Plugging Data Constructs into Paradigm-Specific Languages: towards an Application to UML *

Egidio Astesiano, Maura Cerioli and Gianna Reggio

DISI–Dipartimento di Informatica e Scienze dell'Informazione,
Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy,
e-mail: {astes,cerioli,reggio}@disi.unige.it

Abstract. We are interested in the composition of languages, in particular a data description language and a paradigm-specific language, from a pragmatic point of view. Roughly speaking our goal is the description of languages in a component-based style, focussing on the data definition component. The proposed approach is to substitute the constructs dealing with data from the "data" language for the constructs describing data that are not specific to the particular paradigm of the "paradigm-specific" language in a way that syntax, semantics as well as methodologies of the two components are preserved.
We illustrate our proposal on a toy example: using the algebraic specification language CASL, as data language, and a "pre-post" condition logic *à la Hoare*, as the paradigm specific one.
A more interesting application of our technique is fully worked out in [16] and the first step towards an application to UML, that is an analysis of UML from the data viewpoint, following the guidelines given here, is sketched at the end.

Introduction

Languages, including specification languages, oriented to particular paradigms, like for instance object-oriented, (higher-order) functional, concurrent languages, frequently have poor, or insufficiently investigated constructs to specify the used data, sometimes even no constructs at all.

This is somehow reasonable, because the main efforts of the language designers are spent, especially if the language is part of an open research proposal, on the most interesting constructs, those qualifying the language itself and the attention of the users should focus on the paradigm-specific part. Moreover, in many cases it is possible to realize the interesting data structures using the paradigm-specific constructs (for instance, static data by object-oriented classes,

* Partially supported by CoFI, ESPRIT Working Group 29432, and Murst - Saladin (Software Architecture and Languages to coordinate Distributed Mobile Components).

queues and stacks by CCS processes, see, e.g., [15]), though this solution is not very efficient and leads to unnaturally complicated specifications (programs).

Though many designers (mainly in the "formal method community") regard the data part as not relevant, this opinion cannot be shared from the pragmatic point of view of an end user, because any realistic application requires some nontrivial data. Indeed, for instance, the first attempt to use CCS [15] for some realistic applications, precisely protocol specification, led to design LOTOS [12], where powerful constructs for data were provided by an algebraic specification language ACTONE [7]. Moreover, in the literature of the last decades many attemtps to combine paradigm-specific languages with data languages have been presented. For example, combinations of an algebraic specification language with

- a process calculus: other than LOTOS, we have PSF, [13, 14], a combination of a process calculus similar to ACP with ASF (Algebraic Specification Formalism) [4];
- Petri nets [17]: many proposals see, e.g., [18, 2, 6, 23];
- statecharts [9]: see in [16] a combination obtained following the method proposed here.

But, there are also combinations using Z for the data part, for example with statecharts [5, 24], and with an object-oriented language as Object-Z [19].

On the other hand, fixing some (even powerful) data constructs in a paradigm-specific language can restrict the community of potential users, if some of them are already familiar with different style(s) of data description or if some community has very specific needs in terms of datatypes.

Thus, the best solution, in our opinion, is to divorce the choice of the language for the data definition (from now on DL) from that of the language for paradigm-specific constructs (from now on PSL) as much as possible, aiming at a sort of modular language design, or *component based* language design, where the data part can be plugged in.

Since the existing languages have not been introduced in this style, we first have to cope with the problem of extracting from their descriptions the *components* we want to combine. Therefore, we have to analyze the constructs of a paradigm-specific language in order to identify the "just-data" part, that we will replace by our DL. Section 1 will be devoted to the description of a layering method.

Once we have fixed the components we want to combine, the next step is the definition of (the semantics of) the resulting composition.

Several examples of compositions in literature are the origin of new theories and techniques, to get a more powerful framework, where the constructs of both languages can be given a semantics (e.g., algebras as states, concurrent ML).

Another approach quite widespread to solve this problem is using the features of one language/formalism to express concepts of both the data and the paradigm-specific levels, in particular, for instance, coding one of the languages into the other (e.g., the use of Z to specify processes seen as sets of traces; the use of a higher-order logic language to specify axiomatically processes seen as infinite traces modelled as functions; LTL [1], where the algebraic specification techniques and related constructs are extended to handle processes seen as labelled transition systems).

Neither solution is palatable to us, for pragmatic reasons. Indeed, in either case one or both the starting languages (or at least their semantics) is redefined, so that tools, results, theorems and "pieces of code" cannot be simply inherited but have to be somehow adjusted (validated). In the most drastic cases, even the "programming" techniques cannot be lifted from the original framework(s) to the combination and this is clearly unacceptable from the user point of view. We want to keep the original semantics of each construct, building a mechanism to pass the results of computations from one framework to the other.

Therefore, in Section 2, we will sketch a method to integrate the semantics of the two parts that does not require new theory. Of course there is no guarantee that such a method works in general; indeed, we need (very few and most reasonable) assumptions on the way the syntax and the semantics of the two starting languages are given. We will use as a running example the integration of the CASL algebraic specification language and of HLL a simple logic *à la Hoare*.

1 A Data-Driven Language Taxonomy

The problem we are tackling is the composition of two languages: ideally, one of them, say PSL, having constructs for some specific paradigm, that we will call *paradigm-specific language*, but with questionable or even missing constructs for datatypes; the other, say DL, providing a rich language for describing datatypes, that we will call *data language*. It is interesting to note that from any paradigm-specific language having a reasonable set of constructs for data definition, like for instance ML, the data sublanguage can be extracted and used as DL in our construction.

In Table 1 and Table 2 we present examples of both categories of languages.

CASL [21]	algebraic specification language
Z [20]	mathematical oriented notation
ML (the purely functional sublanguage of) [11]	functional programming language
PASCAL data sublanguage	the PASCAL sublanguage consisting of the type declarations and of the expressions

Table 1. Data Languages

1.1 Data (Part of a) Language

In order to illustrate the concepts on our running example, let us start with a short introduction on the data language we are going to use, that is CASL.

