Integration of Formal Datatypes within State Diagrams

Christian Attiogbé*, Pascal Poizat**, and Gwen Salaün*

*IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, 44322 Nantes Cedex 3, France
email: {attiogbe,salaun}@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/perso/permanents/salaun/

**LaMI - UMR 8042 CNRS et Université d'Evry Val d'Essonne, Genopole
Tour Évry 2, 523 Place des terrasses de l'Agora, 91000 Évry, France
email: poizat@lami.univ-evry.fr
http://www.lami.univ-evry.fr/~poizat/

Abstract. In this paper, we present a generic approach to integrate datatypes expressed using formal specification languages within state diagrams. Our main motivations are (i) to be able to model dynamic aspects of complex systems with graphical user-friendly languages, and (ii) to be able to specify in a formal way and at a high abstraction level the datatypes pertaining to the static aspects of such systems. The dynamic aspects may be expressed using state diagrams (such as UML or SDL ones) and the static aspects may be expressed using either algebraic specifications or state oriented specifications (such as Z or B). Our approach introduces a flexible use of datatypes. It also may take into account different semantics for the state diagrams.

Keywords: Formal Methods Integration, State Diagrams, Algebraic Specifications, Z, B.

1 Introduction

The joint use of formal and semi-formal specification languages is a promising approach, with the objective of taking advantage of both approaches: specifier-friendliness and readability from semi-formal approaches, high abstraction level, expressiveness, consistency and verification means from formal approaches. In this paper, we propose an approach dealing with this issue. It enables one to specify the different aspects of complex systems using an integrated language.

Static and functional aspects are specified using static formal specification languages (algebraic specifications [6], state oriented languages such as Z [24] or B [1]). This makes the verification of specifications possible but also the datatypes description at a very high abstraction level. The flexibility we propose at the static aspects specification level enables the specifier to choose the formal language that is the more suited to this task: either the one (s)he knows well, the one with tools, or the one that makes possible the reuse of earlier specifications.

Dynamic aspects (*i.e.* behaviour, concurrency, communication) are modelled using state diagrams. Our proposal is generic. Different dynamic semantics may be taken into account, hence our approach may be used for Statecharts [17], for the different

(yet growing number) of UML state diagrams semantics, [21, 20, 25, 19] for instance, and more generally for any state / transition oriented specification. In our approach, the specification is control-driven: the dynamic aspects are the main aspects within a specification and state how the static aspects datatypes are used. On a larger scale, our work deals with the formal specifications integration and composition issues, where we yet have some general results [3]. At the global specification level, our approach therefore also addresses the consistency of the static and dynamic parts.

This paper is structured as follows. In Section 2, we present the syntactic extensions used to integrate formal datatypes within state diagrams. Section 3 deals with the generic integration semantics. To end, Section 4 concludes the paper and presents some perspectives.

2 Syntax

We here present the syntactic extensions we add to state diagrams to take into account the formal datatypes integration. We advocate for a control-driven approach of mixed specification. This means that the main part of a specification is given by the dynamic aspects modelling (behaviours and communications). The static aspects are encapsulated within the dynamic aspects.

We first add module[1] importation and local variables declaration extensions to state diagrams. Both are done using *data boxes* (Fig. 1), a kind of UML-inspired note which is usually used to give additional informations to a diagram in a textual form. The `IMPORT` notation is introduced to indicate which data modules are imported as well as the language used to write their contents (algebraic specifications, Z schemas, B machines, or other). Such a language is called a *framework* in our approach and is used to define the specific evaluation functions which enable us to evaluate the data embeddings into state diagrams. Variables are also declared and typed in the data boxes. Since modules often contain several type definitions, and since types with the same name may be defined in two different modules, the type of a variable may be prefixed with the name of a module in order to avoid conflicts.

