of instances of the process that may exist at the same time. A process instance may be created by any other instance of any process type, but can be terminated only by itself. To address different instances of processes, each instance is equipped with a unique identifier of the predefined type PId of *process identifiers* (including the unique value null which is not used for an existing process). Additionally, each process instance contains four predefined variables of corresponding subtypes of PId whose values are changed automatically:

- self: the instance itself,
- sender: the sender instance of the most recently consumed signal,
- parent: the instance that created this one, and
- offspring: the most recently created instance.

Communication between processes is established via (asynchronous) FIFO channels with an optional delay and a certain capacity, via (synchronous) signal routes, or combinations of both. Each process instance owns an (asynchronous) input queue.

Signal types are declared in the same way as local variables (e.g. signal ack ({0,1}) or dcl x ({0,1}), respectively). As explained in the introduction, we restrict the allowed types to the set of Boolean values, the set of integers (both with the usual operations), and the set PId. In addition, we allow these types for parameters of processes, procedures, and signals, i.e., for the list of formal parameters (value parameters prefixed by the keyword fpar) of a process declaration and for the list of value parameters (in) as well as for the list of reference parameters (in/out) of a procedure declaration.

State transitions describe the behaviour of an SDL process. They determine the combinations of different kinds of actions (such as tasks x := 0, outputs data (x) via d, process creations Client (y) procedure invocations SendPackage or decisions parent) which may be executed if a certain input statement (e.g. ack (y)) is executable in a certain state (e.g. wait), and the next state. Initially, each process executes a special transition starting in the initial state ().

In the remaining part of this section, we give our running example. It describes a simple ARQ (**A**utomatic **R**epeat re**Q**uest) communication protocol with alternating acknowledgement. The top level of the specification (cf. Fig. 2) comprises two processes, Client (Client (1,2)) and Server (Server (1,1)). At any time there exists exactly one instance of the Server process, whereas up to two Client processes may be instantiated at the same time.

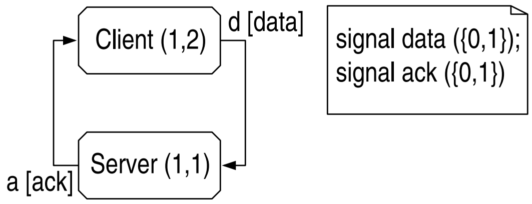


Fig. 2: The ARQ-System.

Fig. 3 shows a more detailed level. An alternating sequence of 0 and 1 (modelled by a signal type data with parameter type $\{0, 1\}$) should be transmitted. Each Client instance is responsible for the correct transmission of exactly one data package with the label x (which is the formal parameter) via the signal route d to the Server. The Server returns random acknowledgements via the signal route a specifying the receiver (sender) explicitly.

The initial instance (parent=null) sets the value of x to 0. The Client instance continues to send x (within the procedure SendPackage) until receipt of the corresponding acknowledgement. Then it creates the next Client instance (passing the next label as parameter) and terminates ($\times$) afterwards. The Server receives the data packages. Errors are modelled in the Server process by the non-deterministic choice between the answers, 0 and 1.

3 M-nets

The algebra of M-nets was introduced in [5] as an abstract and flexible metalanguage for the definition of the semantics of concurrent programming languages. The most distinguishing feature of the M-nets is given by the rich set of composition operators they provide. These allow – as usually composition operators in process algebras do – the compositional construction of complex nets from simple ones, thereby satisfying various algebraic properties.

Annotations of places (sets of allowed tokens), arcs (multisets of variables or values or tuples of variables and values), and transitions (occurrence conditions – called value terms) support the unfolding of an M-net into an elementary net. Communication capabilities are denoted by labels of transitions (action terms), while labels of places (called status) denote their interface capabilities. A status can be 'entry', 'exit' or 'internal'.

For the purpose of handling dynamic creation and termination of processes as well as procedures, we have to extend the notion of M-nets by allowing entry- and exit-places with complex (i.e. non-singleton) types, and by introducing additional composition operators, namely transition substitution and relabelling. Moreover, the definitions of some of the old operators have to be changed accordingly, thereby loosing the property of commutativity with unfolding (which is not needed in our approach). For this reason, we call the changed operations 'weak'.

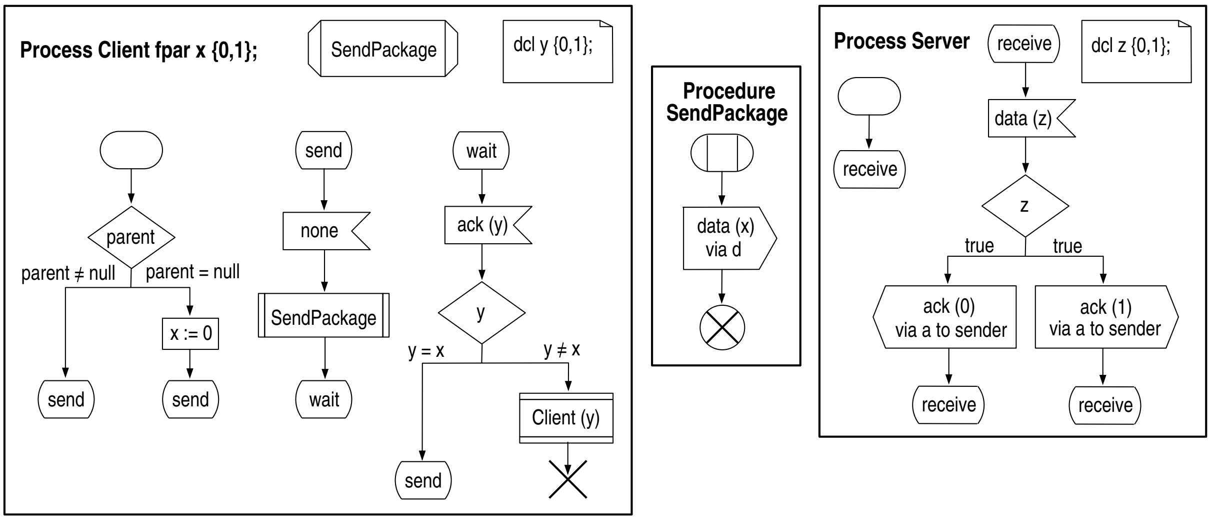


Fig. 3: The *Client* and the *Server* Process.

3.1 Auxiliary definitions

Let *Val* be a fixed and suitably large set of *values*. In our context, it is sufficient to assume that all integers, the Boolean values **true** and **false**, and the tokens • and †, as well as tuples of these are contained in *Val*. Let *Var* be a countably infinite set of variables. Variables are interpreted by values of *Val*.

We assume the existence of a fixed but sufficiently large set $\mathbf{A}$ of *action symbols*. The arity $ar(A)$ which is associated with each action symbol $A \in \mathbf{A}$ describes the number of its parameters. The elements of $\mathbf{A}$ are grouped into pairwise conjugates by the bijection $\overline{}: \mathbf{A} \to \mathbf{A}$, called *conjugation*, satisfying

$$\forall A \in \mathbf{A}\colon\ \overline{A} \neq A, \overline{\overline{A}} = A, \text{ and } ar(A) = ar(\overline{A}).$$

Finally, an *action term* is, by definition, a construct $A(\tau_1, \ldots, \tau_{ar(A)})$, where $A \in \mathbf{A}$ and $\tau_j \in Var \cup Val$ for $1 \leq j \leq ar(A)$.

3.2 Definition of M-nets

An *M-net* N is a triple (P, T, ι) such that P is a set of *places*, T is a set of *transitions* with $P \cap T = \emptyset$, and ι is a function with domain $P \cup (P \times T) \cup (T \times P) \cup T$ (called *inscription*) such that:

- $\forall p \in P\colon \iota(p) = (\lambda_p \mid \alpha_p)$, where $\lambda_p \in \{e, i, x\}$ is called the *(place-)label* or *status* and $\alpha_p \subseteq Val^{k_p}$ (for some $k_p \in \mathbb{N}$) is a nonempty set called the *place-annotation* or the *type* of p. k_p is called arity $ar(p)$ of p.
- $\forall t \in T\colon \iota(t) = (\lambda_t \mid \alpha_t)$, where λ_t is a finite multiset of action terms called its *label* and α_t is a term called its *annotation* or *value term* or *occurrence condition*.
- $\forall (p, t) \in (P \times T)$: $\iota(p, t) \in \mathcal{M}_f(\{(a_1, \cdots, a_k) | a_1, \cdots, a_k \in Var \cup Val\})$ with $k = ar(p)$ (similarly for arcs $(t, p) \in (T \times P)$), i.e., the inscription of (p, t) is a finite multiset of tuples of variables and values respecting the type of place p. The meaning of $\iota(p, t) = \emptyset$ is that there is no arc leading from p to t.

Furthermore, we require that there exists at least one entry and one exit place; that entry places have no incoming arcs and exit places no outgoing arcs. During the construction of the semantics we allow arbitrary types for entry and exit places of a net N, called the *interface type* $IT(N)$. But, we will only combine nets with the same interface type; and finally, the resulting net always has the interface type $\{\bullet\}$. An *initial marking* is a marking under which each entry place is marked with the same nonempty set of elements of its type.

Fig. 4 [2] shows a part of an M-net with one entry place Pd, one internal place Pe, one transition Tc and two arcs. The *transition rule* for M-nets is explained informally on the example: If Pd is marked with a pair $(3, 2)$, the variables in the arc inscriptions (pid and πid) can only be bound to 3 and 2, respectively. Thus, an occurrence of Tc carries the pair $(3, 2)$ to Pe. However, Tc can occur in infinitely many modes, because the action term contains variables ($'X$ and X') which are not sufficiently bound. E.g., $X' = 2 \wedge {'X} = 3$ is a possible binding. Note that our semantics will be given in such a way, that action terms are only used during the composition of the nets. The final nets will contain neither action terms nor free variables.

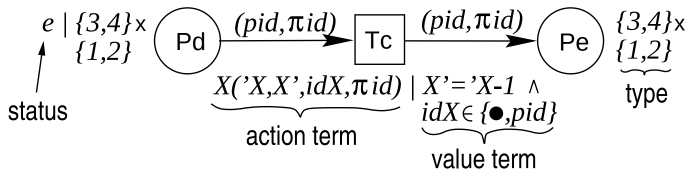


Fig. 4: M-net example.

M-nets can be unfolded into Petri boxes which are equivalent to elementary M-nets, i.e., M-nets in which all places have a singleton type, all transition labels consist only of variable free action terms and all occurrence conditions are empty. A place is unfolded into one place for each possible value and a transition is unfolded into one transition for each possible variable binding which fulfils the occurrence condition and respects the type of adjacent places. Markings are unfolded analogously.

