An Action Semantics for the D2L Design Description Language

Christina von Flach G. Chavez
e-mail: flach@inf.puc-rio.br

Sylvia de Oliveira e Cruz
e-mail: sylvia@inf.puc-rio.br

Sergio E.R. de Carvalho*

PUC-RioInf.MCC22/00 June, 2000

Abstract: D2L is a language for expressing object-oriented designs. It provides mechanisms for the representation of classes and their relationships, object life-cycles, scenarios, state-dependent behavior and object communication in general, as well as some less conventional features, such as several new semantics for object behavior and some predefined automatic relationships between objects.

This work reports our ongoing effort in developing a complete formal specification for D2L using Action Semantics. The first stage of this effort comprises the formal specification of D2L dynamic semantics. With the complete action-semantic specification for D2L, we hope to provide an executable model for designs expressed in D2L that can be simulated and also automatically translated to source code in some object-oriented languages.

Keywords: object-oriented language, design description, formal specification, Action Semantics

Resumo: D2L é uma linguagem para descrever projetos orientados a objetos. D2L oferece mecanismos para a representação de classes e seus relacionamentos, ciclos de vida de objetos, cenários, comportamento baseado em estado e comunicação entre objetos. D2L também oferece recursos de programação menos convencionais, tais como, diversas formas de definir comportamento de objetos e a criação automática de algumas relações pré-definidas entre objetos.

Este trabalho descreve uma iniciativa em andamento de formalização de D2L utilizando-se Action Semantics. O primeiro estágio de tal iniciativa corresponde à formalização da semântica dinâmica de D2L. Com a especificação formal completa de D2L utilizando Action Semantics, esperamos fornecer um modelo executável para projetos escritos em D2L, que poderão ser simulados e também traduzidos automaticamente para linguagens de programação orientadas a objetos.

Palavras-chave: linguagem orientada a objetos, descrição de projetos, especificação formal, Action Semantics

---

*in memoriam

1 Introduction

In the last decade, several object-oriented development methods appeared, each with its own terminology, notation and processes. Since these methods have been built around the same concepts, some attempts to unify them emerged naturally.

D2L (*D*esign *D*escription *L*anguage) [Car97] is a language for expressing object-oriented designs. It reifies not a new methodology or unification of features, but rather a carefully organized object-oriented discourse, which can be visually and textually realized. D2L has been designed to be the textual realization for such object-oriented discourse. It is currently embedded in 2GOOD (*2*nd *G*eneration *O*bject *O*riented *D*evelopment) [CCO98], a CASE tool that supports graphical descriptions of object-oriented designs (discourse's visual realization), providing mechanisms to represent classes and their relationships, object communication, state-dependent behavior and scenario diagrams, as well as a framework for transformational development from design to implementation. The fact that both graphical design tool and textual design description language reflect exactly the same object-oriented discourse promotes development without discontinuities, documentation, and traceability.

In its original purpose, D2L code was expected to be automatically obtained from 2GOOD designs, and also automatically transformed into some object-oriented language source code, to be eventually compiled, generating executable code. This second transformation step was instantiated with C++ as the target object-oriented language, and was implemented using TXL [CCH95], a semantic-preserving transformational system. This configuration was experimented in pilot projects but it was noticed a kind of programmer's resilience with respect to writing code using an intermediate language. Moreover, TXL execution was very inefficient (both time and space-consuming): the transformation of a D2L program with 10 classes (a small number for a real-life project) was almost impracticable. Finally, the generated C++ code was quite unreadable. The need for a D2L compiler was rather obvious. Nevertheless, D2L does not have an implemented compiler yet.

In the last few months, new extensions have been proposed to 2GOOD/D2L, mainly to provide design-level support to aspects [CL00a] and extensions to pronoun-based communication [CL00b]. These two parallel efforts are centered around D2L and require a complete and unambiguous formal description of the language. Besides, we are interested in providing an executable model for designs expressed in D2L that can be not only translated to source code in some object-oriented languages (including C++), but also simulated prior to translation. Again, a formal description of D2L can be very useful. Having these goals in mind, the formalization of D2L turned out to be an intrinsic part of both efforts [CL00a,CL00b].

Although Denotational Semantics has been the most popular tool among programming language researchers for studying and describing several aspects of programming languages, we have adopted the Action Semantics framework to develop the formalization of D2L. Actions Semantics provides a high-level notation to formally describe the semantics of programming languages, which gives a novice the intuition about what is being formalized. Furthermore, Action Semantics specifications are compositional and modular. Hence, they can be straightforwardly modified to reflect language design changes, and to reuse parts of an existing language definition in the specification of similar, related languages. With these features, the specification of the new proposed extensions may be easily incorporated into D2L specification, with low impact over the previous specification.

