LABORATOIRE DE MÉTHODES INFORMATIQUES

Integration of Formal Datatypes within State Diagrams

Christian Attiogbé [1], Pascal Poizat [2], Gwen Salaün[1]

[1] IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, 44322 Nantes Cedex 3, France
email: `{attiogbe,salaun}@irin.univ-nantes.fr`

[2] 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`

— *SPECIF* —

UNIVERSITÉ D'EVRY
VAL D'ESSONNE

RESEARCH REPORT

N^{o} 83-2002

October 2002

LaMI, UMR 8042 CNRS, Genopole & Université d'Évry Val d'Essonne
Tour Évry 2, 523 place des terrasses de l'Agora
F-91000 ÉVRY

Christian Attiogbé, Pascal Poizat, Gwen Salaün
Integration of Formal Datatypes within State Diagrams
34 p.

Les rapports de recherche du Laboratoire de Méthodes Informatiques sont disponibles aux formats PostScript® et PDF® à l'URL :
`ftp://ftp.lami.univ-evry.fr/pub/publications/reports/index.html`

Research reports from the Laboratoire de Méthodes Informatiques are available in PostScript® and PDF® formats at the URL:
`ftp://ftp.lami.univ-evry.fr/pub/publications/reports/index.html`

APS8302.tex – Integration of Formal Datatypes within State Diagrams – 22/10/2002 – 13:04

Integration of Formal Datatypes within State Diagrams

Christian Attiogbé, Pascal Poizat, Gwen Salaün

Abstract

In this technical report, we present a generic approach for the integration of 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. We herein present first the formal foundations of our approach and then a case study is used to demonstrate its pragmatism.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Formal and Semi-formal Specifications Integration, State Diagrams, UML, Algebraic Specifications, Z, B

Contents

1 Introduction

Semi-formal notations such as the UML [59] and SDL [34] have now made a breakthrough in software engineering, mainly because of their user-friendliness through graphical notations and adaptability. For instance, the UML enables one to describe the different aspects of complex systems (static aspects – including datatypes and functional aspects –, dynamic aspects – including concurrency and communication –, and architectural aspects). For this purpose, the UML is using different kinds of diagrams inherited from earlier methods and notations. However, the UML still lacks a formal semantics [63, 11] and even if some diagrams are formalized (see Section 4), the whole combination of the different diagrams is not. This issue is an important one as one would want to ensure that a whole system described through different diagrams is consistent. This "multi-diagram consistency problem" is nowadays an issue for a lot of works [13, 61]. Verifications on multi-diagrams UML specifications is also, if not impossible, very restricted. For all these reasons, UML formalization is actually an issue for a lot of works, mainly related with the Precise UML Group [36].

In the meantime, formal methods have now been successfully used for several decades to describe systems in a more mathematical and unambiguous way. The main interest of formal methods and formal languages is that their semantics is well-defined, hence making the verification of systems possible. Moreover, formal specifications also enable one to describe systems at a higher abstraction level, dealing with the "what" (system properties) and not the "how" (implementation details). However, as a counterpart, formal methods are often said to be hard to learn and to put into practice.

The joint use of formal and semi-formal specification languages therefore seems to be 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 using an integrated language the different aspects of complex systems.

Static and functional aspects are specified using static formal specification languages (algebraic specifications [14], state oriented languages such as Z [70] or B [5]). This makes the verification of specifications possible but also the description of datatypes 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 the reuse of earlier specifications possible.

Dynamic aspects (*i.e.* behaviours, 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 [38], for the different (yet growing number) of UML state diagrams semantics [51, 49, 74, 10, 73, 41], 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.

In a larger scale, our work deals with the formal specifications integration and composition issues, where we yet have some general results [9, 66, 65, 64]. At the specification global level, our approach therefore also addresses the static-dynamic parts consistency issues.

This paper is structured as follows. In Section 2, we present the formal foundations of our approach: syntactic links between state diagrams and datatypes extensions, and integration semantics. Section 3 then illustrates in a pragmatic way how our approach may be used to specify a simple system (a gas station). In Section 4, we present related works and compare our approach with them. To end, Section 5 concludes our work and presents some future perspectives.

2 Formal Foundations of the Combination

In this section we present the formal aspects of our integration of formal datatypes within state diagrams. We first present the extension syntax for this integration, and then study its semantics.

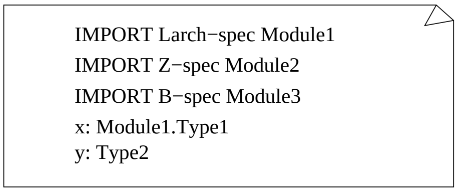
2.1 Syntax

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 importation and local variables declaration extensions to state diagrams. Both are done using *data boxes* (Fig. 1), some 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 state diagrams data embeddings.

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.



```
IMPORT Larch−spec Module1
IMPORT Z−spec Module2
IMPORT B−spec Module3
x: Module1.Type1
y: Type2
```

Figure 1: Local declarations of data into state diagrams

The general form of a state diagram transition is given in Figure 2.

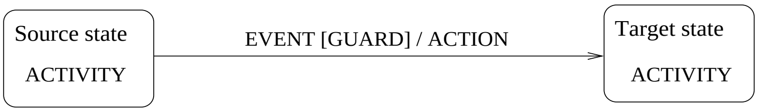


Figure 2: State diagram transition

Datatype extensions (*data expressions*) may therefore appear in both states and transitions (Tab. 1). In states, our extensions correspond to activities (actions such as entry/exit or local modifications, and events). In transitions, our extensions 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 with data expressions within and (iv) make assignments of data expressions to the local variables. The two latter ones 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.

Data expressions may be either variables, terms for algebraic specifications or operation applications for state oriented specifications (such a VDM, Z or B). As far as the formal language for the static aspects is concerned, the only constraint is to have some well-defined evaluation mechanism. 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 specifications 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 means to solve such inconsistencies. As a general way of verifying these constraints, we developped a *meta-type* concept using meta-typing rules (see the semantics below). Badly meta-typed terms may not be evaluated within our approach. Finally, we highlight the fact that the meaning of static specifications has to be consistent, and we assume this working hypothesis for the definition of semantic rules. In Table 1, we summarize our transitions extensions.

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

Table 1: Links between state diagrams and data

Then, in Table 2, we give an abstract syntax for these extensions. The `VAR`, `TYPE`, `PREDICATE` and `DATA-TERM` nonterminals correspond respectively to variable names, type names (sorts), predicate and data expressions.

```
LINK         ::=  [EVENT] [[GUARD]] [/ ACTION+]
EVENT        ::=  event-name [(VAR-DECL+)]
VAR-DECL     ::=  VAR: TYPE
GUARD        ::=  PREDICATE
ACTION       ::=  EMISSION | ASSIGNMENT
EMISSION     ::=  [VAR] ^ event-name [(DATA-TERM+)]
ASSIGNMENT   ::=  VAR := DATA-TERM
```

Table 2: BNF Grammar of links

2.2 Semantics

In this part, our goal is to give an integrated 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 [51, 49, 74, 10, 73, 41, 38] 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 constructs may then be instantiated for a specific kind of state diagram. In our semantics we will for example 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 constructs 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 kind of state diagrams and their underlying semantics could be considered. The meaning of the state diagrams extension is expressed using an operational semantics. Therefore, the semantics of a non-extended state diagram D has to be given in term 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 (describing 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, yet concrete, piece of specification. Specific notations are introduced throughout the semantic definitions.

Meta-Typing Rules. These rules are needed in order to evaluate data expressions appearing in diagrams, and more precisely to perform evaluation using the adequate evaluation function, that is the one dedicated to the meta-type (*e.g.* Larch, CASL, Z, B) of the term.

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 therefore 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.* $\rhd_X$) will denote their interpretation within the context of the corresponding framework (*e.g.* $\rhd_B$ denotes the B evaluation function).

The meta-typing rules are given in Figure 3.

$$\frac{\begin{array}{c}\texttt{IMPORT}\ X-SPEC\ M \in DeclImp(D)\\ def(T,M)\\ x : T \in DeclVar(D)\end{array}}{x ::_D X}\ META-VAR$$

$$\frac{\begin{array}{c}\texttt{IMPORT}\ X-SPEC\ M \in DeclImp(D)\\ def(op,M)\\ \forall\, i \in 1..n\ .\ t_i ::_D X\end{array}}{op\ \ t_1\ \ \ldots\ \ t_n ::_D X}\ META-TERM$$

Figure 3: Meta-typing rules

The $META-VAR$ rule is used to give a meta-type to variables using data local declarations. The $META-TERM$ 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 set of terms, since syntactically there are some differences between algebraic and state oriented formal specification languages.