3.3 Composition operations

For M-nets with complex interface type we have (at least) two different ways to define e.g. sequential composition, namely either by first unfolding the nets and then composing the results, or the other way round. Fig. 5 shows both possibilities: Ordinary sequential composition, which enforces that the first net hast to terminate completely before the second can start, is depicted in the upper part. An apparently different operator – which is called weak sequential composition in this paper[3] – results in an independent propagation of different tokens (cf. bottom part). Analogous considerations apply to the other vertical composition operators (choice composition, iteration, transition substitution). Note, that for parallel composition, there is no difference between the ordinary and the weak

[2] This figure and the following ones are simplified. Brackets around arc annotations and action terms, empty sets and labels of internal places are omitted.

[3] Weak sequential composition is similar to the notion of layered composition in [23].

version. In this paper, the weak compositions will be used, because they fit better into our approach.

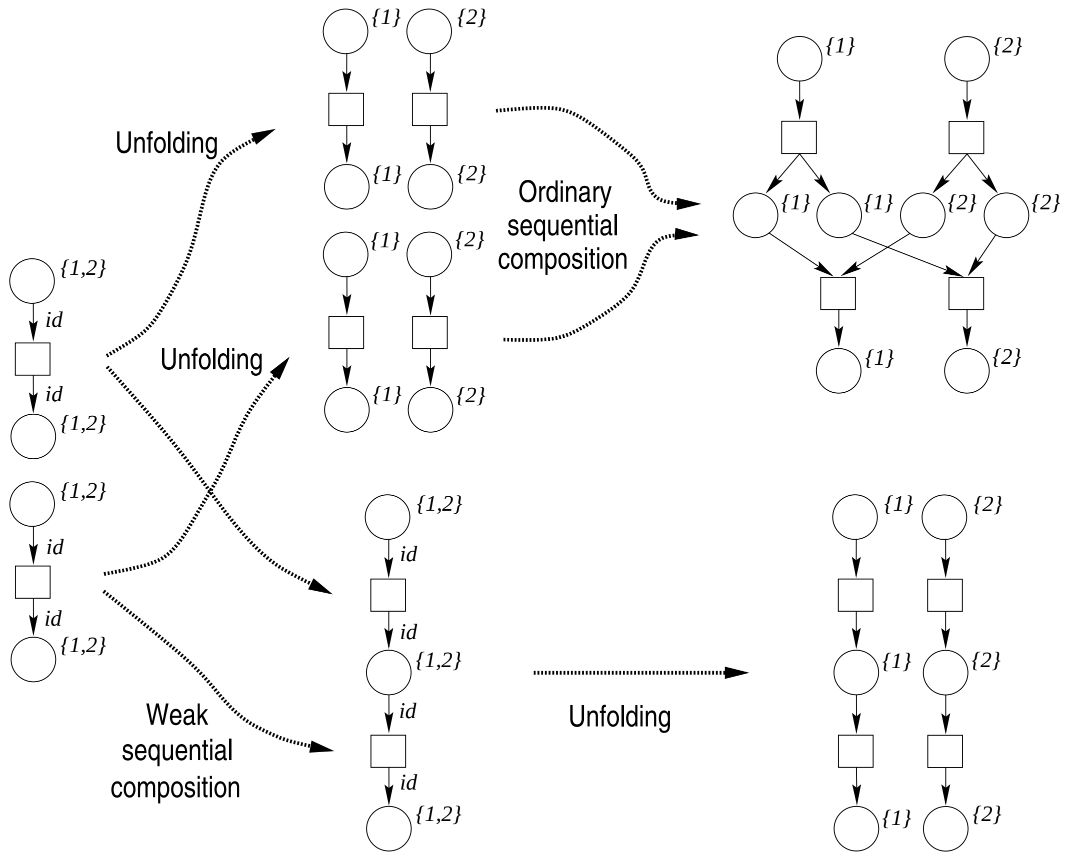


Fig. 5: Difference between weak and ordinary sequential composition.

We use the auxiliary operator *place merging* ($P_1 \otimes P_2$) to define the composition operations. This operator will only be applied to sets of places with the same interface type, because we will apply composition operators only to nets with the same interface type. Place multiplication of P_1 and P_2 is defined by

$$P_1 \otimes P_2 = \{p_{ij} \mid p_i \in P_1 \textit{ and } p_j \in P_2\},$$

such that all p_{ij} are of the same type $IT(N)$ having the label i if $\lambda_{p_i} \neq \lambda_{p_j}$ and λ_{p_i} otherwise. Note, that place multiplication is associative and commutative.

Parallel composition ($N_1 \| N_2$) (cf. Fig. 6) is defined as independent juxtaposition.

Weak sequential composition ($N_1; N_2$) (cf. Fig. 6) merges the exit places of the first net with the entry places of the second.

Weak choice ($N_1 \square N_2$) (cf. Fig. 6) merges the entry places and the exit places of the nets, respectively.

The *weak iteration* construct $[N_1 * N_2 * N_3]$ (cf. Fig. 7) produces the effect of one execution of N_1, followed by zero or more executions of N_2, followed by one execution of N_3[4].

[4]We were able to simplify the iteration operator of [5] here, because iteration is not applied to M-nets with initial parallelism.

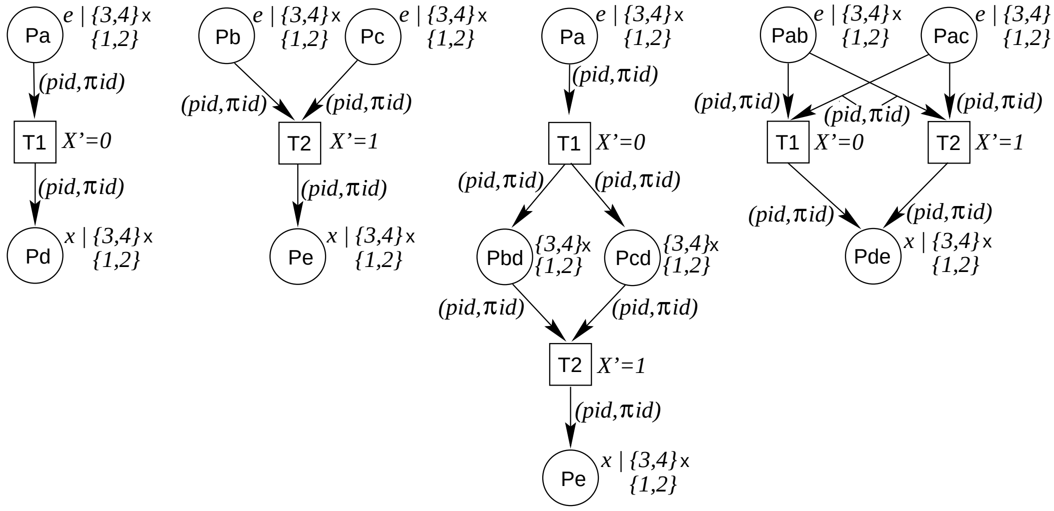


Fig. 6: From left to right: $N_1 \| N_2$, $N_1; N_2$ and $N_1 \square N_2$.

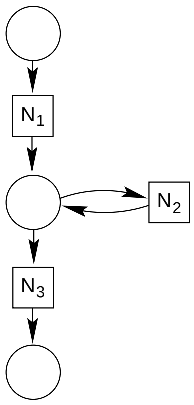


Fig. 7: Scheme of the Iteration.

Transition substitution is defined as follows: Let R be a transition of an M-net N_{R}, such that all places adjacent to R have the type IT. R is substituted by the M-net N with interface type IT (written as $N_{\mathsf{R}}\,[\mathsf{R} \leftarrow N]$) in the following way: R is replaced by N, all entry places of N are merged with the input places of R and all exit places of N are merged with the output places of R, all arcs which are connected to R are replaced by new arcs connected to the corresponding new places of N. Fig. 8 shows an example.

A *relabelling* function $\Gamma^{P}_{\text{var-list}}$, which is applied to an M-net N, is parameterised with a name of a procedure (P) and a list of variable names (var-list). It does the following substitutions for each occurrence of a variable $X \in$ var-list in N (see Fig. 8):

1. Each action term $X('X, X', idX, \pi id)$ is substituted with $X_P('X_P, X_P', pid, \pi id)$.
2. Each occurrence of X, $'X$ or X' within an occurrence condition is substituted with X_P, $'X_P$ or X_P', respectively.
3. Each occurrence of the term $idX \in \{\bullet, pid\}$ within a value term is deleted.

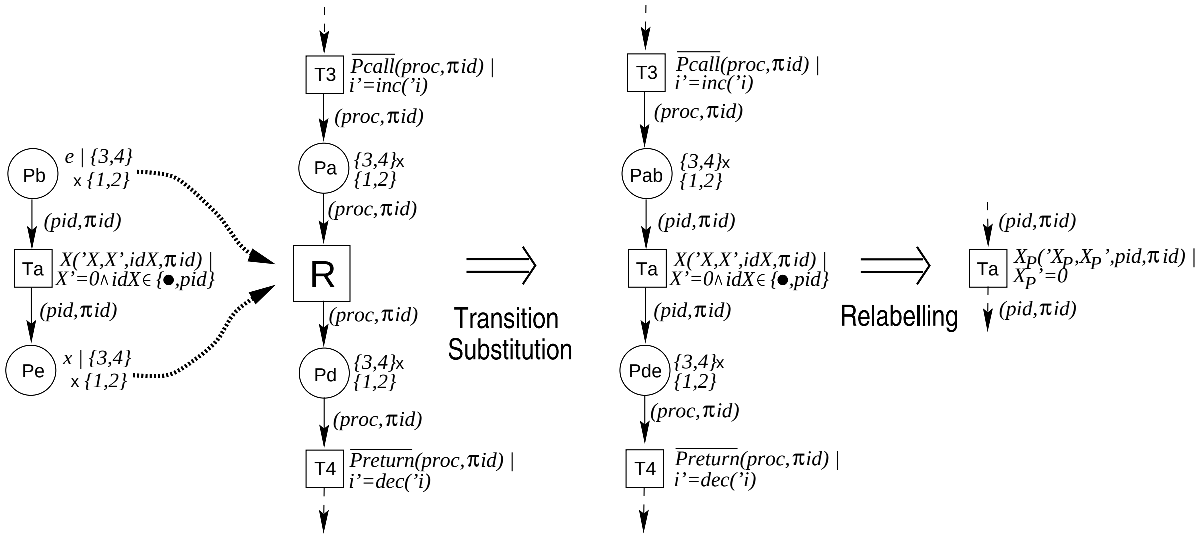


Fig. 8: Transition substitution and relabelling. From left to right: N, N_{R}, $N_{\mathsf{R}}\,[\mathsf{R} \leftarrow N]$ and $\Gamma^{P}_{\{X\}}(N_{\mathsf{R}}\,[\mathsf{R} \leftarrow N])$.