This paper is structured as follows. In section 2 we provide a brief description of D2L. Section 3 presents some parts of the current specification stage. In section 4, we present some specification problems found up until now and in section 5, we compare our work to previous similar approaches. Finally, in section 6 we discuss the actual specification stage and the next steps to complete the specification.

2 Overview of D2L

D2L is a textual design description language for object-oriented systems, whose main purpose is to minimize discontinuities from design to code. D2L is a strongly-typed, concurrent, pure object-oriented language. It provides common features found in ordinary object-oriented languages, such as classes, inheritance and polymorphism. Less conventional features are also provided by D2L. For example, D2L objects are polymorphic and can satisfy requests in several different ways: procedurally, answering asynchronous, handshake and future messages, cooperatively and exceptionally. Concurrency in an inherent aspect in D2L. Other D2L features include automatically created relationships among objects, denoted by pronouns, and the textual realization of state-dependent behavior.

D2L design criteria include simplicity and familiarity. The language is small and most of its object-oriented constructs can be explained in terms of well-known constructs provided by other object-oriented languages.

In the present section, we elaborate only those features that will be covered in section 3.

2.1 Programs

From a static point of view, a D2L program is a collection of possibly nested classes, declared in the program or brought in from libraries. There is no external referencing environment (no main program, external data or functions). This is a pure object-oriented environment.

Class collections can be constructed for execution or for storage in libraries. D2L applications are essentially brokered [BMR+96]: an executable program must have at least one top-level class containing a procedure called "Main". One of these classes, indicated via the execution environment, is the *main class* in the program. On the other hand, a class library does not need (but may contain) one or more classes with defined "Main" procedures.

From a dynamic point of view, a D2L program is a collection of concurrent interacting objects. The execution of a program begins with the application of the "Main" procedure to an automatically created object of the *main class*.

2.2 Classes and Interfaces

Classes model objects, prescribing their structure and behavior. Interfaces do not model objects; they only define behaviors to be implemented by classes.

Classes can be concrete, abstract or generic. Abstract classes allow the declaration but not the creation of objects, since at least one of its operations is not fully defined. These classes are useful in inheritance hierarchies, promoting polymorphism. Concrete classes allow the declaration and creation of objects; they may inherit from abstract classes (but not vice-versa), implementing incomplete operations, and thus allowing the creation of objects. Generic or parameterized classes are class models, which can be instantiated with argument classes.

```
CLASS SuperFoo<S>;    INTERFACE IntFoo1;     INTERFACE IntFoo2;
  ...                   ...                    ...
END CLASS               PROCEDURE p1;          PROCEDURE p2;
                        ...                    ...
                      END INTERFACE          END INTERFACE


CLASS AbstractFoo;    CLASS Foo;             CLASS GenericFoo <S,T>
  ...                   OBJECT               INHERITS SuperFoo<S>
  PROCEDURE p;            SuperFoo<INT> f;   IMPLEMENTS IntFoo1,IntFoo2;
END CLASS               END OBJECT             ...
                        ...                    PROCEDURE p1;
                        PROCEDURE p;           ...
                        ...                    END PROCEDURE
                        END PROCEDURE          PROCEDURE p2;
                        ...                    ...
                                               END PROCEDURE
                      END CLASS              END CLASS
```

Fig. 1. An Example

D2L provides simple inheritance between classes, creating a super-type/sub-type relationship, and multiple inheritance between classes and interfaces, creating interface/implements relationships.

Figure 1 presents several examples of D2L classes. `Foo` is a concrete class with a member `f` of type `SuperFoo<INT>`. The class `SuperFoo<INT>` corresponds to the generic class `SuperFoo<S>`, with every ocurrence of `S` replaced by `INT`. The generic class `GenericFoo` inherits from class `SuperFoo`, introducing one more generic parameter `T`; parameter correspondence is nominal, not positional. The class `GenericFoo` also implements two interfaces: `IntFoo1` and `IntFoo2`. `AbstractFoo` is an abstract class, since it declares a procedure named `P` with no defined body.