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. Environments memorize the datatypes managed by extended state diagrams;
- $\boxed{\mathcal{Q}}$ is the set of collections[1], $\boxed{\mathcal{Q}}[EVENT^{?}]$ 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 $\rhd_X$

[1] 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.

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 on global states are given in Figure 4.

$$\frac{\begin{array}{c} act-eval(a_1, S, D) = S' \\ act-eval(a_2 \ldots a_n, S', D) = S'' \end{array}}{act-eval(a_1 \ldots a_n, S, D) = S''} \; EVAL-SEQ$$

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

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

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

Figure 4: Action evaluation rules

The $EVAL-SEQ$ rule is used to evaluate the actions in sequence. $a_1 \ldots a_n$ is a syntactic 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 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[2] 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 (the term t is rewritten in v). For state oriented languages, the interpretation of the rule is slightly different. The variable x represents the state of the system. 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.

The partition we made between the different kinds of semantics rules enables one to add some more action expressions. Let us illustrate this on a conditional (*if-then-else*) expression whose semantics is given in Figure 5. Notations $TRUE_X$ and $FALSE_X$ denote truth values within the X framework.

This process may be used to extend the action language in order to take into account the specific constructs of a given language, such as LOTOS [40] or SDL [34] for instance. Hence, our approach even though focusing on the state diagrams may be easily extended to consider a wide range of languages involving static and dynamic aspects.

[2] $\boxed{\uplus}$ 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.

$$\frac{\begin{array}{c}\exists\, X\ .\ cond ::_D X\\ E \vdash cond \rhd_X TRUE_X\\ act\!-\!eval(a_1 \ldots a_n, <\Gamma, E, Q_{in}, Q_{out}>, D) = S'\end{array}}{act\!-\!eval(if\ cond\ then\ a_1 \ldots a_n\ else\ b_1 \ldots b_m, <\Gamma, E, Q_{in}, Q_{out}>, D) = S'}\ EVAL-IF1$$

$$\frac{\begin{array}{c}\exists\, X\ .\ cond ::_D X\\ E \vdash cond \rhd_X FALSE_X\\ act\!-\!eval(b_1 \ldots b_m, <\Gamma, E, Q_{in}, Q_{out}>, D) = S'\end{array}}{act\!-\!eval(if\ cond\ then\ a_1 \ldots a_n\ else\ b_1 \ldots b_m, <\Gamma, E, Q_{in}, Q_{out}>, D) = S'}\ EVAL-IF2$$

Figure 5: if-then-else evaluation rules

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 term of an LTS $(\underline{INIT}, \underline{STATE}, \underline{TRANS})$ where states are extended states, with:

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

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 term 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.* $INIT$, $STATE$, $\boxed{INIT}$ 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 term of *configurations* [51, 49, 73] 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 in Figure 6.

$$\begin{array}{l}
base(event\!-\!name(x_1 : T_1, \ldots, x_n : T_n)\ \ guard\ \ /\ \ a_1 \ldots a_m)\\
\qquad = base_{event}(event\!-\!name(x_1 : T_1, \ldots, x_n : T_n))\ \ /\ \ base_{action}(a_1 \ldots a_m)\\
base_{action}(a_1 \ldots a_n) = base_{action}(a_1) \ldots base_{action}(a_n)\\
base_{action}(\varepsilon_{act}) = \varepsilon_{act}\\
base_{action}(x := t) = \varepsilon_{act}\\
base_{action}(receiver\hat{\ }event\!-\!name(t_1, \ldots, t_n)) = receiver\hat{\ }base_{event}(event\!-\!name(t_1, \ldots, t_n))\\
base_{event}(event\!-\!name(x_1 : T_1, \ldots, x_n : T_n)) = event\!-\!name\\
base_{event}(event\!-\!name(t_1, \ldots, t_n)) = event\!-\!name
\end{array}$$

Figure 6: Definition of the *base* function

A first rule (Fig. 7) 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 ($\boxed{\emptyset}$).

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

Figure 7: Initialisation rule

The meta-type of variables is used to define the notion of type in term 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. 8) 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.

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

Figure 8: Stuttering steps rule

The next dynamic rule (Fig. 9) 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.

However, sometimes there are no events to trigger transitions. Such a case is dealt with by Figure 10 rule, where the corresponding transition of $\underline{TRANS}(D)$ is labelled by the stuttering step label (ε).

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.

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, Fig. 2). 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 others 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. An important thing to note is that these modifications of the state diagrams extended states may

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

Figure 9: Basic dynamic rule (event reception)

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 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. 11) 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 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 set of 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:

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

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

Figure 10: Basic dynamic rule (no event reception)

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

Figure 11: Opening dynamic rule

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, Fig. 12) that expresses that whenever an emission event is taken out of a given diagram (D_i) output event collection (*i.e.* present in E_{out_i}), 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}).

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

Figure 12: Communication constraint formula

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

Semantics of $\triangleright_X$. The semantics of $\triangleright_X$ depends on the various cases of data management, *i.e.* how are formalized evaluation functions. We have at our disposal several kinds of evaluation functions depending on the data specification language: axiomatic or state oriented. Here, explanations are splitted into two outcomes. We respectively offer several valuable insights into the evaluation function formalization for algebraic specifications (Larch and CASL) and state oriented languages (Z and B).

Algebraic Specifications. Concerning algebraic specifications, rewriting system is chosen as the evaluation function. This choice is justified since it is suitable to an operational semantics, and accordingly enables us to remain in a pragmatic and executable context. There are two cases: either the evaluation function is already available (formalized by hand or computed in a tool) and the designer has only to study this existing work, or (s)he has to build the function step by step. The first case is the most widespread possibility; indeed languages often come with their rewriting system. Furthermore, this case is more relevant because our goal is to deal with real cases and to point the work towards pragmatic results. For both cases, the evaluation function is mainly computed from the algebraic axioms appearing in the datatypes declaration. Signatures are used too to perform static semantics verification such as type consistency. The first solution aims at using the o wn ordering mechanisms of the language. We illustrate this idea on two algebraic languages which are Larch and CASL.

Larch [37] is a multi-site project, exploring methods, languages, and tools for the practical use of formal specifications. Much of the early work was done at MIT. Larch has at its disposal a theorem prover LP [35] which is able to automatically orient equations into rewrite rules without users having to enter explicit ordering commands. LP provides three types of ordering mechanisms for orienting equations into rewrite rules: two registered orderings (the **dsmpos** and **noeq-dsmpos** orderings), a polynomial ordering, and three "brute-force" ordering procedures. Here, we detail the registered orderings. These orderings are chosen because there are sufficient enough to illustrate the computation of the function. Furthermore, both next kinds of ordering mechanisms are difficult to use and produce a nonterminating set of rewrite rules. LP's registered orderings use information in a registry to orient equations. When no commutative or associative-commutative operators are involved, these orderings guarantee that the resulting rewrite rules terminate. There are two kinds of information in a registry: height information (relating pairs of operators) and status information (assigning relative weights to the arguments of operators with arity greater than one). The reader may refer to [35, 32] to have detailed informations concerning the registered ordering mechanism.

CASL [12] (Common Algebraic Specification Language) is an expressive algebraic language aimed at specifying requirements and design for conventional software. From CASL, simpler languages (*e.g.* for interfacing with existing tools) are to be obtained by restriction. The main features of its design are as follows: many-sorted basic specifications, structured specifications, architectural specifications, and specification libraries. A loose semantics for CASL specifications is described in [29], but the meaning of CASL terms could be given using rewriting too. The rewriting is performed using a set of rewrite rules deduced from the CASL specifications. This can be achieved following the conceptual steps defined in [42] whose goal is the execution of CASL equational specifications with the ELAN rewrite engine [20]. Both basic and structured specifications are considered, even though some restrictions are assumed (subsorting and partiality features). The translation from CASL to ELAN is performed using intermediate formats (FCasEnv and Efix ATerms). Then, a last step of translation transforms Efix ATerms into the ELAN executable format (REF programs).

About the second case (by hand), the axioms or equations have to be oriented into rewrite rules to obtain the rewriting system. There are two ways to compute this set of rewrite rules. It can be obtained from algebraic axioms by applying general ordering algorithms as those described in [43]. This computation is not automated and must be performed by hand. Most of these algorithms ensure termination and confluence of the computed rewriting system. To apply these algorithms, we restrict ourselves to the initial semantics of the datatypes because the interpretation of a set of axioms by a rewriting system has really sense only in the case of initial model [18]. For algebraic specification languages with loose semantics, it is possible to restrict them to their initial algebra so as to simplify the process.