3.4 Synchronisation and restriction

Communication is performed by transition synchronisation. The intuition is that synchronisation of a net w.r.t. an action symbol (N **sy** A) is performed through a series of basic synchronisations. During a basic synchronisation two corresponding action terms ($A(\cdots)$ and $\overline{A}(\cdots)$) are considered. The communication is performed by a most general unifier which renames the variables in the action terms appropriately.

The *restriction* $N' = N$ **rs** A of a net N w.r.t. an action symbol A removes all transitions whose annotations contain action terms $A(\cdots)$ or $\overline{A}(\cdots)$ together with adjacent arcs.

The sequence $N' = (N$ **sy** $A)$ **rs** A of first synchronisation and then restriction of a net N w.r.t. an action symbol A is called *scoping* $N' = [\{A\} : N]$.

This scoping mechanism is used for block structuring. Fig. 9 shows a typical example. A local variable is accessed within a procedure. Tb is the access transition within the variable net and Tc (which was already explained in Fig. 4) corresponds to the access of the variable within the procedure. The renaming of the variable idX to pid (according to the $idX \in \{\bullet, pid\}$ part of the value term) ensures that the correct instance of X is accessed. Note that the free variables are bound during the scoping.

4 Translating SDL specifications into M-nets

In this section, a semantic function ζ will be defined which associates to each SDL system S an M-net $\zeta(S)$, the HL net semantics of S. $\zeta(S)$ may be unfolded into a Petri net $U(\zeta(S))$, the LL semantics of S. The semantic function satisfies the property that for each system S the LL semantics $U(\zeta(S))$ is a safe LL Petri net. We will explain the definition of ζ by applying it top-down to our running example.

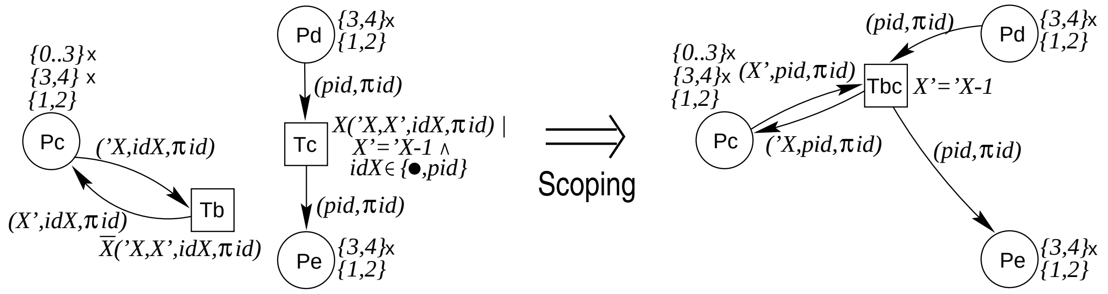


Fig. 9: Scoping example: $N_1 \| N_2$ and $[\ \{X\} :\ (N_1 \| N_2)\ ]$.

At a first look the figures will appear complex and too detailed. This comes from the fact that we provide the basis for a complete understanding of the semantics, rather than reducing it to the main ideas. We suggest therefore to look first only at those parts of the figures which are explained in the text.

4.1 General remarks

We will explain the translation of procedures and processes by considering the type of the tokens which are passed in the control flow. If neither procedures nor processes (with different instances) are involved, it is sufficient to use black tokens (•). If different process instances are involved, process instance identifiers (πids) rather than black tokens are used. If both, procedure and process instances, are involved we use tuples (pid,πid) containing a procedure identifier (pid).

Moreover, we use the '•' as a pid, in the case of a control flow token outside any procedure, and as a πid, in the case of a control flow token of the net for the global SDL machine (which, e.g., initialises channels and signal routes and creates the initial process instances).

One of the major challenges is to give the complete definition in a coherent way. In order to achieve this, it is necessary to use two additional parameters for the semantic function ζ, namely a set of pids and a set of πids.

Within the definition of ζ, we will use some auxiliary functions to determine some sets:

- $\mathcal{Q}(\Pi)$ contains the states of Π including the internal states • (for the intermediate state) and † (for the dead – i.e., going to terminate – state).
- $\mathcal{ID}(\Pi)$ contains the πids of Π.
- $\mathcal{ID}_s(\Pi)$, $\mathcal{ID}_r(\Pi)$, $\mathcal{ID}_p(\Pi)$, $\mathcal{ID}_o(\Pi)$ contain the πids that can send signals to Π, receive signals from Π, create Π, or be created by Π, respectively. Note that • is contained in these sets in order to represent null.
- $\mathcal{SIG}(\Pi)$ and $\mathcal{SIG}(C)$ contain the signals that can be sent to the process Π or be transmitted via the channel C, respectively. Note that • is contained in these sets in order to represent empty entries.
- $\mathcal{ID}(P)$ is the set containing the pids of procedure P.

In the semantic model of SDL, there are several types of infinite objects. The capacity of a channel ($\mathcal{CAP}(C)$) and the capacity of the input queue of a process ($\mathcal{CAP}(\Pi)$) are unbounded and, the maximal number of instances of a process ($\mathcal{MAX}(\Pi)$) or of a procedure ($\mathcal{MAX}(P)$) may also be unbounded. Since we are mainly interested in model checking (in addition to simulation), we assume that for each procedure and for each process there is a finite bound which limits the number of instances which may be active at the same time. Also, we impose a finite bound on the capacity of channels and input queues. Often, a finite bound is explicit part of the problem specification. Otherwise, if it exists at all, it may be determined using the model checker by checking with an increasing upper bound until, e.g., the queue cannot be filled completely.

4.2 Semantics of an SDL system

The M-net semantics of an SDL system is (by definition) the parallel composition of M-nets for:

1. each process,
2. each signal route,
3. each channel and
4. the global control.

This parallel composition is depicted for the ARQ system in an abstract way in the upper part of Fig. 10. Furthermore, the figure depicts that the nets for procedures, variables, formal parameters and the input queue are contained in the nets for the processes (e.g. Client contains SendPackage, y, x, Input queue). The net ζ(*Global control*) is not shown. It contains a sequence of: initialisation transitions (one for each process, procedure, channel and signal route); one creation transition for each initial instance of each process (in the ARQ example one Client and one Server); exactly one transition for the simultaneous termination of all the processes; and the termination transitions (the same as initialisation).

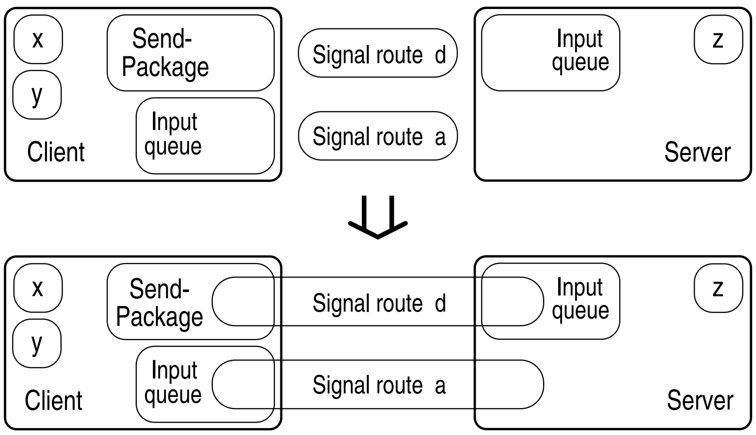


Fig. 10: Translation scheme.

The bottom part of Fig. 10 shows that parts of the nets synchronise during the following scoping (e.g., the input queue of Client with the signal route a). The scoping is done w.r.t.:

1. the initialisation, creation and termination actions of each process;
2. the initialisation and termination actions of each procedure, channel and signal route;
3. the input (from outside the process) actions of the input queues of each process denoted, e.g., by *Client!*; (scoping w.r.t. output (to the inside of the process) actions is done within the M-net for the process the input queue belongs to); and
4. the input (or receive) actions of each signal route and of each channel denoted, e.g., by *a!*; (the output or forward actions collapse with input actions of another signal route, channel or input queue of a process).

Hence, the semantics for the whole *ARQ* system is described by the following equation[5]:

$$\begin{aligned}&\zeta(ARQ, \{\bullet\}, \{\bullet\}) = \\ &\quad [\{Client_{init}, Client_{create}, Client_{term}, Server_{init}, Server_{create}, Server_{term}, \\ &\quad SendPackage_{init}, SendPackage_{term}, a_{init}, a_{term}, d_{init}, d_{term}, Client!, Server!, a!, d!\} : \\ &\qquad \zeta(Client, \{\bullet\}, \mathcal{ID}(Client)) \mid\mid \zeta(Server, \{\bullet\}, \mathcal{ID}(Server)) \mid\mid \\ &\quad\ \zeta(a, \{\bullet\}, \{\bullet\}) \mid\mid \zeta(d, \{\bullet\}, \{\bullet\}) \mid\mid \zeta(Global\ control, \{\bullet\}, \{\bullet\})\ ]\end{aligned}$$

4.3 Semantics of an SDL process

The main part of the semantic function of a single process Π is the process net. Before we go into the details of this net we want to explain its basic structure shown in Fig. 11. The left most part deals with the initialisation and termination of the process net. In particular, a stack for the handling of the πids is initialised and terminated, respectively. This stack is accessed by a creation and a resume part. Besides performing the initialisation (and termination, respectively) of the implicit variables, the parameters, the state, and the input queue for one instance of the process, the body part of this instance is enabled (or disabled, respectively).

The process net is constructed w.r.t. the definition of the in-parameters (in SDL processes do not have in/out-parameters) contained in *fpar-list* and the name of the process (further information is retrieved by the auxiliary functions such as $\mathcal{ID}(\Pi)$ or $\mathcal{SIG}(\Pi)$). Fig. 12 shows the detailed net which contains:

1. a part for the global initialisation and termination (P1, T1, T2, and P2);
2. a stack for the handling of πids (P3 counts the number of active instances, and P4 stores the πids of available (i.e., not active) instances together with their position in the stack.);
3. a subnet for the main part of the process containing:
   (a) T3 and T14 for the creation of a process instance. They synchronise with the corresponding creation transitions in the control flow exchanging the πids of

[5] So far we have used a simplified version of the function ζ in order to enhance readability. But, for the compositional definition of the semantics it is important that the function ζ has three parameters. The second and the third parameter correspond to the *pid* and the πid sets, respectively.