A class declaration may optionally contain the following members:

- nesting: a class A may be contained in a class B, having full visibility of its features, and being protected from accesses external to B;

- object structure: a record of attribute declarations, each containing the name of an attribute and its modeling class name. Corresponding to each object there exists, at execution-time, a block of data for its attributes. At object creation time, a default initial value is assigned to each object. Object components are automatically protected from external accesses, being visible only to class operations. This protection can be explicitly relaxed by programmers;
- common structure: another record, statically allocated, unique and visible to all class objects, through class operations; direct access to this structure is denied to class users, but may be explicitly given by programmer's option;
- state component: a special object component, modeling the state-dependent behavior of class objects;
- class constants: read-only class objects;
- class messages: declarations of messages that class objects may send to pronouns;
- class exceptions: declarations of exceptions that class objects may propagate;
- object behavior: declarations of the operations applicable to class objects.

2.3 Objects

Objects may be declared as members of other objects, in the class common area, locally in operations, as operation parameters, and as operation results. They can be collected in containers, such as arrays and lists (objects of instantiated generic classes).

All D2L objects are polymorphic and may change class at execution-time (always in the hierarchy rooted by their declaration classes). These objects are heap allocated, and their life cycles are programmer controlled.

Each object is controlled by a *monitor*, in charge of handling several aspects relating to its behavior, as for example message receiving, guard evaluation, and operation scheduling [CCO00].

The modeling of an object's monitor starts when the monitored object is created. At this time, the monitor acquires as components a descriptor and a message queue to be used at execution time. The descriptor will record information such as the objects current class. The message queue will hold the operation requests the object receives. The monitor intercepts all object messages, organizing them in its message queue, and sends to the object the next message to be executed. If no operation is found that corresponds to a message, perhaps due to some polymorphic transformation suffered by the object, then the message is lost. The monitor needs to control a thread and an execution stack to fulfill the expected operation concurrency.

Real time features, guard evaluation, message priorities and exception handling are all part of D2L. They are not described here due to space considerations.

Pronouns. Certain automatic relationships between objects are created at execution-time: *creator-creature* and *parent-son*. The creator is an object that, while servicing a request, creates another object, its creature. The parent is an object that, in its structure declaration, contains the declaration of its son.

These relationships are represented in D2L by the generic designators CREATOR and PARENT, respectively. Using these designators, class decoupling can be increased, if classes model objects that relate to the outside world by sending asynchronous messages to their creators or parents (their clients).

The automatic creation of parent-son and creator-creature relationships relieves programmers of complicated code that must be written to allow generic references of this kind (as recommended in the Composite and Chain of Responsibility patterns [G$^+$95]). It also alleviates the restriction that objects must know the identifiers of their message-receiving collaborators.

2.4 Object behavior

D2L objects can satisfy requests in several different ways: procedurally, answering asynchronous, handshake and future messages, cooperatively, and exceptionally. Special syntax and semantics exist for expressing this varied object behavior; it is up to designer to select the most appropriate in each situation.

The procedural or methodical behavior is that found in other object-oriented languages: the clients execution is suspended until the invoked procedure execution terminates.

Objects in D2L may receive asynchronous messages, messages with handshake protocols, or messages with future synchronism. Asynchronous message receiving does not block the client; handshake message receiving blocks the client until the reception is acknowledged; future message clients may self-block to receive parameters from the server object.

Coroutines and iterators can also be declared in classes, offering a cooperative behavior for objects. Iterators are associated to loops, controlled by objects whose next values are computed outside the loop, in the iterator provided by their modeling classes. Threads are also useful here, mainly to save local execution environments and resume addresses.

Finally, to handle exceptional conditions that may arise during program execution, D2L allows for the definition of exception handlers at the operation level and in classes. Class exception handlers specify another handling scope, being automatically applied to the receiving object if no local handler is defined for some raised exception.

2.5 State-dependent behavior

In object-oriented systems, the behavior of an object, when responding to a stimulus, may depend on the values of its components at that time, or on its state. Several object-oriented methods use states and transitions to define object behavior, when this behavior depends on the past history of stimuli received by objects (see Schaler-Mellor [SM94], ROOM [SGW94], Rumbaugh [R$^+$91], Booch [Boo95]).

A state is actually an abstraction of the past requests made to an object, and transitions indicate behavior requests (they usually cause state changes). A state transition diagram [Har87] or table is used in most methods to represent valid sequences of such stimuli. To describe this feature textually, D2L adapts the State pattern described in [G$^+$95].

