PH-Helper – a Syntax-Directed Editor for Hoshimi Programming Language, HL

Mariano Luzza[1], Mario Marcelo Beron[1], and Pedro Rangel Henriques[2]

1 **National University of San Luis**
San Luis, Argentina
{mluzza,mberon}@unsl.edu.ar
2 **Universidade do Minho**
Braga, Portugal
prh@di.uminho.pt

Abstract

It is well known that students face many difficulties when they have to learn programming. Generally, these difficulties arise from two main reasons: i) the kind of exercises proposed by the teacher, and ii) the programming language used for solving those problems. The first problem is overcome by selecting an interesting application domain for the students. The second problem is tackled by using programming languages specialized for teaching.

Nowadays, there are many programming languages aimed at simplifying the learning process. However, many of them still have the same drawbacks of traditional programming languages: the language used to write the statements is different from the programmers' native language; and the syntactic rules impose many tricky restrictions not easy to follow.

This paper presents an approach for solving the problems previously mentioned. The approach consists of using: an *application domain* motivating for the student, the Project Hoshimi (PH); and a *programming environment*, PH-Helper that is a simple and user-friendly syntax-directed editor and compiler for Hoshimi Language (HL), the actual PH programming language.

1998 ACM Subject Classification D.2.6 Programming Environments

Keywords and phrases Syntax-directed Editors, Visual Programming Enviroments, DSL

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.71

1 Introduction

Since the early days of programming, people realized how difficult is to teach programming principles and imperative programming languages.

Many researchers in computer science and didactics have been working over this problem. On one hand, searching for its causes, relating difficulties with students background and courses curricula, and looking after the definition of the ideal profile for a successful computing student. On the other hand, designing languages and sketching programming environments that can overcome student barriers. Prolog [26, 4, 27, 21, 7, 8] (and other declarative programming languages), Logo [23, 31, 32][1], and Scratch [25, 17][2] are some of

[1] A detailed description can be found at http://en.wikipedia.org/wiki/Logo:(programming_language). See also the project homepage at http://stager.org/logo.html or http://el.media.mit.edu/logo-foundation/. Complementary information is available at http://mckoss.com/logo/.
[2] A detailed description can be found at http://en.wikipedia.org/wiki/Scratch-(programming_language). See also the project homepage at http://scratch.mit.edu/.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 71–89

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the more relevant contributions coming out from this search effort. As fully discussed in [29], Alice [13, 14, 5, 20][3] can be compared to Scratch as a teaching tool for introductory computing. It uses 3D graphics and a drag-and-drop interface to facilitate a more engaging, less frustrating first programming experience. The underlying idea is to create a 3D programming environment that makes it easy to create an animation for telling a story, playing an interactive game, or a video to share on the web.

However the problem is far away from being overridden. Even nowadays, all over the world students face tremendous difficulties when they are introduced to programming. Obviously the most difficult step is the ability to understand problem statement and to write the algorithm; this is precisely the focus of the teaching activity. But unfortunately other minor issues emerge that are strong obstacles that obstruct students progress. One is the use of keywords in English language; another one is the (sometimes) complex details concerning language syntax, like punctuation symbols of composed statements structure. Visual programming languages and environments attempt to surmount these barriers that many times lead beginners to loss motivation to accomplish programming tasks [22, 30, 24, 1]. However visual programming does not scale properly and this approach had an impact far below the one expected 20 years ago.

Structured editing[4], also called *Syntax-directed editing* [18] or even *Language-based editing*, is another relevant approach that can actually help programmers to cope agilely with programming languages and overcome the referred syntactic idiosyncrasies [12, 16, 15]. The basic idea is to develop text editors that are aware of a specific language structure or syntax. In [11, 6] Henriques et.al. review the set of *Language-based tools*, like the editors under discussion, that can be derived and automatically generated from the language's context free grammar. Also in [19] the topic is explored to develop a meta-language based editor to create and analyze grammars. Some *Structured Editors* are *reactive* and others are *proactive*.