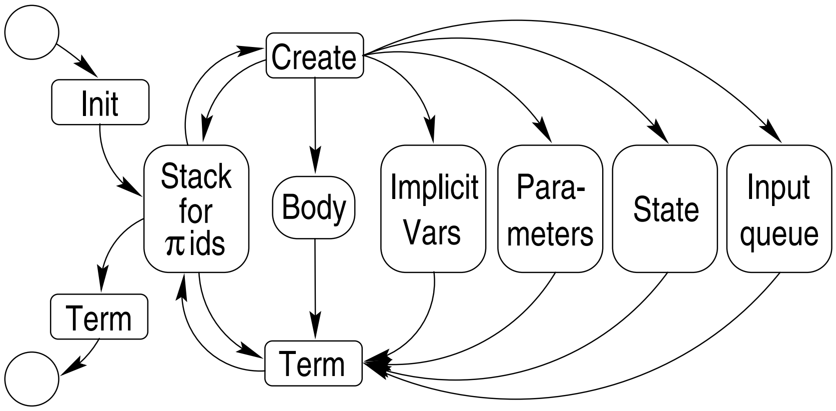


Fig. 11: Abstract process net

the new process and the parent, as well as the value of the formal parameters (only *pin* in the given net). T14 deals with the case that the maximum number of process instances is already active (in which case no new instance is created - • represents Null).

(b) T4 for the termination of a process instance;

(c) a special transition R_Π to be substituted by the M-net $\zeta(body(\Pi))$ for the body of the process Π.

4. places for the implicit variables self, parent, offspring and sender (P5, P6, P7, and P8). The action terms of the corresponding transitions (T5, T6, T7, and T8) are used to change their values. The first parameter corresponds to the value before, the second to the value after a change, the third to the *pid*, and the fourth to the πid.

5. a place for each formal in-parameter in *fpar-list* together with the appropriate transition to change its value (P9 and T9);

6. a place for the set $Q(\Pi)$ of states of the process together with an appropriate transition for state changes (P10 and T10). Note that in addition to the states defined explicitely in the declaration of the process, $Q(\Pi)$ contains the implicit states • and †, denoting *intermediate state* and *terminating state*, respectively.

7. a part for the input queue of the process consisting of:
   (a) P11, which counts (for each instance) the number of signals which are contained in the queue;
   (b) P12, which contains (for each instance) the signals;
   (c) an input (from the outside of an instance) transition (T11). The action term is parameterised with the signal and the πids of the sender and the receiver.
   (d) an output (to the inside of the instance) transition (T12);
   (e) a shift transition (T13) which shifts signals (together with the πids of sender and receiver) within the queue.

Nets for all procedures which are declared within Π are put in parallel with the process net. The result is scoped w.r.t. self, parent, offspring, sender, q (for the state), *in?* (for

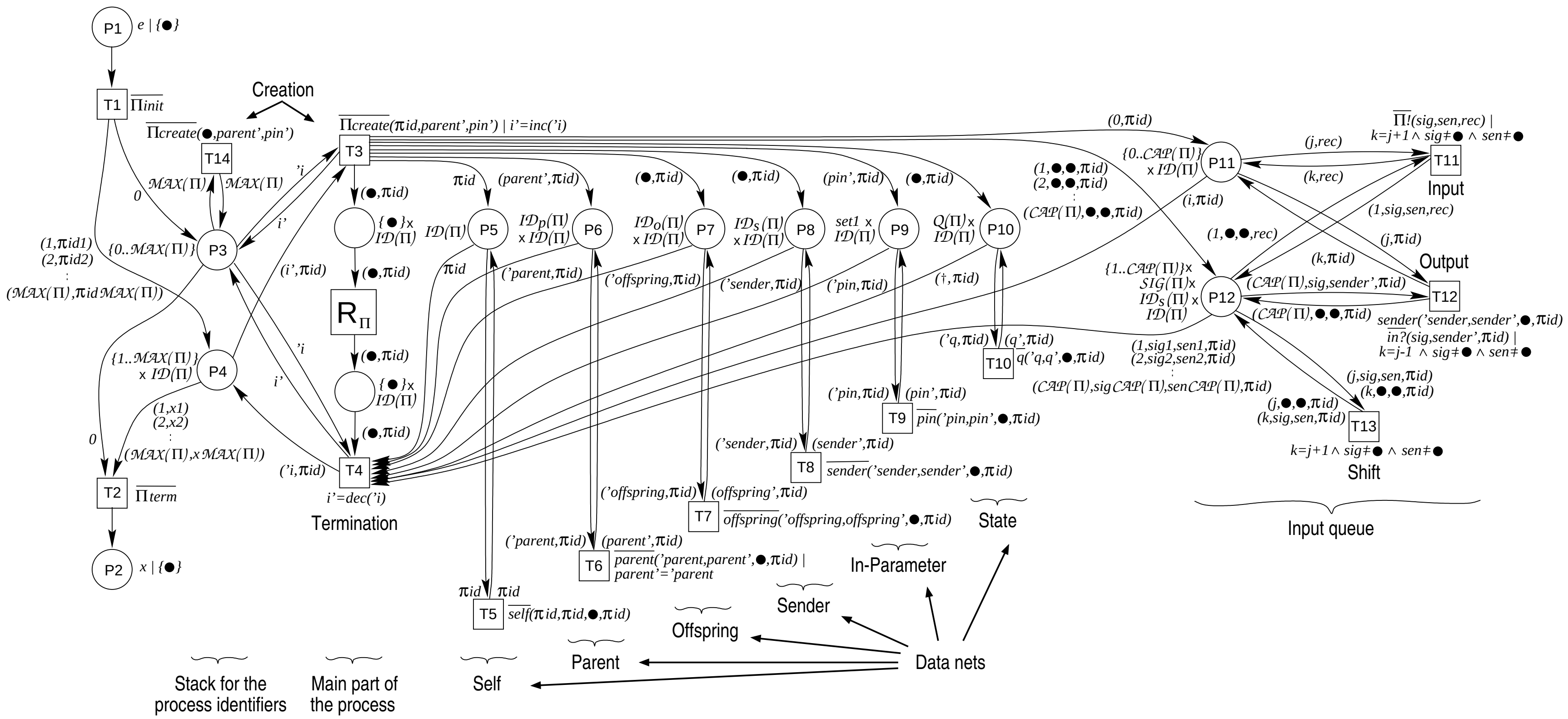


Fig. 12: SDL process net for a process Π with one parameter *pin* with type *set1*.

the output action of the input queue), *call* and *return* for each procedure, all relabelled in/out-parameters (of the procedures), all in-parameters of the process and all variables which are declared in the process.

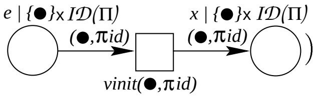
For the Client process of our running example this corresponds to the following equation (remember that $t \leftarrow N$ denotes substitution of (M-net) transition t by the M-net N):

$$\zeta(Client, \{\bullet\}, \mathcal{ID}(Client)) =$$
$$[\ \{\mathsf{self}, \mathsf{parent}, \mathsf{offspring}, \mathsf{sender}, q, in?, SendPackage_{call}, SendPackage_{return}, x, y\} :$$
$$N_{Client}(x{:}\{0,1\})\ [\ \mathsf{R}_\Pi \leftarrow (\zeta(body(Client), \{\bullet\}, \mathcal{ID}(Client)))\ ]\ \|$$
$$\zeta(SendPackage, \mathcal{ID}(SendPackage), \mathcal{ID}(Client))\ ]$$

4.4 Semantics of the body of an SDL process

The M-net $\zeta(body(\Pi), \{\bullet\}, \mathcal{ID}(\Pi))$ for the body of Π, in turn, consists of:

1. a part for each declaration 'dcl v : *set*;' of a variable, given by a special M-net $M_{data}(v, set, \{\bullet\}, \mathcal{ID}(\Pi))$, called *data net* (which is explained later);
2. a sequence of:
   (a) a part for the initialisation of each variable v

   $\gamma^I(v, \{\bullet\}, \mathcal{ID}(\Pi)) =$ 

);

   (b) a part for the control flow of the process $(\zeta(control(\Pi), \{\bullet\}, \mathcal{ID}(\pi)))$;
   (c) a part for the termination of each variable v $(\gamma^T(v, \{\bullet\}, \mathcal{ID}(\Pi)))$.

These are put in parallel and then scoped w.r.t. the action symbols for the initialisation and termination of the variables declared in Π. For the Client process we get:

$$\zeta(body(Client), \{\bullet\}, \mathcal{ID}(Client)) =$$
$$[\ \{y_{init}, y_{term}\} : M_{data}(y, \{0,1\}, \{\bullet\}, \mathcal{ID}(Client))\ \|$$
$$(\gamma^I(y, \{\bullet\}, \mathcal{ID}(Client)); \zeta(control(Client), \{\bullet\}, \mathcal{ID}(Client)); \gamma^T(y, \{\bullet\}, \mathcal{ID}(Client)))\ ]$$

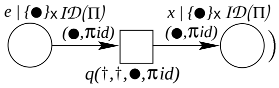
4.5 Semantics of the control part of an SDL process

The control part of an SDL process consists of an (SDL) start transition and a set of other (SDL) transitions which either terminate the process or yield the next state. This is reflected in the semantics by an iteration construct.

- The (SDL) start transition is translated into the start net of the iteration.
- The main part of the iteration is the choice of the nets for the other (SDL) transitions and for an implicit transition. The net for the implicit transition consists of only one transition which covers the cases for which no SDL transitions are specified, i.e., the consumption of certain signals in certain states.
- The exit part is an M-net (N^Π_{exit}) which ensures that the iteration is only exited if the state $q = \dagger$ is reached. This state is produced by the (M-net) transition that corresponds to the (SDL) termination.

For the control part of the Client process, which consists of a start transition *start* and two transitions $transition_1$ and $transition_2$ we have (cf. Fig. 13)[6]:

$$\zeta(control(Client), \{\bullet\}, \mathcal{ID}(Client)) =$$
$$[\ \zeta(start, \{\bullet\}, \mathcal{ID}(Client))$$
$$*\ (\ \zeta(transition_1, \{\bullet\}, \mathcal{ID}(Client))\ \Box\ \zeta(transition_2, \{\bullet\}, \mathcal{ID}(Client)))$$
$$*\ N_{exit}^{Client}\ ]$$

with (N_{exit}^{Π} = 

)

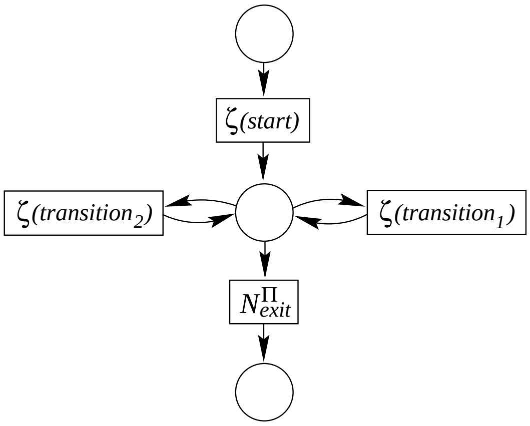


Fig. 13: Scheme for the transition loop (for the Client process).