The State pattern consists basically in representing diagrams' states as classes in an inheritance hierarchy, and diagrams' transitions as method requests. A *state component* is an instance of the root class in this hierarchy. Any request sent to an object that contains a state component is delegated to it. This state component plays the role of navigating in this hierarchy, i.e., it is responsible for the implementation of its parent's state-dependent behavior.

3 D2L Action Semantics

In this section, we present the first results related to the provision of a specification for D2L dynamic semantics using Action Semantics.

3.1 Classes

As stated in section 2.2, D2L classes can be concrete, abstract or generic. In Java, a class declaration introduces a new reference type, which denotes a class. Java Action Semantics [BW99] specifies that class declarations are elaborated through the semantic function `elaborate` that binds each declared class name to a value of sort `class`. In D2L specification, generic classes motivated us to adopt a distinct approach to elaborate class declarations. A generic class is a class template that when used with actual parameters, as for example, in object declarations, instantiates a new concrete class with its class parameters bound to the given actual parameters. The reference to the generic class must be kept as well as the reference to the instantiated class. The solution we propose in D2L Action Semantics is to encapsulate each class declaration as a *class abstraction*.

```
class-abstraction = abstraction [giving a class]
                                [using the given type*]
```

A class abstraction is an abstraction that, when enacted for the first time, introduces a new reference type, denoted by a class. Further enactions of a previously enacted class abstraction give the corresponding introduced reference type. The `given type*` yields the actual parameters that will be bound to class parameters, for generic classes.

The following sort describes a D2L class in Action Semantics:

```
class = class of (derived-name, interface-bindings,
                  nested-classes, class-variable-bindings,
                  instance-variable-allocator,
                  state-component-allocator, class-constants,
                  class-messages, class-exceptions,
                  method-bindings, class?)
```

- `derived-name` specifies the name of the class (suffixed by the names of the classes used as actual parameters, if the class is generic).
- `interface-bindings` is a mapping for the direct superinterfaces of the class;
- `nested-classes` is a mapping for the nested classes of the class;
- `class-variable-bindings` is a mapping for the class variables of the class;
- `class-constants` is a mapping for class constants;

- `class-messages` is a mapping for class message signatures of the class;
- `class-exceptions` is a list of class exceptions names;
- `method-bindings` is a mapping for the method abstractions of the class;
- `instance-variable-allocator` and `state-component-allocator` are abstractions that, when enacted, allocate storage for the instance variables and for the state component of the class, respectivelly;
- `class` is the direct superclass of the defined class.

Class Elaboration. Class elaboration is performed by a semantic function that takes class declarations as arguments and gives an action that binds each declared class name to a class abstraction.

```
elaborate _ :: Class-Declaration ->
                   action [binding]
                          [using current bindings]

(*) elaborate [[ "CLASS" S:Class-Specifier
                 H:Inheritance-Clause
                 P:Implements-Clause?
                 B:Class-Body? "END" "CLASS" ]] =
       bind the simple name of S to the class-abstraction of (
         the closure of the abstraction of (
         furthermore
         | actualize class parameters of S              (1)
         thence
         | | give the derived-name of S                 (2)
         | then
         | | | give the class bound to                  (3)
         | | |   the given derived-name
         | | or
         | | | | check not (there is a class bound      (4)
         | | | |   to the given derived-name)
         | | | and then
         | | | | | | give the given derived-name and
         | | | | | | enact (application
         | | | | | |  (the class-abstraction bound to the
         | | | | | |   simple name of H) to the arguments of H)
         | | | | | then
         | | | | | | recursively bind the given derived-name#1
         | | | | | | to the class of (the given derived-name#1,
         | | | | | |                  ...   , the given class#2)
         | | | | and then
         | | | | | give the class bound to the given derived-name)).
```

Fig. 2. Class elaboration

Figure 2 presents the semantic function `elaborate`[1] defined for class declarations. The sequence of actions encapsulated by a class abstraction implements the following steps:

[1] In the semantic equation (*), we suppose the inheritance clause is always present. Assume there is also an equation where the inheritance clause is not present.

1. The class parameters defined in `S` are bound to the classes given as actual parameters;
2. The derived name of the class is established. For example, the generic class `GenericFoo<S,T>` has a simple name, `GenericFoo`. The instantiated class `GenericFoo<INT,REAL>` has a derived name, `(GenericFoo, (INT, REAL))`.
3. If there is a class bound to the class derived-name, this class is returned.
4. If there is not a class bound to the class derived-name, a new class is created and then returned.
   (a) The enaction of the class abstraction bound to the class' superclass (declared in the inheritance clause `H`) applied to the superclass' arguments may introduce its recursive instantiation, if it has not been instantiated yet, or returns the class bound to the superclass derived-name. `the arguments of H` yields a list of the actual parameters bound to the class parameters names declared in the inheritance clause `H`.
   (b) The class members declared in the class body are elaborated; details are omitted here, but the specification of class members elaboration is quite similar to the one found in [BW99].
   (c) The derived-name of the class is bound to its corresponding sort `class`.

