A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes

Christian Attiogbe, Pascal Poizat, Gwen Salaün

A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes

Christian Attiogbé, Pascal Poizat, and Gwen Salaün

***Abstract*— Separation of concerns or aspects is a way to deal with the increasing complexity of systems. The separate design of models for different aspects also promotes a better reusability level. However, an important issue is then to define means to integrate them into a global model. We present a formal and tool-equipped approach for the integration of dynamic models (behaviours expressed using state diagrams) and static models (formal data types) with the benefit to share advantages of both: graphical user-friendly models for behaviours, formal and abstract models for data types. Integration is achieved in a generic way so that it can deal with both different static specification languages (algebraic specifications, Z, B) and different dynamic specification semantics.**

***Index Terms*— Formal methods, languages, integrated environments, state diagrams, specification techniques, operational semantics, tools.**

I. Introduction

THE increasing complexity of systems (size, distribution and communication, number of interacting entities) has led in the last years to numerous proposals of expressive structuring mechanisms such as modules, viewpoints, components, software architectures or models. The corresponding entities are designed separately which increase their reusability while making their integration more complicated.

In this article we tackle *horizontal integration* which means the integration of models representing different concerns and possibly written in different languages. Rather than relying on a separate integration description which would make it necessary for the system designer to know yet another language, we propose to define a semantic framework for the integration *within* one of the languages of features from the other ones, yielding an *integrated*, or *mixed* language. Such a semantic framework is a mandatory preliminary step to be able to build tools dedicated to model integration.

The concerns we are interested in are the main ones in systems, *i.e.*, on one hand their *static aspects* (data types and related operations) and on the other hand their *dynamic aspects* (behaviours, concurrency and communication). In our approach, integration is *control-driven*: dynamics is the main aspect and drives the way data types (static aspects) are used. In this way, our proposal enforces the consistency of the static and dynamic parts.

We advocate the specification of static aspects using formal data description languages (*e.g.*, algebraic specifications [1]–[3], state-oriented languages such as Z [4] or B [5]). They allow the description of data types at a high abstraction level and the verification of their specification. Regarding dynamic aspects, we propose the use of state diagrams since such semi-formal notations (*e.g.*, UML [6] or Statecharts [7]) have now made a breakthrough in software engineering, mainly because of their user-friendliness through graphical notations and adaptability.

Semi-formal graphical languages lack a widely accepted formal semantics, and formal description languages are often said to be hard to learn and put into practice. Their joint use is a pragmatic approach which takes advantage of both: user-friendliness and readability from graphical approaches, high abstraction level, expressiveness, consistency and verification means from formal approaches.

Our integration approach is generic with respect to the static and the dynamic aspects. The language flexibility we propose for the static aspect specification enables the specifier to choose the formal languages that are the more suited to this task: either the ones (s)he is used to, the ones equipped with tools, or the ones that make the reuse of earlier specifications possible. Our approach makes the joint use of several static specification languages possible. This is an important feature as there is no universal modelling language and therefore different parts of the data used in systems may be more adequately written in different languages. Different dynamic semantics may be taken into account. Our approach may be used for Statecharts [7], [8], for different UML state diagram semantics, [9]–[12] for instance, and more generally for other state / transition based languages.

An early version of this work has been presented in [13]. The syntax part has been improved thanks to a motivating example, a formal grammar of transition extensions and more explanations. More details on the semantic framework and rules have also been added. Consistency and completeness of the dynamic semantic rules have been proven. Finally a section presents xCLAP, an animation prototype for extended state diagrams.

The article is structured as follows. Section II presents the syntactic extensions used to integrate formal data types within state diagrams. In Section III, the semantic foundations of our approach are formalised. Section IV demonstrates how the semantic framework can be instantiated. Section V overviews the xCLAP prototype tool. In Section VI, we present related works and compare our approach to them. Finally, Section VII concludes the article.

C. Attiogbé is with LINA FRE 2729 CNRS, Université de Nantes, France.
P. Poizat is with IBISC FRE 2873 CNRS, Université d'Evry Val d'Essonne and ARLES project, INRIA Rocquencourt, France.
G. Salaün is with VASY project, INRIA Rhône-Alpes, France.

II. Syntactic Aspects

In this section we present the extensions needed in state diagrams to enable their integration with data types, yielding *Extended State Diagrams* (ESDs). We advocate for a control-driven approach of integration. This means that dynamic behaviours, namely state diagrams, describe the main part of the specification whereas data types are handled by this behavioural specification.

State diagrams are used to graphically represent finite state machines. They can be used to specify the behaviour of various entities, from computer programs to business processes. Initial states are represented using filled circles. Final states are represented using hollow circles. Rectangles with round angles are used to represent states, which can be named. Transitions between states are represented with arrows. In addition to its source states (at least one) and its target states (any number), a transition may optionally support an event, a condition or guard, and an action list. Transition labels correspond to these three elements: EVENT `[GUARD] / ACTIONS`. The intuitive semantics of transitions is as follows: when the event is produced while the source states are active and the guard is true, then actions are executed and the target states are activated. Additional notations can be used to write dynamic models in a more concise way, *e.g.*, hierarchical states (also called OR-states) represented as states which contain a state diagram, or concurrent states (also called AND-states) to denote concurrent execution zones in state diagrams, represented using dashed lines between the zones. We refer to [6], [7] for a comprehensive description of state diagrams notations.

Let us start with a simple ESD specification (Fig. 1) to introduce what an integrated ESD specification would look like. It is a producer/consumer system where diagrams `D1` and `D2` are consumers which receive resources (`x`) and consume them. `D1` consumes all resources at once, while `D2` consumes them one by one. Diagram `D3` is the producer which sends resources to `D1` and `D2`, either sending one to each or two to `D2`. It also counts the number of provided resources (`n`).

The data part of the specification is described in our example using Larch algebraic specifications [1] (see `STORE` in Fig. 2 which keeps track of given resources). An *algebraic data type specification* is made up of a set of *sorts* (types) definitions together with *operators* on these sorts. Constants correspond to 0-ary operators. Operators with a result sort that corresponds to the sort being defined are called *constructors*. Operators and variables enable one to build *terms*, *e.g.*, if `1` and `plus` are natural numbers constructors (`1` being a 0-ary constructor and `plus` being a binary one), and if `x` is a natural number variable, then a possible term is `plus(x,1)`. The constructors that can generate all the terms corresponding to the values of a given sort are called *generators*. It could be for example `0`, `1` and `plus` for natural numbers. In our example, generators are `new` and `append`. The profiles of operators are first given and then their semantics are provided thanks to axioms, *e.g.*, an axiom such as `plus(x,y) = plus(y,x)` states the commutative property of addition. Note that variables in algebraic specifications are just placeholders in axioms and do not correspond to a state space for the sort they are defined in; hence they are not initialised. Axioms can be used in a functional fashion, *i.e.*, `op(args) = term`. In such a case, the `args` (arguments) part of axioms is usually defined inductively on the generators of the argument types. This can be observed in Fig. 2 where the `update` operator is defined using the two `STORE` generators, namely `new` (first axiom) and `append` (two last axioms). Algebraic specifications given as such are executable and can be transformed into code [14]. More details on algebraic specifications can be found in [2].

```
declare sort STORE
declare operators
   (* creates an empty store *)
   new : -> STORE
   (* adds a client record *)
   append : STORE, NAT, NAT -> STORE
   (* updates a client record *)
   update : STORE, NAT, NAT -> STORE
..
(* generators *)
assert sort STORE generated by new, append;
declare variables store: STORE,
                  client,client2: NAT,
                  amount,amount2: NAT
assert
   update(new,client,amount) = append(new,client,amount);
   (client==client2)
   => update(append(store,client,amount),client2,amount2)
      = append(new,client,amount+amount2);
   ~(client==client2)
   => update(append(store,client,amount),client2,amount2)
      = append(update(store,client2,amount2),client,amount);
..
```

Fig. 2. Producer/consumer static model

Data types are used in state diagrams to enable data encapsulation (`n` in `D3`), communication and value passing (`x` received in `D1` and `D2`). A more realistic example of such an integration of static and dynamic models is presented in [15] where more complex data types and ESDs are used.

As shown in the example, the state diagrams notation has to be extended in two ways to take into account formal data types: (i) data boxes are associated to state diagrams and (ii) data expressions appear in transitions.

Data boxes have two goals: they are used to import modules (a module being a collection of one or several data type definitions), and to declare variables locally to a state diagram. Note that the initialisation of variables is performed in transitions. Data boxes are inspired from UML notes which are usually used to give additional information to a diagram in a textual form. A data box is made up of a list of module importations and variable declarations. The `IMPORT` notation indicates which data modules are imported as well as the language used to write them out (*e.g.*, the Larch algebraic specification language as in our example, but also Z schemas or B machines). This language is called a *framework* in our approach and determines which function has to be used to evaluate the data embedded into the state diagram. Variables are also declared and typed in data boxes. Since modules often contain several type definitions, and since types with the same name may be defined in different modules, the type of a variable may be prefixed with the name of the module it is defined in to avoid conflicts.