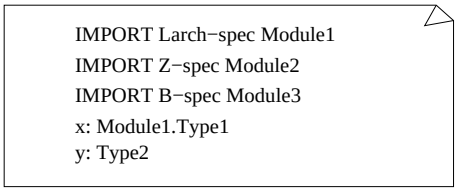

Fig. 1. Local declarations of data into state diagrams

The general form of a state diagram transition is made up of two states (which can contain activities) and a label with an event, a guard and an action to do when the transi-

[1] A module (general definition) is a collection of one or several descriptions of datatypes or behaviours.

tion is triggered. Datatype extensions (*data expressions*) may appear in both states and transitions. In states, our extensions correspond to activities (actions such as entry/exit or local modifications, and events). In transitions, our extensions (Tab. 1) enable one to (i) receive values in events and then store these values into local variables, (ii) guard transitions with data expressions, (iii) send events containing data expressions and (iv) make assignments of data expressions to the local variables. The last two take place in the action part of transitions. Possible extended activities within states are not directly taken into consideration in our approach. However, such activities can be viewed as a specific case of transitions between states.

Part of label	Kind of interaction	Example with data
EVENT	reception	*event-name*$(x_1{:}T_1,\ldots,x_n{:}T_n)$
GUARD	guard	*predicate*$(t_1,\ldots,t_n)$
ACTION	emission	*receiver* $\hat{}$ *event-name*$(t_1,\ldots,t_n)$
ACTION	local modification	$x{:=}t$

Table 1. Links between state diagrams and data

Data expressions may be either variables, terms for algebraic specifications or operation applications for state oriented specifications. As far as the formal language for the static aspects is concerned, the only constraint is to have some well-defined evaluation mechanism. The reason for this is that we are interested in operational semantics to address in a generic way operational issues of mixed specifications (animation, equivalences/bisimulations, model checking) which have been addressed for specific mixed specification languages (*e.g.* LOTOS [11]). Denotational issues may be treated with denotational approaches based on institutions for example [16, 2] and would enable to ease this evaluation mechanism constraint. Our approach makes possible the joint use of several static formal languages at the same time. However, a strong mix of constructs from several languages (such as importing a Z module within an algebraic specification, or using algebraic specification variables in a Z operation application) is prohibited in order to avoid possible semantic inconsistencies. Our goal is neither to advocate for such complex combinations, nor to develop a mean to solve such inconsistencies. As a simple way to detect them, we develop a *meta-type* concept using meta-typing rules (see the semantics below). Terms which are not meta-typed (*i.e.* inconsistent) will not be able to be used in the dynamic rules.

3 Semantics

In this section, our goal is to give a formal semantics to state diagrams extended with formal datatypes in a way that has been presented in the syntactic part. We do not aim at formalizing some specific kind of state diagram, which has already successfully been done, see [21, 20, 25, 19, 17] for example. We rather aim at being able to reuse different existing state diagram semantics. Therefore, our semantics is presented in such a way that generic concepts may then be instantiated for a specific kind of state diagram. In

our semantics we will deal with properties such as "the event pertains to the input event collection of the state diagram", which could thereafter be instantiated for a specific state diagram language into "the event is the first element of the input queue associated with the state diagram process". Such generic concepts are represented in our semantics using boxes (*e.g. event* $\boxed{\in}\ \mathcal{Q}_{in}$).

Using this generic approach, we preserve a very general description of extended state diagrams. In this way, all kinds of state diagrams and their underlying semantics could be considered. The only constraint is that the semantics of a non-extended state diagram *D* has to be given in terms of a Labelled Transition System (LTS) ($\boxed{INIT}(D)$, $\boxed{STATE}(D)$, $\boxed{TRANS}(D)$), which is currently always the case. We may then define precisely the meaning of the integration of data into dynamic diagrams using *extended states* evolutions and *evaluation functions*.

Our semantics is given using rules which may be decomposed into: meta-typing rules (badly meta-typed terms may not be evaluated), action evaluation rules (describing the effects of actions on extended states), dynamic rules (formalizing the individual evolution of an extended state diagram), open systems rules (describing the effect of putting extended state diagrams within an external environment), and global system and communication rules (putting things altogether). We will also discuss the evaluation functions associated with the different static specification languages one may use. To end this section on the semantics of our approach, we will illustrate the application of our semantic rules on a simple piece of specification. Specific notations are introduced throughout the semantic definitions.