The results of the elaboration of the classes presented in Figure 1 are given below.

Source Action

```
elaborate [[ CLASS SuperFoo<S> ; ]]    before
elaborate [[ CLASS GenericFoo<S,T>
             INHERITS SuperFoo<S> ; ]] before
elaborate [[ CLASS Foo ; ]]
```

Bindings

```
{ SuperFoo -> abstraction ... }
{ GenericFoo -> abstraction ... }
{ Foo -> abstraction ... }
```

Enaction of a class abstraction occurs every time a class name is used, as for example, in object declaration (see next section); it gives a new named reference type or the type previously bound to the corresponding class name.

3.2 Objects

Objects in D2L are inherently concurrent dynamic entities. In D2L Action Semantics, each object is modeled as a sort `object`.

```
object = object of (class, agent, creator)
```

In the sort `object`, we have the following elements:

- `class` is the modeling class of the object;

– `agent` corresponds to an agent subordinated to the object. Storage for the object's instance variables is allocated in the storage associated to this agent. The object's message queue corresponds to the agent's message buffer;
– `creator` is another object that, servicing some request, has created the object.

The reference to `creator` in the sort `object` is redundant, since it must be defined in the agent's local bindings. It is used there for documentation purposes.

Object Declaration. Object declarations allocate storage that will hold references to objects, but do not allocate storage for objects themselves; they must be explicitly created. The syntax for object declaration consists of a class name (denoting the modeling class) followed by an identifier (denoting the variable that will hold the object reference).

Object declaration is performed by a semantic function `elaborate` that takes an object declaration as its argument and gives an action that binds the given identifier to a variable. This variable is allocated to hold a reference to an object or a value of some predefined class (INT, REAL, etc.).

```
elaborate _ :: Object-Declaration ->
       action [binding | storing]
              [using current bindings | current storage] .

(*) elaborate [[ C: Modeling-Class-Name I: Identifier ]] =
   | | elaborate the type denoted by C
   | then
   | | allocate a variable of the given type
   | |   initialized to the default-value of the given type
   then
   | bind the token of I to the given variable .
```

The action `elaborate the type denoted by C` (defined in the semantic function `elaborate` given above) corresponds to the enaction of the class abstraction bound to `C`, applied to the given arguments, if any, and gives a `class`.

```
elaborate the type denoted by _ :: Modeling-Class-Name ->
    action [binding | giving type] [using current bindings]

(*) elaborate the type denoted by
      [[ I: Identifier "<" C:Class-Instances ">" ]] =
        enact (application (the class-abstraction bound
                            to the token of I)
               to the actual parameters of C) .
```

The next example shows the outputs after performing the action `elaborate the type denoted by` applied to `GenericFoo<INT, REAL>`.

Source Action

```
elaborate the type denoted by [[ GenericFoo<INT, REAL> ]]
```

Bindings

```
-- bindings for SuperFoo, GenericFoo, ...
(SuperFoo,INT) -> class of (...)
(GenericFoo,(INT,REAL)) -> class of ((GenericFoo,(INT,REAL)),
                                     ... , (SuperFoo, (INT)) )
```

Transients

```
class of (...) -- the class bound to GenericFoo<INT, REAL>
```

Object Creation. Objects in D2L must be explicitly created, using the operation `CREATE`. During object creation, an agent is subordinated to it. The object's agent receives an offer for a contract containing a protocol that defines all the distinct object behaviors provided in D2L (asynchronous, future and handshakes messages, coroutines, etc.).