In the first class, the editor knows the language syntax but it does not guide the programmer; the programmer is free to write what he wants, and during or after writing the editor uses *highlighting*, *indentation* or other visual techniques to enhance the language keywords and the text structure. WinEdt[5] is just one example of this first class.

> WinEdt is a powerful and versatile text editor for Windows with a strong predisposition towards the creation of [La]TeX documents. It is used as a front-end for compilers and typesetting systems, such as TeX, HTML, etc. WinEdt's highlighting schemes can be customized for different modes and its spell checking functionality supports multi-lingual setups, with dictionaries (word-lists) for many languages.

Autocompletion—the ability to complete the words that are being typed, according to the context that restrict the choices[6]—is another feature offered by modern reactive language-sensitive editors that improve significantly the typesetting process reducing simultaneously the error-proneness. Emacs[7] and Vim[8], as well as TexMaker[9] are examples of text editors that

---

[3] See also the project homepage at `http://www.alice.org/` or `http://www.cs.duke.edu/csed/alice/aliceInSchools/`.

[4] Look at `http://en.wikipedia.org/wiki/Structure_editor` for a large discussion on that topic.

[5] See the homepage at `http://www.winedt.com/` for more details, or visit `http://www.winedt.org/`.

[6] See `http://en.wikipedia.org/wiki/Autocomplete` for more details and examples.

[7] See the homepage at `http://www.gnu.org/software/emacs/` or a guided tour at `http://www.gnu.org/software/emacs/tour/`.

[8] More information at `http://www.vim.org/` or `http://www.yolinux.com/TUTORIALS/LinuxTutorialAdvanced_vi.html`.

[9] For details look at `http://www.xm1math.net/texmaker/`.

offer autocompletion. *Autoreplace* is a related feature that involves automatic replacement of a particular string with another.

Also the code completion feature provided by Microsoft, known as IntelliSense, is similar to autocompletion but at a semantic level. IntelliSense editors are context sensitive and are aware of all program identifiers so far declared; in this way, they are able to suggest those identifiers that can be used in a certain place inside the program,

From the perspective of the work described in the paper, proactive structured editors are more interesting. In the second class, the editor is aware of the language syntax and it actively guides the programmer along the typesetting. At each moment, the editor knows what can be written so it shows the possibilities and after the user choice it goes on in the same way until it reaches an identifier or constant whose specific value just the programmer knows; this is the only thing he actually has to type. Obviously this kind of editors offer an effective help to the programmer that does not need to have a complete knowledge of the programming language syntax. Moreover this family of tool can be derived directly from the grammar and the editors can be generated automatically. Tim Teitelbaum was a pioneer [28] in this area. Relevant work in this field—automatic generation of syntax-directed editors—but concerned with visual editors, like the one (PH-Helper) discussed in this paper, has been done by Arefi et.al. [2, 3].

Another concern that is highly relevant when teaching computer programming is the problem statement by itself (its context, application area, and goals): many times, the problems proposed by teachers are not challenging enough, failing to motivate students. Without a strong motivation and practical validation, is even more difficult to attract students for this complex task. So, this topic (the choice of interesting and effective problems) is another issue that shall be taken into account towards a successful learning process. Regarding this point, we believe that start programming courses teaching Domain Specific Languages (DSL) instead of choosing immediately a General Purpose Language (GPL)can be a wise approach to overcome this last difficulty. A DSL is a formal language just a GPL is, so it is possible to reach the same learning objectives—like problem understanding, data structures choice, algorithm development, code implementation following elegant ways and good practices—but supported in a smaller, higher-level language (more abstract and concise), designed specially for a more restricted domain that can be more natural and appealing for students. Opposed to the proposals described above—as Prolog, Logo, Scratch, Alice—and many other that although innovative are sill GPL s, our proposal is a DSL aiming at taking profit from the benefits just set above.

Project Hoshimi [10][10], that will be introduced in section 2, is another trial to surpass part of the programming struggles above identified. Created by Richard Clark for the 2005 Microsoft's Imagine Cup contest, Project Hoshimi is a game based on .Net technology. The basic idea is to create a scenario (in the context of human bloody system) and ask students to program small robots that can navigate through the body and protect him from some diseases. The scenario is attractive and based on common sense and the task is challenging enough. The robots programming is done using a small set of primitive and intuitive commands.