Meta-Typing Rules. These 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 its meta-type (*e.g.* Larch, CASL, Z, B). In the following, $\mathcal{D}$ corresponds to the set of extended state diagrams. Within the rules we are in the context of a specific diagram *D* pertaining to $\mathcal{D}$. The states of *D* are denoted by $STATE(D)$, its initial states by $INIT(D)$ (which is a subset of $STATE(D)$), and its transitions by $TRANS(D)$. $DeclImp(D)$ and $DeclVar^{!}(D)$ denote respectively the imports and the variables declarations of the diagram *D* data box. $DeclVar^{?}(D)$ is the set of (typed) variables received in events. *DeclVar* 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* for 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.* $\triangleright_B$ denotes the B evaluation function). The meta-typing rules are given in Figure 2.

The Figure 2.*a* rule is used to give a meta-type to variables using data local declarations. The Figure 2.*b* rule 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 syntactically there are some differences between algebraic and state oriented formal specification languages.

$$\frac{\texttt{IMPORT}\ X - SPEC\ M \in DeclImp(D) \quad def(T, M) \quad x : T \in DeclVar(D)}{x ::_D X}\ (a) \qquad \frac{\texttt{IMPORT}\ X - SPEC\ M \in DeclImp(D) \quad def(op, M) \quad \forall\, i \in 1..n\ .\ t_i ::_D X}{op\ \ t_1\ \ \ldots\ \ t_n ::_D X}\ (b)$$

Fig. 2. Meta-typing rules

Action Evaluation Rules. This set of rules deals with the effect of actions on the extended states used to give semantics to extended state diagrams. Let us first give a definition of these states. $EVENT^?$ is the set of all *input events*, whose general form is $event-name(value_1, ..., value_n)$, that is a concrete instantiation with values of an abstract event parameterized 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 extended state diagrams 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[2], $\boxed{\mathcal{Q}}[EVENT^?]$ the set of input events collections, and $\boxed{\mathcal{Q}}[EVENT^!]$ the set of output events collections.

Collections are introduced to memorize events exchanged between diagrams. Q_{in_D} (resp. Q_{out_D}) is used to denote a collection associated to diagram D to store input (resp. output) events. The $E \vdash e \rhd_X v$ notation means that using the evaluation defined in the X framework, v is a possible evaluation of e using the environment E for substituting the free variables. More details concerning the semantics of $\rhd_X$ will be given in the remainder. 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 Figure 3.

The $EVAL - SEQ$ rule is used to evaluate the actions in sequence. $a_1 \ldots a_n$ is an abstract notation for a sequence of actions as we do not wish to set here some particular concrete syntax for these. The $EVAL - NIL$ rule states that doing no action does not

[2] A collection is an abstract structure which may be instantiated for a given type of state diagram by a queue, a set or a multiset for example.

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

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

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

Fig. 3. Action evaluation rules

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[3] into the state diagram output event collection.

The $EVAL-ASSIGN$ rule may need some explanations. Roughly speaking, assignments update the local environment. The understanding of this rule in an algebraic specification framework is quite natural (v is an interpretation of t). For state oriented languages, the interpretation of the rule is slightly different. x is a state variable. The value v denotes the new state obtained from the environment and after the evaluation of t. The notation used in this rule is specific to our approach and slightly different of the usual notation (*i.e.* pointed or with side effect). However, we wish to have at our disposal a common notation for the different considered languages.

Dynamic Rules. This set of rules deals with the dynamic evolution of a single state diagram. We introduce a special event, ε denoting (as usual) a stuttering step.

$$EVENT^{?+} = EVENT^{?} \cup \{\varepsilon\}$$

The state diagram evolutions are given in terms of a 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 extended state diagrams 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 a LTS ($\boxed{INIT}(D)$, $\boxed{STATE}(D)$, $\boxed{TRANS}(D)$). For some kind of state diagrams, there is a correspondence of the notion of states (*i.e.* $INIT$,

[3] $\boxed{\uplus}$ denotes an abstract union operation which may be instantiated differently depending on the type of state diagram semantics we want: union, put in front of a queue, etc.