Data expressions appear in transitions which are extended to (i) receive values in the EVENT part and

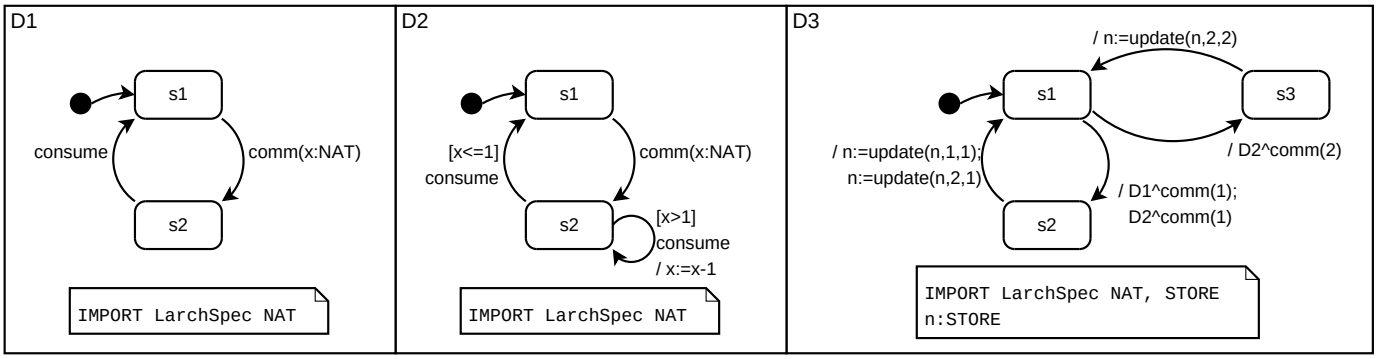

Fig. 1. Producer/consumer system

TABLE I
GRAMMAR OF TRANSITIONS WITH DATA EXPRESSIONS

TRANSITION	::=	[EVENT] [[GUARD]] [/ ACTION [; ACTION]*]
EVENT	::=	event-name [(PARAM [, PARAM]*)]
PARAM	::=	var : type
GUARD	::=	DATA-TERM
ACTION	::=	EMISSION | ASSIGNMENT
EMISSION	::=	receiver^event-name [(DATA-TERM [, DATA-TERM]*)]
ASSIGNMENT	::=	var :=DATA-TERM
DATA-TERM	::=	var | operation [(DATA-TERM [, DATA-TERM]*)]

then to store these values into local variables, *e.g.*, *event-name*(x_1:T_1,…,x_n:T_n), (ii) guard transitions with data expressions, *e.g.*, *predicate*(t_1,…,t_n), (iii) send events containing data expressions, *e.g.*, *receiver^event-name*(t_1,…,t_n), and (iv) make assignments of data expressions to local variables, *e.g.*, x:=t. Points (iii) and (iv) take place in the ACTIONS part of transitions. Table I gives a formal grammar of transitions handling data terms.

Data expressions may be either variables, terms for algebraic specifications or operation applications for state-oriented specifications. We recall that constants correspond to 0-ary operations. As far as the formal languages for the static aspects are concerned, the only constraint is to have some well-defined evaluation mechanism. The reason is that we are interested in providing a generic specification framework formally defined using an operational semantics. Hence, the design and implementation of dedicated tools, such as the one we present in Section V, can be tackled. Our approach makes the joint use of several static formal languages possible. However, a mix of constructs from several languages (such as the importation of a Z module within an algebraic specification, or using algebraic specification variables in a B operation application) is not allowed to avoid possible semantic inconsistencies. As a simple way to detect them, we develop a *meta-type* concept using meta-typing rules (see Section III-A). Terms which are not meta-typed, and therefore inconsistent, cannot be used in the dynamic rules.

To sum up, an *ESD specification* is given as a set of static models (data type modules) together with a set of dynamic models (ESD). Each ESD possibly includes data definitions (module importations and variable declarations). Any syntactic construction of state diagrams (*e.g.*, hierarchy, histories, inter-level transitions) can be used at the specification level provided that this construction is taken into account and formalised in the semantics considered for the non-extended notation (see configurations in Section III-C, page 6).

III. SEMANTIC ASPECTS

In this section, our goal is to give a formal semantics to state diagrams extended with formal data types as presented in the syntactic part.

We do not aim at formalising some specific kind of (non-extended) state diagram, which has already successfully been done, see [8]–[12] for example. We rather aim at being able to reuse different existing state diagram semantics. Therefore, our semantics is presented in a way such that generic concepts, represented in our semantics using a boxed notation, may be instantiated for a specific kind of state diagram semantics. A generic property such as “the event pertains to the input event collection of the state diagram” is for example represented as *event* $\boxed{\in}$ $\mathcal{Q}_{in}$, without assuming any specific additional constraint (such as an ordering of events in collections). It may however thereafter be instantiated into “the event is the first element of the input queue”, *event* $\in \mathcal{Q}_{in}$, thus taking into account ordering.

Using this generic approach, several kinds of non-extended state diagrams and their underlying semantics can be considered. The only constraint is that this semantics has to be given in terms of a Labelled Transition System (LTS), *i.e.*, a tuple (*INIT*, *STATE*, *TRANS*) with initial states, states and transitions which are tuples (source state, label, target state). Labels correspond to TRANSITION in Table I.

We define an *operational semantics* for ESD specifications since such a semantics is well suited for the definition of tools.

This semantics is based on LTSs. Getting such a semantics in one step is a complex task as different elements have to be taken into account: action semantics using *evaluation functions* and their effect on the *extended state space* of individual ESDs, storing of events, individual behavioural semantics of ESDs, relations between ESDs and their (open) environment, and finally communications between several ESDs at the system level. Therefore, we propose to achieve a semantics *incrementally*, *i.e.*, in several steps, taking into account at each step new elements presented above. This semantic "separation of concerns", yielding a separation of (groups of) semantic rules, as a side-effect enables one to reuse specific rules and replace or specialise other ones to deal with specific needs.

It is important to notice that in our approach both the syntactic pieces (state diagrams) and the semantic ones (semantic execution models) we incrementally build are LTSs. However, the incremental semantics process will bring at each step more information and semantics to enrich these LTS states and transitions. The semantics steps and the corresponding LTSs are summarised in Fig. 3, with syntactic pieces on the left and the incrementally built semantic ones on the right. Different notations are used to denote the different kinds of LTSs:

- simple notation, $(INIT, STATE, TRANS)$, for syntactic pieces;
- boxed (generic) notation, $(\boxed{INIT}, \boxed{STATE}, \boxed{TRANS})$, for the semantics of non-extended state diagrams we build on;
- underlined notation, $(\underline{INIT}, \underline{STATE}, \underline{TRANS})$, for the behavioural models of individual ESDs, taking into account data encapsulation and event collections for communication. These LTSs are generated by the *dynamic rules* (Fig. 3(1), Section III-C) which rely on *meta-typing* (Section III-A) and *action evaluation rules* (Section III-B). Moreover, the dynamic rules use, Fig. 3(c), a given (chosen) semantics for non-extended diagrams (denoted by $||\ .\ ||$ in the figure) obtained in Fig. 3(a) and (b). To be able to reuse this semantics in our approach, we have the need for specific (*i.e.*, dependent on the non-extended diagrams semantics we take into account) "forget" functions (they forget the extensions in ESDs), which are denoted by $\boxed{\downarrow}$. These functions distribute over LTS triples (both syntax and semantics ones), *e.g.*, $\boxed{\downarrow}(INIT, STATE, TRANS) = (\boxed{\downarrow} INIT, \boxed{\downarrow} STATE, \boxed{\downarrow} TRANS)$. Moreover, as we only extend transitions (see Section II), we have $\boxed{\downarrow} INIT = INIT$ and $\boxed{\downarrow} STATE = STATE$, and hence $\boxed{\downarrow}(INIT, STATE, TRANS) = (INIT, STATE, \boxed{\downarrow} TRANS)$. When dealing with a given non-extended diagram semantics (such as for example the [12] one, used in Section IV for illustration purposes), we will use a specific $\boxed{\downarrow}$ forget function (this means that the $\boxed{\downarrow}$ function has to be instantiated for specific non-extended semantics);
- *open* exponent notation, $(\underline{INIT}^{open}, \underline{STATE}^{open}, \underline{TRANS}^{open})$, for (open) models of individual ESDs, taking into account their relation with the environment. These LTSs are generated by the *open system rule* (Fig. 3(2), Section III-D);
- and finally over-lined notation, $(\overline{INIT}, \overline{STATE}, \overline{TRANS})$, for the complete semantics of an ESD specification, *i.e.*, a set of communicating ESDs. These LTSs are generated by the *global system and communication rules* (Fig. 3(3), Section III-E) which, putting these ESDs (quantified by i) individual open models altogether, yield a global model for the whole ESD specification. The use of forget functions (a) and non-extended semantics (b), and the application of dynamic rules (1), (c), and open systems rules (2) is performed independently for each diagram within the system. It is the global system and communication rules (3) which yield a semantics for the whole system.

$$\frac{(\texttt{IMPORT}\ XSpec\ M) \in DeclImp(D) \quad def(T, M) \quad x : T \in DeclVar(D)}{x ::_D X} \ (a)$$

$$\frac{(\texttt{IMPORT}\ XSpec\ M) \in DeclImp(D) \quad def(op, M) \quad \forall\, i \in 1..n\ .\ t_i ::_D X}{op\ \ t_1\ \ \ldots\ \ t_n ::_D X} \ (b)$$

Fig. 4. Meta-typing rules

In the sequel we present more formally each group of rules. We will also discuss the evaluation functions associated with the different static specification languages one may use. We will end with a proof of consistency and completeness for dynamic rules.

A. Meta-Typing Rules

The meta-typing rules are needed in order to detect multiple language inconsistencies and to be able to perform the evaluation of a term using the adequate evaluation function, that is the one dedicated to the framework corresponding to the meta-type of this term (*e.g.*, Larch, Z, B). In the following, $\mathcal{D}$ is the set of ESDs. Rules apply to a diagram D belonging to $\mathcal{D}$. The states of D are denoted by $STATE(D)$, its initial states by $INIT(D)$, and its transitions by $TRANS(D)$. $DeclImp(D)$ and $DeclVar^!(D)$ denote respectively the data modules importations and the variable declarations which appear in the diagram D data box. $DeclVar^?(D)$ is the set of (typed) variables received in events. $DeclVar(D)$ is the union of $DeclVar^?(D)$ and $DeclVar^!(D)$. A diagram D may be given syntactically by a tuple ($INIT$, $STATE$, $TRANS$, $DeclImp$, $DeclVar^!$). $def(x, M)$ is true if x is defined within the module M. We use T for usual types and X for meta-types. The notation $t ::_D X$ means that t has X as meta-type within the diagram D. Throughout the semantic part, operators suffixed with meta-types (*e.g.*, $\triangleright_X$) will denote their interpretation within the context of the corresponding framework (*e.g.*, $\triangleright_Z$ denotes the Z evaluation function). Semantic rules have the general form $\frac{H}{C} name$ where H is a set of premises and C, the consequent of the rule, is a conjunction of predicates.