State Oriented Languages. We give here the evaluation functions for Z and B are successively presented. Other languages such as VDM could be also taken into account using related approaches. Note that Z and B are seen through their language aspect and not the methodological one.

The B method is a collection of mathematically based techniques for the specification, design and implementation of software modules. Systems are modeled as a collection of interdependent abstract machines. An abstract machine is described using the Abstract Machine Notation (AMN). Now, we discuss the computation of the evaluation function from state oriented specification techniques, and especially from B abstract machines. For this issue, we inspire ourselves from the B-Book [5], and from a more recent work of Bert and Cave [17]. In the latter, the authors study several ways to build finite LTS from B abstract systems: enumeration of states, symbolic evaluation and set constraints, abstract interpretation, and so on. A LTS is suitable to depict an evaluation function.

The evaluation function is defined using the enumeration of states. This is the easiest way to define the function, and the underlying set of states and transitions. Given a B machine, we need the set of variables, the invariant, the initialisation, and the list of operations. The behaviour of the machine is viewed as a LTS. Each state denotes a set of data with values, and more precisely a finite set of variable/value couple. These couples are deduced from the variables and initialisations of the B machine. Names of operations are labels of transitions and symbolize the evolution from one state to another. The state space of the LTS is the set of states which satisfy the invariant. From the machine initialisation clause (depicted here using *INIT*), the initial states are deduced. In the formulas below, the symbols $\langle\rangle$ and $[]$ are the temporal operators used in the formalization of some B dynamic aspects [6].

$$S_0 = \{x_j \mapsto v_j \mid \langle INIT\rangle(x_j = v_j)\} \;\; \forall j \in 1 \ldots n$$

Then, from each state satisfying the invariant and for each operation enabled in the initial state, the successors are the states with new values obtained by the application of the operation body (*i.e.* the corresponding generalized substitution).

$$S_{k+1} = \{x_j \mapsto v'_j \mid [S_k](\langle OP_i\rangle(x_j = v'_j))\} \;\; \forall i \in 1 \ldots m \;\; \forall j \in 1 \ldots n$$

According to the B-Book, this rule means that if $(x_j \mapsto v_j)$ is in S_k then (v_j, v'_j) is in $\mathrm{rel}_{x_j}(OP_i)$, where $\mathrm{rel}_{x_j}(OP_i)$ is the binary relation which relates the values of x_j before and after the substitution OP_i. The properties of the evaluation function are the well-known ones on transition systems: deadlock, finite states, reachability, and so on. Here, we mainly are interested in the fact that there are no deadlocks in the transition system in order to be able to apply the evaluation function (moving from one state to another applying any operation of the machine). Yet, it is possible that the evolution in the LTS has no effect on the data. Another important property is the finite looping of the function. The termination is ensured because the evaluation of one operation induces a single step in the transition system evolution. Last but not least, the transition system is not finite in most of cases, but it is not really a shortcoming seeing that the generation of the whole automaton is useless here.

Z [70] 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 $\triangleright_Z$ is then defined as: $E \vdash l \triangleright_Z s' \Leftrightarrow \exists\, s \subseteq E \,.\, s \xrightarrow{l} s' \in TRANS_z$. The properties are like defined above for B.

To conclude the model oriented discussion, let us remark that the different steps followed for both languages could be gathered in a same approach. Indeed, they have the same underlying semantic model (LTS) introduced above.

Application. In this part we illustrate the application of our semantic rules on a simple example with three communicating state diagrams (Fig. 13).

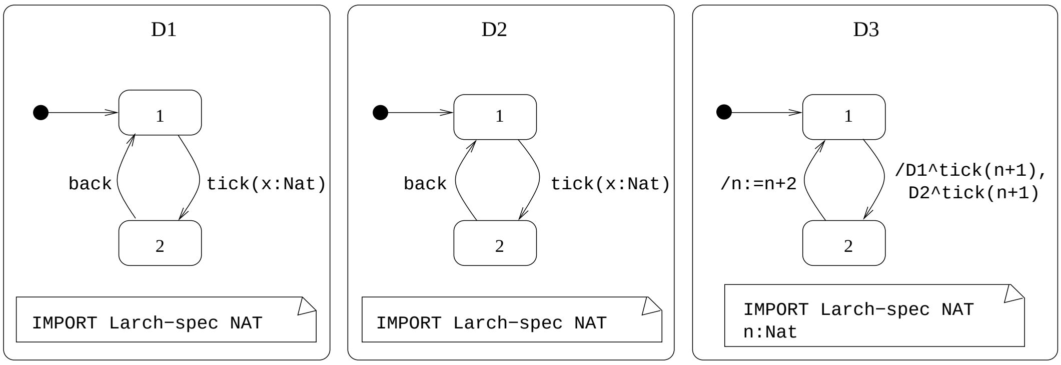


Figure 13: A simple data extended state diagrams system

We first have to choose a non-extended state diagram semantics, like for example the semantics of UML state diagrams given by Jürjens [41]. $\boxed{INIT}$, $\boxed{STATE}$ and $\boxed{TRANS}$, used to define a semantics to non-extended state diagrams, are instantiated from [41]. $\boxed{INIT}(D)$, for example, correspond to the initial states computed using the **SCInitialize(D)** rule formalized in [41]. However, in this example we will still use the abstract notations to keep the rules readable. Each generic concept within our rules is instantiated by some concrete concept as presented in Table 3. Here, the event collections are queues, together with the usual operations on them.

Abstract operators	$\boxed{\emptyset}$	$\boxed{\mathcal{Q}}$	$\boxed{\in}$	$\boxed{\setminus}$	$\boxed{\uplus}$
Concrete operators	nil	Queue	car	cdr	append

Table 3: Instantiation of the generic operators

Our example uses natural numbers (sort `Nat`) defined in an usual way using `0` and `succ`. `Nat` is written using the input language of the LP theorem prover [35], which is a subset of Larch. Within this framework, the evaluation function corresponds to term rewriting, *i.e.* $\triangleright_{Larch-spec} \equiv \rightsquigarrow^*_R$ with R being the term rewriting relation. The rewrite rules are obtained from the algebraic specification axioms applying the **noeq-dsmpos** LP ordering command [35]. $TRUE_{Larch-spec}$ corresponds to *true* in LP.

As a first example of rule application, Figure 14 gives an example of a simple dynamic evolution. This is an instantiation of the $DYN-E\emptyset$ rule (Fig. 10) 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 4 (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.

An example of communication is given in Figures 15, 16 and 17. 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 15 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 queue. 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. 14) and then the open system rule (expressing the queues possible evolutions). We do not give the whole proofs here by lack of place. Figure 16 states that the communication constraint is verified for Figure 15 global evolution and therefore this evolution is a legal evolution for the global system (Fig. 17). Note that in the rules, the abstract concepts have been instantiated by the Table 3 concrete concepts, with *queue* being the *Queue* constructor.

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

Figure 14: Independent dynamic evolution

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

Figure 15: Events exchange

$$\begin{array}{c} CC(S_1 \overset{\varepsilon}{\rightarrow}_{\mathit{queue}(\mathtt{tick(u)}),\mathit{nil}} S_1', S_2 \overset{\varepsilon}{\rightarrow}_{\mathit{queue}(\mathtt{tick(u)}),\mathit{nil}} S_2', S_3 \overset{\varepsilon}{\rightarrow}_{\mathit{nil},\mathit{queue}(\mathrm{D1}\hat{\ }\mathtt{tick(u)},\mathrm{D2}\hat{\ }\mathtt{tick(u)})} S_3') \Longleftrightarrow \\ (\mathrm{D1}\hat{\ }\mathtt{tick(u)} = \mathit{car}(\mathit{queue}(\mathrm{D1}\hat{\ }\mathtt{tick(u)}, \mathrm{D2}\hat{\ }\mathtt{tick(u)})) \wedge \mathrm{D}_1 \in \cup_{i \in 1..3} \mathrm{D_i} \Longrightarrow \\ \mathtt{tick(u)} = \mathit{car}(\mathit{queue}(\mathtt{tick(u)})))\wedge \\ (\mathrm{D2}\hat{\ }\mathtt{tick(u)} = \mathit{car}(\mathit{queue}(\mathrm{D1}\hat{\ }\mathtt{tick(u)}, \mathrm{D2}\hat{\ }\mathtt{tick(u)})) \wedge \mathrm{D}_2 \in \cup_{i \in 1..3} \mathrm{D_i} \Longrightarrow \\ \mathtt{tick(u)} = \mathit{car}(\mathit{queue}(\mathtt{tick(u)}))) \end{array}$$