[10] A detailed description can be found at `http://fr.wikipedia.org/wiki/Project_Hoshimi`. See also the project homepage at `http://www.projethoshimi.fr`. A Guide to The Imagine Cup Project Hoshimi is available at `http://blogs.msdn.com/b/edunhill/archive/2007/10/22/guide-to-the-imagine-cup-project-hoshimi-ai-competition.aspx`.

This paper is about an editing environment, called PH-Helper, that we are developing as a front-end for Project Hoshimi that aims at overwhelming the remaining difficulties. Namely, PH-Helper adds some more power to Hoshimi language (see section 3) and provides a syntax-directed editor—section 4 fully discusses it—that helps beginners with the language syntax avoiding boring errors. From a pedagogical perspective, a syntax-directed editor can also be adequately instrumented to force, or just to give advices, on the use of good programming practices like proper name conventions for the different type of identifiers, indentation, code commenting, etc. This topic is something that we plan to explore in a future version. We also discuss two other functionalities provided by PH-Helper: the compiler that generates C# to allow running the developed programs; and an XML exporter (see section 5). In order to demonstrate how PH-Helper works a case study is presented (see section 6).

Section 2 is a bit long description of the main characteristics of Project Hoshimi. It was included with two purposes: to make the paper more self-contained; and to make more natural to describe the extensions introduced, the code generation strategy and mainly to justify the decisions concerned with PH-Helper editor and enhance the syntactic aids it offers. Of course the reader can skip it and go directly to the other sections, coming back just on demand.

2 Project Hoshimi, an Overview

As said above and can be read at `http://fr.wikipedia.org/wiki/Project_Hoshimi`, Project Hoshimi (PH for short) is a computer game aimed at promoting creative use of programming languages and tools. PH is useful for:

- Conceptualizing *programming* as a creative activity through strategic simulation.
- Teaching *object oriented programming* and .Net technology.

In general terms, PH is a game whose goal is to cure human diseases using a set of NanoBots. Nanobots are small robots that can be injected in blood flow system and are able to solve health problems found in the human body. Each NanoBot has "artificial" intelligence for healing partially certain diseases. So we can said that the game consists in the *strategic simulation* of NanoBots behavior. Artificial Intelligence (AI) behavior is provided by the student programming those strategies in .Net technology.

2.1 Game Environment

The game is carried out inside of human body, where:

Game map is a tissue. Each map has 200 x 200 positions and the NanoBots have two possible movements: Horizontal and Vertical.

Game area is composed of blood (red), bones (gray), nerves (blue) and impassable sectors (black).

AZN area contains molecules employed for curing diseases. These molecules must be collected by a special kind of NanoBots known as NanoCollector. This area is never empty.

Hoshimi Points (HPo) are the disease zones; in this places the NanoNeedles are elaborated. Each HPo can receive up to 100 AZN molecules.

Injection Points (IPo) are the places where NanoBots are inserted. These points should not be changed during the game.

2.2 NanoBots

NanoBots are the central, or main, entities in the game. For this reason they are described in next subsections.

2.2.1 Kind of NanoBots

There are several kinds of NanoBots, namely:

NanoIntelligence (NanoI): this kind of NanoBot has two functionalities: i) Create all other NanoBots in the community, and ii) Give instructions to the other NanoBots. Each community (a team in the context of the game) has only one NanoI that is the first inserted inside the human body. It can move but it can not shoot.

NanoNeedle: this kind of NanoBot is created at Hoshimi Points in order to provide AZN molecules to the system. It belongs to the static defense; so it can not move but it can shoot.

NanoCollector: this kind of NanoBot collects AZN molecules and transfers them to the NanoNeedles. It is the only class that can move and shoot.

NanoExplorer: this kind of NanoBot is aimed at doing recognition task. This class has the widest vision range and the fastest movements, however it can not shoot.

NanoBlocker: this kind of NanoBot is used for diminishing enemy movements by changing blood density. It can not move and shoot.