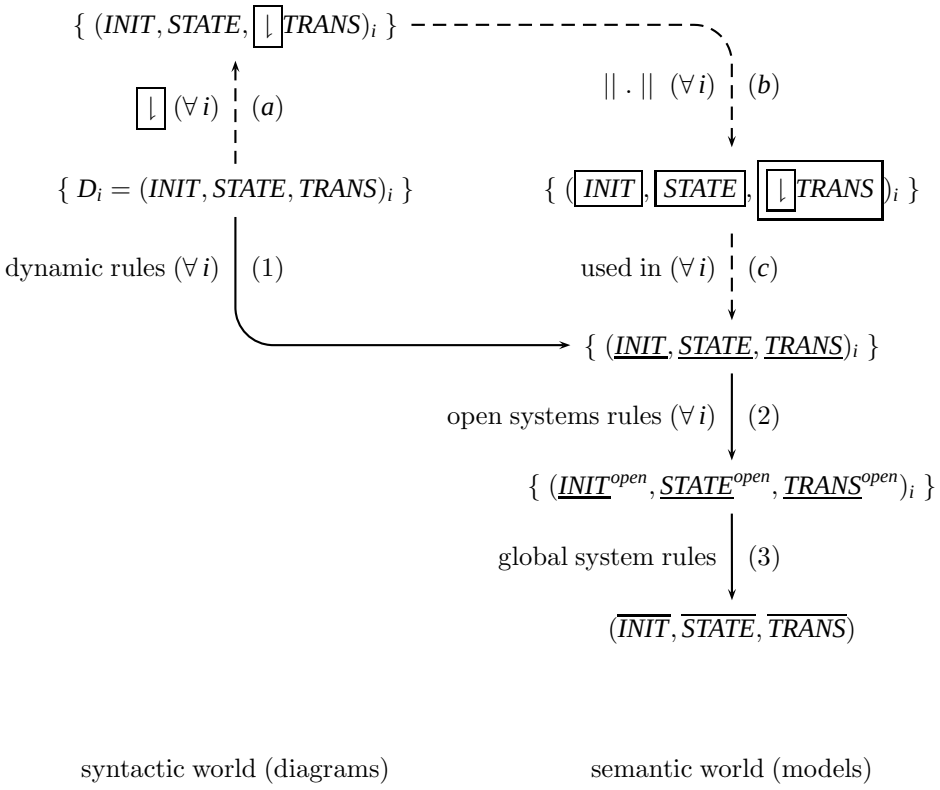

Fig. 3. Obtaining of ESDs semantics

The rule in Fig. 4(a) is used to give a meta-type to variables using local declarations and variable receptions. The rule in Fig. 4(b) gives the meta-type of a construction from the meta-types of elements which compose it. *op* $t_1 \ \ldots \ t_n$ is an abstract notation to denote the application of an operation to a list of terms, since there are some syntactic differences between algebraic and state-oriented formal specification languages.

B. Action Evaluation Rules

This set of rules deals with the effect of actions on the extended states used to give semantics to ESDs. Let us first give a definition of these states. $EVENT^?$ is the set of all *input events*, whose general form is $event\text{–}name(value_1, ..., value_n)$, that is a concrete instantiation with values of an abstract event parameterised by variables (*e.g.*, $e(0)$ is an instantiation of $e(x : NAT)$). $EVENT^!$ is the set of all *output events*, whose general form is $receiver \ \hat{} \ event - name(value_1, ..., value_n)$. $EVENT$ is the set of all events, that is: $EVENT = EVENT^? \cup EVENT^!$. The set of extended states for an ESD is defined as:

$$\mathcal{S} \subseteq \boxed{STATE}(D) \times \mathcal{E} \times \boxed{\mathcal{Q}}[EVENT^?] \times \boxed{\mathcal{Q}}[EVENT^!]$$

where

- $\boxed{STATE}(D)$ is the set of states used to give a semantics to the non-extended state diagram D;
- $\mathcal{E}$ is the set of environments, which are finite sets of pairs (x, v) denoting that the variable x is bound to the value v;
- $\boxed{\mathcal{Q}}$ is the set of collections, $\boxed{\mathcal{Q}}[EVENT^?]$ the set of input event collections, and $\boxed{\mathcal{Q}}[EVENT^!]$ the set of output event collections.

Collections are introduced to memorise events exchanged between diagrams. In the sequel, $Q_{in_D} \in \boxed{\mathcal{Q}}[EVENT^?]$ (respectively $Q_{out_D} \in \boxed{\mathcal{Q}}[EVENT^!]$) is used to denote a collection associated to a diagram D to store input (respectively output) events. Contrary to existing models storing events in collections (such as SDL using queues), we use two collections rather than a single one. Input and output collections will be used separately until the semantics of communication between several diagrams is taken into account linking up the output collections of some diagrams with the input collections of other ones. This enables one to adapt more easily parts of the semantics, for example to take different communication semantics into account (*e.g.*, there is no need to have explicit external buffers to model asynchronous communication). The $E \vdash t \rhd_X v$ notation means that using the evaluation defined in the X framework, v is a possible evaluation of t using the environment E for substituting the free variables in t. More details concerning the semantics of $\rhd_X$ will be given in Section III-F. Furthermore, if E and E' are environments then EE' is the environment in which variables of E and E' are defined and the bindings of E' overload those of E. We recall that symbols in boxes depict abstract structures and operations to be instantiated for a given type of state diagram. S will thereafter be used to denote an element of $\mathcal{S}$, and Γ_D to denote an element of $\boxed{STATE}(D)$. When there is no ambiguity on D, we use Γ for Γ_D, Q_{in} for Q_{in_D}, and Q_{out} for Q_{out_D}. The rules describing the evaluation of actions are given in Fig. 5.

$$\frac{act\text{-}eval(a_1,S,D)=S' \quad act\text{-}eval(a_2;\ \ldots;\ a_n,S',D)=S''}{act\text{-}eval(a_1;\ \ldots;\ a_n,S,D)=S''}\ EVAL\text{-}SEQ$$

$$\frac{}{act\text{-}eval(\varepsilon_{act},S,D)=S}\ EVAL\text{-}NIL$$

$$\frac{\begin{array}{c}\forall i\in 1..n \\ t_i ::_D X_i \\ E\vdash t_i \rhd_{X_i} v_i\end{array}}{\begin{array}{l}act\text{-}eval(rec\hat{}e(t_1,\ldots,t_n), \\ \qquad <\Gamma,E,Q_{in},Q_{out}>,D) \\ =<\Gamma,E,Q_{in},Q_{out}\boxed{\uplus}\{rec\hat{}e(v_1,\ldots,v_n)\}>\end{array}}\ EVAL\text{-}SEND$$

$$\frac{\begin{array}{c}t ::_D X \\ E\vdash t\rhd_X v\end{array}}{\begin{array}{c}act\text{-}eval(x:=t,<\Gamma,E,Q_{in},Q_{out}>,D) \\ =<\Gamma,E\{x\mapsto v\},Q_{in},Q_{out}>\end{array}}\ EVAL\text{-}ASSIGN$$

Fig. 5. Action evaluation rules

The $EVAL-SEQ$ rule is used to evaluate the actions in sequence. The $EVAL-NIL$ rule states that doing no action does not change the global state. The event emissions are dealt with by the $EVAL-SEND$ rule, which expresses that the effect of sending an event is to evaluate its arguments and then put it into the state diagram output event collection. $\boxed{\uplus}$ denotes an abstract union operation which may be instantiated differently depending on the type of state diagram semantics one wants: union of sets, adding in front of a queue, etc. The $EVAL-ASSIGN$ rule updates the local environment replacing the previous value of variable x by the new one (v, evaluation of t).

C. Dynamic Rules

This set of rules deals with the dynamic evolution of a single state diagram. We introduce a special event, ε denoting a stuttering step: $EVENT^{?+}=EVENT^{?}\cup\{\varepsilon\}$.

State diagram evolutions are given in terms of an LTS ($\underline{INIT}, \underline{STATE}, \underline{TRANS}$) where states are extended states, with:

$$\underline{STATE}\subseteq\mathcal{S}$$
$$\underline{INIT}\subseteq\underline{STATE}$$
$$\underline{TRANS}\subseteq\underline{STATE}\times EVENT^{?+}\times\underline{STATE}$$

We recall that ESDs may be given syntactically as a tuple ($INIT$, $STATE$, $TRANS$, $DeclImp$, $DeclVar^{!}$) and that the semantics of non-extended state diagrams are given in terms of an LTS ($\boxed{INIT}(D)$, $\boxed{STATE}(D)$, $\boxed{TRANS}(D)$). For some state diagram semantics, there is a direct correspondence between the syntactic and semantic notions of states (*i.e.*, $INIT$, $STATE$, $\boxed{INIT}$ and $\boxed{STATE}$ have the same type). However, for others (such as the UML state diagrams due to their hierarchical constructs), the semantics of non-extended state diagrams are given in terms of the more general concept of *configurations* [9]–[11] which are sets of *active* states. Hence, we define the *active* function which is used to know if a state is active in a configuration (or more generally in some element of $\boxed{STATE}$): $active(\gamma,\Gamma)$ is true if the γ state is active in the Γ configuration.

$$\frac{\begin{array}{c}\gamma_0\in INIT(D) \\ \Gamma_0\in\boxed{INIT}(D) \\ active(\gamma_0,\Gamma_0) \\ DeclVar^{!}(D)=\cup_{i\in 1..n}\{x_i:T_i\} \\ \forall i\in 1..n\ .\ x_i ::_D X_i\ .\ v_i :_{X_i} T_i\end{array}}{<\Gamma_0,\cup_{i\in 1..n}\{x_i\mapsto v_i\},\boxed{\emptyset},\boxed{\emptyset}>\in\underline{INIT}(D)}\ DYN-INIT$$

Fig. 6. Initialisation rule

$$\frac{S\in\underline{STATE}(D)}{S\xrightarrow{\varepsilon}S\in\underline{TRANS}(D)}\ DYN-\varepsilon$$

Fig. 7. Stuttering step rule

A first rule (Fig. 6) is used to obtain the initial extended states which correspond to the initial states of the non-extended state diagram underlying semantics ($\boxed{INIT}(D)$) extended with initial values for the variables and empty input and output collections ($\boxed{\emptyset}$). The meta-type of variables is used to define the notion of type in terms of a specific framework. The notation $v :_X T$ denotes the fact that, within the X framework, v is a value of type T.

A second rule (Fig. 7) is used to express stuttering steps. These steps denote an ESD which does not evolve and will be used when putting state diagrams in an open system environment. This rule should be taken with care when verifying liveness properties (such as inevitability of events) as it may cause livelocks (infinite sequences of stuttering steps). In such a case, verification tools should enable the assertion of fairness or progress hypotheses before verification takes place, in such a way that traces with infinite sequences of stuttering steps are not taken into consideration.