Figure 16: CC instantiation

$$\frac{\begin{array}{c} T = (S_1 \overset{\varepsilon}{\rightarrow}_{\mathit{queue}(\mathtt{tick(u)}),\mathit{nil}} S_1', S_2 \overset{\varepsilon}{\rightarrow}_{\mathit{queue}(\mathtt{tick(u)}),\mathit{nil}} S_2', S_3 \overset{\varepsilon}{\rightarrow}_{\mathit{nil},\mathit{queue}(\mathrm{D1}\hat{\ }\mathtt{tick(u)},\mathrm{D2}\hat{\ }\mathtt{tick(u)})} S_3') \\ T \in \Pi_{i \in 1..3} \underline{\mathit{TRANS}}^{open}(D_i) CC(T) \end{array}}{T \in \overline{\mathit{TRANS}}(\cup_{i \in 1..3} \mathrm{D_i})}$$

Figure 17: Example of transition in the final semantic model

3 Case Study: A Gas Station

In this section, we illustrate our approach with a concrete, simple but realistic example. It addresses a gas station equipped with several pumps. We successively deal with the requirements, the analysis and the modelling of static and dynamic aspects. For this case study we choose the UML state diagram notation to specify the dynamic aspect and both Larch and Z to model static aspects. Therefore, the foundations of Section 2 serve as the basis for this specific combination and are interpreted with regard to these formalisms semantics. According to the static part, we refer mainly to the semantics of Z and Larch; concerning the dynamic part, we refer to one of the proposed semantics of UML diagrams. It is the formalization of Jürjens [41] which is fixed for our work. The focus may be put on interactions between diagrams. Informally, for an UML model, interactions are viewed through collaboration diagram and/or sequence diagrams associated to the set of objects which participate in the accomplishment of a task. The corresponding informal semantics is defined as a sequence of communications which contains a collection of partially ordered messages. Each message specifies a communication between a sender and a receiver. In the considered formal semantics, the state diagrams D associated to each object, are precisely defined with their states and transition relation. Interactions between diagrams are formalized and communications are treated using queues to exchange messages. In the following case study, we deal with UML state diagrams and their interactions, in conformity with the fixed semantics. The combined use of our formalization (in Section 2) above the fixed semantics is already illustrated in Section 2 with an instantiation of our semantic rules.

3.1 Requirements and Analysis

Requirements. The case study is about the specification of a gas station behaviour. The station only provides self service by card payment and not payment at a cashier's desk. The necessary transactions for authentication and effective payment (data transmission to card management center) are not in the scope of this study. There are only three pumps at the gas station. Each pump delivers a type of fuel: two-star fuel, lead-free fuel and diesel. The station also has a control board equipped with a card reader for payments, several button keys for options selection, a numerical keyboard for inputs, a control screen and a ticket printer. The gas station manages all these equipments and furnishes to users the desired quantity of the fuel they selected. The station enables the user to enter his/her card number via the numerical keyboard, to select the fuel type and to choose or not the ticket printing.

Analysis and Working Hypothesis. From these requirements, we clarify the functionalities of the gas station considered here as the main system: users identification and authentication, fuel delivering and payment. We make explicit the necessary conditions to ensure these functionalities and the interactions implied with the environment. According to this analysis, we identify and model the various components. Therefore, we describe the global behaviour expected for the gas station. We distinguish two main parts in the analysis. A static part is devoted to the data contained in the system. A dynamic part deals with the system evolution through its components.

Static Part. To deliver fuel to users without sudden interruption during the distribution, the station must have a sufficient quantity of each type of fuel when it is put on service. We assume that there are several tanks. An unique tank is associated to each pump, that means to each type of fuel. Two different pumps are associated to two different tanks. A pump is off (*i.e.* out of service) when the quantity of fuel in the associated tank is less than a threshold to be defined. Therefore, the station cannot deliver this type of fuel before a refill of its tank. To guarantee a smooth working, the system should know at any time the tank state. By the way, this state should be maintained according to the achieved operations (fuel delivering, fuel refilling). To compute the amount to be paid and print the ticket, the system must have the price by litre for each type of fuel and the delivered quantity. A record of these informations is necessary. To summarize, we have to specify tanks, fuels and pumps to model the static part. Cards are not treated because authentication and payment are not in the scope of the study.

Dynamic Part. The functioning of the station is triggered by the user. When the user inserts its card, an authentication procedure follows. If the authentication succeeds then the user has to select a fuel type and validate or correct his/her choice. These interactions put emphasis on a component: a *card manager*. According to the user choice, the station initializes the screen, displays the litre price of the selected pump and the maximal quantity of fuel which can be delivered. Then the user must disengage the pump and serves itself. The amount to be paid and the quantity are simultaneously displayed during the delivering. At the end of the delivering, when the user engages the pump, the station prints the ticket if the user had selected this option. All these interactions reveal a subsystem we call *pumps manager*. This latter exchanges information with the user. We assume that when a pump is on, it guarantees the delivering of the associated fuel. For this reason, the station checks for the tank state after each service and either the pump remains on or is set off if there is not enough fuel quantity. Thus, we identify a *tanks manager* which has interactions with the pumps manager.

We design the global system as a concurrent composition of the three main independent components: a card manager, a pumps manager and a tanks manager. Each of them is described in the sequel. Other subsystems are considered as external: users and a subsystem that provides continuously, the quantity of fuel and the corresponding amount to be paid for a distribution.

3.2 Static Aspects Modelling

In this case study, we specify the data with the input language of the LP theorem prover and with the Z language. We specify in LP the natural and the real numbers. We use Z to specify data concerning pumps and tanks handling. We give here the Z specifications used in the diagrams of the dynamic part. The complete specification is analysed with Z/EVES for static correctness (type checking and syntax analysis) and static semantics (domain control).

Data Modelling for the Tank Management. Several Z schemas model the tank management. For each tank, we record the fuel, the minimal and maximal quantity it can contain, and its current quantity. There is a bijection between the pumps (*pumpsId*) and the fuel types (*ZFuel*). Various operations are described to specify the update of the tank contents.

$$\textit{pumpsId} == \{1, 2, 3\}$$

Pumps are modelled in our system as natural numbers. A pump is identified using a unique natural number.

$$\textit{ZBool} ::= \textit{tt} \mid \textit{ff}$$

The values *tt* (true) and *ff* (false) characterise the Boolean type.

$$\textit{ZFuel} ::= \textit{twostar} \mid \textit{leadfree} \mid \textit{diesel}$$

The enumeration of the three fuel types enables us to specify the fuels.

$$\begin{array}{|l}
\textit{fuel} : \textit{pumpsId} \rightarrowtail\!\!\!\!\twoheadrightarrow \textit{ZFuel} \\
\hline
\textit{fuel}(1) = \textit{twostar} \\
\textit{fuel}(2) = \textit{leadfree} \\
\textit{fuel}(3) = \textit{diesel}
\end{array}$$

Each pump delivers a unique fuel type. For this purpose, we use the function *fuel*. Therefore, given a fuel type, we can get back the associated pump with the reverse function *fuel*~.

$$\begin{array}{|l|}
\hline
\textit{Tank} \\
\hline
\textit{tfuel} : \textit{ZFuel} \\
\textit{minCapacity} : \mathbb{Z} \\
\textit{maxCapacity} : \mathbb{Z} \\
\textit{quantity} : \mathbb{Z} \\
\hline
\textit{quantity} < \textit{maxCapacity} \ \wedge \ \textit{quantity} > \textit{minCapacity} \\
\hline
\end{array}$$

A tank is characterized by its type of fuel, its minimal and maximal capacity and, its current quantity of fuel.

$$\begin{array}{|l|}
\hline
\textit{TanksM} \\
\hline
\textit{tank} : \textit{pumpsId} \rightarrow \textit{Tank} \\
\hline
\forall \textit{pp} : \operatorname{dom} \textit{tank} \bullet (\textit{tank}(\textit{pp})).\textit{tfuel} = \textit{fuel}(\textit{pp}) \\
\hline
\end{array}$$

This schema allows us to manage all the pumps in our system. A tank is associated to each pump.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{InitTanksM}} \\
\mathit{TanksM}' \\
\hline
\mathit{tank}' = \emptyset \\
\hline
\end{array}$$