Object creation is performed by a semantic function `evaluate` that takes an expression for object creation as argument (a class name followed by the keyword `CREATE`) and gives an action that creates an instance of the given class name, denoted by an `object`.

```
evaluate _ :: Object-Creation ->
    action [giving an object | ... | communicating]
           [using current buffer]

(*) evaluate [[ C:Modeling-Class-Name "CREATE" ]] =
    | elaborate the type denoted by C
    then
    | | | give the given type
    | | and
    | | | | | | offer a contract [to any agent]
    | | | | | |    [containing abstraction of object-behavior]
    | | | | | then
    | | | | | | receive a message[containing an agent]
    | | | | and
    | | | | | give the instance-variable-allocator of given type
    | | | then
    | | | | | send a message [to the agent yielded by
    | | | | |    the contents of the given message#1][containing
    | | | | |    the given instance-variable-allocator#2]
    | | | | and
    | | | | | give the agent yielded by
    | | | | |  the contents of the given message#1
    | then
    | | give the object of (the given type#1, the given agent#2,
    | |                      the object bound to the self-token) .
```

Figure 3 presents `object-behavior`, the protocol offered to objects' agents. While performing this protocol, each subordinated agent:

1. Sends its identification to the contracting agent;
2. Waits for an abstraction corresponding to the instance-variable-allocator;
3. Enacts this abstraction, allocating storage for the object's instance variables;
4. Produces agent's local bindings, and finally,
5. Executes a loop that continuously waits for a message containing a request for a service with optional arguments, and then selects the suitable protocol depending on the kind of the request received.

```
method = procedure | async | handshake | future |
         exception | coroutine | iterator .

object-behavior = action .

(*) object-behavior =
    | | send a message [to the contracting-agent]            (1)
    | |                [containing the performing agent]
    | | and then
    | | receive a message [from the contracting-agent]       (2)
    | |                   [containing an abstraction]
    | then
    | | enact the abstraction yielded by                     (3)
    | |   the contents of the given message
    before
    | | | bind self-token to the performing agent            (4)
    | | before
    | | | bind creator-token to the contracting agent
    | hence
    | unfolding                                              (5)
    | | | receive a message [from any agent]
    | | |   [containing msg(method, arguments)]
    | | then
    | | | | check (method in msg is handshake) then
    | | | | send a message [to the contracting-agent]
    | | | |                [containing the ack-signal]
    | | | | and then  ...
    | | | |  ... << the behavior of handshake method >>
    | | | | then unfold
    | | | or ... << other behaviors  >>
```

Fig. 3. The object-behavior protocol

An Example: Handshake. Suppose that, during its execution, `obj1` sends a handshake message to `obj2`: `obj2 <- handshakeMethod`. The handshake protocol states that the sender `obj1` is supposed to wait for an `ack-signal` from the receiver `obj2` in order to proceed.

The Action Semantics description for the handshake protocol on the client's side (in this case, `obj1`) is given in Figure 4. The action `receive a message [from the agent of the given object#1] [containing the ack-signal]` is specific to the handshake protocol on the client's side; in fact, it must be embedded in a multiple `or` action combinator (similar to the one defined in `object-behavior`, that checks for each defined behavior in D2L, and chooses one, based on the type of the message that has been sent).

```
evaluate _ :: Application -> action
                [giving a value? | ... ]
                [using current bindings ... ]

(*) evaluate [[ T:Application-Target "<-"
               O:Operation-Name ";" ]] =
      | | give the receiver denoted by T
      | then
      | | | give the given object
      | | and
      | | | | give the class of the given object
      | | | then
      | | | | give the method bound to the token of O
      | | | | in the given class
      then
      | | send a message[to the agent of the given object#1]
      | |               [containing msg(the given method#2)]
      | and then
      | | receive a message [from the agent of the given object#1]
      | |                   [containing the ack-signal]
      ...
```

Fig. 4. Method Call

4 Specification Problems

The communicative/hybrid facets (agents and send/receive actions) have been quite suitable to model the behavior of D2L objects. Agents provide the desirable intrinsic asynchronism demanded by D2L and the agent's message buffer plays the role of the object's message queue. The semantics of the `receive` action (busy-waiting semantics), however, has imposed some difficulties in modeling the various kinds of synchronous communication provided by D2L.

The modeling of the well-known procedural synchronous behavior, for example, has presented some problems. In a straighfoward implementation for this behavior, the sender object remains blocked (after the synchronous message dispatch) until it receives a "end" message from the receiver object. Within this implementation, however, in case the receiver object sends, directly or not, other synchronous message to sender object, this message will not be received, because the sender object is blocked waiting for the original "end" message. This behavior is not the desired one, of course, since ordinary synchronous communication (by message exchange between objects) shall simulate procedural or methodical behavior, where there is no blocking, but simply pushing in a execution stack. On the other hand, freeing the synchronous message sender object to receive any other messages, corresponds by no means to the desired behavior, since we do not want to allow this object to receive, from a third object, any message that is not concerned to the current execution line.