The next dynamic rule (Fig. 8) expresses the general evolution

$$\frac{\begin{array}{c}S\in\underline{STATE}(D)\quad S=<\Gamma,E,Q_{in},Q_{out}> \\ \gamma\in STATE(D)\quad \gamma'\in STATE(D) \\ l=e(x_1:T_1,\ldots,x_n:T_n)\ g\ /\ a_1;\ \ldots;\ a_m \\ \gamma\xrightarrow{l}\gamma'\in TRANS(D)\quad active(\gamma,\Gamma) \\ e(v_1,\ldots,v_n)\boxed{\in}Q_{in} \\ Q'_{in}=Q_{in}\boxed{\setminus}\{e(v_1,\ldots,v_n)\} \\ \Gamma\xrightarrow{\boxed{\downarrow}l}\Gamma'\in\boxed{\downarrow TRANS}(D) \\ \forall i\in 1..n\ .\ x_i ::_D X_i\ .\ v_i :_{X_i} T_i \\ E'=E\cup_{i\in 1..n}\{x_i\mapsto v_i\} \\ g ::_D X\quad E'\vdash g\rhd_X TRUE_X \\ act\text{-}eval(a_1;\ \ldots;\ a_m,<\Gamma,E',Q'_{in},Q_{out}>,D) \\ \qquad =<\Gamma,E'',Q'_{in},Q'_{out}> \\ S'=<\Gamma',E'',Q'_{in},Q'_{out}>\end{array}}{S'\in\underline{STATE}(D)\wedge S\xrightarrow{e(v_1,\ldots,v_n)}S'\in\underline{TRANS}(D)}\ DYN-E$$

Fig. 8. Basic dynamic rule (with event reception)

triggered when an event is read from the state diagram input event collection. This event may carry data values that are put into the variable environment of the state diagram. $TRUE_X$ denotes the truth value within the X framework. Once again, boxed elements are abstract concepts to be instantiated for a given type of state diagram semantics. $e \boxed{\in} \mathcal{Q}$ denotes that the event e is in the collection $\mathcal{Q}$. Possible instantiations are: e is in $\mathcal{Q}$ ($e \in \mathcal{Q}$), e is the first/top element in $\mathcal{Q}$ ($e = car(\mathcal{Q})$), e is the element in $\mathcal{Q}$ with the highest priority, and so on. $\mathcal{Q} \boxed{\setminus} e$ denotes, in the same way, the (abstract) removal of e from $\mathcal{Q}$. As explained in the Section III introduction, to be able to reuse the semantic information yield by the set of transitions of the non-extended state diagram semantics, $\boxed{TRANS}(D)$, we rely on a $\boxed{\downarrow}$ function.

This big-step semantic rule corresponds to the Run To Completion (RTC) step found in different state diagram semantics. Its meaning is the following:

if (premises)

- S is in the set of (semantic) states of the dynamic model,
- with Γ being the configuration part of this state, and (syntactic) state γ being active in Γ,
- there is a transition with label $l = e(x_1 : T_1, \ldots, x_n : T_n)\ g\ /\ a_1;\ \ldots;\ a_m$ originating from γ and going to a (syntactic) state γ', and a corresponding transition (in the basic semantic model) from Γ to Γ',

and (conditions for the transition to be triggered and completion)

- if event e, corresponding to the transition label, is present in the diagram input event collection, then reception variables are bound to the received values, and the guard g is evaluated,
- if g is true, then actions are performed, yielding a new (semantic) target state, S'

then (conclusion)

- S' is in the set of states of the dynamic model,
- and the transition from S to S', labelled by concrete values, is in its set of transitions.

However, sometimes events are not needed to trigger transitions. Such a case is dealt with by the rule in Fig. 9, where the corresponding transition of $\underline{TRANS}(D)$ is labelled by the stuttering step label (ε). The only difference with the previous rule is that the premises concerning the received event are not needed.

The *DYN–E* and *DYN–E*$\emptyset$ rules deal with the more general forms of state diagram transitions (*i.e.*, with the EVENT [GUARD] / ACTIONS and [GUARD] / ACTIONS forms). Rules for restricted forms of transitions (*e.g.*, without guard) may be obtained in an easy way from these general rules (*e.g.*, consider the guard to be true). An operational semantics is obtained from this model associating to D its LTS ($\underline{INIT}(D)$, $\underline{STATE}(D)$, $\underline{TRANS}(D)$), and then using for example an usual trace semantics denoted by:

$$|| D ||_{oper} = TR(\underline{INIT}(D), \underline{STATE}(D), \underline{TRANS}(D)).$$

TR denotes the set of traces computed from the LTS, see [16] for comprehensive definitions on trace semantics. These traces can be used, for example, to check LTL temporal formulas over the models.

$$\frac{\begin{array}{c} S \in \underline{STATE}(D) \quad S =< \Gamma, E, Q_{in}, Q_{out} > \\ \gamma \in STATE(D) \quad \gamma' \in STATE(D) \\ l = g\ /\ a_1;\ \ldots;\ a_m \\ \gamma \xrightarrow{l} \gamma' \in TRANS(D) \quad active(\gamma, \Gamma) \\ \Gamma \xrightarrow{\boxed{\downarrow} l} \Gamma' \in \boxed{\downarrow} TRANS\ (D) \\ g ::_D X \quad E \vdash g \rhd_X TRUE_X \\ act{-}eval(a_1;\ \ldots;\ a_m, < \Gamma, E, Q_{in}, Q_{out} >, D) \\ =< \Gamma, E', Q'_{in}, Q'_{out} > \\ S' =< \Gamma', E', Q'_{in}, Q'_{out} > \end{array}}{S' \in \underline{STATE}(D) \wedge S \xrightarrow{\varepsilon} S' \in \underline{TRANS}(D)}\ DYN-E\emptyset$$

Fig. 9. Basic dynamic rule (without event reception)

D. Open System Rule

This rule is used to express what happens when a state diagram is put into an open system environment. Basically, some events may be received from the environment and some others may be sent to it. As far as the input and output event collections of a given state diagram are concerned, this means that input events are put into its input event collection and output events are taken out of its output event collection. Note that these modifications of the extended states may appear while the state diagram does a transition (*i.e.*, following the $DYN-E$ and $DYN-E\emptyset$ rules) but also if it does nothing. To be able to represent this, we may use ε transitions (rule $DYN-\varepsilon$). More formally, the semantics of a state diagram in an open system is defined as the traces of the LTS ($\underline{INIT}^{open}(D), \underline{STATE}^{open}(D), \underline{TRANS}^{open}(D)$), that is denoted by:

$$|| D ||^{open}_{oper} = TR(\underline{INIT}^{open}(D), \underline{STATE}^{open}(D), \underline{TRANS}^{open}(D)).$$

For a given diagram D, we define:

$$\begin{array}{c} \underline{INIT}^{open}(D) = \underline{INIT}(D) \\ \underline{TRANS}^{open}(D) \subseteq \underline{TRANS}(D) \times \boxed{\mathcal{Q}}[EVENT^{?}] \times \boxed{\mathcal{Q}}[EVENT^{!}] \\ \underline{STATE}^{open}(D) = \underline{INIT}^{open}(D) \cup TARGET(\underline{TRANS}^{open}(D)) \end{array}$$

$\underline{STATE}^{open}$ is obtained from initial states ($\underline{INIT}^{open}$) and reachable states (target states of the $\underline{TRANS}^{open}$ transitions). The *DYN–OPEN* rule (Fig. 10) defines the modifications in the collections that may take place in an open system semantics. In this rule, the label l matches the two possible things the state diagram may do during the collection modification: a classical transition (rules $DYN-E$ and $DYN-E\emptyset$) or no internal state modification at all (which is the reason for rule $DYN-\varepsilon$).

$\boxed{\mathcal{P}}(ES)$ denotes the collection obtained from the powerset of ES. ES_{in} (respectively ES_{out}) in open transitions (members of $\underline{TRANS}^{open}(D)$) is used to keep track of what has been put into the input event collection (respectively taken out of the output event collection) of the state diagram. $S \xrightarrow{l}_{ES_{in}, ES_{out}} S' \in \underline{TRANS}^{open}(D)$ is used as a shorthand notation for $(S, l, S', ES_{in}, ES_{out}) \in \underline{TRANS}^{open}(D)$.

$$\frac{< \Gamma, E, Q_{in}, Q_{out} > \xrightarrow{l} < \Gamma', E', Q'_{in}, Q'_{out} > \in \underline{TRANS}(D) \quad ES_{out} \boxed{\subseteq} Q_{out} \quad ES_{in} \boxed{\subseteq} \boxed{\mathcal{P}}(EVENT^?)}{< \Gamma, E, Q_{in}, Q_{out} > \xrightarrow{l}_{ES_{in}, ES_{out}} < \Gamma', E', Q'_{in} \boxed{\uplus} ES_{in}, Q'_{out} \boxed{\setminus} ES_{out} > \in \underline{TRANS}^{open}(D)} \; DYN-OPEN$$

Fig. 10. Open system rule

E. Global System and Communication Rules

The last set of rules puts things altogether. The idea is that a global system made up of several ESDs (denoted $\cup_{i\in 1..n} D_i$) evolves as its components evolve. Once again we define the operational semantics of the system to be the traces of an LTS ($\overline{INIT}(\cup_{i\in 1..n} D_i)$, $\overline{STATE}(\cup_{i\in 1..n} D_i)$, $\overline{TRANS}(\cup_{i\in 1..n} D_i)$), that is denoted by:

$$TR(\overline{INIT}(\cup_{i\in 1..n} D_i), \overline{STATE}(\cup_{i\in 1..n} D_i), \overline{TRANS}(\cup_{i\in 1..n} D_i)).$$

$\overline{INIT}$, $\overline{STATE}$ and $\overline{TRANS}$ are defined as:

$$\overline{INIT}(\cup_{i\in 1..n} D_i) = \Pi_i \underline{INIT}^{open}(D_i)$$
$$\overline{TRANS}(\cup_{i\in 1..n} D_i) = \{t \in \Pi_i \underline{TRANS}^{open}(D_i) \mid CC(t)\}$$
$$\overline{STATE}(\cup_{i\in 1..n} D_i) = \overline{INIT}(\cup_{i\in 1..n} D_i) \cup TARGET(\overline{TRANS}(\cup_{i\in 1..n} D_i))$$

$\overline{TRANS}$ is obtained from the product of the $\underline{TRANS}^{open}$ sets of each state diagram of the system, restricting this product with a communication constraint (CC) which expresses that whenever an emission event is taken out of a given diagram (D_k) output event collection (*i.e.*, present in ES_{out_k}), and if the receiver of this emission is a member (D_j) of the system, then this receiver has the event being put into its input event collection (ES_{in_j}).

$$CC(S_1 \xrightarrow{l_1}_{ES_{in_1}, ES_{out_1}} S'_1, \ldots, S_n \xrightarrow{l_n}_{ES_{in_n}, ES_{out_n}} S'_n) \Leftrightarrow$$
$$\forall k \in 1..n \;.\; \forall D_j \hat{}\, e \boxed{\in} ES_{out_k} \;.\; D_j \in \cup_{i\in 1..n} D_i \Longrightarrow e \boxed{\in} ES_{in_j}$$

Other specific communication constraints may be defined in order to take different communication semantics into account. Broadcast communication can be defined using a syntax for multiple receivers (*e.g.*, $\{D_1, \ldots, D_n\} \hat{}\, e$) and refining the CC constraint to express the delivery of the event in all the concerned diagrams input collections at once. Synchronous communication can be expressed restricting the size of collections to one. (A)synchronous, binary and broadcast communication have been implemented in the xCLAP tool (see Section V). Events with expiry dates can be modelled tagging events with time-stamps. CC may then take these time-stamps into account. Systems with (message) transit durations can be designed in the same way or adding intermediate components whose role is to simulate the time duration of the transit.