STATE, $\boxed{INIT}$ and $\boxed{STATE}$ have the same type). However, for the others (such as the UML state diagrams), the semantics of non-extended state diagrams are given in terms of *configurations* [21, 20, 25] which are sets of *active* states. The *active* function is used to know if a state is active in a configuration (or more generally in some element of $\boxed{STATE}$). $active(\gamma, \Gamma)$ yields true if γ is active in Γ.

To be able to reuse the semantic information yield by $\boxed{TRANS}(D)$, we have first to define a function that maps members of $TRANS(D)$ (extended transitions) to members of $\boxed{TRANS}(D)$. This function, *base*, is defined inductively on the structure of the extended diagram notation [8]. A first rule (Fig. 4) 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 ($\boxed{\emptyset}$).

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

Fig. 4. Initialization rule

The meta-type of variables is used to define the notion of type in terms of a specific framework, which is noted $v :_X T$. Hence, the notation $\exists\, v :_X T$ denotes the fact that, within the X framework, v is a value of type T. A second rule (Fig. 5) is used to express stuttering steps. These steps denote an extended state diagram doing no evolution and will be used when putting state diagrams in an open system environment.

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

Fig. 5. Stuttering steps rule

The next dynamic rule (Fig. 6) expresses the general evolution triggered when an event is read from the state diagram input event collection. This event may carry data values that are put into the state diagram variables environment. $TRUE_X$ denotes the truth value within the X framework.

However, sometimes there are no events to trigger transitions. Such a case is dealt with by Figure 7 rule, where the corresponding transition of $\underline{TRANS}(D)$ is labelled by the stuttering step label (ε).

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

Fig. 6. Basic dynamic rule (event reception)

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. $e \boxed{\setminus} \mathcal{Q}$ denotes, in the same way, the (abstract) removal of e out of $\mathcal{Q}$. $\boxed{TRANS}(D)$ is the set of transitions of the non-extended state diagram semantics. The $DYN-E$ and $DYN-E\emptyset$ rules deal with the more general forms of state diagrams transitions (*i.e.* with the EVENT [GUARD]/ACTION and [GUARD]/ACTION 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 easily obtained from this model associating to D its LTS ($\underline{INIT}(D)$, $\underline{STATE}(D)$, $\underline{TRANS}(D)$), and then using an usual trace semantics (TR) for example:

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

Open System Rules. This set of rules is used to express what happens when a single state diagram is put into an open system environment. Mainly, the intuition we want to express is that some events may be received from the environment and some other may be send to it. As far as the input and output event collections of the state diagrams are concerned, this means that things are put into the input collection and things are taken out of the output collection. Something important to note is that these modifications of the extended states may appear meanwhile 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

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

Fig. 7. Basic dynamic rule (no event reception)

represent this, we may use the ε transitions (rule $DYN-\varepsilon$). Let us now formalize this intuition. As usual in our approach, the semantics of a state diagram in an open system will be defined as the traces of the LTS $(\underline{INIT}^{open}(D), \underline{STATE}^{open}(D), \underline{TRANS}^{open}(D))$, that is:

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

For a given diagram D, we state that:

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

with functions *SOURCE* and *TARGET* respectively denoting the source and target extended states of transitions. A unique rule, $DYN-OPEN$ (Fig. 8), is then defined to express the collection modifications 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 collections modification: nothing (rule $DYN-\varepsilon$) or a classical transition (rules $DYN-E$ and $DYN-E\emptyset$).

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

Global System and Communication. We may now define the last set of rules, putting things altogether. The idea is that a global system made up of several extended state diagrams (denoted $\cup_{i \in 1..n} D_i$) will evolve as its component evolve. Once again we will define the operational semantics of the system to be the traces of a 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:

$$\frac{\begin{array}{c} < \Gamma, E, Q_{in}, Q_{out} > \xrightarrow{l} < \Gamma', E', Q'_{in}, Q'_{out} > \in \underline{TRANS}(D) \\ \exists\, E_{out} \subseteq Q_{out} \\ \exists\, E_{in} \subseteq \boxed{\mathcal{P}}(EVENT^{?}) \end{array}}{< \Gamma, E, Q_{in}, Q_{out} > \xrightarrow{l}_{E_{in}, E_{out}} < \Gamma', E', Q'_{in} \boxed{\uplus} E_{in}, Q'_{out} \boxed{\setminus} E_{out} > \in \underline{TRANS}^{open}(D)} \; DYN\text{–}OPEN$$