In this initialization schema, we start with an empty function for tank. This state variable is initialized and updated by the operation *UpdPumpTank*. We present the operation *UpdQtty* which updates the remaining fuel quantity of each pump and the operation *OkThreshold* which checks the fuel quantity.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{UpdPumpTank}} \\
\Delta \mathit{TanksM} \\
\mathit{pp}? : \mathit{pumpsId} \\
\mathit{fuel}? : \mathit{ZFuel} \\
\mathit{minc}? : \mathbb{Z} \\
\mathit{maxc}? : \mathbb{Z} \\
\mathit{qtty}? : \mathbb{Z} \\
\hline
\mathit{minc}? < \mathit{qtty}? \land \mathit{qtty}? < \mathit{maxc}? \\
\mathbf{let}\ \mathit{ctank} == (\mu\, \mathit{Tank} \mid \mathit{tfuel} = \mathit{fuel}? \land \mathit{minCapacity} = \mathit{minc}? \\
\qquad\qquad \land\ \mathit{maxCapacity} = \mathit{maxc}? \land \mathit{quantity} = \mathit{qtty}?) \bullet \\
\qquad\qquad\qquad \mathit{tank}' = \mathit{tank} \cup \{\mathit{pp}? \mapsto \mathit{ctank}\} \\
\hline
\end{array}$$

This operation initializes the tank associated to a given pump.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{UpdQtty}} \\
\Delta \mathit{TanksM} \\
\mathit{pp}? : \mathbb{N} \\
\mathit{qtty}? : \mathbb{Z} \\
\hline
\mathit{pp}? \in \mathit{pumpsId} \\
(\mathit{tank}'(\mathit{pp}?)).\mathit{quantity} = (\mathit{tank}(\mathit{pp}?)).\mathit{quantity} + \mathit{qtty}? \\
\hline
\end{array}$$

This operation updates (increases and decreases) the fuel quantity of a given pump. The quantity can be negative to decrease or positive to increase. The input variables $\mathit{pp}?$ and $\mathit{qtty}?$ can be replaced by those in the diagrams using the Z notation $\mathit{UpdQtty}[\mathit{pp}/\mathit{pp}?;\mathit{nqt}/\mathit{qtty}?]$. Similarly, the output variables of operation schemas are replaced by local variables to retrieve operation results.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{OkThreshold}} \\
\Xi \mathit{TanksM} \\
\mathit{pp}? : \mathbb{N} \\
\mathit{threshold}? : \mathbb{Z} \\
\mathit{res}! : \mathit{ZBool} \\
\hline
\mathit{pp}? \in \mathit{pumpsId} \\
(((\mathit{tank}(\mathit{pp}?)).\mathit{quantity} < \mathit{threshold}?\ \lor \\
\qquad (\mathit{tank}(\mathit{pp}?)).\mathit{quantity} = \mathit{threshold}?) \land \mathit{res}! = \mathit{tt}) \lor \\
(((\mathit{tank}(\mathit{pp}?)).\mathit{quantity} > \mathit{threshold}?)\ \land\ \mathit{res}! = \mathit{ff}) \\
\hline
\end{array}$$

This operation checks the fuel quantity in a tank with regard to a set threshold. The result is positive if the remaining quantity in the indicated pump ($\mathit{pp}?$) is greater than the indicated threshold ($\mathit{threshold}?$).

These data specifications will be used to model the behaviours of the gas station components. Accordingly, we gather all these specifications in a data module called `Z-GSdata`. Indeed, we can import this module so as to access predefined types as well as the defined types, state schemas and operations schemas. Note that only one diagram uses the initialization schema in order to have only one instance of the system state. The other diagrams use operation schemas and data sent through message parameters. For the readability of the following figures we use special definitions for predefined types: $ZInt == \mathbb{Z}$ and $ZNat == \mathbb{N}$.

3.3 Dynamic Aspects Modelling

The behaviours of card manager, pumps manager and tanks manager are modelled using UML state diagrams. Consequently, the semantics for diagrams interactions is the one of UML diagrams but with regard to the Jřjens’s formalization. In our modelling, we consider the possible interactions between the various diagrams in order to build the scheduling of events and actions, and to establish links with previously described data. The datatypes are considered and used as indicated in Section 2. They appear in guards like `[pstate = tt]`, as event parameters like `updPumpState(pstate:  ZBool)`, in actions for example `/PumpsManagerˆactivatePump(ap)` and in diagrams note as local variables. These latter variables have types imported (`IMPORT ...`) from modules specified in the static part. In this case study, different type names are used in modules. Therefore, there is no need to prefix the type name by the module name to avoid confusion.

Let us now detail the basic components of our system. We start with the card manager (see Fig. 18). It models events which permit to insert a card, input the code and select the desired fuel. Afterwards, it interacts with the pumps manager so as to deliver the fuel. Several interaction termination cases are possible: the authentication fails (`error`), the delivering is correctly terminated (`endDelivering`) and the ticket is printed if the user has selected this option (`printTicket`), or the delivering has not been possible (the pump is not disengaged within sixteen seconds) and the interactions terminate with error (`pumpError`). Note that two kinds of events are used in the diagrams below. The basic ones have no parameters (from UML state diagram). They express either an internal evolution of the state diagram (for example `insertCard` and `inputCode`) or an interaction between diagrams (for example `/CardManagerˆpumpError`). The extended ones can bear parameters for example `askTicket(b: Zbool)`. They express interactions with other diagrams.

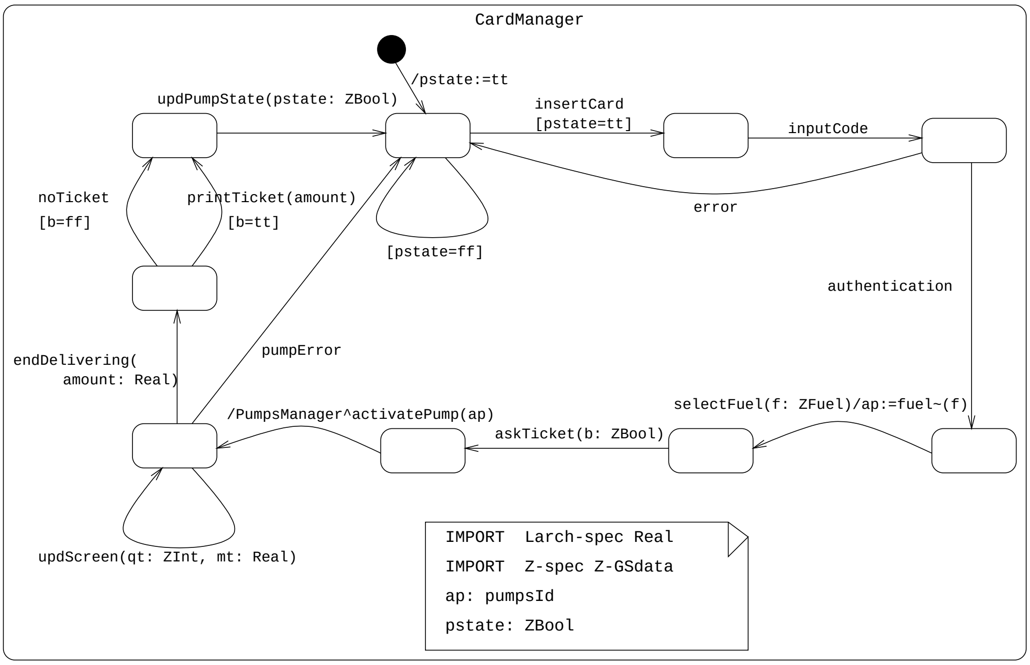


Figure 18: Card Manager Diagram

The diagram of the pumps manager (see Fig. 19) uses local variables `pumpsThreshold, maxAmount` and `t`. The variable `pumpsThreshold` represents the minimal fuel quantity needed to maintain the pump on; it is a set threshold. Note that we have external events that correspond to interactions between the system and its environment, particularly with the user which triggers the pump functioning. These events are not completely enumerated. There are for example `insertCard` in the card manager (see Fig. 18) and `distribution` in the pumps manager (see Fig. 19). The computation of the amount to be paid (`amount`) is achieved by an external subsystem and the value is communicated via the event `receiveAmount` in Figure 19. It is also the case for the distributed quantity (`servedqtty`). The specification of this subsystem could be simply done by considering a state diagram with a local data type handling the couples of a *fuel* and the corresponding *price by litre*. This aspect is hidden through the event `receiveAmount`. The pumps manager ensures the fuel distribution by interacting with the card manager and with the tanks manager. When a distribution is initiated, the user has sixteen seconds to disengage the pump and to serve itself. After this delay an error is notified. The pumps manager has the maximum fuel quantity an user can serve; it can then interrupt the distribution (`endDistribution`). The card manager interacts with the user for a service, for example through the events `insertCard` and `askTicket`. During a distribution, there are a lot of synchronizations between the pumps manager and the card manager so as to update the control screen (`updScreen`) with the fuel quantity and the corresponding amount. The pumps manager engages the event `/CardManager^updScreen(servedqtty, amount)` and the other does the event `updScreen(...)`. This is an UML state diagrams interaction. It meets the general communication formalization presented in Section 2. These interactions with the user are continued until the end of the distribution (`endDistribution`) and the pump engagement (`engage`). Then, the pumps manager locks the pump and synchronizes itself with the tanks manager to update the fuel quantity (`decreaseQtty`), and finally, it communicates with the card reader to complete the distribution (`endDelivering`). In the pumps manager, the pump state is updated (`/CardManager^updPumpState`) after the test of the quantity available after each distribution (`/TanksManager^checkThreshold`).