F. Semantics of Evaluation Functions ($\triangleright_X$)

Several kinds of evaluation functions may be defined depending on which data specification language is used. We discuss here how these functions may be obtained for algebraic specifications and for state-oriented languages.

Algebraic specifications. Rewriting is chosen as the evaluation function for algebraic specifications. This choice is justified since it is suitable to an operational semantics, which accordingly enables us to remain in a pragmatic and executable context, and to design and implement tools. Algebraic specification languages often come with their rewriting system. Larch is equipped with a theorem prover, LP [17], which implements equational term rewriting. Specifications written in CASL [3] can be partially transformed into rewrite rules. This can be achieved executing CASL equational specifications within the ELAN rewrite engine [18].

State-oriented language. We present here the evaluation function for Z. Other languages such as B, VDM, Object-Z, or Alloy, may be taken into account following a similar process.

Z is a mathematical notation based on set theory and first order predicate calculus. Z schemas allow to structure data and operation specifications. A schema is made up of a declaration part (a set of typed variables) and a predicate part built on these variables. State schemas define state spaces. The semantics of a state schema is a set of bindings between the schema variables and values such that the predicate holds. A complete Z specification also has an initialisation schema which defines initial values for the variables.

The Z evaluation function is defined compiling Z specifications into LTSs. Let z be a Z specification defined with a state schema $SSch_z$, an initialisation schema $ISch_z$, and a set of operation schemas. The LTS $(INIT_z, STATE_z, TRANS_z)$ associated to the Z specification is obtained as follows. Its set of states ($STATE_z$) corresponds to the $SSch_z$ semantics state space. The set of initial states ($INIT_z$) of the LTS is the subset of $STATE_z$ with elements that satisfy the predicate of $ISch_z$. Finally, each operation schema predicate, used to relate the bindings of two states, defines a set of transitions labelled by operation applications. The set of transitions of the LTS ($TRANS_z$) is the union of all these transitions. The evaluation function $\triangleright_Z$ is then defined as: $E \vdash l \triangleright_Z s' \Leftrightarrow \exists s \subseteq E \;.\; s \xrightarrow{l} s' \in TRANS_z$. This formula states that the application l of an operation yields a state schema s' iff the schema s, on which this operation is applied and which is present in the evaluation environment E, is related to s' by l in the set of transitions $TRANS_z$.

G. Consistency and Completeness

In this section we prove that the set of dynamic rules is consistent and complete. This dynamic rules system (DRS) forms the core of the semantics aspects considered in Section III. As far as the system evolution is concerned, the dynamic rules are based on transition labels that appear in labelled transition

systems. We consider transition labels from a syntactic point of view *i.e.,* a set *Label* that contains all the syntactic forms of labels computed from TRANSITION in Table I.

In the following we use these syntactic forms of the labels to identify the different semantic rules considered for the evolution of state diagrams. We focus on the $DYN-E$ and $DYN-E\emptyset$ rules (Fig. 8 and 9) to deal with the labels which have the general forms: EVENT [GUARD] / ACTIONS and [GUARD] / ACTIONS. These rules are representative members of DRS since the other rules are specific forms of them. Each one of these rules covers a pair of cases that we would have if we omitted the GUARD (the *TRUE* value being the default one). We recall that each dynamic rule has the general form $\frac{H}{C}name$ where H is a set of premises and C, the consequent of the rule, is a conjunction of predicates.

Consistency. The DRS rule system is consistent iff its constituent rules are not contradictory and they all lead to a correct state of the evolving system (namely a diagram D). To establish the consistency we have to prove that there is only one dynamic rule that deals with a given evolution label and that the evolution captured by each dynamic rule leads to a correct state (a member of $\underline{STATE}(D)$).

Proof. We relate the dynamic (operational semantics) rules with the syntactic structure of the considered evolution labels. Therefore the rules are selected according to the occurrence of a specific evolution label, acting as a discriminant, in their hypothesis. This is formalised as follows:

$$\forall\, i,\ r_i = \tfrac{H_i}{C_i}\mathsf{n_i} \in DRS\ .\ \exists\, el \in Label,\ el \in H_i\ .\\ \forall\, j,\ j \neq i,\ r_j = \tfrac{H_j}{C_j}\mathsf{n_j} \in DRS\ .\ el \notin H_j$$

The premises of each DRS rule contain one specific evolution label of *Label*; hence the related rules $DYN-E$ and $DYN-E\emptyset$ cannot be enabled simultaneously to handle the evolution of a diagram D. Moreover, the consequences of all the dynamic rules in DRS are made up of a predicate with the form $s \xrightarrow{l} s' \in \underline{TRANS}(D)$; it expresses the evolution from a state s into another state s'. It follows that s' is in the range of $\underline{TRANS}(D)$, $\underline{STATE}(D)$, thus it is a correct state of D.

Completeness. The DRS rule system is complete iff at least one dynamic rule deals with each evolution label of the considered transition diagram. That is, for every label *el*: either *el* belongs to the premises of a rule (*i.e.,* $el \in H$) or *el* does not belong to them (*i.e.,* $el \notin H$).

Proof. In our formalisation of the dynamic rules, there is exactly one rule that deals with each label: $e(x_1 : T_1, \cdots, x_n : T_n)g/a_1;\ \cdots;\ a_m$ is part of the premises of $DYN-E$; in the same way, $g/a_1;\ \cdots;\ a_m$ is part of the premises of $DYN-E\emptyset$.

We cover with the rules $DYN-E$ and $DYN-E\emptyset$ (and their omitted special cases), all the evolution cases including the exchanges with the environment.

The last part of the completeness proof concerns the cases where there is no explicit evolution. The stuttering rule ($DYN-\epsilon$) captures this step; that means, the system stays in the correct current state.

IV. Application

In this section, our semantics for ESD are illustrated on the producer/consumer example we introduced in Section II.

We first have to choose a non-extended state diagram semantics, for example the semantics of UML 1.4 state diagrams given by Jürjens [12]. This semantics is based on Abstract State Machines (ASM), and formalises state diagrams as a set of active states which can evolve depending on the state of their queues. In particular, this evolution is given using two rules, `SCInitialize(D)` and `SCMain(D)`, where the later one consists of selecting the event to be fired, executing it, and then executing the rules for the internal actions. The formal interpretation of these actions is given by ASM rules as well. Our semantic framework applies on top of this semantics. Being given this non-extended state diagram semantics, we now have to define an instantiation $\lfloor$ of the $\boxed{\lfloor}$ function. It is defined inductively on the structure of the transition labels of the extended diagram notation (Fig. 11).

$$\begin{aligned}
&\lfloor\, event{-}name(x_1 : T_1, \ldots, x_n : T_n)\ guard\ /\ a_1;\ \ldots;\ a_m\\
&\quad = \lfloor_{event}\ event{-}name(x_1 : T_1, \ldots,\\
&\qquad\qquad x_n : T_n)\ /\ \lfloor_{action}\ a_1;\ \ldots;\ a_m\\
&\lfloor_{action}\ a_1;\ \ldots;\ a_n = \lfloor_{action}\ a_1;\ \ldots;\ \lfloor_{action}\ a_n\\
&\lfloor_{action}\ \varepsilon_{act} = \varepsilon_{act}\\
&\lfloor_{action}\ x := t = \varepsilon_{act}\\
&\lfloor_{action}\ receiver\hat{}\,event{-}name(t_1, \ldots, t_n)\\
&\qquad\qquad = receiver\hat{}\ \lfloor_{event}\ event{-}name(t_1, \ldots, t_n)\\
&\lfloor_{event}\ event{-}name(x_1 : T_1, \ldots, x_n : T_n) = event{-}name\\
&\lfloor_{event}\ event{-}name(t_1, \ldots, t_n) = event{-}name
\end{aligned}$$

Fig. 11. Definition of the $\lfloor$ function on labels for the [12] semantics

Event collections are instantiated by queues, together with their usual operations. Hence, each generic concept within our rules is instantiated by some queue-related concrete one (*Queue* type for $\boxed{\mathcal{Q}}$, *nil* for $\boxed{\emptyset}$, *queue* operator denoting the effect of constructors $\boxed{\uplus}$ and $\boxed{\setminus}$, $\in$ for $\boxed{\in}$ and $\subseteq$ for $\boxed{\subseteq}$).

The example imports modules written in Larch. Within this framework, the evaluation function corresponds to term rewriting, *i.e.,* $\rhd_{Larch\text{–}spec} \equiv \leadsto_R^*$ with R being the set of rewrite rules (oriented module axioms). $TRUE_{Larch\text{–}spec}$ corresponds to `true` in Larch.