STATECHART [9, 10]	visual notation for reactive systems
CCS with value passing [15]	specification language for processes communicating by handshaking along channels
CCS with value passing and parametric channels	specification language for processes communicating by handshaking along channels, where also channel names may be communicated
Petri nets [17]	visual notation for concurrent parallel systems
PASCAL	imperative programming language
Linear time temporal logic	a logic for processes
ML [11]	functional programming language
UML plus OCL [22]	visual object oriented notation (a subset/profile having a formal semantics)

Table 2. Paradigm Specific Languages

Casl (Common Algebraic Specification Language) The specification language CASL has been designed within the CoFI [1] initiative, an open group with representatives from almost all the European groups working in this area, started by ESPRIT BRA COMPASS in cooperation with IFIP WG 1.3 (Foundations of Systems Specification) and recently evolved in an ESPRIT Working Group.

Though CASL is reasonably concise and expressive, restrictions (e.g., for interfacing with existing tools) and extensions (e.g., for dealing with specific applications, like reactive systems) are expected to be defined. Indeed, CASL has been planned from the beginning as a family of languages coherent on the common parts and what we call CASL should be more precisely called *basic* CASL.

We have no pretension to describe the syntax nor the semantics of CASL, for which we refer to [21, 3] or to the web site:
`http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/index.html`.
We shortly list the main features of CASL, as a reminder for readers already familiar with the language.

Basic specifications allow the declaration (definition) of sorts, subsorts, functions (both partial and total) and predicates. Terms are built, starting from typed variables, by applying functions to terms of any subsort of the expected argument sort (that is, cohercion to the supersort is implicit) or projecting a term t of some sort onto a subsort s (t *as* s). Terms are used to state axioms, that are first-order formulae built from equations, predicate applications (with the same rules as for function application), definedness assertions, and subtyping membership predicates. Finally, datatype declarations are provided for concise specification of sorts together with some constructors and (optional) selectors.

The semantics of a basic specification consists of a signature of the form $(S, TF, PF, P, \leq_S)$ (where S is the set of *sorts*, TF and PF are respectively *total* and *partial functions*, P are *predicates* and $\leq_S$ is the *subsorting relation*) and a class of many-sorted partial first-order structures, i.e., algebras where the functions are partial or total, and where also predicates are allowed.

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

Structured specifications provide constructs for translation, reduction, union, (free) extension[2] of specifications. Generic specifications may also be defined. The semantics of structured specifications belongs to the same domain as that of basic specifications.

Other important features of CASL are the *architectural specifications* and the *specifications libraries*, that are not relevant here.

Data Language Taxonomy The parts of a language concerning data are basically those used to describe types (if the language is typed, of course), their elements (the data used in the computations) and to express conditions on the data. Therefore, we are working in a context where we have the following *ingredients*.

Types The types are sets, and are denoted by *type expressions* (in CASL, sorts), that are built using *type constructors*, whose semantics is a function resulting in a set (of values in a suitable universe). Such type constructors are sometimes user defined and introduced by definitional constructs. Moreover, we have a set of rules describing how the type constructors are used to build correct type expressions.

In CASL there are no type constructors, but only a definitional mechanism for specifications, giving the semantics of sorts together with that of functions and predicates.

Elements The elements are the individual elements of types, called *data*. Analogously to the type case, data are denoted by *expressions*, that are built using *operations*, interpreted as functions on data (so having types as source and target) and such operations may be user defined and introduced by definitional constructs. For instance, in CASL operations correspond to functions, introduced by function declarations and function definitions or as constructors/selectors of data types. Again we have rules expressing how operations are used to build correct expressions. In most cases, we are interested not only in closed (constant) expressions, but in open expressions as well, used for instance as bodies of functions (the variables play the role of parameters) or in conditions (the variables are somehow quantified). The evaluation of such expressions is based on concepts like *environments* (or *evaluations*) and the mechanism is always the same: variables are added as possible basic expressions and the evaluation of expressions is based on an extra parameter (the environment) used to give the value of this new basic case, while the semantics of other constructs remains the same, with this silent extra parameter.

In CASL expressions are called terms and are built by function application, starting from constants and (logical) variables. The functions that can be used are those declared in specifications, embedding into supersorts and projections onto subsorts.

Conditions The conditions are logical formulae used to express properties on data, and are built using *predicates* (i.e., functions yielding values in a set of

[2] Initiality is a special case of this construction.

special *truth values*), *logical connectives* and *quantifiers*. In particular, such predicates are quite commonly user defined and introduced by definitional constructs; this is the case of CASL, where predicates can be user defined. Again we have rules expressing how conditions are built using these ingredients.

In CASL conditions correspond to the non-terminal `FORMULA`, whose elements are partial first-order formulae.

In the following we will use *data language* to denote languages having only the above constructs. In the literature there are several examples where data languages are actually the declarative part of larger languages and we first have to evict that part.

1.2 Paradigm-Specific Language

Let us now sketch a rather simple variant of Hoare's logic to be used as our running example of paradigm-specific language, apt to express requirements on elementary imperative programs, consisting of sequences of statements using only global variables (programs here for short).

HLL (Hoare's Logic Language) A HLL specification consists of a collection of imperative variable declarations, together with a logical constraint on state transformation.

Syntax

HSPEC	::=	use DEC*; constr COND
DEC	::=	VAR : TYPE;
TYPE	::=	B_TYPE | Pointer B_TYPE
B_TYPE	::=	Int | Real
COND	::=	ATOM | (COND) | COND $\wedge$ COND | COND $\vee$ COND | $\neg$COND | COND $\Rightarrow$ COND
ATOM	::=	EXP $=$ EXP | IEXP $<$ IEXP | REXP $<$ REXP
EXP	::=	IEXP | REXP | nil | VAR
IEXP	::=	NUM | IEXP $+$ IEXP | IEXP $-$ IEXP | IEXP $*$ IEXP | LEXP
REXP	::=	R-NUM | REXP $+$ REXP | REXP $-$ REXP | REXP $*$ REXP | REXP/REXP | LEXP
LEXP	::=	VAR | VAR$\uparrow$ | @VAR | @VAR$\uparrow$

The actual form choosen to represent variables and numbers is immaterial; hence we do not list the corresponding productions.

$\uparrow$ denotes pointer dereferencing and `@` the initial value of variables.