4.6 Semantics of an SDL state transition

We will not describe the compositional derivation of the semantics of a state transition here, but merely explain some interesting points by considering the net semantics of the control part of the Client process (shown in Fig. 14). The part which corresponds to the two branches of the start transition is shown in the upper part (P1, T1, P2, T2, T3, P3, T4, P4, T5 and P5). The first state transition (cf. second transition in the left part of Fig. 3) corresponds to the loop consisting of T6, P6, T7, P7, T8, P8, T9 and P5. Finally, the left part of the net (T10, P9, T11, P10, T12, T13, P11, T14, P12, T15 and P5) corresponds to the second state transition.

We will consider the first state transition of the Client process (shown in the middle of the left part of Fig. 3) consisting of three parts which are composed sequentially: Consuming of input none in state send, call of procedure SendPackage, and entering of state wait. These parts are represented in the semantic model by four transitions which are also connected sequentially. The first one (T6) changes the state (currently send) to undefined ($\bullet$). The corresponding action synchronises with the data net for the state. Furthermore, the variable for sender is set to πid (which is self) in order to be conform with the SDL specification for an input none. The second one causes a call to the procedure

[6] We have omitted the part for the implicit transition because all combinations of possible inputs and states are already covered by the SDL specification.

SendPackage, the third one (T8) a resume. Finally, the fourth one (T9) changes the state from undefined to wait.

T4 shows that the semantics of the individual parts of an (SDL) transition (such as a variable access x:=0) is always constructed in the same way, regardless whether they are part of a process or of a procedure. The adjacent arcs are always annotated with (pid, πid), but the action term contains idx (instead of pid) together with the condition that idx is either pid (representing access of a local variable) or $\bullet$ (representing global for the process). The order of the scoping during the construction of the overall semantics (first w.r.t to local variables and then w.r.t. global variables) ensures that idx is bound correctly. The types of the adjacent places are determined by the second and the third parameter of the function ζ.

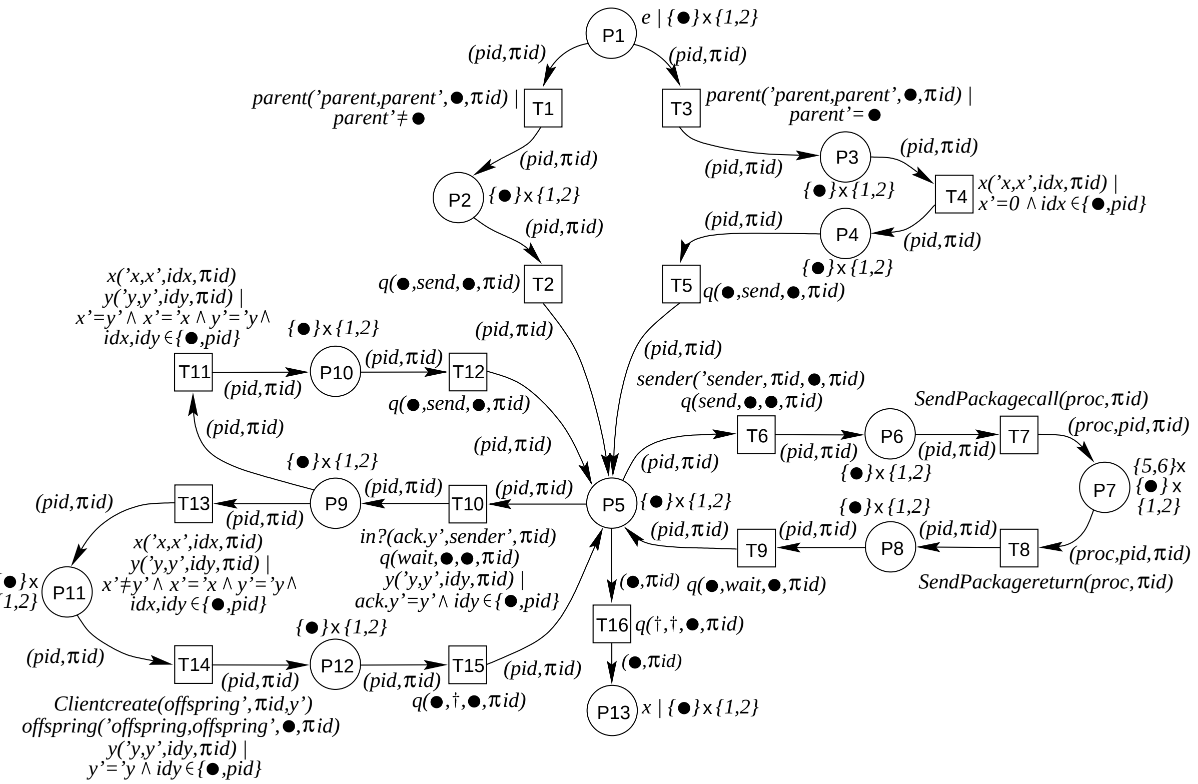


Fig. 14: Net for the control part of the *Client* process.

4.7 Semantics of an SDL variable

The semantics of a variable declaration 'dcl X : set;' is given by a special parameterised M-net (cf. Fig. 15). The first parameter of the net is the name of the variable, the second the type, the third the set of pids, and the last the set of πids. Thus, the above declaration yields $M_{data}(X, set, \{\bullet\}, \mathcal{ID}(\Pi))$ if X is declared within a process Π itself, and $M_{data}(X, set, \mathcal{ID}(P), \mathcal{ID}(\Pi))$ if X is declared inside a procedure P of a process Π.

The data net consists of four parts. T10 initialises an instance of the variable, and T13 terminates it. P12 stores the values of the variable instances (one for each combination of an pid and a πid). Finally, T12 is the counterpart for the variable access transitions

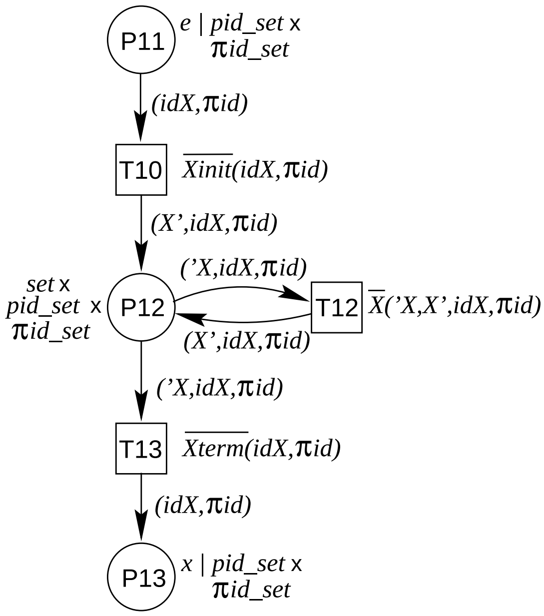


Fig. 15: The parameterised data net $\mathrm{M}_{data}(X, set, pid\_set, \pi id\_set)$.

within the control flow (e.g. T4 in Fig. 14). In Fig. 9 an example for the corresponding synchronisation was given.

4.8 Semantics of an SDL procedure

The semantics for the declaration of a procedure P with the formal parameter list *fpar-list* inside a process Π is given by the *procedure net* N_P^{Π}(*fpar-list*). This net is similar to the process net, but simpler, because a procedure has neither an own input queue, nor own implicit variables self, sender, parent, and offspring, nor an own set of states. In Fig. 16 the semantics of a procedure P with one in-parameter (*pin* with type *set1*) and one in/out-parameter (*pinout* with type *set2*) which is declared within the process Π is shown. It contains the following parts:

1. a part for the global initialisation and termination (P1, T1, T2, and P2);
2. a stack for the handling of *pid*s (P3 counts the number of active instances, and P4 stores the *pid*s of available (i.e., not active) instances together with their position in the stack.);
3. a subnet for the main part of the procedure containing:
   (a) T3 for the creation of an instance which pops a *pid* from the stack and increments the number of active instances.
   (b) T4 for the termination of an instance which pushes the *pid* back to the stack and decrements the number of active instances.
   (c) a special transition R_P to be substituted by the M-net for the body of P $\zeta(body(P), \mathcal{ID}(P), \mathcal{ID}(\Pi))$ which is first relabelled with respect to the in/out-parameter.

4. a place for the formal in-parameter *pin* together with the appropriate transition to change its value (P5 and T5);

The procedure net is then scoped w.r.t. the access actions for the in-parameters. For the general case of a procedure P with the in-parameters pin_1, ..., pin_n with types set_1, ..., set_n (respectively) and the in/out-parameters $pinout_1$, ..., $pinout_m$ we get the following equation:

$\zeta(P, \mathcal{ID}(P), \mathcal{ID}(\Pi)) = [\{pin_1, ..., pin_n\} :$
$N_P^{\Pi}(pin_1 : set_1, ..., pin_n : set_n)[\ \mathsf{R}_P \leftarrow \ \Gamma^P_{\{pinout_1, ..., pinout_m\}}(\zeta(body(P), \mathcal{ID}(P), \mathcal{ID}(\Pi)))\ ]$

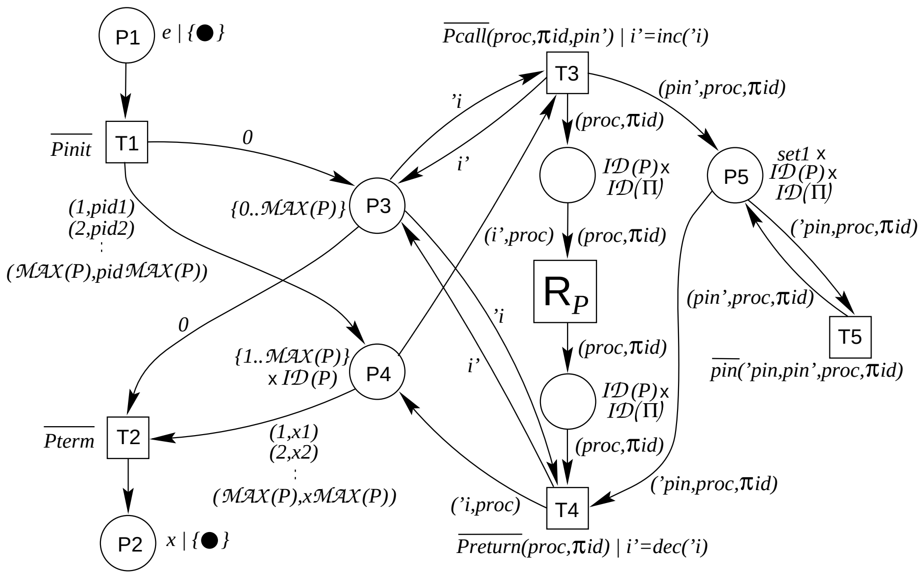


Fig. 16: The parameterised procedure net $N_P^{\Pi}(\textit{fpar-list})$.