As a first example of rule application, Fig. 12 denotes a simple dynamic evolution. This is an instantiation of the $DYN-E\emptyset$ rule (Fig. 9) without guard. It represents an independent evolution of diagram D3 from state s2 to state s1. The premises of the rule denote the different conditions to be fulfilled to compute the new extended state S' from the initial one S, and as a consequence the corresponding transition of the dynamic evolution semantic model. The `w` value bound to the `n` variable corresponds to its normal form after performing the two `update` calls, supposing its value before was `v`. Rewriting is performed through the $act-eval$ application, using rule $EVAL-SEQ$ for the sequence of actions and rule $EVAL-ASSIGN$ twice, first with premise $\{(\mathtt{n},\mathtt{v})\} \vdash \mathtt{update}(\mathtt{n},1,1) \leadsto_R^* \mathtt{w_0}$ and second with premise $\{(\mathtt{n},\mathtt{w_0})\} \vdash \mathtt{update}(\mathtt{n},2,1) \leadsto_R^* \mathtt{w}$.

$$CC(S_1 \xrightarrow{\varepsilon}_{queue(\texttt{comm}(1)),nil} S'_1, S_2 \xrightarrow{\varepsilon}_{queue(\texttt{comm}(1)),nil} S'_2, S_3 \xrightarrow{\varepsilon}_{nil,queue(\mathrm{D1}\texttt{\^{}comm}(1),\mathrm{D2}\texttt{\^{}comm}(1))} S'_3) \Longleftrightarrow$$
$$(\mathrm{D1}\texttt{\^{}comm}(1) \in queue(\mathrm{D1}\texttt{\^{}comm}(1), \mathrm{D2}\texttt{\^{}comm}(1)) \wedge \mathrm{D}_1 \in \cup_{i\in 1..3}\mathrm{D_i} \Longrightarrow \texttt{comm}(1) \in queue(\texttt{comm}(1))) \wedge$$
$$(\mathrm{D2}\texttt{\^{}comm}(1) \in queue(\mathrm{D1}\texttt{\^{}comm}(1), \mathrm{D2}\texttt{\^{}comm}(1)) \wedge \mathrm{D}_2 \in \cup_{i\in 1..3}\mathrm{D_i} \Longrightarrow \texttt{comm}(1) \in queue(\texttt{comm}(1)))$$

Fig. 15. Instantiation of the *CC* rule

$$\frac{T = (S_1 \xrightarrow{\varepsilon}_{queue(\texttt{comm}(1)),nil} S'_1, S_2 \xrightarrow{\varepsilon}_{queue(\texttt{comm}(1)),nil} S'_2, S_3 \xrightarrow{\varepsilon}_{nil,queue(\mathrm{D1}\texttt{\^{}comm}(1),\mathrm{D2}\texttt{\^{}comm}(1))} S'_3) \quad T \in \Pi_{i\in 1..3}\underline{TRANS}^{open}(D_i) \quad CC(T)}{T \in \overline{TRANS}(\cup_{i\in 1..3}\mathrm{D_i})}$$

Fig. 16. Example of transition in the global semantic model

Note that, as said earlier, the abstract concepts have been instantiated in the rules by concrete concepts, with, for example, *queue* being the *Queue* constructor.

The rule in Fig. 10 ($DYN - OPEN$) may be used to denote the effect of the external environment on D3, having both output events taken out of its output event queue (Fig. 13).

An example of construction of a global transition is then given in Fig. 14, 15 and 16. It describes the asynchronous communication on `comm` between D1, D2 and D3 (which sends the `comm` events). In this example, the diagrams D1 and D2 are initially in state s1 and the diagram D3 is in state s2.

Fig. 14 describes a conjunction of evolutions for the three diagrams where D3 has events taken out of its output queue (obtained in Fig. 13, using the rule in Fig. 10) whereas D1 and D2 have events put into their input queues (which could be obtained using the rule in Fig. 10 too). The conclusion of the rule builds a global transition which may pertain to the global semantic model. However, the communication constraint has not been checked yet.

Fig. 15 instantiates the *CC* communication constraint and demonstrates that the Fig. 14 asynchronous communication is possible due to the compatibility between open system transitions of the different diagrams. Here, *CC* is instantiated with $k = 3$. For $k = 1..2$, the ES_{out_k} are *nil* hence the remainder of the *CC* rule instantiation is empty.

Finally, Fig. 16 uses the last two results to show that a global transition, verifying the communication constraint, is defined in the final semantic model.

V. The xCLAP Tool

xCLAP (extended CLAP) [19] is a prototype animator for ESDs. So far, it restricts to flat ESDs, *i.e.*, ESDs where there are no concurrent (AND) states (concurrency and communication is dealt with between diagrams), no hierarchical (OR) states, and no entry/exit actions (these can be taken into account adding specific transitions to the diagrams in a preprocessing step). These choices have been made to devote more time to the interactions between behaviours and data types.

A. Principles

xCLAP builds on CLAP [14], [20] a class library for the description, product and reachability analysis of transition systems. xCLAP focuses on the interactive animation of ESDs as done for process algebras in tools such as CWB-NC [21] for CCS or CADP [22] for LOTOS. xCLAP may be used to perform the construction of LTSs from ESD specifications thus making it possible afterwards to perform model-checking or reachability analysis using CLAP features and its extension for abstract analysis techniques [23].

An overview of the way xCLAP works (from the user point of view) is given in Fig. 17. Class diagrams, modelled in the UML SMW tool [24] are used to model ESDs (a state diagram is associated to each ESD class), data modules (a documentation note is associated to each module class) and their relations. The SMW reification of MOF is then used to translate SMW models[1] into models which are read by xCLAP and animated.

User interface. xCLAP can be parameterised at runtime (see Fig. 18, left): import of ESDs, but also definition of frameworks and configuration of the operational semantics (synchronous or asynchronous communication, binary or n-ary communication, specific treatment of received variables). During the animation process (see Fig. 18, right), the user can see the local state of all diagrams, possible transitions and their effect, and then choose a transition to be executed. The interactions with data evaluation tools are completely transparent for the user.

Encoding of the semantics. xCLAP is written in the object-oriented Python language and organised in six packages: `ESD` (extension classes for CLAP related to ESD), `SIGNATURE` and `TYPING` (related to operations on data description files and typing/meta-typing), `EVALUATION` (evaluation mechanisms and interface with external tools), `ANIMATION` (operational semantics) and `UI` (user interface). The implementation of the operational semantics follows the structure of our sets of rules encoded in these different packages. xCLAP relies on a specific API of classes and methods which can be used to change or extend the semantics. For more technical information we refer to the xCLAP manual [19].

Term evaluation ($\triangleright$). Term evaluation is performed in xCLAP trough specific interface modules with external evaluation tools. The rewriting of algebraic terms is performed with LP which automatically transforms equations into rewrite rules (using the `dsmpos` and `noeq-dsmpos` registered orderings [17], [25]). Hence, xCLAP users do not have to give explicit ordering commands. The Z evaluation function we presented in Section III-F relies on ZANS [26], an animation tool for Z specifications which allows the evaluation of expressions and predicates. ZANS is used in xCLAP jointly with the ZTC type checker [27] which checks for syntactic and typing errors in Z specifications.

[1] A translation from XMI to xCLAP was also possible but less efficient.

$$\frac{\begin{array}{c} S \in \underline{STATE}(\mathrm{D3}) \\ S =< \Gamma_{D3}, \{(\mathtt{n},\mathtt{v})\}, nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) > \\ l = / \ \mathtt{n} := \mathtt{update}(\mathtt{n},1,1);\ \ \mathtt{n} := \mathtt{update}(\mathtt{n},2,1) \\ s2 \xrightarrow{l} s1 \in TRANS(\mathrm{D3}) \\ active(s2, \Gamma_{D3}) \\ \Gamma_{D3} \xrightarrow{\varepsilon_{act}} \Gamma'_{D3} \in \boxed{\downarrow TRANS}(\mathrm{D3}) \\ act{-}eval(\mathtt{n} := \mathtt{update}(\mathtt{n},1,1);\ \ \mathtt{n} := \mathtt{update}(\mathtt{n},2,1), \\ < \Gamma_{D3}, \{(\mathtt{n},\mathtt{v})\}, nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) >, \mathrm{D3}) =< \Gamma_{D3}, \{(\mathtt{n},\mathtt{w})\}, nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) > \\ S' =< \Gamma'_{D3}, \{(\mathtt{n},\mathtt{w})\}, nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) > \end{array}}{S' \in \underline{STATE}(\mathrm{D3}) \wedge S \xrightarrow{\varepsilon} S' \in \underline{TRANS}(\mathrm{D3})}$$

Fig. 12. Independent dynamic evolution, rule $DYN - E\emptyset$, for D3

$$\frac{\begin{array}{c} S =< \Gamma_{D3}, \{(\mathtt{n},\mathtt{v})\}, nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) > \\ S' =< \Gamma'_{D3}, \{(\mathtt{n},\mathtt{w})\}, nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) > \\ S \xrightarrow{\varepsilon} S' \in \underline{TRANS}(\mathrm{D3}) \\ queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) \subseteq queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) nil \subseteq \mathcal{P}(EVENT^?) \end{array}}{S \xrightarrow{\varepsilon}_{nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1))} < \Gamma'_{D3}, \{(\mathtt{n},\mathtt{w})\}, nil, nil > \in \underline{TRANS}^{open}(D3)}$$

Fig. 13. Instantiation of the $DYN - OPEN$ rule for D3

$$\frac{\begin{array}{c} S_1 =< \Gamma_{D1}, \emptyset, nil, nil > \\ S'_1 =< \Gamma_{D1}, \emptyset, queue(\mathtt{comm}(1)), nil > \\ T_1 = S_1 \xrightarrow{\varepsilon}_{queue(\mathtt{comm}(1)), nil} S'_1 \in \underline{TRANS}^{open}(D1) \\ S_2 =< \Gamma_{D2}, \emptyset, nil, nil > \\ S'_2 =< \Gamma_{D2}, \emptyset, queue(\mathtt{comm}(1)), nil > \\ T_2 = S_2 \xrightarrow{\varepsilon}_{queue(\mathtt{comm}(1)), nil} S'_2 \in \underline{TRANS}^{open}(D2) \\ S_3 =< \Gamma_{D3}, \{(\mathtt{n},\mathtt{v})\}, nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1)) > \\ S'_3 =< \Gamma'_{D3}, \{(\mathtt{n},\mathtt{w})\}, nil, nil > \\ T_3 = \ \ S_3 \xrightarrow{\varepsilon}_{nil, queue(\mathrm{D1}\hat{}\,\mathtt{comm}(1), \mathrm{D2}\hat{}\,\mathtt{comm}(1))} S'_3 \in \underline{TRANS}^{open}(D3) \\ T = (T_1, T_2, T_3) \end{array}}{T \in \Pi_{i \in 1..3} \underline{TRANS}^{open}(D_i)}$$