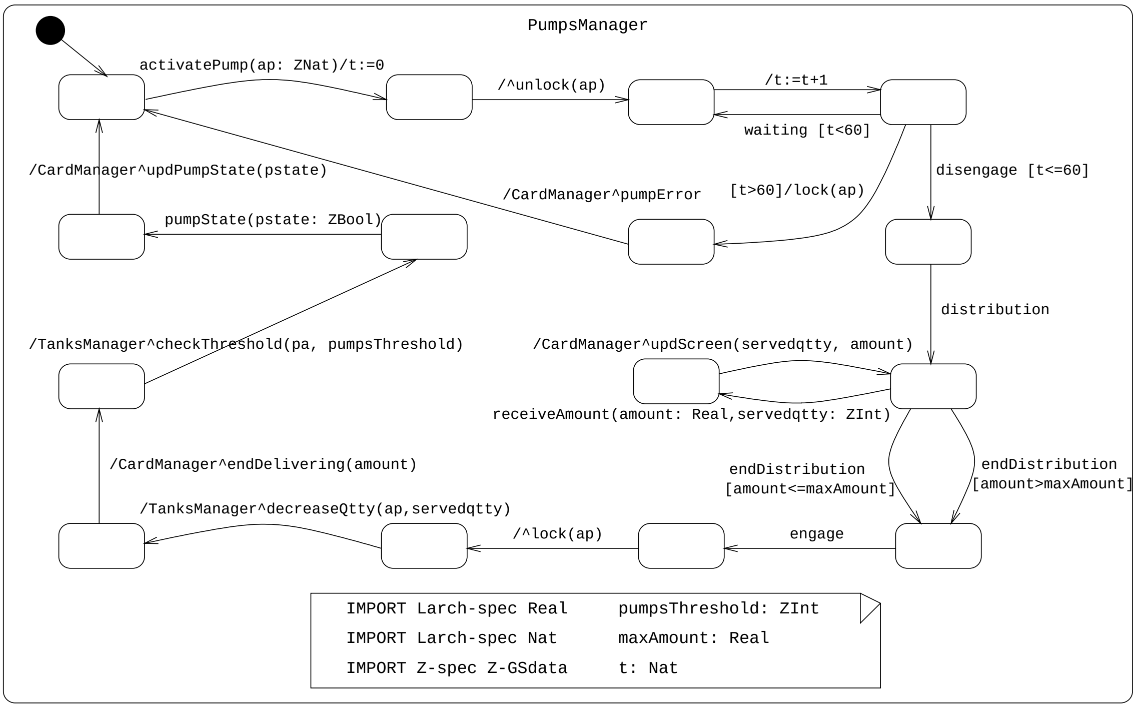


Figure 19: Pumps Manager Diagram

The tanks manager uses the data module `Z-GSdata` and the variable `pstate` as local data. The tanks manager interacts with the pumps manager (`decreaseQtty`) and with the environment in order to fill in (`increaseQtty`) with fuel. It also achieves the update operations on the tanks (`UpdQtty`) and the test of fuel level (`checkThreshold`).

We show in this case study how to practically integrate formal datatypes within state diagrams on the theoretical basis provided in Section 2. Data are beforehand formalized and analysed using Z and LP. Then, within the scope

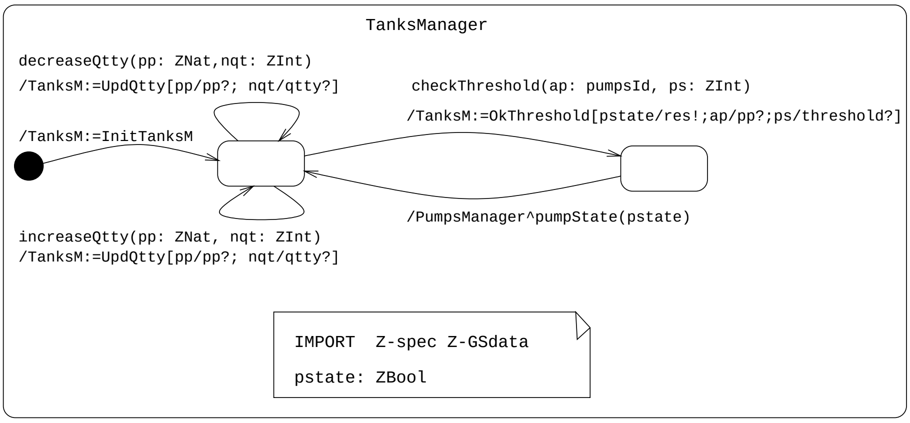

TanksManager
decreaseQtty(pp: ZNat,nqt: ZInt)
/TanksM:=UpdQtty[pp/pp?; nqt/qtty?]
checkThreshold(ap: pumpsId, ps: ZInt)
/TanksM:=OkThreshold[pstate/res!;ap/pp?;ps/threshold?]
/TanksM:=InitTanksM
/PumpsManager^pumpState(pstate)
increaseQtty(pp: ZNat, nqt: ZInt)
/TanksM:=UpdQtty[pp/pp?; nqt/qtty?]
IMPORT Z-spec Z-GSdata
pstate: ZBool

Figure 20: Tanks Manager Diagram

of the UML notation, state diagrams are used to model behaviours of managers which interact by exchanging messages on the basis of a fixed formal semantics.

4 Related Works

In this section, we compare our approach with existing works for the integration or combining of state diagrams (either UML or Statecharts ones) with formal specification languages dedicated to the static aspects. As we already stressed in the introduction, existing works enforce the use of a unique static formal description language, hence are less flexible than our approach. We also therefore have a better level of reusability for our components.

State Diagrams and OCL. As a complement to the description of datatypes using class diagrams (that, in spite of several formalizations [36], have the major drawback to raise the UML multi-diagram consistency problem), the usual extension of state diagrams with datatypes is the use of OCL (Object Constraint Language) constraints. However, OCL is mainly adapted to express constraints on internal data, or on data associated with classes and objects. Hence OCL is not a real language to describe formal abstract datatypes. Nevertheless, several works are interested in the formalization of OCL [24, 26].

Several works address the verification of state diagrams in the reactive or real-time modelling framework. In such a case, the state diagrams are extended with basic datatypes, usually integers, to model clocks. These works, as [31, 52] for example, usually proceed in a two-step approach. First the state diagrams (eventually a restriction) are formalized by translating them into the input languages of model checkers or hybrid verification tools such as Uppaal [48]. Then, the verification process may take place. If these approaches are interesting as far as taking temporal aspects into account is concerned, they do not satisfy us as far as they do not enable one to use real abstract datatypes (*i.e.* others than the clocks) since such datatypes may not be translated efficiently into the tools input languages.

State Diagrams and Z. Büssow and Weber propose a practical methodology for the specification of real-time control systems [75, 22]. Their model, μSZ, is made up of three views to deal with the complexity of such systems and to separate their aspects: (i) the architectural view is specified with object and class diagrams, (ii) the reactive view is specified using Statecharts, and (iii) the functional view is specified using Z. Links between the different views lead to proof obligations to ensure a semantic compatibility.

Other works have taken interest in linking UML class diagrams and Z. Examples are RoZ [33], or FuZed [21] which was initially based on Fusion and has been applied to UML later on. These approaches proceed using a translation into Z. Recently, the RoZ approach has been combined with state diagrams to address the static / dynamic consistency issue [53]. This is an approach complementary to ours since we also want to address this issue. However, the focus of our approach is more put on the dynamic part of systems (operational semantics) and then we try to be as general as possible as far as the static part is concerned (Z may be used but algebraic specifications as well).