Thus, we have the following for the procedure SendPackage of the Client process:

$\zeta(SendPackage, \mathcal{ID}(SendPackage), \mathcal{ID}(Client)) =$
$N^{\Pi}_{SendPackage}()\ [\ \mathsf{R}_P \leftarrow$
$\Gamma^{SendPackage}_{\emptyset}(\zeta(body(SendPackage), \mathcal{ID}(SendPackage), \mathcal{ID}(Client)))\ ]$

Note that the construction of the net for the body of the procedure is similar to the the construction for the body of a process.

4.9 Semantics of an SDL procedure call

The semantics of a call to a procedure P which is declared inside a process Π, is given by a special M-net N_{Pcall}. E.g., if the formal parameters of P are given by *fparamlist*, then we have the following equation for a call with actual parameters *actparamlist*:

$\zeta(\mathsf{call}\ P\ (actparamlist), pid\_set, \pi id\_set) =$
$N_{Pcall}(fparamlist, actparamlist, pid\_set, \pi id\_set)$.

Fig. 17 shows the net

$N^{\Pi}_{Pcall}((\mathsf{fpar\ in}\ pin\ set1\ \mathsf{in/out}\ pinout\ set2), (X, Z), pid\_set, \pi id\_set)$

of a call to the procedure P of Fig. 16 with actual parameters X and Z, respectively. This net consists of parts for:

1. *Initialisation*: T7 initiates a call. It synchronises at the same time with the *Pcall* transition T3 of the procedure net (cf. Fig. 16) and with the access transition T12 of the data net (cf. Fig. 15) for the actual in-parameter X. Thus, occurrence of the resulting transition causes the actual value of X to appear on P5 in the procedure net and, in addition, forwards the control token, extended by the *pid* of the just created procedure instance, to P8 and P9.

2. *Handling of in/out-parameters*: T8 synchronises with each transition inside the procedure net accessing *pinout* and with the access transition of the data net of Z. Hence, each access to *pinout* causes a corresponding access to the actual parameter Z. Note that this mechanism also works in the case where Z is not a variable, but another – or even the same – in/out-parameter, which might occur with recursive calls or calls within nested procedures. Corresponding to the relabelling, which is applied to the procedure net, *pinout* is renamed to $pinout_P$. This avoids possible name clashes between the formal in/out-parameter *pinout* and a variable, which might be declared in the environment. Each in/out-parameter has its own subnet (consisting of P8 and T8) within each call net. Otherwise, simultaneous accesses to different in/out-parameters would not be possible.

3. *Termination*: T9 synchronises with the return transition T4 of the procedure net.

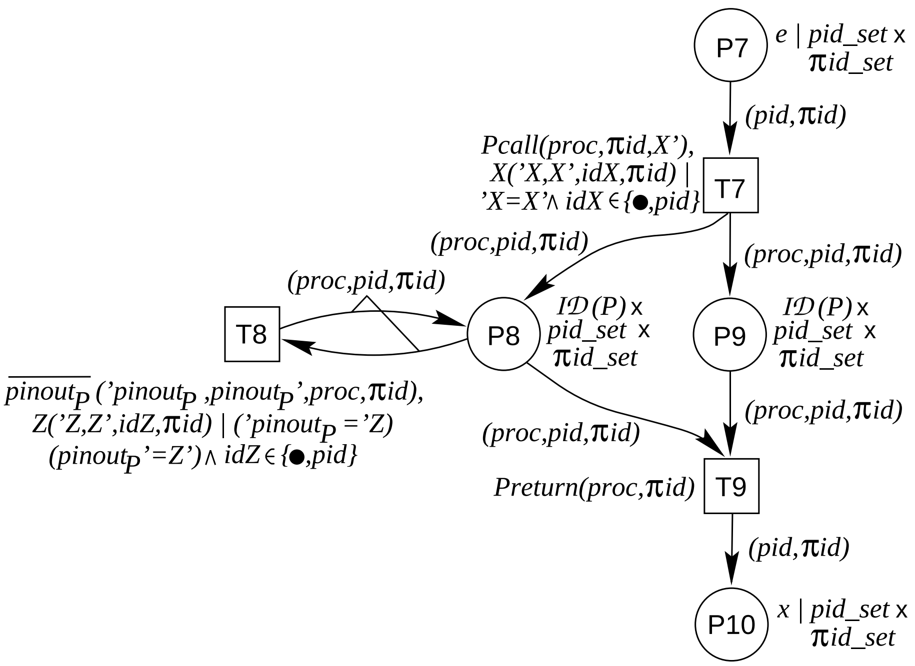


Fig. 17: M-net for the call $P(X,Z)$ of procedure P with the reference parameter *pinout*.

4.10 Semantics of an SDL channel

In SDL, communication between processes is modelled by channels and signal routes. Moreover, according to the standard semantics of SDL, each process has an implicit input queue which acts like a channel and which is represented in the M-net semantics of the process (cf. right part of Fig. 12).

A channel may receive input from processes, or another channel, or a signal route. Although there may be multiple input (or *From*) parts, signals are always forwarded to *To* which may be a process (possibly the same as the sender), or another channel, or a signal route. Signals have a certain type $\mathcal{SIG}(C)$ and the type of a channel is $\mathcal{SIG}(C) \times \mathcal{ID}_s(C) \times \mathcal{ID}_r(C)$. ($\mathcal{ID}_s(C)$ denotes the set which contains the πids of all processes whose signals may be transmitted via C, and $\mathcal{ID}_r(C)$ contains the πids of all processes to which signals may be directed via C.) Signals are forwarded after a non-deterministic delay (signal routes cover the case of no-delay channels). The semantics of a channel C is defined by:

$$\zeta(C, \{\bullet\}, \{\bullet\}) = M_{channel}(C, To)$$

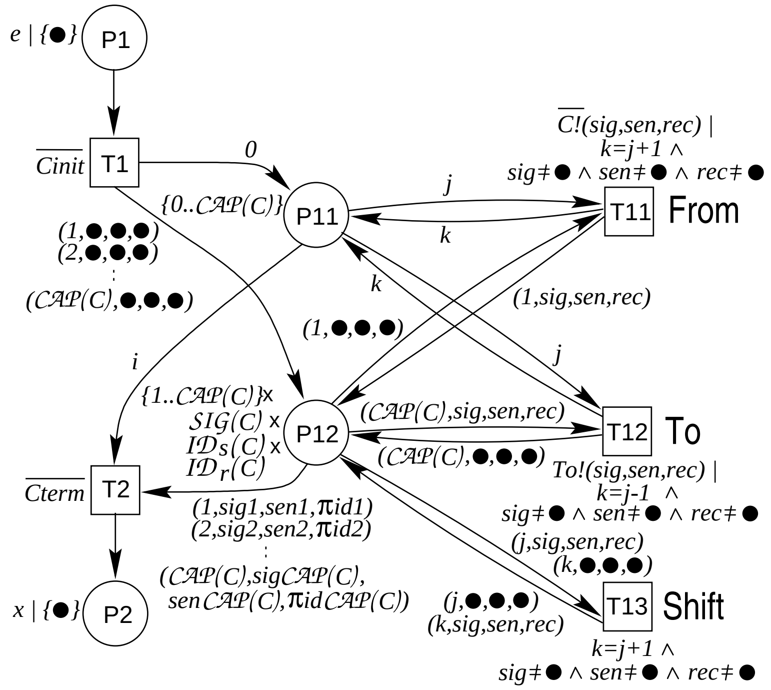


Fig. 18: M-net $M_{channel}(C, To)$ for a channel C.

The channel net $M_{channel}(C, To)$, which is parameterised with the name of the channel C and the name of the output side *To*, is shown in Fig. 18. It consists of the following parts:

1. *Initialisation*: The channel is initialised by an occurrence of T1, which synchronises with a global initialisation transition. Thereby P11 which counts the number of actual entries in the queue is initialised with 0, and the central place P12, which simulates the buffer, is initially filled with an empty entry (denoted by $(\bullet, \bullet, \bullet)$) for each of the $\mathcal{CAP}(C)$ many positions.

2. *'From' part of the channel*: New entries are written to the first position of the buffer by T11 which also updates the current number of entries and which synchronises with an output transition of the M-net of a possible sender.
3. *'To' part of the channel*: Output to somewhere is always done at the last position. It is performed by T12, which also updates the current number of entries and which synchronises with the input transition of *To* which may, e.g., be the input queue of a process.
4. *Shift*: Entries may be shifted from one position to the next (if this position is empty) by means of T13.
5. *Termination*: The channel may be terminated by occurrence of T2, which empties P11 and P12 and has to synchronise with an appropriate transition of the net for the global control.

4.11 Semantics of an SDL signal route

The semantics of a signal route R is similar to the semantics of a channel:

$$\zeta(R, \{\bullet\}, \{\bullet\}) = M_{route}(R, To)$$

However, the net $M_{route}(R, To)$ for a signal route R is much simpler (cf. Fig. 19). It provides synchronous (no-delay) communication of signals with type $\mathcal{SIG}(R)$ between $\mathcal{ID}_s(R)$ and $\mathcal{ID}_r(R)$. Synchronism is achieved by providing only one transition for both, the *From* part and the *To* part. The type $\{\bullet\}$ is sufficient for P2 because no entries have to be stored.

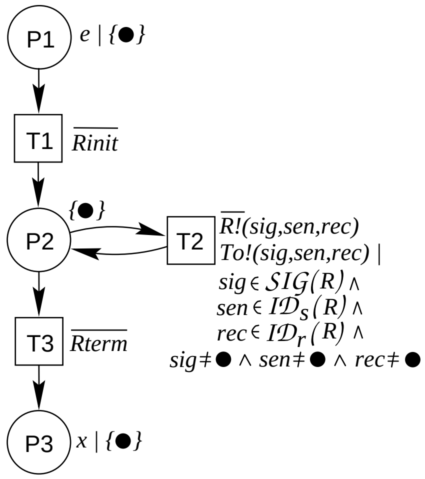


Fig. 19: M-net $M_{route}(R, To)$ for a signal route.