Fig. 8. Open dynamic rule

$$|| \cup_{i \in 1..n} D_i \; ||^{open}_{oper} = TR(\overline{INIT}(\cup_{i \in 1..n} D_i), \overline{STATE}(\cup_{i \in 1..n} D_i), \overline{TRANS}(\cup_{i \in 1..n} D_i))$$

We now have to define $\overline{INIT}$, $\overline{STATE}$ and $\overline{TRANS}$.

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

$\overline{STATE}$ is obtained from initial states (*INIT*) and reachable states (target states of transitions). $\overline{TRANS}$ 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 E_{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 (E_{in_j}).

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

Other specific communication contraints may be defined in order to take different communication semantics into account.

Semantics of $\triangleright_X$. Several kinds of evaluation functions may be defined depending on which data specification language is used. In this paper, we focus on state oriented languages and especially on Z. Comprehensive explanations concerning algebraic specifications and B are reported in [8].

Z [24] is a mathematical notation based on set theory and first order predicate calculus. Z defines schemas to structure data specifications and operation specifications. A schema is made up of a declaration part (a set of typed schema variables) and a predicate part built on these variables. The semantics of a state schema consists in a set of bindings between the schema variables and values such that the predicate holds. State schemas define state spaces. A complete Z specification also uses an initialization schema which defines initial values for the variables.

The idea to define the Z evaluation function is to consider LTSs associated with Z specifications. This is possible since Z follows a model oriented approach. For this purpose, we use both the state schema, the initialization schema and the operation schemas of Z specifications. Let z be a Z specification defined with a state schema $SSch_z$, an

initialization schema $SInit_z$, and a set of operation schemas. We may then define the associated LTS. Its set of states ($STATE_z$) corresponds to the $SSch_z$ semantics state space. The set of initial states of the LTS is the subset of $STATE_z$ with elements that satisfy the predicate of $SInit_z$. 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 $\rhd_Z$ is then defined as: $E \vdash l \rhd_Z s' \Leftrightarrow \exists s \subseteq E \,.\, s \xrightarrow{l} s' \in TRANS_z$.

Application. In this part we illustrate the application of our semantic rules on a simple example with three communicating state diagrams (Fig. 9).

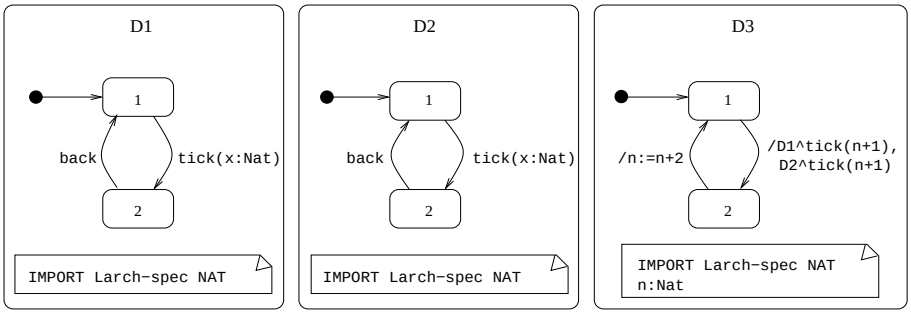

Fig. 9. A simple data extended state diagrams system

We first have to choose a non-extended state diagram semantics, like for example the semantics of UML state diagrams given by Jürjens [19]. $\boxed{INIT}$, $\boxed{STATE}$ and $\boxed{TRANS}$, used to define a semantics to non-extended state diagrams, are instantiated from [19]. $\boxed{INIT}(D)$, for example, correspond to the initial states computed using the **SCInitialize(D)** rule formalized in [19]. However, in this example we will still use the abstract notations to keep the rules readable. Each generic concept within our rules ($\boxed{\emptyset}$, $\boxed{\mathcal{Q}}$, $\boxed{\in}$, $\boxed{\setminus}$, $\boxed{\uplus}$) is instantiated by some concrete concept (resp. nil, Queue, car, cdr, append). Here, the event collections are queues, together with the usual operations on them.