The rules for building correct expressions are the standard ones, about the correct typing (and existence) of variables, and are based on static environments $SENV = [\texttt{VAR} \to_p \texttt{TYPE}]$, associating types with variables. In the following, we use $\to_p$ to denote partial functions and, for each $f : A \to_p B$, $D(f) \subseteq A$ to denote the definition domain of f.

Static environment are created by declarations accordingly to the following rules.

$$\frac{}{\vdash \Lambda \rhd \emptyset} \qquad \frac{\vdash decs \rhd se}{\vdash decs\ x : t; \rhd se[t/x]}\ x \notin D(se)$$

Let us now simply sketch some judgments using static environments. Notice that the judgments on expressions are explicitly giving the type, using the form $se \vdash exp : t$, while those on conditions, for instance, are simply stating their correctness, using the form $se \vdash cond$.

$$\frac{}{se \vdash x : se(x)}\ x \in D(se) \qquad \frac{se \vdash x : t}{se \vdash \mathtt{@}x : t}$$

$$\frac{se \vdash x : \mathtt{Pointer}\ t}{se \vdash x\uparrow : t} \qquad \frac{se \vdash x : \mathtt{Pointer}\ t}{se \vdash \mathtt{@}x\uparrow : t}$$

$$\frac{}{se \vdash \mathtt{nil} : \mathtt{Pointer}\ t} \qquad \frac{se \vdash exp_1 : t \quad se \vdash exp_2 : t}{se \vdash exp_1 = exp_2}$$

Semantics HLL is used to express properties on *programs*, written in any given imperative language. Programs are classified accordingly to the typed variables they use, i.e., a static environment, and their semantics describes a state transformation.

In the following, let $\wp$ denote the power set function and $\mathbb{B}$ the domain of standard first-order logic; we will denote truth by T and falsity by F, and use the standard connectives of such logic with standard notation as well. Moreover, let LOC be a denumerable set (of locations) and $\underline{n} \notin LOC$ a distinct value corresponding to the `nil` pointer.

$$\begin{array}{l}
VALUE = \mathbb{R} \cup \mathbb{Z} \cup LOC \cup \{\underline{n}\} \\
ENV = [\mathtt{VAR} \to_p LOC] \\
STATE = [LOC \to_p VALUE] \\
\mathcal{P} : \mathtt{HSPEC} \to [[(ENV \times STATE) \to_p STATE] \to \mathbb{B}] \\
\mathcal{T} : \mathtt{TYPE} \to \wp(VALUE) \\
\mathcal{D} : \mathtt{DEC}^* \to SENV \\
\mathcal{C} : \mathtt{COND} \to [(ENV \times STATE \times STATE) \to \mathbb{B}] \\
\mathcal{E} : \mathtt{EXP} \to [(ENV \times STATE \times STATE) \to_p VALUE]
\end{array}$$

Since locations are not typed and there are elements (like $\underline{n}$) having more than one type, we have to introduce a mechanism to match the static information to environments and states.

For each static environment se let $(ENV \times STATE)_{se}$ consist of all pairs (ρ, σ) s.t. $D(se) = D(\rho)$ and for all x s.t. $se(x) = t$ if $\rho(x) \in D(\sigma)$ then $\sigma(\rho(x)) \in \mathcal{T}(t)$. Moreover, let $Prog_{se}$ be the set of *programs* correctly using the typed variables provided by se, that is $Prog_{se} \subseteq [(ENV \times STATE)_{se} \to_p STATE]$, and for all $p \in Prog_{se}$ if $p(\rho, \sigma) = \sigma'$ then $(\rho, \sigma') \in (ENV \times STATE)_{se}$.

We define the semantic functions by induction, here we just quote some rules, omitting for instance the meaning of operations and predicates that we assume obvious.

- $\mathcal{P}(\mathtt{use}\ decs;\ \mathtt{constr}\ cond)(p) = T$ iff
  $p \in Prog_{\mathcal{D}(decs)}$ and $\mathcal{C}(cond)_{\rho,\sigma,p(\rho,\sigma)} = T$ for all $(\rho, \sigma) \in D(p)$.
- $\mathcal{D}(decs) = se$ iff $\vdash decs \triangleright se$

– $\mathcal{T}(\mathtt{Int}) = \mathbb{Z}, \mathcal{T}(\mathtt{Real}) = \mathbb{R}, \mathcal{T}(\mathtt{Pointer}\ t) = LOC \cup \{\underline{n}\}$
– the semantics of logical connectives is standard.Let us see, for instance, the rule for the equality atoms:

$$\mathcal{C}(exp_1 = exp_2)_{\rho,\sigma,\sigma'} \quad = \quad \begin{cases} T \text{ if } \mathcal{E}(exp_1)_{\rho,\sigma,\sigma'} = v = \mathcal{E}(exp_2)_{\rho,\sigma,\sigma'} \\ F \text{ otherwise} \end{cases}$$

– the rules for the integer and real operations are the obvious ones, relying on the definition of the corresponding mathematical functions; moreover we have

$$\begin{array}{ll} \mathcal{E}(x)_{\rho,\sigma,\sigma'} = \sigma'(\rho(x)) & \mathcal{E}(x\uparrow)_{\rho,\sigma,\sigma'} = \sigma'(\sigma'(\rho(x)))) \\ \mathcal{E}(@x)_{\rho,\sigma,\sigma'} = \sigma(\rho(x)) & \mathcal{E}(@x\uparrow)_{\rho,\sigma,\sigma'} = \sigma(\sigma(\rho(x)))) \end{array}$$

Paradigm Specific Language Taxonomy Consider now a paradigm-specific language PSL. A first distinction we make on its constructs is given by the context needed in order to give them a semantics. Indeed, we have a *data* part, whose semantics is based on set(s) of values, functions and predicates, and a *paradigm-specific* part, whose semantics requires information about the context they are used in. For instance, if we consider HLL, we have that the constructs for `B_TYPE`, those for `REXP` and `IEXP` giving the real and integer operations, and `COND` are the data part, while pointer dereferencing and imperative variables are the paradigm-specific part, as they need an environment and a state in order to be evaluated.

Let us make a more refined analysis within each category of constructs. The results of this analysis for the paradigm-specific languages in Table 2 will be sketched in Table 3.