State Diagrams and B Method. The first proposal for the combination of state diagrams and B is [67] which aims at using state diagrams to design in a graphical way reactive systems. Therefore, it proposes a translation of Statecharts into the B Abstract State Machines notation. Using this translation, the author gives the Statecharts a formal semantics in term of B machines. The main contribution of this work is to enable one to specify systems using the user-friendly Statechart notation and then translate them into B which thereafter constitutes an homogeneous framework for the subsequent steps of the formal development such as refinement and security properties analysis. The *iState* tool implements the translation of Statecharts into B but also into several programming languages [68, 69].

Lano *et al* [46, 47] propose a method to support the graphical design of systems using B, together with a design methodology to express the structuring of reactive systems in B. Only a simplified version of Statecharts (RSDS) is taken into account in order to avoid some problems arising with their different semantics (mainly related to the event sequences). Their main goal is to use the development support of the B method for reactive systems.

Ledang and Souquières [50] propose contributions to the modelling of UML state diagrams in B. They focus on the event (*i.e.* no activities) part, with a new approach to model events, either deferred or not. Moreover, a modelling of intra diagram communications is presented. In their PhD theses, Meyer [54] and Nguyen [57] have also worked on the translation of state diagrams into B.

An important thing to note is that, as with approaches combining state diagrams with Z, the works combining them with B usually proceed using a translation into the static specification language, which thereafter is used as a unique formal development framework. Only recently, Laleau and Polack [45] have addressed the issue to have a two-way translation process.

The main difference between our approach and 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 another. We think this approach is more suited to a reuse of proofs. It enables the specifier to reuse a proof related to one aspect (hence done in a framework (s)he knows, with tools dedicated to this aspect) within a proof related to the whole (global) system. We hope this could enable at last some kind of multi-language compositional specification. Using integrated translation approaches, the specifier has to make his/her whole proofs within a unique framework (which may not be adequate).

State Diagrams and Algebraic Specification Languages. Reggio and Repetto propose Casl-Chart [62], a visual formal specification language for the modelling of reactive systems. It combines Statecharts (following the STATEMATE semantics [38]) with the CASL algebraic specification language [12]. 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. The semantics of the combined language is given in term of the two basic languages semantics.

Our work is therefore close to Casl-Chart. However, we propose a more flexible approach at the static specification level since we enable the specifier to use (at the same time) several formalisms. Our approach is also more flexible at the dynamic specification level since we propose a reusable semantic framework for the integration of formal datatypes within dynamic formalisms.

5 Conclusion and Perspectives

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 chose 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. Unlike existing approaches, our proposal advocates an higher level of abstraction. Therefore, it enables to take into account the different UML state diagrams semantics, and more generally Statecharts semantics or any states and transitions formalisms such as SDL or the recent works on symbolic transition systems [39, 23, 25]. Our approach is also more flexible as it permits to use different datatype description languages, hence increasing the reusability level of specifications.

As far as the dynamic part is concerned, specifications obtained using our approach may be verified using model checking or specification animation (direct use of STATEMATE abstracting away from the datatypes, translation of the dynamic semantics into specific tools – such as CWB-NC [27] – input languages). As far as the static part is concerned, one may use theorem provers or toolboxes dedicated to the static languages that are being used within the integration (*e.g.* CATS [4] for CASL, Larch Prover [35] for Larch, AtelierB [1] or B-Toolkit [2] for B, Z/EVES [3] for Z).

A first perspective addresses the 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 [30] and Isabelle [58], taking inspiration from the logic and institution for mixed systems presented in [7], from [8, 72, 16, 71, 10] works for the dynamic part and [19, 56, 44] works for the static part. To translate a mixed system specification into PVS, we have to express into PVS input language both basic static and dynamic aspects of the system, their interactions and then propose proof strategies taking into account the mixed aspect of specifications. Up to now, we know how to translate into PVS specifications combining process algebras and algebraic specifications. Yet, we still have to experiment this embedding on concrete case studies, develop the proof strategies and then undertake proofs on the specifications using these strategies.

We think approaches translating the whole specification into the static aspects formal language [68, 69, 50, 60] could also be used but they have the drawback not to enable some kind of proof interaction. That is, after the translation of the dynamic aspects into the static language has been achieved, if the specifier has to interact with the prover during the verification process then (s)he has to do it in the static language, no reverse translation is done. This makes the verification process more difficult. The recent works of Laleau and Polack [45] are a first step to take this interactive two-way mixed verification issue into account. More generally, the issue is about mixed proof integration, that is give means to the specifier to reuse proofs made separately on different aspects into the global system proof process (taking of course into account the fact that different languages are been used).

We yet have developed a tool dedicated to the animation of specifications combining CCS with abstract datatypes. This tool, ISA [15], is quite generic over the datatype (*i.e.* static) language which is concerned. There just has to be an evaluation function defined for the chosen language. This function is coded as an external module which is then used by the main ISA program. Up to now, we have made experiments with the Python (object oriented and functional) language. In the near future, we plan to deal with algebraic specifications (with initiality) through the use of an external rewrite engine (ELAN [20]). The next step will then be to extend ISA in order to deal with several dynamic specification languages. The experiments we have already made (with CCS and Python, or with UML state diagrams and formal datatypes as described in this paper) could then be described as a specific instantiation of this generic framework.

Another future work is related to a generalization of our approach in order to be able to combine different formalisms based on the integration of formal datatypes within state / transition systems (LOTOS [40], SDL [34], Korrigan [25]). Indeed, our approach is quite general and flexible yet as far as the dynamic and static aspects are concerned. This should enable us to take into account a large family of mixed specification languages and compose them. Specific works dealing with relating different heterogeneous specification languages given in term of institutions (the recent works of Mossakowski [55] for example) are not directly relevant here since there is not always an institution defined for the mixed specification languages we know (*e.g.* LOTOS). In [7] however, we made a first step in this direction, defining an institution with refinement for mixed specification languages.

References

[1] http://www.atelierb.societe.com/index_uk.html.

[2] http://www.b-core.com/btoolkit.html.

[3] http://www.ora.on.ca/z-eves/.

[4] http://www.tzi.de/cofi/Tools/CATS.html.

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

[6] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, ed., *Proc. of the 2nd International B Conference (B'98)*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128, France, 1998. Springer-Verlag.

[7] M. Aiguier, F. Barbier, and P. Poizat. A Logic for Mixed Specifications. Technical report 73-2002, LaMI, Germany, 2002. Presented at WADT'2002. Available at ftp://ftp.lami.univ-evry.fr/pub/specif/poizat/documents/RR-ABP02.ps.gz.

[8] M. Allemand. Verification of Properties Involving Logical and Physical Timing Features. In *Génie Logiciel & Ingénierie de Systèmes & leurs Applications (ICSSEA'00)*, 2000.

[9] 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.

[10] D. B. Aredo. Semantics of UML Statecharts in PVS. In *Proc. of the 12th Nordic Workshop on Programming Theory (NWPT'00)*, Norway, 2000.

[11] E. Astesiano. UML as Heterogeneous Multiview Notation. Strategies for a Formal Foundation. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, eds., *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98) – Workshop on Formalizing UML. Why? How?*, Canada, 1998.

[12] 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.

[13] E. Astesiano, M. Cerioli, and G. Reggio. Consistency Issues in Multiview Modelling Techniques. Invited talk at WADT'2002, Germany, 2002.

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

[15] C. Attiogbé, A. Francheteau, J. Limousin, and G. Salaün. ISA, a Tool for Integrated Specifications Animation. Available at http://www.sciences.univ-nantes.fr/info/perso/permanents/salaun/ISA/isa.html.

[16] T. Basten and J. Hooman. Process Algebra in PVS. In W. R. Cleaveland, ed., *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 270–284, The Netherlands, 1999. Springer-Verlag.

[17] D. Bert and F. Cave. Construction of Finite Labelled Transition Systems from B Abstract Systems. In *Proc. of the Second International Conference on Integrated Formal Methods (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 235–254, Germany, 2000. Springer-Verlag.

[18] M. Bidoit. *Pluss, un langage pour le développement de spécifications algébriques modulaires*. PhD, Université de Paris-Sud – Centre d'Orsay, 1989.

[19] J.-P. Bodeveix, M. Filali, and C. Munoz. A Formalization of the B Method in Coq and PVS. In *B Users Group Meeting – Applying B in an industrial context: Tools, Lessons and Techniques (FM'99)*, pages 32–48, France, 1999. Springer-Verlag.

[20] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, eds., *International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, France, 1998. Elsevier Science.

[21] J.-M. Bruel and R. B. France. Transforming UML models to Formal Specifications. In P.-A. Muller and J. Bézivin, eds., *Proc. of the International Conference on the Unified Modelling Language: Beyond the Notation (UML'98)*, number 1618 in LNCS, France, 1998. Springer-Verlag.