5 Verification of SDL specifications

In this section we exploit the different verification methods which are available if the presented approach for a compositional translation of SDL specifications into M-nets is used. By considering our running example:

- We show that our translation allows the application of 'state of the art' model checking algorithms by giving the results of various tests; and
- we provide an intuition of the additional advantage that the compositionality of our semantics is very important for the verification of large client-server systems.

Although, the implementation of the compiler has not been completely finished, we were able to produce the M-net semantics by using the various utilities which are available in the PEP tool [6]. Finally, this resulted in an M-net with 64 places, 36 transitions and 235 arcs and a corresponding low-level net with 137 places, 106 transitions and 905 arcs.

5.1 Automatic verification

Model checking is a widely accepted method for detecting errors in specifications of distributed systems. For reasonably sized systems, functional as well as qualitative temporal properties may be proven or disproven (fully automatically) by model checking based on a suitable semantics of the system. The most important model checking approaches for the verification of low-level Petri nets are:

1. Partial order based model checking [11, 20]: In contrast to other approaches this verification algorithm does not distinguish interleavings and is therefore often very efficient. Another advantage of this method is that the input (the finite prefix of the branching process of the net, called McMillan-unfolding or just 'prefix' [10]) has to be constructed only once for each SDL system. The major drawback of the method is that efficiency is gained partly by restricting the expressiveness of the supported logic.
2. Sleep set and bitstate techniques [21]: Another partial order based approach is implemented in the SPIN verification package. The sleep set method may be combined with a bitstate compression technique. A distinctive feature is that the verification does not necessarily construct the whole state space if an error is found earlier. Pohl implemented an interface based on a compilation of Petri nets into the input language PROMELA [26].
3. Reachability graph based model checking: This is the most well-known verification technique which first constructs the reachability graph for the whole system and then analyses it in a second step. For the purpose of verifying bigger client-server systems, analysis of non-reduced state graphs is infeasible. Therefore, mainly two reduction methods – 'stubborn set' and 'symmetric' reduction – are implemented in the CTL model checker belonging to the new version of the INA tool [27].
4. BDD based model checking [8]: Wimmel provided an interface to the SMV verification package by compiling low-level Petri nets into the SMV formalism [30]. This extended transition relation analysis also supports a stronger logic (namely CTL).
5. Linear programming based analysis [25]: Esparza and Melzer suggested an alternative method for reachability analysis which is based on integer programming using invariant and trap equations.

We have been able to apply these different methods to our running example using the PEP tool which includes all of them [19]. We profited from the possibility to use a dedicated reference scheme which is specific for the presented semantics and similar to the one presented in [18]. Thus, we were able to express the properties in terms of the SDL specification. For instance, we were able to refer to

- a particular instance of a process (e.g., *Client_1*),
- its implicit variables (e.g., *Client_1.sender*),
- its state (e.g., *Server_1.state*),
- the contents of its queue and the number of elements stored in its queue (for instance, *Client_1.queue[1].sig* and *Client_2.no_queue*)
- its formal parameters (e.g., *Client_1.par_x*),
- its variables (e.g., *Server_1.z*), and
- the number of active instances of a process (for instance, *Client.no_instances*).

Furthermore, we were able to transform the verification results back to the SDL level, getting information such as:

- a certain process was initialised,
- a signal '1' from the second instance of the Client process was written into the input queue of the Server process via the signal route d, or
- a call of the procedure SendPackage was performed.

We applied the different verification techniques to the ARQ system using a relatively slow 40 MHz SUN SPARC 10. 32 MB main memory had been sufficient. In the following we will give the details. In particular, we will give the time needed for the verifications (in seconds[7]).

In a first step, we made the necessary pre-calculations. We first constructed the prefix within 3.2 sec. It consists of 2180 conditions and 591 events. But, the most important characteristic w.r.t. verification is the number of 47 cut-off events and 49 maximal configurations which is an evidence for very efficient model checking. As a side effect of this calculation we were able to automatically eliminate all the dead transitions (which can never occur) together with consequently isolated places in the low-level net as well as in the M-net. This resulted in the size given above.

Furthermore, we constructed the stubborn set reduced reachability graph in approximately 8 sec. It consists of 576 states and 620 transitions. It turned out that the construction of a symmetrically reduced reachability graph is not appropriate for this example.

In a second step, we used different analysis techniques to check whether the resulting low-level net is deadlock-free. All of them figured out that this is true which is particularly interesting because, as pointed out in [9], '*Even for small systems with few clients and servers, especially in synchronous message passing environments, the potential for deadlock is very high.*':

[7]Unfortunately some of the programs do not display the used time. Thus, some of the given values are not 100 % accurate.

- A prefix based algorithm (first described by McMillan and optimised by Römer) in 17.69 sec.
- The integrated SPIN package in approximately 1 sec[8]. During this analysis SPIN exploited the whole (sleep set reduced) state graph of 5039 states and 8793 transitions.
- The integrated SMV package in 12.2 sec. We should mention that we speeded up the verification by a factor of more than 100 by using information about components of the low-level net. We were only able to retrieve this information due to the compositionality of our translation and the dedicated reference scheme. The size of the BDD (BDD nodes representing transition relation: 16392 and reachable states: 3753) may give a hint about the complexity.

In a third step, we made a couple of *safety* analysis, i.e., we checked whether or not certain (local or global) states of the whole ARQ system are reachable.

We would like to mention that some of the non-reachable states have already been reported by the elimination of dead transitions and isolated places during the calculation of the prefix. For instance, this produced the information:

- that none of the input queues ever contains more than one signal; or
- that none of the processes terminates (which should not be confused with a *return*).

We used the prefix based model checker to verify that each of the process instances can reach all its SDL states. This check is crucial because a lot of modelling errors can be detected like this very easily. Tab. 1 shows how we expressed this using the $\Diamond$-operator (possible sometimes in the future) as well as the results.

Formula	Result	T (s)
$\Diamond$ *Client_1.state = send*	TRUE	0.11
$\Diamond$ *Client_1.state = wait*	TRUE	0.09
$\Diamond$ *Client_2.state = send*	TRUE	0.10
$\Diamond$ *Client_2.state = wait*	TRUE	0.08
$\Diamond$ *Server_1.state = receive*	TRUE	0.07

Tab. 1: Reachability of states.

We expressed some other safety properties using also the $\Box$-operator (always):

- No process instance can ever be in the (internal) state † while its input queue is not empty. E.g.:
  $$\Box(\neg((Server\_1.state = \dagger) \wedge (Server\_1.no\_queue > 0)))$$
  The verification for the three instances took 0.3 sec.
- If both Client instances are active, one of them must be in one of the internal states • or †: $\Box((Client.no\_instances = 2) \Rightarrow$

[8]This is the time for the verification itself. The generation of the executable code took approximately 50 sec.

$$(Client\_1.state = \bullet \vee Client\_1.state = \dagger \vee$$
$$Client\_2.state = \bullet \vee Client\_2.state = \dagger)).$$

The prefix model checker proved this in 0.43 sec, the LP based analyser in less than 1 sec and the INA CTL model checker also in less than 1 sec.

In a fourth step, we checked a couple of (in the terminology of some people) *liveness* properties. A state in an SDL system may be called live iff it is always reachable. Correspondingly, a system may be called live iff all its states are live. Tab. 2 gives the results.

Formula	Result	T (s)
$\Box(\Diamond\ Client\_1.state = send)$	TRUE	0.35
$\Box(\Diamond\ Client\_1.state = wait)$	TRUE	0.37
$\Box(\Diamond\ Client\_2.state = send)$	TRUE	0.40
$\Box(\Diamond\ Client\_2.state = wait)$	TRUE	0.43
$\Box(\Diamond\ Server\_1.state = receive)$	TRUE	0.35

Tab. 2: Liveness of states.

An alternative way to prove the absence of deadlocks is to check that two different (and therefore mutual exclusive) states of a process are always reachable, e.g.

$$(\Box(\Diamond\ Client\_2.state = wait) \wedge (\Box(\Diamond\ Client\_2.state = \bullet))$$

because this implies that their is no deadlock. In our example, this is indeed the fastest possibility to verify that the resulting net is deadlock-free. The check took only 0.8 sec.

Finally, we checked some real *progress* properties which are not expressible using the $\Box$ and the $\Diamond$ operator, but in CTL. The following tests have been particularly interesting because they produce counter examples.

- It is not always the case that eventually the first instance of the Client process is in the state *wait* while the local variable x has the value 1 and the input queue of that instance contains 0 as the first element:

  $$\mathsf{AG}\ \mathsf{AF}\ (Client\_1.state = wait \wedge Client\_1.par\_x = 1 \wedge$$
  $$Client\_1.queue[1].sig = 0)$$

  The verification took approximately 1-2 sec (plus 50 sec for the generation of the executable) with SPIN, 88 sec with SMV, and approximately 1-2 sec with INA. Transforming back the resulting firing sequence to the SDL level, we retrieved the information that an instance of the Client process may continuously send '0' and that the Server instance may continuously reply with the (wrong) '0' acknowledgement.
- It is not always the case that after a Client instance has reached the point in the control flow directly after data (x) via d while the variable x has the value '1', this instance reaches eventually the point in the control flow directly after ack (y) while the variable y has the value '0'. Intuitively, this means that a send does not have to be acknowledged eventually. The test took approximately 2 sec (plus 55 sec for the generation of the executable) with SPIN, 56 sec with SMV, and approximately 1-2 sec with INA. The error trail gave a similar hint to a non-fair sequence as above.

Note, that the last two checks yield *true* if the CTL operators AG and AF are replaced by $\Box$ and $\Diamond$, respectively.

5.2 Compositional verification

To find proof rules which profit from the compositionality of the M-net semantics, remains a topic for further research. In this paper we can only illustrate what should be possible by means of a proof rule for a special safety property as a simple example. Consider again our running example. The instances of the Client process satisfy the following mutual exclusion property: At any time at most one instance of the Client process is in the state send. This property may be expressed by the formula

$$\Box(\neg(Client\_1.state = send \wedge Client\_2.state = send))$$

Of course, the ARQ-system may be checked against this formula using the same model checker as above. But, in typical client-server systems with a large server process and a huge number of different relatively small client processes this may be infeasible due to the well known 'state explosion' problem. In this situation compositionality may help. In our example, e.g., this property only depends on very few parts of the ARQ system, basically only on the Client process and its global initialisation as well as the creation of its initial instance. Therefore, it suffices to restrict the model to the Client process, to compute its semantics, and to check the formula against the resulting (in general much smaller) M-net.