Data Constructs

paradigm-specific datatypes $\mathsf{PSL}_{\mathrm{par-spec}}$ are the datatypes that are specific to the language paradigm (i.e., such that some construct of the language cannot be properly used/defined if we drop one of them).
Some examples are time for statecharts, boolean for CCS with value passing and ML, sets of object identities for UML, and pointers, with the constant `nil`, for HLL.

basic data language $\mathsf{PSL}_{\mathrm{basic}}$ is the sublanguage of PSL used to handle the data that are not paradigm-specific, that is those that are not crucial for the language and can be safely substituted; it is a data language.
This part is empty for several paradigm-specific languages (e.g., classical Petri nets), or it is very poor, consisting just of a few predefined datatypes. Some examples are integers and reals for HLL, the part of ML concerning the "datatypes" and the "abstract datatypes", and the generic values for CCS with value passing.

The combination of $\mathsf{PSL}_{\mathrm{basic}} \oplus \mathsf{PSL}_{\mathrm{par-spec}}$ is the sublanguage of PSL that is a data language.

Paradigm-Specific Constructs

pseudo-data constructs $\mathsf{PSL}_{\mathrm{pseudo}}$ are those language constructs (constituents) used to build correct expressions (type expressions, conditions) that are not directly denoting a data or an operation but need some extra input to yield one.
Some examples are updatable variables in imperative languages (yielding a value for given environment and state), function parameters in functional languages (yielding a value for given environment built by the call), the system function returning the current time (yielding a value given the state), the "happened" construct of statecharts (checking whether an event has happened and yielding a boolean value for given history of the system). In particular, in HLL, we have imperative variables and pointer dereferencing.

kernel constructs $\mathsf{PSL}_{\mathrm{ker}}$ are those language constructs (constituents) that are not involved in data description, though they can *use* data.
Some examples are statements and subroutines in imperative languages, methods, classes and objects in object-oriented languages, the diagrams of visual languages, as UML and STATECHART. In HLL this part is the main construct of `HSPEC`.

2 A Method for the Combination

Let us now shortly sketch how we can safely realize the combination of the two languages, that will be made at the level of the abstract syntax, as it is where their structure is more evident. We cannot, of course, give a method for the combination of *any* pair of languages; thus let us start with some assumptions.

2.1 Assumption on DL and PSL

First, we assume that both in the data and in the paradigm-specific languages, there are just one syntactic category corresponding to type expressions, one corresponding to expressions and one corresponding to conditions. This requirement makes clear the correspondence among the syntactic categories of DL with the corresponding categories of PSL.

For each of these categories, the possible productions are in the following forms:

```
TYPE    ::= TYPE-CONST(TYPE*) | TYPE-ITER(DEC*, {TYPE | EXP | COND}*)
EXP     ::= OP(EXP*) | EXP-ITER(DEC*, {TYPE | EXP | COND}*) | VAR
COND    ::= PRED(EXP*) | CON(COND*) | COND-ITER(DEC*, {TYPE | EXP | COND}*)
DEC     ::= VAR : TYPE;
```

where `TYPE-CONST` are the type constructors, `OP` are operations, `PRED` are predicates and `CON` are logical connectives. That is, we have two kinds of constructs for each non-terminal, that are the application of some constructor to its arguments and iterators, binding typed variables (representing elements) in open terms.

PSL	data part		paradigm-specific part	
	PSL_{basic}	$\mathsf{PSL}_{par-spec}$	PSL_{pseudo}	PSL_{ker}
STATECHART	integers, chars, ...	time, *states* (of the chart), *events*	variables, happened, is_in_state, since, before, at	visual part, actions
CCS with value passing	generic values	booleans	variables used in input prefixes	combinators ($\|$, +, ...)
CCS with parametric channels	generic values	booleans, *channels*	variables used in input prefixes	combinators ($\|$, +, ...)
Petri nets		sets of tokens with operations $\subseteq$, $\cup$, $-$ and $\{\_\}$		visual part
PASCAL	integers, reals, array, record...	boolean, pointer types, ordered types, enumeration types	constants, variables, parameters of functions and procedures, function calls, pointer dereferencing	declarations except those for types, statements
UML with OCL	OCL basic types	classes (evaluated as sets of object identities), *associations*, *states* of statecharts, *type of all types*, collections (sequences, sets, bags), boolean, iterators, has_type, ...many others	attribute selection, right/left application of association, all instances of a class, operation parameters...many others	UML
Linear time (homogeneous) temporal logic	a unique type, with functions and predicates defining the logical language, logical connectives and rigid quantifications		flexible variables, temporal connectives	
ML	basic types, datatypes, abstract datatypes	boolean, functional and product types	function parameters	function declaration

where the spurious type are emphasized.

Table 3. Taxonomy of Paradigm-Specific Languages

Some examples of iterators are λ (returning functional values), the logical quantifiers, the subsort definition by means of a condition in CASL, and the set-comprehension notation. As usual, the language will consist only of elements without free occurrences of variables.

The static semantics has to provide correctness judgments, using static environment memorizing context information (as the types of the introduced variables).

In general, languages are given by grammars that do not comply to the proposed schema. Therefore, we may have to massage somehow the abstract syntax in order to put it in the correct form, moving some information into the static semantics part.

This is, for instance, the case of HLL; so we rephrase its grammar accordingly to the wanted pattern[3].

```
HSPEC     ::=  use DEC*; constr COND
DEC       ::=  VAR : TYPE;
TYPE      ::=  Int  | Real | Pointer TYPE
COND      ::=  PRED(EXP*) | CON(COND*)
PRED      ::=  =|<
CON       ::=  ∧ | ∨ | ¬  |⇒
EXP       ::=  OP(EXP*) | VAR | @EXP | EXP↑
OP        ::=  + | ∗ | − | / | nil | ...
```

Extra judgements have to be added to the static semantics, like for instance those restricting the application of the pointer type constructor to basic types only.

$$\frac{}{\vdash \texttt{Int} : BType} \qquad \frac{}{\vdash \texttt{Real} : BType} \qquad \frac{\vdash t : BType}{\vdash \texttt{Pointer}\ t : PType}$$

Moreover, some rules have to be restricted, because some verifications originally managed by the context-free grammar have to be incorporated.