Fig. 14. Event exchange

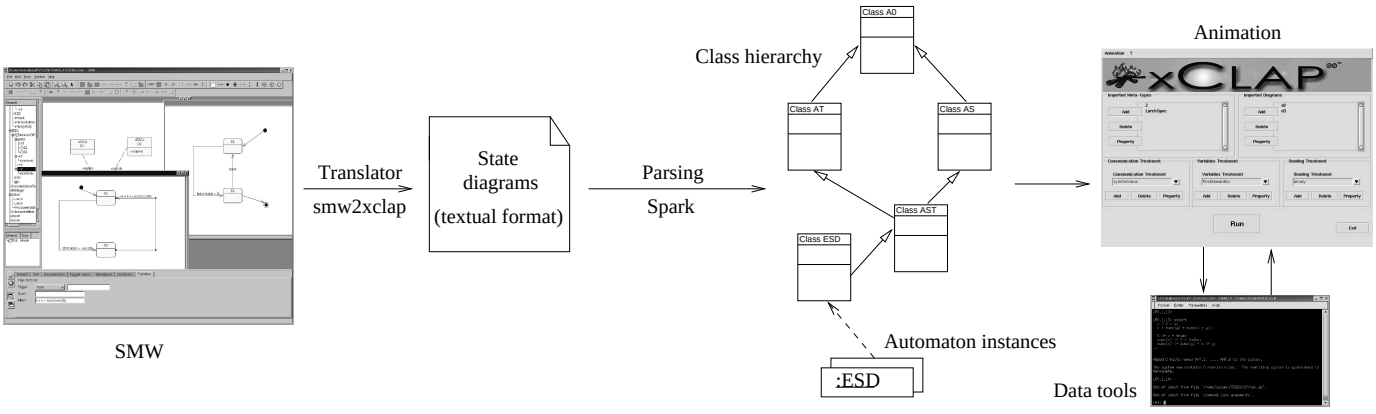

Fig. 17. xCLAP overview

B. Genericity and Extension Mechanisms

The generic elements of our semantics have been encoded using classes (some of them being abstract ones) with a specific API. These elements can be instantiated using subclassing and by overriding methods. In the sequel we briefly describe how different instantiations or extensions can be achieved, following the same order in which we presented our rules.

Data types and term evaluation ($\triangleright$). Changing an evaluation tool, *e.g.*, using ELAN [28] instead of LP to rewrite

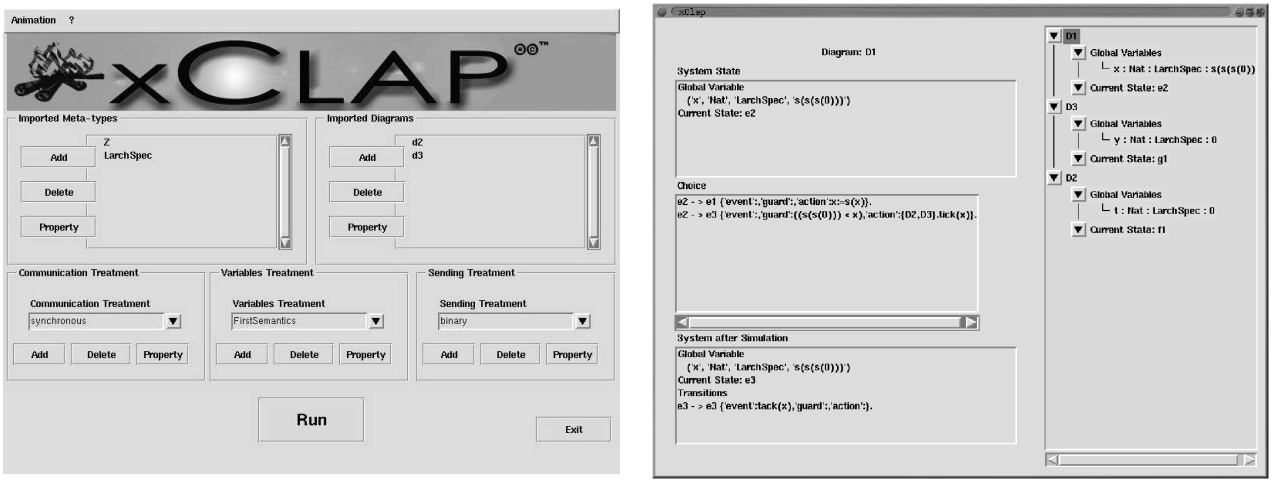

Fig. 18. xCLAP windows (left: parameters, right: animation)

algebraic terms, is possible with a limited impact on the xCLAP code (the tool call in the corresponding evaluation interface module has to be changed). Taking into account a new data framework, *e.g.*, B, is done as follows. A class has to be defined in the `TYPING` package to be able to scan and parse B expressions. A class is also added to the `SIGNATURE` package to define mechanisms for parsing signatures out of a B module. The evaluation of B expressions is achieved adding a class in the `EVALUATION` package that serves as a front-end to an external tool for the effective evaluation.

Collection related operators. Collection policies (the interpretation of creating, adding, removing, etc.) are enforced using collection classes. The encoding of the semantics only addresses the collections through this API. To change a collection policy, once a new collection class has been defined, an instance of it has to be passed to the constructor of `AnimatedDiagram`, the class that represents the ESD under animation (mainly its extended states, denoted ***STATE*** in the semantics).

Meta-typing and action semantics. The set of meta-typing rules has not to be changed. The only possible open point in action semantics (*EVAL*− rules) is the change in the collection policy through a change in the `AnimatedDiagram` constructor call when running xCLAP (see above).

Dynamic rules and communication semantics. Dynamic rules are dealt with locally (*DYN* − *E* and *DYN* − *E*0 rules) by the `LocalAnimator` class (it builds local transitions for a given ESD) and globally (*DYN* − *OPEN* and *CC* rules) by the `OverallAnimator` class (it builds global transitions from local ones). `LocalAnimator` and `OverallAnimator` are parameterised by instances of three subclasses of abstract classes reifying (i) the communication mode (synchronous/asynchronous), (ii) the binding mode (binary/n-ary), and (iii) the treatment of reception variables. Different semantics can be taken into account by defining new subclasses for the three abstract classes. This principle has been already used and different communication semantics can be taken into account in xCLAP through its parameterising window (see above, user interface). The last open point is the encoding of the *CC* rule in the `OverallAnimator` class. Changing the *CC* rule can be achieved also by sub-classing the classes which parameterise `OverallAnimator`.

VI. Related Work

In this section, we compare our approach with existing works on the combination of state diagrams with data description languages.

State diagrams and OCL. The frequent extension of state diagrams with data types is the use of OCL (Object Constraint Language) constraints. OCL is a complement to the description of data types using class diagrams but not a real language for the abstract description of data types. Nevertheless, several works are interested in its formalisation [29], [30].

State diagrams and state-oriented specifications. There are numerous works combining state diagrams with Z [31], [32] or with B [33]–[35]. In both cases, a translation into a static formalism (Z or B) is used which thereafter constitutes an homogeneous framework for the subsequent steps of the formal development. Laleau and Polack [36] addressed the issue to have a two-way translation process. The RoZ approach [32] has been combined with state diagrams to address the static / dynamic consistency issue. This is an approach complementary to ours. However, our focus is more on the dynamic part of systems (operational semantics), and we try to be more general as far as the static part is concerned (Z may be used but algebraic specifications as well). The main difference between our approach and these approaches using Z or B is that we try to use the different semantics of the dynamic and the static parts without translating one into the other.

State diagrams and algebraic specifications. In [37], a conceptual framework to plug data description languages into paradigm-specific ones is presented. The authors illustrate their approach with the Casl-Chart formalism [38] which combines Statecharts (following the STATEMATE semantics [8]) with the CASL algebraic specification language. A Casl-Chart specification is made up of data types written in CASL and

several Statecharts that may use algebraic terms written from these data types in events, guards and actions. The semantics of the combined language is given in terms of the semantics of the two basic languages. Our work is therefore close to Casl-Chart, but our approach is more flexible at the dynamic specification level since, more than the integration of algebraic specification into a given semantics of state diagrams, we propose a reusable semantic framework for the integration of formal data types within dynamic formalisms. Casl-Chart deals with the full expressiveness of Statecharts as in our semantics (using configurations). xCLAP can only deal with flat ESD. However, tools for Casl-Chart are, as far as we know, limited to the CASL ones, *i.e.*, parsers, type checkers and translators into high order logic theorem provers.

All the works mentioned above enable one to use only a single data description language. They are also less flexible for the dynamic aspect since our approach has been defined to take into account various state diagram semantics. We also have better reuse possibilities for our modules which can be imported in other application contexts.

VII. Conclusion

The separate design of concern models is a way to tackle the complexity of systems and promote the reusability of these models. Yet, it requires the definition of formally grounded techniques for their integration. In this article we have proposed a semantic framework for the integration of static and dynamic aspects through the extension of state diagrams with formal data types. This joint use of a graphical notation with formal languages enables one to take advantage of both approaches.

The proposed framework is generic in the sense that it can deal with various state diagrams semantics and different static specification languages provided that the first one is given in terms of (some form of) LTS and that the latter support the definition of term evaluation mechanisms. Genericity is achieved through abstract semantic rules and abstract concepts used in their formalising. Rules can be constrained to deal with specific semantics, *e.g.*, communication rules may describe different communication models between diagrams. Interaction between ESDs through external descriptions (sequence diagrams or synchronised products) has been experimented [39] and fully integrated into the framework.

The use of an operational semantics enabled the design and implementation of xCLAP, a prototype tool for the animation of our integrated formalism. Its UML graphical front-end and its back-end enable specifiers to exploit specification and animation in a user-friendly way even if by now it presents restrictions on the structure of ESDs taken as input (flat state diagrams). As in [40], the use of flattening algorithms as a preliminary step is a possible solution to this limitation. As far as validation and verification are concerned, models can be animated but also verified using model checking techniques, using xCLAP as a preprocessor to build a global state-space LTS and verifying it afterwards with dedicated tools such as the CADP tool-box. A main drawback of integrated formalisms is the state explosion which may appear when integrating data types within behaviours. Specific abstract analysis techniques have to be developed for such integrated models [23]. Another solution is to verify concern models independently. ESDs can be translated into specific tools input languages abstracting away from the data types. The data types can be verified using theorem provers or tools dedicated to the chosen static languages. This approach has been followed in a more complex case study where Z data types have been type-checked using Z/EVES [15]. A remaining issue is the integration of verification results for the separate models into global results for the whole system.