[22] R. Büssow and M. Weber. A Steam-Boiler Control Specification with Statecharts and Z. In J.-R. Abrial, E. Börger, and H. Langmaack, eds., *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler*, volume 1165 of *Lecture Notes in Computer Science*, pages 109–128. Springer-Verlag, 1996.

[23] 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.

[24] M. V. Cengarle and A. Knap. A Formal Semantics for OCL 1.4. In M. Gogolla and C. Kobryn, eds., *Proc. of the Fourth International Conference on the Unified Modeling Language (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 118–133, Canada, 2001. Springer-Verlag.

[25] C. Choppy, P. Poizat, and J.-C. Royer. A Global Semantics for Views. In T. Rus, ed., *Proc of the 8th International Conference on Algebraic Methodology And Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180, USA, 2000. Springer-Verlag.

[26] T. Clark and J. Warmer, eds.. *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[27] 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.

[28] CoFI. The Common Framework Initiative for Algebraic Specification and Development, electronic archives. Notes and Documents accessible by WWW[3] and FTP[4].

[29] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics (version 1.0). Note S-9 in [28], 2000.

[30] 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.

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

[32] N. Dershowitz. Orderings for Term-Rewriting Systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

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

[3] `http://www.brics.dk/Projects/CoFI`

[4] `ftp://ftp.brics.dk/Projects/CoFI`

[34] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.

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

[36] The Precise UML Group. `http://www.cs.york.ac.uk/puml/`.

[37] 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*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[38] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[39] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[40] ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

[41] 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.

[42] H. Kirchner and C. Ringeissen. Executing CASL Equational Specifications with the ELAN Rewrite Engine. Note T-9 in [28], November 2000.

[43] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds., *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, UK, 1992.

[44] S. Kolyang, T. Santen, and T. Wolff. A Structure Preserving Encoding of Z in Isabelle/HOL. In J. Von Wright, J. Grundy, and J. Harrison, eds., *Proc. of the International Conference on Theorem Proving in Higher Order Logic (TPHOL'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 283–298, Finland, 1996. Springer-Verlag.

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

[46] K. Lano, K. Androutsopoulos, and D. Clark. Structuring and Design of Reactive Systems Using RSDS and B. In T. S. E. Maibaum, ed., *Proc. of the International Conference on Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lecture Notes in Computer Science*, pages 376–390, Germany, 2000. Springer-Verlag.

[47] 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)*, pages 25–34, UK, 2000. IEEE Computer Society Press.

[48] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[49] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini and R. Gorrieri, eds., *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.

[50] H. Ledang and J. Souquières. Contributions for Modelling UML State-Charts in B. In M. Butler, L. Petre, and K. Sere, eds., *Proc. of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *Lecture Notes in Computer Science*, pages 109–127, Finland, 2002. Springer-Verlag.

[51] J. Lilius and I. P. Paltor. The Semantics of UML State Machines. Technical report 273, Turku Centre for Computer Science, 1999. Available at `http://www.tucs.fi/Publications/techreports/TR273.php`.

[52] J. Lilius and I. P. Paltor. UML: A Tool for Verifying UML Models. In *Proc. of the International Conference on Automated Software Engineering (ASE'99)*, pages 255–258, USA, 1999. IEEE Computer Society. Extended version available at `http://www.tucs.fi/Publications/techreports/TR272.php`.

[53] O. Maury, C. Oriat, and Y. Ledru. Invariants de liaison pour la cohérence de vues statiques et dynamiques en UML. In *Actes de la conférence sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'01)*, pages 23–37, Nancy, France, 2001.

[54] E. Meyer. *Développements formels par objets : utilisation conjointe de B et d'UML*. PhD, LORIA – Université Nancy 2, 2001.

[55] T. Mossakowski. Simplified Heterogeneous Specification. Available at `http://www.informatik.uni-bremen.de/~till/publications.html`, 2002.

[56] T. Mossakowski and B. Krieg-Brückner. Static Semantic Analysis and Theorem Proving for CASL. In F. Parisi-Presicce, ed., *Recent Trends in Algebraic Development Techniques, 12th Workshop on Algebraic Development Techniques (WADT'97)*, volume 1376, pages 333–348. Springer-Verlag, 1998.

[57] H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD, CEDRIC – CNAM, 1998.

[58] T. Nipkow and L. C. Paulson. Isabelle-91. In D. Kapur, ed., *Proc. of the 11th International Conference on Automated Deduction (CADE'92)*, volume 607 of *Lecture Notes in Computer Science*, pages 673–676, USA, 1992. Springer-Verlag.

[59] Object Management Group. *Unified Modelling Language Specification, version 1.4*, September 2001.

[60] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL: A CASL Extension for Dynamic Reactive Systems - Summary. Technical report DISI-TR-99-34, DISI - Università di Genova, 1999. Revised February 2000.

[61] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, ed., *Proc. of the International Conference on Fundamental Approaches to Software Engineering (FASE'2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 171–186, Italy, 2001. Springer-Verlag.

[62] G. Reggio and L. Repetto. Casl-Chart: A Combination of Statecharts and of the Algebraic Specification Language Casl. In T. Rus, ed., *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *Lecture Notes in Computer Science*, pages 243–257, USA, 2000. Springer-Verlag.

[63] G. Reggio and R. Wieringa. Thirty one Problems in the Semantics of UML 1.3 Dynamics. In R. France, J.-M. Bruel, B. Henderson-Sellers, A. Moreira, and B. Rumpe, eds., *Proc. of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99) – Workshop "Rigorous Modelling and Analysis of the UML: Challenges and Limitations"*, USA, 1999.

[64] G. Salaün, M. Allemand, and C. Attiogbé. Formal Framework for a Generic Combination of a Process Algebra with an Algebraic Specification Language: an Overview. In *Proc. of the 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, IEEE Computer Society Press, pages 299–302, China, 2001.

[65] G. Salaün, M. Allemand, and C. Attiogbé. A Method to Combine any Process Algebra with an Algebraic Specification Language: the π-Calculus Example. In *Proc. of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, IEEE Computer Society Press, England, 2002.

[66] G. Salaün, M. Allemand, and C. Attiogbé. Specification of an Access Control System with a Formalism Combining CCS and CASL. In *Proc. of the 7th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'02)*, IEEE Computer Society Press, USA, 2002.

[67] E. Sekerinski. Graphical Design of Reactive Systems. In D. Bert, ed., *Proc. of the 2nd International B Conference (B'98)*, volume 1393 of *Lecture Notes in Computer Science*, pages 182–197, France, 1998. Springer-Verlag.

[68] E. Sekerinski and R. Zurob. iState: A Statechart Translator. In M. Gogolla and C. Kobryn, eds., *Proc. of the 4th International Conference on the Unified Modeling Language (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 376–390, Canada, 2001. Springer-Verlag.

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

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

[71] H. Tej and B. Wolff. A Corrected Failure-Divergence Model for CSP in Isabelle/HOL. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, eds., *Proc. of the International Formal Methods Europe Symposium (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, Austria, 1997. Springer-Verlag.

[72] I. Traoré. An Outline of PVS Semantics for UML Statecharts. *Journal of Universal Computer Science (JUCS)*, 6(11):1088–1108, 2000.

[73] M. van der Beeck. Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn, eds., *Proc. of the 4th International Conference on the Unified Modeling Language (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 406–421, Canada, 2001. Springer-Verlag.

[74] D. Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In A. Corradini and H.J. Kreowski, eds., *Proc. of the 1st International Conference on Graph Transformation (ICGT'2002)*, volume 2505 of *Lecture Notes in Computer Science*, Spain, 2002. Springer-Verlag.

[75] M. Weber. Combining Statecharts and Z for the Design of Safety-Critical Control Systems. In M. Gaudel and J. Woodcock, eds., *Proc. of the 3rd International Symposium of Formal Methods Europe (FME'96)*, volume 1051 of *Lecture Notes in Computer Science*, pages 307–326, UK, 1996. Springer-Verlag.

LABORATOIRE DE MÉTHODES INFORMATIQUES

Integration of Formal Datatypes within State Diagrams

Christian Attiogbé, Pascal Poizat, Gwen Salaün

Abstract

In this technical report, we present a generic approach for the integration of 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. We herein present first the formal foundations of our approach and then a case study is used to demonstrate its pragmatism.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Formal and Semi-formal Specifications Integration, State Diagrams, UML, Algebraic Specifications, Z, B

LaMI, UMR 8042 CNRS, Genopole & Université d'Évry Val d'Essonne
Tour Évry 2, 523 place des terrasses de l'Agora
F-91000 ÉVRY