The key points are, that the Server process does not create instances of the Client process and that this method may introduce new behaviour but it does not restrict any behaviour. In principal, we analyse the Client process in a completely arbitrary environment taking, for instance, into account that the input queues of the instances may be filled randomly. This approach can not be used for the verification of all kinds of properties, but as the model checking (with the prefix checker in 0.8 sec) gave the result 'true' in our special case, we have proven the mutual exclusion property for the ARQ-system independently of the size of the Server process. This is particularly interesting because a server process may obviously be much more complex than in our small example.

5.3 Interactive verification

In addition to the automatic and compositional verifications we also made an interactive verification in order to show another area of future research which is connected to our semantics. We wanted to prove that the Server process alternatingly receives and sends. This property is (in general) not expressible in CTL. Therefore, we used the synchronisation mechanism of the M-net model.

The language-theoretic approach to model checking based on automata (cf. [16]) expresses a formula as a Büchi automaton and synchronises it with the automaton which is verified. In [12, 29] this approach had been adapted to Petri nets. In a similar way we labelled the transitions which correspond to the send and the receive with send and receive action terms, respectively, and combined the result with a net similar to the one shown in Fig. 20.

We then calculated the prefix and figured out that all the transitions which resulted from a second send and a second receive in a row (T4 and T6, resp.) are dead. This is an elegant proof (which may also be done automatically) that the Server process alternatingly receives and sends.

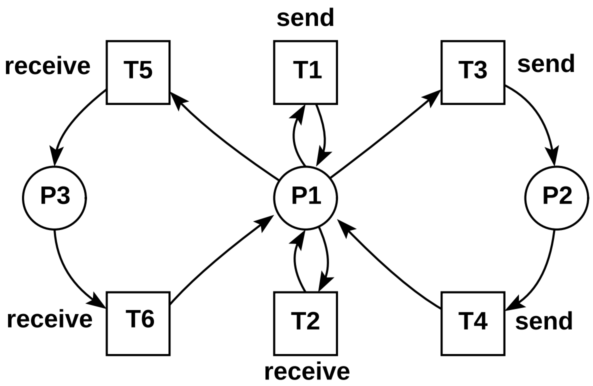


Fig. 20: Abstract formula net.

6 Related work

There have been earlier approaches to defining a Petri net semantics for (parts of) the SDL language; some examples are [13, 17, 22, 24]. All of these have their own merits, but, as we claim, none of them at the same time satisfies the requirements of compositionality and transparency and allows the application of the most important 'state of the art' model checking packages.

The Petri net semantics given in this paper can be considered as an extension of the one presented in [1]. We adapted and extended the techniques developed in [14] for the handling of procedures in the parallel programming language $B(PN)^2$ in order to be able to handle procedures and dynamic creation and termination of processes in SDL.

An orthogonal extension of [1] (which may as well be combined with our approach) is presented in [15]. Real-time requirements are added, and a compositional semantics for the resulting language is given in terms of Time-M-nets, which may be used to check also quantitative temporal properties of SDL-specifications.

In the EMMA (Extendible Multi Method Analyzer) project [22], a tool is constructed for the analysis of TeleNokia SDL (TNSDL), which is a dialect of SDL-88. An SDL system is translated into a formal validation model using high-level Petri nets. This model is analysed with the PROD reachability graph analyser [28]. The aim is to cover full TNSDL, i.e., the tool is planned to manage TeleNokia-specific applications. Hence, in contrast to our approach, not so much emphasis is laid on conformance with the published standard semantics of SDL. E.g., SDL transitions are handled as atomic actions in the EMMA approach and are represented by a single transition at the Petri net level. This is a suitable solution for their special purposes, but not in general. Especially in the area of client-server systems, it is often necessary to build systems whose SDL transitions are not atomic. The non atomicity of a real (executable) implementation should be a good reason for reflecting it in the semantics which is used for the verification. Otherwise, in many cases, errors hidden inside a transition cannot be properly localised in the design phase.

[13] describes the use of Petri nets for analysis and formal verification of SDL specifications within the SITE (SDL Integrated Tools Environment) project. Again, industrial applicability, and not semantic foundation, is the primary goal. For the analysis of SDL

specifications, the class of *SDL Time Nets* is introduced, which extends place/transition nets by guards and time intervals for transitions, and data structures for (special) places. An SDL system is translated into an SDL time net. This covers all features of SDL'92 without any essential restrictions. To yield small net models, also this approach deviates from the standard semantics of SDL at some points. E.g., in the translation of a process declaration, the implicit variables self, sender, parent and offspring are neglected. Also, the guard functions for net transitions and the data structures for process variables and signal parameters have to be coded into C-programs by the user. Thus, the same holds true as for the EMMA approach: it is an appropriate solution for their special purposes, but not a general one.

The EPOCA environment is described in [9]. Client-server programs are developed in DISC (DIStributed C), rather than as SDL-specifications. But the analysis is based – similar to our approach – on a compositional Petri net semantics. Since the semantics is given in terms of generalised stochastic Petri nets, it allows to reason about quantitative properties. They base the analysis of qualitative properties, e.g. deadlock-freeness, on reachability analysis of the net model. Since – in contrast to our approach – tests in a DISC-program are modelled by nondeterminism in the semantics, deadlock analysis faces the problem of 'non-faults' (which are deadlocks that can never happen in a real execution of the program). This problem does not arise in our semantics, since program variables are represented explicitly.

7 Conclusion and future work

Based on the algebra of M-nets, a flexible and powerful high-level metalanguage, we have presented the main ideas of a fully compositional and transparent semantics of SDL specifications. This Petri net semantics covers dynamic creation and termination of processes as well as (also recursive) procedures, and allows all types of parameters available in SDL. We have followed the standard semantics of SDL as given in Annex F of [7] as closely as possible.

The benefits of our approach are twofold. On one hand, we have given a completely precise semantic definition avoiding, e.g., ambiguities which are inherent to textual descriptions. On the other hand, we have shown how verification of SDL specifications may be based on our approach.

As an example, we have checked various properties of a client-server system very efficiently, using existing verification tools, which are all integrated in the verification component of the **PEP** tool. It turned out, that the semantics is helpful for proving properties automatically as well as for interactive proofs. Moreover, as our example suggests, it supports the derivation of compositional proof rules, which remains an interesting topic for further research.

Acknowledgement: We would like to thank Eike Best and anonymous referees for comments on the paper.

References

[1] P. Amthor, H. Fleischhack, and J. Tapken. *MOBY – more than a Tool for the Verification of SDL-Specifications*, Technical Report, Universität Oldenburg, 1996.

[2] E. Best. *Partial Order Verification with PEP. Proc. of POMIV'96, Princeton.*

[3] E. Best, R. Devillers, and J. G. Hall. The Box Calculus: a New Causal Algebra with Multi-Label Communication. *Advances in Petri Nets 92*, *LNCS* 609, 21–69. Springer.

[4] E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. An M-Net Semantics of $B(PN)^2$. *Proc. of STRICT'95*, 85–100. Springer.

[5] E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel, and E. Pelz. A Class of Composable High Level Petri Nets. *Proc. of ATPN'95, Torino*, *LNCS* 935, 103–118. Springer.

[6] E. Best and B. Grahlmann. *PEP: Documentation and User Guide.* Universität Hildesheim. Available together with the tool via: http://www.informatik.uni-hildesheim.de/~pep.

[7] CCITT. *Specification and Description Language*, CCITT Z.100, Geneva, 1992.

[8] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic Model Checking. *Proc. of CAV'96, New Brunswick*, *LNCS* 1102, 419–422. Springer.

[9] S. Donatelli, N. Mazzocca, and S. Russo. Client-server programs analysis in the EPOCA environment, *Distributed Systems Engineering Journal*, **3**, 1996.

[10] J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. *Proc. of TACAS'96*, 1996.

[11] J. Esparza. *Model Checking Using Net Unfoldings*, 151–195. No 23 in Science of Computer Programming. Elsevier, 1994.

[12] J. Esparza and S. Melzer. *Model Checking LTL using Constraint Programming. Proc. of ATPN'97*, *LNCS* 1248, 1–20. Springer. An extended version is available as SFB-Bericht 341/07/97 A, TU München, 1997.

[13] J. Fischer, E. Dimitrov, and U. Taubert. *Analysis and Formal Verification of SDL'92 Specifications using Extended Petri Nets*, Technical Report, Humboldt-Universität zu Berlin (1995).

[14] H. Fleischhack and B. Grahlmann. *A Petri Net Semantics for $B(PN)^2$ with Procedures. Proc. of PDSE'97*, Boston, IEEE Computer Society Press.

[15] H. Fleischhack and J. Tapken. An M-Net Semantics for a Real-Time Extension of μSDL. *Proc. of FME'97*, Springer.

[16] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems. LNCS 1032*, Springer. 1996.

[17] J. Grabowski. *Statische und dynamische Analysen für SDL-Spezifikationen auf der Basis von Petri-Netzen und Sequence-Charts*, Diploma Thesis, Universität Hamburg 1990.

[18] B. Grahlmann. The Reference Component of PEP. *Proc. of TACAS'97, Enschede*, *LNCS 1217*, 65–80, Springer.

[19] B. Grahlmann. The PEP Tool. *Proc. of CAV'97, Haifa*, *LNCS*, Springer.

[20] B. Graves. Computing Reachability Properties Hidden in Finite Net Unfoldings. *Proc. of FST&TCS'97.*

[21] G. Holzmann and D. Peled. The State of SPIN. *Proc. of CAV'96, New Brunswick. LNCS 1102*, 385–389. Springer.

[22] N. Husberg, M. Malmquist, and T. Jyrinki. *Emma: An SDL Analyzer Using High Level Petri Nets*, Technical report, Helsinki University of Technology, 1996.

[23] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *Proc. of CONCUR'91*, *LNCS 527*, 298–316, Springer.

[24] M. Lindqvist. *Translation of the Specification Language SDL into Predicate/Transition Nets*, Licentiate's Thesis, Helsinki University of Technology, 1987.

[25] St. Melzer and J. Esparza. Checking System Properties via Integer Programming. *Proc. of ESOP'96 LNCS 1058*, 250–264, Springer.

[26] C. Pohl. *Integration automatenbasierter Verifikation aus SPIN in PEP.* Diploma Thesis, Universität Hildesheim 1997.

[27] P. H. Starke. *INA: Integrated Net Analyzer.* Handbuch, cf. http://www.informatik.hu-berlin.de/~starke/ina.html.

[28] K. Varpaanniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. *PROD Reference Manual*, Technical Report 13, Helsinki University of Technology, 1992.

[29] F. Wallner. *Model Checking LTL Using Net Unfoldings*, Technical Report, Technische Universität München, 1997.

[30] G. Wimmel. A BDD-based Model Checker for the PEP Tool. *Technical Report, University of Newcastle upon Tyne*, 1997.