This is the case, for instance, of the rule for variable declaration, where the correctness of the type expression has to be checked as well

$$\frac{\vdash decs \rhd se}{\vdash decs\ \ x : t; \rhd se[t/x]}\ x \notin D(se) \text{ and } (\vdash t : BType \text{ or } \vdash t : PType)$$

or pointer dereferencing, where the expression has to be a pointer variable.

$$\frac{se \vdash x : \texttt{Pointer}\ t}{se \vdash x\uparrow : t}\ x \in \texttt{VAR} \text{ and } \vdash \texttt{Pointer}\ t : PType \text{ or } \vdash t : PType$$

There is no need for changing the semantics, as the language generated is still the same. The only difference between this version and the original one is in the way the language is defined, because several "errors" that were intercepted by a finer context free grammar are now captured by the static semantics rules.

Having the static checks of either language given in this way allows us to immediatly apply them to the terms produced by constructs of the other language as well, while if the same requirements are captured using more terminals

[3] This is the case for CASL as well, but we are not going into the details of the rephrasing that is straightforward but long to be described.

we should somehow distribute the new cases (productions) among the auxiliary non-terminals, possibly introducing other categories as well.

In several examples the construction schema for `TYPE` (`EXP`, `COND`) requires some extra ingredients (other syntactic categories). In such cases the standard solution is to introduce *spurious* types, with very restricted uses, whose semantics (that is a set) has to be provided as well. For instance, the condition **is_in**(st), that appear in the data language of statecharts, uses st that is a state of the statechart and, thus, not a data of the language. Technically, we can deal with this, by introducing a spurious type `STATE`, denoting the set of the states of the statechart itself. This technique is also followed in the constraint language of UML (OCL), where classes, associations etc. are introduced as extra types.

The (dynamic) semantics has to be given compositionally following the structure of the abstract syntax, only for correct elements, of course.

2.2 Adding Paradigm-Specific Data

To combine a language PSL with a data language (a data component) DL means essentially to replace $\mathsf{PSL}_{\mathrm{basic}}$ with DL. This substitution is done in two steps, accordingly to the stratification among the constructs of PSL induced by our analisys on the use of data in PSL.

The first step is the combination of the "new" data from DL with the paradigm-specific data of PSL, $\mathsf{DL}_{\mathrm{ext}} = \mathsf{DL} \oplus \mathsf{PSL}_{\mathrm{par-spec}}$, while the second one will be discussed in the next session.

There are (at least) two cases, general enough to jointly cover most pragmatic examples, that guarantee a safe combination of DL and $\mathsf{PSL}_{\mathrm{par-spec}}$:

- the datatypes in $\mathsf{PSL}_{\mathrm{par-spec}}$ can be implemented or realized in DL; in particular this is the case when they already are included in DL (e.g., a boolean type or a time type realized by a Casl specification or a Z type).
  This is indeed the technique we adopt for the combination of HLL and Casl, using the following specifications:
  - the first is the axiomatization of the domain of locations; as we only know only that it is a denumerable set, we inductively build it starting from the "first location" and a "successor" operation.

    spec `LOC` = **freetype** `Loc` ::= `loc`$_0$ | `new(Loc)`
  - using the above specification of locations, we build the data type of pointers for any given type (sort) s, that is (up to isomorphism) the disjoint union of locations and the extra value to represent the nil operation.

    spec `POINTER`[**sort** s] **given** `LOC` =
    freetype `Pointer`[s] ::= `nil` | **sort** `Loc`

  These "standard" declarations should be considered implicitly added to any program of DL used in the combination language. If this approach is possible, it is definitely the most convenient, as the combination $\mathsf{DL}_{\mathrm{ext}}$ is the same as DL, in the sense that each data description in it is also a data description in DL, and hence tools, theorems and methodologies can be directly reused.

- the types in DL are defined by means of type constructors, that expect as input only other types and hence their semantics is a function over sets of data. In this case we simply have to add to DL some new type constructors, with the related operations and predicates, corresponding to the datatypes in $\mathsf{PSL}_{\mathrm{par-spec}}$ and their integration is given by the fact that the DL type constructors can deal with any set[4] given in input, so in particular with the semantic interpretation of the paradigm-specific datatypes.
  This setting is general enough; it is satisfied for instance by most programming languages, but not by standard algebraic specification languages (where the semantics of type expressions is given together with, and mutually dependent on, the semantics of operations and predicates).

At the end of this phase, we have built a new data language describing a class of structures. Each structure in the class provides the semantics for type expressions, operations and predicates; hence it is some kind of algebra (first-order structure). Now, this "algebra" has to be used as basis for the semantics of the rest of the combination language.

2.3 Adding the Paradigm-Specific Part

We have to adjust the pseudo-data of PSL, as well as the kernel constructs, to work on the data language $\mathsf{DL}_{\mathrm{ext}}$.

In general this is possible, because their semantics is described in a sort of generic or polymorphic compositional style, working on any structure (collection of sets, functions and predicates, so basically an algebra) satisfying some requirements. In most cases, the kind of algebras built by $\mathsf{DL}_{\mathrm{ext}}$ (from now on DL-algebras) is not the same as the kind of algebras expected by the paradigm-specific constructs of PSL (from now on PSL-algebras). Therefore, we have to provide a *conversion function* from DL-algebras into PSL-algebras. This is usually possible if all concepts expressible in DL-algebras are expressible in PSL-algebras as well. For instance, if we use as DL the algebraic specification language CASL, then the DL-algebras are partial algebras. Now, if we want to combine CASL with an imperative language, whose semantics is given in terms of domains, then we have to transform the representation of undefinedness of partial algebras into the representation of undefinedness of domains, that is the bottom element. Since it is straightforward to make such conversion (by standard strict totalization techniques), we are able to combine such languages. But if we try to combine CASL with a paradigm-specific language that has no notion of partiality, then this cannot be done preserving the semantics and the methodology of the paradigm-specific language. In such case, hence, our method cannot be applied.

If we are able to make the conversion from DL-algebras into PSL-algebras, then the compositionality of the semantics allows to delegate the evaluation

[4] Of course some compatibility among the universes fixed for the semantics of the two languages has to be assumed.

of subcomponents to the (known) semantics of either DL_{ext} or PSL, and the genericity guarantees that the semantic description of the leading operation is able to cope with the results of the subcomputations.