Our example uses natural numbers (sort `Nat`) defined in an usual way using `0` and `succ` [8]. `Nat` is written using the input language of the LP theorem prover [15], which is a subset of 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. The rewrite rules are obtained from the algebraic specification axioms applying the **noeq-dsmpos** LP ordering command [15]. $TRUE_{Larch\text{–}spec}$ corresponds to *true* in LP.

As a first example of rule application, Figure 10 gives an example of a simple dynamic evolution. This is an instantiation of the $DYN-E\emptyset$ rule (Fig. 7) without guard. It represents an independent evolution of diagram D3 from state 2 to state 1. The `w` value bound to the `n` variable is supposed to be the normal form of the `n+2` term. This evaluation is performed through the $act-eval$ application, and the underlying semantic rule

EVAL−SEND of Figure 3 (having as premise: $\{(\mathtt{n},\mathtt{v})\} \vdash \mathtt{n}+2 \rightsquigarrow_R^* \mathtt{w}$). The conclusion of the rule denotes the state and transition construction of the diagram model.

$$\frac{\begin{array}{c} S \in \underline{STATE}(\mathrm{D3}) \\ S =< \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, \emptyset, nil > \\ l = / \ \mathtt{n} := \mathtt{n}+2 \\ \gamma_3 \xrightarrow{l} \gamma_3' \in TRANS(\mathrm{D3}) \\ active(\gamma_3, \Gamma_3) \\ \Gamma_3 \xrightarrow{\varepsilon_{act}} \Gamma_3' \in \boxed{TRANS}(\mathrm{D3}) \\ act{-}eval(\mathtt{n} := \mathtt{n}+2, < \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, nil, nil >, \mathrm{D3}) =< \Gamma_3, \{(\mathtt{n},\mathtt{w})\}, nil, nil > \\ S' =< \Gamma_3', \{(\mathtt{n},\mathtt{w})\}, nil, nil > \end{array}}{\begin{array}{c} S' \in \underline{STATE}(\mathrm{D3}) \\ S \xrightarrow{\varepsilon} S' \in \underline{TRANS}(\mathrm{D3}) \end{array}}$$

Fig. 10. Independent dynamic evolution

An example of communication is given in Figures 11, 12 and 13. It describes the asynchronous communication on `tick` between D1, D2 and D3 (which sends the `tick` events). In this example, we assume that the diagrams D1 and D2 are initially in state 1 and that the diagram D3 is in state 2 with the two sent events in its output queue. Figure 11 describes a conjunction of evolutions for the three diagrams where D3 has events taken out of its output queue whereas D1 and D2 have events put into their input queues. Such individual evolutions (here in premises) could have been proven correct, independently for each diagram, using dynamic evolution rules (such as the one given in Fig. 10) and then the open system rule (expressing the possible queues evolutions). We do not give the whole proof here by lack of place. Figure 12 states that the communication constraint is verified for Figure 11 global evolution and therefore this evolution is a legal evolution for the global system (Fig. 13). Note that in the rules, the abstract concepts have been instantiated by concrete concepts, with *queue* being the *Queue* constructor.