The provision of a `receive` action with no blocking at all (as promised by the Action Notation 2) may facilitate our work, but does not solve the specification problems associated to all D2L synchronous behaviors.

The second problem we have approached refers to class binding. The solution provided to the generic class instance problem, using class abstractions and class late-bindings, has given rise to another kind of problem: the binding of a class name to a recently created `class` entity is done in the local bindings of the agent of the first class instance's creator. This binding and the class variables allocation, however, should be done in a global scope and not in the current local (of agent) scope. This global scope will have to be specified as a separate agent; the class global bindings are all expected to be there. Class variables, in turn, will have to be stored in separate agents, one per class, modeling shared memory among class objects. This agent proliferation further complicates system synchronization and storage management.

5 Related Work

In [Mos96b], Mosses provides an action-semantic description for a subset of the Ada language, where tasks and *rendez-vous* are modeled using the communicative facet. Mosses also reports the use of Action Semantics to describe concurrent languages in [Mos96a].

As stated in section 3, the `class` definition adopted in D2L specification is very similar to the one found in the JAS project [BW99]. The main difference is the class binding time. Since Java does not have generic classes, classes can be bound and their class variables allocated at declaration time. Consequently, classes and their variables are statically created; they already exist when the main object is created. In D2L, classes are late-bound at object creation time. A class and its class variables are created when its first object is created.

The JAS current stage does not approach concurrency aspects, disregarding executions in parallel generated for `java.lang.Thread` objects. This simplification is not useful here since concurrency is an inherent aspect in D2L.

In [Rep91] the standard ML language is extended with the introduction of mechanisms for spawning new processes and for synchronous exchange of values between these processes over typed channels. In [MM94], this extension is described with Action Semantics and analogous to D2L, agents are used in parallel processing. The ML proposal, however, is to provide a low-level language with primitives that allow programmers to control threads and channels. In D2L these features are built-in, so it is up to the language to manage and to have control over them. Furthermore, D2L descriptions may be richer and more complicated due to the several distinct semantics provided for object behavior.

In [PMM95] a parallel object-oriented language – POOL – is described using Action Semantics. Like D2L, this language has integrated parallelism in the object-oriented model by supplying each object with a local independent process that is executed in parallel with all the other objects in the system. The communication mechanism in POOL, however, is only synchronous, unlike D2L, where asynchronous communication is provided. POOL objects are agents and the communication between these agents is made via message exchange as in D2L. However, the *busy-waiting* problem presented in section 4 is not mentioned in [PMM95].

6 Conclusions

This work introduces D2L, a textual design description language for object-oriented systems and presents the first results related to the specification of the dynamic semantics of D2L classes, objects and object behavior using Action Semantics. The main goal of this work is the provision of a complete formal specification for D2L.

Up until now, the contributions of our effort to formalize D2L using Action Semantics are the specification of class declarations using class abstractions, the specification of concurrent objects using agents and also the specification of the several object behaviors provided by D2L. Furthermore, we anticipate that the formalization of a language for expressing object-oriented designs can be very useful to software development as a whole.

The specification of D2L using AS has given us some insights about many language aspects, as well as design decisions that possibly had not been explicitly stated before.

Surprisingly, the task of formalizing D2L using Action Semantics, guided through some useful examples [Mos96b,BW99], has been amazingly easy, especially considering the authors do not have a solid background or previous relevant experience on formalization.

6.1 Future Work

As part of the current specification stage, we expect to be able to describe the whole set of semantics for object communication provided by D2L. The proposed changes in the forthcoming version of Action Notation (AN-2) [LMW00] seem to facilitate our job and to provide solutions to the problems stated in section 4.

In AN-2, message reception is fair and messages are tagged. This sort of non-determinism seems more appropriate to languages with intrinsic asynchronism such as D2L. The primitive actions for explicit agent creation and agent destruction resembles the computation model. Besides, transients passing seems to be less expensive in AN-2, generating more concise specifications, a very important characteristic for languages with rich semantics such as D2L.

To complete the description of D2L dynamic semantics, the following features must be specified: nested classes, visibility, state-dependent behavior, state component inheritance, class constants and exceptions. These features have not been approached in the current stage of D2L specification, but we hope that their introduction will not cause undue difficulties, since other languages' similar features have been modeled using Action Semantics (such as inner classes and access modifiers in Java, for example).

The next stage of this work will comprise the specification of D2L static semantics. After a reliable version of D2L Action Semantics is developed, i.e, verified using ASD or related tools [BMW92,RAT], we will be able to extend D2L with the features proposed in [CL00b] and [CL00a], and give their formal specification using Action Semantics.