The key points are the adaptation of the extra domains present in the semantics for interpreting the pseudo-data to include the "new" values and the uniform addition of such domains as (silent) arguments to the evaluation rules of data operations from DL_{ext}.

For instance, in our toy example, we have to modify the definition of the set *VALUE* to include the universe of Casl values (hence indirectly modifying *STATE* as well). The evaluation rules for variables and pointers, both before and after execution, are formulated disregarding the actual definition of `VALUES`, so they are sufficiently generic to be able to deal with the new values as well.

2.4 The Resulting Combination

If we have successfully finished our process, we have now a language $\mathsf{DL} \oplus \mathsf{PSL}$ where a "program" consists of a declarative part, that is an acceptable declarative part in DL_{ext}, and a "program" in PSL without the declarative part (if any).

The semantics is given in three steps:

- the declarative part is used to find a (class of) DL-algebra(s)
- DL-algebras are transformed into PSL-algebras
- the result of the previous step is used as context to define the semantics of the "PSL-program", using the rules of the semantics of PSL.

Casl $\oplus$ HLL Let us see the final result of our technique on the toy example we are using to illustrate the methodology.

Syntax The overall grammar consists of all productions of the Casl grammar (in the version rephrased accordingly to the assumptions in Sect. 2.1) plus the following.

`HSPEC`	`::=`	`CASL-SPEC` use `DEC`*; constr `COND`
`DEC`	`::=`	`I-VAR` : `TYPE`;
`TYPE`	`::=`	`SORT` (a nonterminal of the Casl grammar)
`COND`	`::=`	all Casl productions for `FORMULA`, with `TERM` replaced by `EXP` and `FORMULA` by `COND`
`EXP`	`::=`	all Casl productions for `TERM`, with `TERM` replaced by `EXP` | `I-VAR` | `@EXP` | `EXP`$\uparrow$

Notice that in the above grammar there are two kinds of variables: `VAR` is the nonterminal for logical variables of the Casl grammar while `I-VAR` is the nonterminal for imperative (updatable) variables. The new cases of the production for `EXP` concern the pseudo-data constructs.

Static Semantics Let us consider a specification *spec* `use` *decs* ; `constr` *cond*.

In order to verify its static correctness, we first use the CASL judgments on a specification *spec+pointers*, where locations and pointers for each sort declared in *spec* are routinely added.

$$
\begin{array}{l}
\textbf{spec } \mathit{spec{+}pointers} = \\
\quad \texttt{LOC}\ \textbf{and} \\
\quad \left.\begin{array}{l}
\texttt{POINTER}[\textbf{sort } s_1]\ \textbf{and} \\
\vdots \\
\texttt{POINTER}[\textbf{sort } s_n]\ \textbf{and}
\end{array}\right\} s_1 \ldots s_n \text{ are the sorts declared in } \mathit{spec} \\
\textbf{then } \mathit{spec}
\end{array}
$$

Assuming that *spec+pointers* is statically correct, this first step produces a signature $\Sigma = (S, TF, PF, P, \leq_S)$, giving the sorts, (total and partial) functions and predicates that can be used in the following parts.

Then, we use the rules for creating static environments from HLL static semantics, replacing

$$
\frac{}{\vdash \texttt{Int} : \mathit{BType}} \qquad \frac{}{\vdash \texttt{Real} : \mathit{BType}} \quad \text{by} \quad \frac{}{\vdash s : \mathit{BType}}\ s \text{ declared in } \mathit{spec}.
$$

Again assuming that *decs* is statically correct, this second step produces a static environment *se* recording which typed imperative variables can be used in the last part.

Finally, we use the CASL judgements replacing `TERM` and `FORMULA` by `EXP` and `COND`, respectively, adding a silent extra parameter, the static environment, to each judgement and adding also the judgements for the pseudo-data constructs:

$$
\frac{x \in D(se) \quad se(x) = s \quad s \in S}{(S, TF, PF, P), se \vdash x \rhd x_s} \qquad \frac{x \in D(se) \quad se(x) = s \quad s \in S}{(S, TF, PF, P), se \vdash \texttt{@}x \rhd (\texttt{@}x)_s}
$$

$$
\frac{x \in D(se) \quad se(x) = \texttt{Pointer}\ s \quad \vdash \texttt{Pointer}\ s : \mathit{PType}}{(S, TF, PF, P), se \vdash x\!\uparrow \rhd (x\!\uparrow)_s}
$$

It is interesting to note that these judgments are the obvious adaptation of those given for HLL to CASL style, where the correctness of expressions (terms) is described translating them into unambiguous forms, with typing captured by decoration, and requiring that all the forms for one term are equivalent (accordingly to the rules given for subsorting).

Semantics

Let us consider a statically correct specification *spec* `use` *decs* ; `constr` *cond*.

Analogously to the procedure adopted for the static semantics, and using the same notation, we give the semantics of the combination language in three steps.

First the semantics of the CASL specification *spec* determines a class of algebras $\mathcal{M}$ on a signature Σ, that is defined by the standard semantics of CASL for the specification *spec+pointers*. Then, *decs* is evaluated, as in the static semantics, producing a static environment *se*. Finally for each algebra in $\mathcal{M}$ we determine the semantics of the remaining part of the program.

Let us fix a model of *spec+pointers*, that is a Σ-algebra $D \in \mathcal{M}$. In the following let us denote the set of valuations of the logical variables (used by the quantifiers) by VAL, with V_O as empty valuation, i.e., the totally undefined function.