$$\frac{\begin{array}{c} S_1 =< \Gamma_1, \emptyset, nil, nil > \quad \wedge \quad S_1' =< \Gamma_1, \emptyset, queue(\mathtt{tick}(\mathtt{u})), nil > \\ T_1 = S_1 \xrightarrow{\varepsilon}_{queue(\mathtt{tick}(\mathtt{u})), nil} S_1' \\ S_2 =< \Gamma_2, \emptyset, nil, nil > \quad \wedge \quad S_2' =< \Gamma_2, \emptyset, queue(\mathtt{tick}(\mathtt{u})), nil > \\ T_2 = S_2 \xrightarrow{\varepsilon}_{queue(\mathtt{tick}(\mathtt{u})), nil} S_2' \\ S_3 =< \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, nil, queue(\mathrm{D1}\hat{}\mathtt{tick}(\mathtt{u}), \mathrm{D2}\hat{}\mathtt{tick}(\mathtt{u})) > \\ S_3' =< \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, nil, nil > \\ T_3 = S_3 \xrightarrow{\varepsilon}_{nil, queue(\mathrm{D1}\hat{}\mathtt{tick}(\mathtt{u}), \mathrm{D2}\hat{}\mathtt{tick}(\mathtt{u}))} S_3' \\ T = (T_1, T_2, T_3) \end{array}}{T \in \Pi_{i \in 1..3} \underline{TRANS}^{open}(D_i)}$$

Fig. 11. Events exchange

$$CC(S_1 \xrightarrow{\varepsilon}_{queue(\mathtt{tick(u)}),nil} S'_1, S_2 \xrightarrow{\varepsilon}_{queue(\mathtt{tick(u)}),nil} S'_2, S_3 \xrightarrow{\varepsilon}_{nil,queue(\mathrm{D1}\hat{}\mathtt{tick(u)},\mathrm{D2}\hat{}\mathtt{tick(u)})} S'_3) \Longleftrightarrow$$
$$(\mathrm{D1}\hat{}\mathtt{tick(u)} = car(queue(\mathrm{D1}\hat{}\mathtt{tick(u)}, \mathrm{D2}\hat{}\mathtt{tick(u)})) \quad \wedge \quad \mathrm{D}_1 \in \cup_{i\in 1..3}\mathrm{D_i} \Longrightarrow$$
$$\mathtt{tick(u)} = car(queue(\mathtt{tick(u)}))) \quad \wedge$$
$$(\mathrm{D2}\hat{}\mathtt{tick(u)} = car(queue(\mathrm{D1}\hat{}\mathtt{tick(u)}, \mathrm{D2}\hat{}\mathtt{tick(u)})) \quad \wedge \quad \mathrm{D}_2 \in \cup_{i\in 1..3}\mathrm{D_i} \Longrightarrow$$
$$\mathtt{tick(u)} = car(queue(\mathtt{tick(u)})))$$

Fig. 12. Instantiation of the *CC* rule

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

Fig. 13. Example of transition in the final semantic model

4 Discussion

Our goal is to propose specifier-friendly integrated languages for mixed specifications, and more generally an integration method suited to the specification of mixed complex systems. We choose to combine state diagrams with formal specification languages devoted to abstract datatypes (algebraic specifications or state oriented specifications such as Z and B). This joint use, in a formal and integrated framework, of a semi-formal notation for dynamic aspects with formal languages for static aspects enables one to take advantage of both approaches: specifier-friendliness and readability from semi-formal approaches, high abstraction level, expressiveness, consistency and verification means from formal approaches. For an example of a case study specified using our approach, the reader may refer to [9].

Comparison. As a complement to the description of datatypes using class diagrams, the usual extension of state diagrams is the use of OCL (Object Constraint Language) constraints. However, OCL is not really suited to the description of formal abstract datatypes. There are also numerous works combining state diagrams with Z or B such as [10, 23]. In both cases, authors proceed using a translation into the static formalism (Z or B) which thereafter constitutes an homogeneous framework for the subsequent steps of the formal development. Casl-Chart [22] combines Statecharts (following the STATEMATE semantics [17]) with the CASL algebraic specification language [4]. A Casl-Chart specification is made up of datatypes written in CASL and several Statecharts that may use these datatypes in events, guards and actions. Astesiano *et al.* [5] suggest a method to compose languages, in particular a data description language and a paradigm-specific language. Their goal is the description of languages in a component-based style, focusing on the data definition component. Unlike all these existing approaches, our proposal enables to take into account the different UML state diagrams semantics, and more generally Statecharts semantics or any states and transitions formalisms such as the SDL [14] or the recent works on symbolic transition systems [18,

11, 12]. Our approach is also more flexible as it permits to use different datatype description languages, hence increasing the reusability level of specifications.