References

[1] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing, *Larch: Languages and Tools for Formal Specification*, ser. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[2] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, Eds., *Algebraic Foundations of System Specification*. Springer-Verlag, 1999.

[3] P. D. Mosses and M. Bidoit, *CASL — the Common Algebraic Specification Language: User Manual*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2004, vol. 2900.

[4] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall International Series in Computer Science, 1992.

[5] J. R. Abrial, *The B-Book*. Cambridge University Press, 1996.

[6] OMG, "UML Superstructure Specification, v2.0," Aug. 2005, document formal/05-07-04.

[7] D. Harel, "Statecharts: A Visual Formalism for Complex System," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[8] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, 1996.

[9] J. Lilius and I. Porres, "Formalising UML State Machines for Model Checking," in *Proc. of the International Conference on the Unified Modelling Language: Beyond the Standard (UML'99)*, ser. Lecture Notes in Computer Science, R. France and B. Rumpe, Eds., vol. 1723. Springer-Verlag, 1999, pp. 430–445.

[10] D. Latella, I. Majzik, and M. Massink, "Towards a Formal Operational Semantics of UML Statechart Diagrams," in *Proc. of the IFIP TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, P. Ciancarini and R. Gorrieri, Eds. Kluwer Academic Publishers, 1999, pp. 331–347.

[11] M. van der Beeck, "Formalization of UML-Statecharts," in *Proc. of the 4th International Conference on the Unified Modelling Language (UML'01)*, ser. Lecture Notes in Computer Science, M. Gogolla and C. Kobryn, Eds., vol. 2185. Springer-Verlag, 2001, pp. 406–421.

[12] J. Jürjens, "A UML Statecharts Semantics with Message-Passing," in *Proc. of the 17th ACM Symposium on Applied Computing (SAC'02)*. ACM, 2002, pp. 1009–1013.

[13] C. Attiogbé, P. Poizat, and G. Salaün, "Integration of Formal Datatypes within State Diagrams," in *Proc. of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE'03)*, ser. Lecture Notes in Computer Science, M. Pezzè, Ed., vol. 2621. Springer-Verlag, 2003, pp. 341–355.

[14] C. Choppy, P. Poizat, and J.-C. Royer, "The Korrigan Environment," *Journal of Universal Computer Science*, vol. 7, no. 1, pp. 19–36, 2001, special issue on Tools for System Design and Verification.

[15] C. Attiogbé, P. Poizat, and G. Salaün, "Specification of a Gas Station using a Formalism Integrating Formal Datatypes within State Diagrams," in *Proc. of the 8th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'03)*, ser. IEEE Computer Society Press, France, 2003.

[16] M. Broy and E.-R. Olderog, *Trace-Oriented Models of Concurrency*, ser. Handbook of Process Algebra. Elsevier, 2001, ch. 2, pp. 101–195.

[17] S. J. Garland and J. V. Guttag, "A Guide to LP, the Larch Prover," Palo Alto, California," Technical Report, 1991.

[18] H. Kirchner and C. Ringeissen, "Executing CASL Equational Specifications with the ELAN Rewrite Engine," November 2000, coFI note T-9, `http://www.daimi.au.dk/~pdm/Common/Notes/T-9/`.

[19] A. Auverlot, C. Cailler, M. Coriton, V. Gruet, and M. Noël, "xCLAP: Animation of State Diagrams with Formal Data, Master's Degree Project, University of Nantes. Tool and documentation available on G. Salaün's webpage." 2003, directed by C. Attiogbé and G. Salaün.

[20] G. Nedélec, M. Papillon, C. Piedsnoirs, and G. Salaün, "CLAP: a Class Library for Automata in Python, Master's Degree Project, University of Nantes. Tool and documentation available on G. Salaün's webpage." 1999, directed by M. Allemand and P. Poizat.

[21] R. Cleaveland, T. Li, and S. Sims, *The Concurrency Workbench of the New Century (Version 1.2)*, Department of Computer Science, North Carolina State University, 2000.

[22] H. Garavel, F. Lang, and R. Mateescu, "An Overview of CADP 2001," *EASST Newsletter*, vol. 4, pp. 13–24, 2001, also available as INRIA Technical Report RT-0254.

[23] P. Poizat, J.-C. Royer, and G. Salaün, "Bounded Analysis and Decomposition for Behavioural Descriptions of Components," in *Proc. of the Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, ser. Lecture Notes in Computer Science, vol. 4037. Springer-Verlag, 2006, pp. 33–47.

[24] R.-J. Back, D. Björklund, J. Lilius, L. Milovanov, and I. Porres, "A Workbench to Experiment on New Model Engineering Applications," in *Proc. of The Unified Modeling Language, Modeling Languages and Applications Conference (UML'2003)*, ser. Lecture Notes in Computer Science, vol. 2863. Springer-Verlag, 2003, pp. 96–100.

[25] N. Dershowitz, "Orderings for Term-Rewriting Systems," *Theoretical Computer Science*, vol. 17, no. 3, pp. 279–301, 1982.

[26] X. Jia, *A Tutorial of ZANS*, DePaul University, 1998.

[27] ——, *ZTC: A Type Checker for Z Notation (User's Guide)*, DePaul University, 1998.

[28] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen, "An Overview of ELAN," in *International Workshop on Rewriting Logic and its Applications*, ser. Electronic Notes in Theoretical Computer Science, C. Kirchner and H. Kirchner, Eds., vol. 15. Elsevier Science, 1998. [Online]. Available: http://www.elsevier.com/locate/entcs/volume15.html

[29] M. V. Cengarle and A. Knap, "A Formal Semantics for OCL 1.4," in *Proc. of the 4th International Conference on the Unified Modelling Language (UML'01)*, ser. Lecture Notes in Computer Science, M. Gogolla and C. Kobryn, Eds., vol. 2185. Springer-Verlag, 2001, pp. 118–133.

[30] T. Clark and J. Warmer, Eds., *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, ser. Lecture Notes in Computer Science, vol. 2263. Springer-Verlag, 2002.

[31] R. Büssow and M. Weber, "A Steam-Boiler Control Specification with Statecharts and Z," in *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler*, ser. Lecture Notes in Computer Science, J.-R. Abrial, E. Börger, and H. Langmaack, Eds. Springer-Verlag, 1996, vol. 1165, pp. 109–128.

[32] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud, "An Overview of RoZ: A Tool for Integrating UML and Z Specifications," in *Proc. of the Advanced Information Systems Engineering Conference (CAiSE'00)*, ser. Lecture Notes in Computer Science, B. Wangler and L. Bergman, Eds., vol. 1789. Springer-Verlag, 2000, pp. 417–430.

[33] E. Sekerinski and R. Zurob, "Translating Statecharts to B," in *Proc. of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, ser. Lecture Notes in Computer Science, M. Butler, L. Petre, and K. Sere, Eds., vol. 2335. Springer-Verlag, 2002, pp. 128–144.

[34] K. Lano, K. Androutsopoulos, and P. Kan, "Structuring Reactive Systems in B AMN," in *Proc. of the 3rd IEEE International Conference on Formal Engineering Methods (ICFEM'00)*. IEEE Computer Society Press, 2000, pp. 25–34.

[35] H. Ledang and J. Souquières, "Contributions for Modelling UML StateCharts in B," in *Proc. of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, ser. Lecture Notes in Computer Science, M. Butler, L. Petre, and K. Sere, Eds., vol. 2335. Springer-Verlag, 2002, pp. 109–127.

[36] R. Laleau and F. Polack, "Coming and Going from UML to B: A Proposal to Support Traceability in Rigorous IS Development," in *Proc. of the 2nd International Z and B Conference (ZB'02)*, ser. Lecture Notes in Computer Science, D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, Eds., vol. 2272. Springer-Verlag, 2002, pp. 517–534.

[37] E. Astesiano, M. Cerioli, and G. Reggio, "Plugging Data Constructs into Paradigm-Specific Languages: Towards an Application to UML," in *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, ser. Lecture Notes in Computer Science, T. Rus, Ed., vol. 1816. Springer-Verlag, 2000, pp. 273–292.

[38] G. Reggio and L. Repetto, "Casl-Chart: A Combination of Statecharts and of the Algebraic Specification Language CASL," in *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, ser. Lecture Notes in Computer Science, T. Rus, Ed., vol. 1816. Springer-Verlag, 2000, pp. 243–257.

[39] G. Salaün and P. Poizat, "Interacting Extended State Diagrams," in *Proc. of the Int. Workshop on Semantic Foundations of Engineering Design Languages (SFEDL'04)*, ser. ENTCS, vol. 115, 2005, pp. 49–57.

[40] A. David, M. O. Möller, and W. Yi, "Formal Verification of UML Statecharts with Real-Time Extensions," in *Proc. of the International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, ser. Lecture Notes in Computer Science, R.-D. Kutsche and H. Weber, Eds., vol. 2306. Springer-Verlag, 2002, pp. 218–232.

Christian Attiogbé received a PhD in Computer Science from the University of Toulouse III in 1992. His is currently associate professor at the University of Nantes, France and researcher at the LINA Laboratory were he is the leader of the Dependable Components and Systems team. His research interests include formal methods for dependable system development. He is engaged in research that includes formal methods integration, multi-paradigm specifications, the B Method, the combination of theorem proving and model checking for multi-facet analysis of concurrent, distributed and reactive systems.

Pascal Poizat is associate professor at the University of Evry and invited researcher in the ARLES project at INRIA, France. He received a PhD in Computer Science from the University of Nantes, France, in 2000. His PhD topics were the integration of static and dynamic aspects in formal description languages, the use of symbolic transition systems and the development of expressive modal logic gluing mechanisms. His current research interests include formal models and verification techniques for component and service based systems, more specifically issues related to coordination and adaptation.

Gwen Salaün is associate researcher in the VASY project at INRIA, France. He received a PhD in Computer Science from the University of Nantes, France, in 2003. His PhD topic was the study of different integration and verification techniques for heterogeneous formal specification languages. He spent one year at the University of Rome "La Sapienza", Italy, addressing the application of formal methods to Web Services design and verification. At INRIA, he has been working on bridges between process calculi, and on the verification of asynchronous circuits and architectures. His research interests include issues related to formal methods and software engineering, specification integration, coordination and adaptation.