The semantics functions evaluating expressions and conditions now need as input both logical variables evaluations (as in CASL) and environment and state (as in HLL).

$$
\begin{array}{l}
VALUE_D = \cup_{s \in Sorts(\Sigma)} D_s \\
VAL_D = [\mathtt{VAR} \to_p VALUE_D] \\
LOC_D \ = D_{\mathtt{Loc}} \\
ENV_D = [\mathtt{I\text{-}VAR} \to_p LOC_D] \\
STATE_D = [LOC_D \to_p VALUE_D] \\
\mathcal{P} : \mathtt{HSPEC} \to [[(ENV_D \times STATE_D) \to_p STATE_D] \to \mathbb{B}] \\
\mathcal{D} : \mathtt{DEC}^* \to SENV \\
\mathcal{C} : \mathtt{COND} \to [(ENV_D \times VAL_D \times STATE_D \times STATE_D) \to \mathbb{B}] \\
\mathcal{E} : \mathtt{EXP} \to [(ENV_D \times VAL_D \times STATE_D \times STATE_D) \to_p VALUE_D] \\
\mathcal{T} : \mathtt{TYPE} \to \wp(VALUE_D)
\end{array}
$$

The semantic functions are defined by induction, using the same rules given, respectively, in CASL and HLL semantics. For instance $\mathcal{T}(t) = D_t$ for each type (sort) t as in CASL.

If necessary, the rules of each language are extended with the extra silent parameters needed by the other language. Like for instance in the definition of the semantics of the main production, where the empty logical variable valuation has to be added to be able to evaluate the constraint:

$$
\begin{array}{l}
\mathcal{P}(spec\ \mathtt{use}\ decs\,;\ \mathtt{constr}\ cond)(p) \quad = \\
\begin{cases} T \text{ if } p \in Prog_{\mathcal{D}(decs)} \text{ and } \mathcal{C}(cond)_{\rho,V_O,\sigma,p(\rho,\sigma)} = T \text{ for all } (\rho,\sigma) \in D(p) \\ F \qquad \text{otherwise} \end{cases}
\end{array}
$$

Let us see a few examples of rules for expressions, that are those where the interaction between CASL and HLL is more significant. The last two are the modification of standard evaluation in algebraic languages (and CASL in particular) for, respectively, function application and logical variable, while the others are the same given for HLL, with a logical variable evaluation as extra (silent) parameter.

$$
\begin{array}{ll}
\mathcal{E}(x)_{\rho,V,\sigma,\sigma'} = \sigma'(\rho(x)) & \mathcal{E}(x\!\uparrow)_{\rho,V,\sigma,\sigma'} = \sigma'(\sigma'(\rho(x))) \\
\mathcal{E}(\mathtt{@}x)_{\rho,V,\sigma,\sigma'} = \sigma(\rho(x)) & \mathcal{E}(\mathtt{@}x\!\uparrow)_{\rho,V,\sigma,\sigma'} = \sigma(\sigma(\rho)) \\
\multicolumn{2}{l}{\mathcal{E}(f(exp_1, \ldots exp_n))_{\rho,V,\sigma,\sigma'} = f^D(\mathcal{E}(exp_1)_{\rho,V,\sigma,\sigma'}, \ldots, \mathcal{E}(exp_n)_{\rho,V,\sigma,\sigma'})} \\
\mathcal{E}(y)_{\rho,V,\sigma,\sigma'} = V(y) &
\end{array}
$$

3 Analysing UML

Though the time is not yet ripe for presenting a full application of the method outlined in this paper to UML, the OMG standard object-oriented notation for specifying, visualizing, constructing, and documenting software systems ([22, 8]), it is interesting to look at UML for the following reasons:

- what is called UML is an outstanding example of a “language design” based on a factorization principle, very much along the lines we have discussed;
- thus, our analysis applied to UML may help understand more deeply its nature and provide a basis for variations and improvements (for example a lot of work for a better standard is underway);
- UML is a *family* of languages, where the members may differ not only because the semantics of some constructs is not fixed (the so-called *semantic variation points*), but also as part of the constructs, e.g., those for the expressions and the actions (the UML teminology for statements), are not defined by the standard but may be imported from other languages, and UML has mechanisms to extend the language itself by defining new constructs (e.g., stereotypes). This encourages the definition of variants using different data languages. Thus, our method of combination, where most work is restricted to the taxonomy of the paradigm-specific language (UML in this case), is especially convenient, because the taxonomy itself could be reused many times.

Unfortunately, an immediate application of our technique to UML is ruled out not only by the lack of a formal semantics for UML, that has not been yet provided, though several attempts are under way, at least for some parts, but also by the visual nature of UML. Indeed, UML “programs” are not strings but instead bunches of diagrams (variants of graphs) possibly annotated with some text. For this reason, its syntax is not given, as usual, by a BNF grammar; instead it is given by the so called *metamodel*, which is essentially an object-oriented description of the UML models[5]. Consequently the “static semantics” of UML is given by means of constraints on such metamodel.

However, we belief that the adaptations required to apply our techniques to UML may be, though non-trivially, worked-out. Therefore, let us sketch briefly how our taxonomy technique apply to UML.

For what concerns the data constructs, following our terminology, we have that UML is, essentially[6], a kernel language without data constructs, whereas all data related constructs can be plugged in to obtain a complete language. The OMG has, however, provided a language for the data constructs called OCL (for Object Constraint Language) [22], so that what is usually known under the name “UML” is instead the combination UML-OCL = OCL $\oplus$ UML. With respect to this combination the UML part is, in our terminology, the kernel UML-$\mathrm{OCL}_{\mathrm{ker}}$.

Thus, the analysis of UML-OCL from a data-constructs viewpoint is in some sense simplified, because the data constructs are already grouped within the OCL component. The only OCL constructs that are not data related are those allowing to attach a constraint (a condition) to an element of an UML model.

[5] Using the UML terminology the UML specifications/programs are called *models*.

[6] We cannot say exactly, because UML allows to define special classes of objects without changeable state and without identity that may be considered a kind of datatypes.

According to our setting, we can thus decompose OCL in the three canonical parts: $\mathrm{OCL} = \mathrm{OCL}_{\mathrm{basic}} \oplus \mathrm{OCL}_{\mathrm{par-spec}} \oplus \mathrm{OCL}_{\mathrm{pseudo}}$

Let us give an idea of what is in each part.

Basic data language $\mathrm{OCL}_{\mathrm{basic}}$ consists of some standard basic predefined types (*Integer*, *Real*), and of a "type declaration mechanism", precisely the possibility of defining simple enumeration types by listing their elements.