Acknowledgements

We wish to thank Prof. Lucena, our advisor, for having "adopted" us without any restrictions, as well as for keeping the work initiated with Prof. Sergio Carvalho in our post-graduate (PhD) programme.

Finally, we dedicate this work to the D2L designer, Prof. Sergio Carvalho, whose bright ideas and remarkable contributions in the field of object-oriented systems, we have the honor (and audacity) to try to keep alive.

References

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stahl. *Pattern-Oriented Software Architecture.* Wiley, 1996.

[BMW92] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer-Verlag, 1992.

[Boo95] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, 2nd edition, 1995.

[BW99] Deryck F. Brown and David A. Watt. JAS: a Java Action Semantics. In *Proc. of 2nd International Workshop on Action Semantics*, pages 43–55, Amsterdam, March 1999. BRICS Notes Series.

[Car97] Sergio E. R. Carvalho. DDL: An Object-Oriented Design Description Language. Technical Report PUC-RioInf.MCC29/97, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Department of Informatics, Rio de Janeiro, Brazil, 1997.

[CCH95] James R. Cordy, Ian H. Carmichael, and Russel Halliday. *The TXL Programming Language.* Legasys Corp., 1995.

[CCO98] Sergio E. R. Carvalho, S. O. Cruz, and T. C. Oliveira. Second Generation Object-Oriented Development. Electronic Notes in Theoretical Computer Science. WWW[2], 1998.

[CCO00] Sergio E. R. Carvalho, Sylvia O. Cruz, and Toacy C. Oliveira. Concurrency Features in 2GOOD/DDL. In *Proc. IDEAS 2000*, pages 100–111, Cancun, Mexico, April 2000.

[CL00a] Christina von Flach G. Chavez and Carlos J. P. Lucena. Design-Level Support for Aspects. Ph.D. Thesis Proposal, Pontifical Catholic University of Rio de Janeiro, Department of Informatics, 2000.

[2] http://www.elsevier/nl/locate/entcs/volume14.html

[CL00b] Sylvia O. Cruz and Carlos J. P. Lucena. Pronoun-based Object Communication. Ph.D. Thesis Proposal, Pontifical Catholic University of Rio de Janeiro, Department of Informatics, 2000.

[G+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.

[Har87] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming 8*, pages 231–274, July 1987.

[LMW00] Soren B. Lassen, Peter D. Mosses, and David A. Watt. AN-2: Revised Action Notation – Informal Summary. DRAFT PROPOSAL AN-2/v0.6, 2000.

[MM94] Peter D. Mosses and Martín A. Musicante. An action semantics for ML concurrency primitives. In *FME'94, Proc. Formal Methods Europe: Symposium on Industrial Benefit of Formal Methods, Barcelona*, volume 873 of *Lecture Notes in Computer Science*, pages 461–479. Springer-Verlag, 1994. WWW[3].

[Mos96a] Peter D. Mosses. Theory and practice of action semantics. In *MFCS '96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science (Cracow, Poland, Sept. 1996)*, volume 1113 of *Lecture Notes in Computer Science*, pages 37–61. Springer-Verlag, 1996. WWW[4], FTP[5].

[Mos96b] Peter D. Mosses. A tutorial on action semantics. 50pp. Tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996). WWW[6], FTP[7], March 1996.

[PMM95] Giovanny L. Palma, Martin Musicante, and Silvio R. L. Meira. A Novel Formal Semantics for a Parallel Object-Oriented Language. In *Proc. of XV Intl. Conf. Of the Chilean Computer Science Society*, Arica, November 1995.

[R+91] J. Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[RAT] Recife Action Tools. WWW[8].

[Rep91] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. SIGPLAN'91, Conf. on Prog. Lang. Design and Impl.*, pages 293–305. ACM, 1991.

[SGW94] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling.* John Wiley, 1994.

[SM94] S. Schlaer and S. Mellor. *Object Life Cycles: Modeling the World in State.* Yourdon Wiley, 1994.

[3] http://www.brics.dk/RS/94/20/
[4] http://www.brics.dk/RS/96/53/
[5] ftp://ftp.brics.dk/Projects/AS/Papers/Mosses96MFCS
[6] http://www.brics.dk/NS/96/14/
[7] ftp://ftp.brics.dk/Projects/AS/Papers/Mosses96DRAFT
[8] http://www.di.ufpe.br/~rat