Perspectives. A first perspective addresses verification issues. If it is yet possible to verify the aspects taken separately, it is important to be able to verify the global system. We are working on the translation of our generic approach for integrated mixed languages and its semantics into higher order logic tools such as PVS [13]. We also have developed a tool dedicated to the animation of specifications combining CCS with abstract datatypes. This tool, ISA [7], is quite generic over the datatype (*i.e.* static) language which is concerned. The next step will then be to extend ISA in order to deal with several dynamic specification languages. Finally, a generalization of our approach represents an interesting challenge in order to be able to combine different formalisms based on the integration of formal datatypes within state / transition systems (such as SDL [14]).

References

1. J. R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. M. Aiguier, F. Barbier, and P. Poizat. A Logic for Mixed Specifications. Technical Report 73-2002, LaMI, Germany, 2002. Presented at WADT'2002.
3. M. Allemand, C. Attiogbé, P. Poizat, J.-C. Royer, and G. Salaün. SHE'S Project: a Report of Joint Works on the Integration of Formal Specification Techniques. In *Proc. of the Workshop on Integration of Specification Techniques with Applications in Engineering (INT'02)*, pages 29–36, France, 2002.
4. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.
5. E. Astesiano, M. Cerioli, and G. Reggio. Plugging Data Constructs into Paradigm-Specific Languages Towards an Application to UML. In T. Rus, editor, *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 273–292, USA, 2000. Springer-Verlag.
6. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of System Specification*. Springer-Verlag, 1999.
7. C. Attiogbé, A. Francheteau, J. Limousin, and G. Salaün. ISA, a Tool for Integrated Specifications Animation. `ISA/isa.html` in Salaün's webpage.
8. C. Attiogbé, P. Poizat, and G. Salaün. Integration of Formal Datatypes within State Diagrams. Technical Report 83-2002, University of Evry, October 2002.
9. 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)*, IEEE Computer Society Press, France, 2003. To appear.
10. R. Büssow and M. Weber. A Steam-Boiler Control Specification with Statecharts and Z. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler*, volume 1165 of *LNCS*, pages 109–128. Springer-Verlag, 1996.
11. M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1):55–61, 2002.

12. C. Choppy, P. Poizat, and J.-C. Royer. A Global Semantics for Views. In T. Rus, editor, *Proc of the 8th International Conference on Algebraic Methodology And Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 165–180, USA, 2000. Springer-Verlag.
13. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *Proc. of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, USA, 1995. Computer Science Laboratory, SRI International.
14. J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.
15. S. J. Garland and J. V. Guttag. A Guide to LP, the Larch Prover. Technical Report, Palo Alto, California, 1991.
16. M. Große-Rhode. Integrating Semantics for Object-Oriented System Models. In F. Orejas, P.G. Spirakis, and J. van Leeuwen, editors, *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *LNCS*, pages 40–60. Springer-Verlag, 2001.
17. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
18. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
19. J. Jürjens. A UML Statecharts Semantics with Message-Passing. In *Proc. of the 17th ACM Symposium on Applied Computing (SAC'02)*, pages 1009–1013, Spain, 2002. ACM Inc.
20. D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini and R. Gorrieri, editors, *Proc. of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 331–347, Italy, 1999. Kluwer Academic Publishers.
21. J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *Proc. of the International Conference on the Unified Modelling Language: Beyond the Standard (UML'99)*, volume 1723 of *LNCS*, pages 430–445, USA, 1999. Springer-Verlag.
22. G. Reggio and L. Repetto. Casl-Chart: A Combination of Statecharts and of the Algebraic Specification Language Casl. In T. Rus, editor, *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 243–257, USA, 2000. Springer-Verlag.
23. E. Sekerinski and R. Zurob. Translating Statecharts to B. In M. Butler, L. Petre, and K. Sere, editors, *Proc. of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *LNCS*, pages 128–144, Finland, 2002. Springer-Verlag.
24. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
25. M. van der Beeck. Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn, editors, *Proc. of the 4th International Conference on the Unified Modelling Language (UML'01)*, volume 2185 of *LNCS*, pages 406–421, Canada, 2001. Springer-Verlag.