Paradigm-specific datatypes $\mathrm{OCL}_{\mathrm{par-spec}}$ is rather rich; indeed it includes two standard types (*String* and *Boolean*); a declaration mechanism for types (the fact that any classifier appearing in the UML model defines a type with the same name of the class), and many *spurious* datatypes, see 2.1, (e.g., *OclState* that is the type whose elements are the states of the state machines appearing in the model).

The most interesting paradigm specific types are however the "collection types", precisley the three collection-like parametric types, *Set*, *Bag*, and *Sequence*, plus the common abstract supertype *Collection*. All such collections types have the usual operations (union, add an element, size, ...) but also some more peculiar ones, as an operation transforming a set/bag into a sequence, which may be nondeterministic.

OCL offers also a rather general *iterator* construct for acting over the values of the collection types, which has several useful particular instantiations (as set-comprehension and a form of universal and existential quantification over collections). Its general form is

collection `->` *iterate*{ *x*`:` *type*`;` *acc*`:` *type'* `=` *exp* `|` *exp'* }

where *x* and *acc* are variables that may appear in *exp'*, and *exp* is a ground expression. The value denoted by this construct is computed as follows:

x iterates over *collection* and *exp'* is evaluated for each elem. After each evaluation of *exp'*, its value is assigned to *acc*. *exp* gives the initial value of *acc*. In this way, the value of *acc* is built up during the iteration over *collection*; when *collection* is a set or a bag the iterator may be nondeterministic too.

Pseudo-data constructs $\mathrm{OCL}_{\mathrm{pseudo}}$ Among these constructs we have the one allowing "navigation": given an object O and an association AS, it returns all objects (a collection) associated wth O by AS. Another one is **`@pre`** used only in post-conditions: it is postfixed to attributes, operations, methods, and associations, and it denotes that such construct should have been computed before the execution of the operation/method call.

Our analysis supports how to approach a variation of UML-OCL, for example modifying the basic data part accordingly to the technique proposed here. However, since UML standard does not include OCL, that is just an example of a data-component among several possibilities, the problem of plugging data into the UML is different from the general case considered before in this paper. Indeed, we could propose different paradigm specific and/or pseudo-data constructs w.r.t. those present in OCL using an analogous technique to that illustrated here for the basic data and still have an acceptable language of the

UML family. We think that it is interesting, however, to first analyse the paradigm specific and pseudo-data components, understand the needs that they cover, and then perhaps to propose new constructs, or just a revised version, covering these aspects.

4 Conclusions

In the course of a two decades experience in the field of formal specifications, we have witnessed the existence of a variety of languages, where the data part not only play different roles, but sometime is partly undefined or inadequated for the applications.

Last, but not least, we are facing the emergence of UML, a de facto standard in system development, where the data are treated separately and even in a non-standardized form. There are, and there will be, attempts to improve the data part of UML, both for sake of clarity and of further expressiveness; here we only mention the CoFI[7] initiative, within which some activity is going on in that direction.

We have been naturally led to propose, first for our personal understanding, a non standard analysis of the way the data part is embodied into a language, to be able to modify a language structure in a modular way. Our proposal may serve as a methodological guideline in the design and the upgrade phase of a language.

References

1. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI–TR–96–20, DISI – Università di Genova, Italy, 1996.
2. E. Battiston, F. De Cindio, and G. Mauri. OBJSA Nets: a Class of High-Level Nets Having Objects as Domains. In G. Rozemberg, editor, *Advances in Petri Nets*, number 340 in Lecture Notes in Computer Science, pages 20–43. Springer Verlag, Berlin, 1988.
3. H. Baumeister, M. Cerioli, A. Haxthausen, T. Mossakowski, P. Mosses, D. Sannella, and A. Tarlecki. Formal Methods '99 - CASL, The Common Algebraic Specification Language - Semantics. Available on compact disc published by Springer-Verlag, 1999.
4. J.A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. Addison-Wesley, 1989.
5. R. Bussow, R. Geisler, and M. Klar. Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study. In E. Astesiano, editor, *Proc. FASE'98*, number 1382 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1998.
6. C. Dimitrovici and U. Hummert. Composition of Algebraic High-Level Nets. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, number 534 in Lecture Notes in Computer Science, pages 52–73. Springer Verlag, Berlin, 1991.

[7] For more information see `http://www.brics.dk/Projects/CoFI/`.

7. H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Specification Language with two Levels of Semantics. Technical Report 83-01, TUB, Berlin, 1983.
8. M. Fowler and K. Scott. *UML Distilled*. Object Technology Series. Addison-Wesley, 1997.
9. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
10. D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw Hill, 1998.
11. R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, LFCS-University of Edinburgh, 1986.
12. I.S.O. ISO 8807 Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS, International Organization for Standardization, 1989.
13. S. Mauw and G.J. Veltink. An Introduction to PSF_d. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89, Vol. 2*, number 352 in Lecture Notes in Computer Science, pages 272 – 285. Springer Verlag, Berlin, 1989.
14. S. Mauw and G.J. Veltink. A Process Specification Formalism. *Fundamenta Informaticae*, (XIII):85–139, 1990.
15. R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
16. G. Reggio and L. Repetto. CASL-CHART: A Combination of Statecharts and of the Algebraic Specification Language CASL. 2000. In this volume. `ftp://ftp.disi.unige.it/person/ReggioG/ReggioRepetto00b.ps`.
17. W. Reisig. *Petri Nets: an Introduction*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, 1985.
18. W. Reisig. Petri Nets and Algebraic Specifications. *T.C.S.*, 80:1–34, 1991.
19. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
20. M. Spivey. *The Z Notation*. Prentice-Hall, 1992.
21. The CoFI Task Group on Language Design. Formal Methods '99 - CASL, The Common Algebraic Specification Language - Summary. Available on compact disc published by Springer-Verlag, 1999.
22. UML Revision Task Force. *OMG UML Specification*, 1999. Available at `http://uml.shl.com`.
23. J. Vautherin. Parallel System Specifications with Coloured Petri Nets and Algebraic Data Types. In G. Rozemberg, editor, *Advances in Petri Nets*, number 266 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1987.
24. M. Weber. Combining Statecharts and Z for the Desgin of Safety-Critical Control Systems. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, number 1051 in Lecture Notes in Computer Science, pages 307–326. Springer Verlag, Berlin, 1996.