NanoContainer: this kind of NanoBot is similar to NanoCollector class but it has bigger capacity. It can move but it can not shoot.

NanoWall: this kind of NanoBot generates a force field. This field can not pass by the enemies. It can not move and shoot.

NanoIPCreator: this kind of NanoBot generates a second injection point. It can move but it can not shoot.

2.2.2 NanoBots Characteristic

The NanoBots have the following characteristics:

Constitution: quantity of health also known as hit points.

Scan: vision distance.

Defense Distance: NanoBots' attack range.

Maximum Damage: maximum damage that a NanoBot can cause to the enemy during one time period.

Container Capacity: quantity of AZN that a NanoBot can store and transport.

Collect Transfer Speed: quantity of AZN that NanoBot can collect and transfer by time period.

All the characteristics aforementioned can be changed by the user using the following rules:

1. The characteristic values can not exceed a maximum value.
2. The addition of characteristic values can not exceed a "Total" established.

2.2.3 NanoBots Commands

The following commands are recognized by NanoBots:

ForceAutoDestruction – This operation is used by the NanoBot for self-destroying. Generally, this command is used when: i) There are many NanoBots or ii) More NanoBots are needed. All NanoBots have this command except the NanoI.

StopMoving – This command is employed for stopping the NanoBot movement. All others actions, such as attack or transport AZN, can be used. Afterward of executing this order other command can be used. This method is used for all NanoBots with movement.

MoveTo(Point) – This command is employed for moving NanoBots until the place indicated by *Point* (a point has two coordinates, `x` and `y`). All NanoBots with movement can run this command.

MoveTo(PointLst) – This command works as *MoveTo(Point)*. However, it receives a sequence of Points as its parameter. This sequence indicates different places where the NanoBots must be moved. All NanoBots with movement can run this command.

DefendTo(Point, int) – This command is used for attacking the point received as parameter. The parameter *int* is used for indicating the number of times that the NanoBot will attack its goal. All NanoBots with attack capability have this command.

CollectFrom(Point, int) – This order is used for collecting molecules AZN. This molecules are gathered from *Point*. The parameter *int* is used for indicating the number of turn that NanoBot will attack this point. All NanoBots with gathering capabilities execute this command.

TransferTo(Point, int) – This instruction allows NanoBots to download molecules AZN in the place indicated by *Point*. The parameter *int* is used for indicating the number of times that the NanoBot will download molecules in this point. All NanoBots with download capabilities execute this command.

Build(Type) – This command is executed by NanoI to create new NanoBots.

The reader interested in knowing more details about these commands shall look at `http://www.projethoshimi.fr/lab/richardc/index.php`—a compilation of tutorials for the competitions.

2.3 Defense Strategies

Depending on the game configuration, it is possible to play with other human player or other enemy controlled by the computer. The defense is scheduled in two levels. The first one is concerned with AI general strategy. The second one allows that a NanoBot attacks a particular position. Generally, the vision range is greater than the attack range, for this reason before attack the NanoBot must put the enemy inside its attack range. Obviously, the NanoBot must not be inside of enemy attack range. In order to create an effective defense strategy, the user must:

1. Surround vulnerable unit with protection, generally using NanoI.
2. Create static sentinels by employing NanoNeedles.
3. Guard the positions and behavior of the enemies.
4. Stay away from enemies.
5. Avoid to be closer of injection points.
6. Use formations highly cohesive.

3 Extending Hoshimi language

The current game platform available has a visual programming interface for developing strategies. This interface is poor, because it only implements the NanoBots basic operations described in section 2.2.3.

However the Project Hoshimi programming language, HL, has some severe restrictions: it does not allow to work with variables, neither supports data structures such as list, hash tables, etc., nor assignments.

PH-Helper gives a practical solution to the problems mentioned above and it provides more flexible statements for implementing strategies.

On one hand, PH-Helper allows to declare and handle variables of primitive (int, Point, etc.) or composed (list, dictionaries, hash tables, etc) types. Variables are defined when the user creates a NanoBot. PH-Helper also extends the original language with the assignment statement; it also allows the programmer to pass variables as parameter to other instructions.

On the other hand, the conditional statements are also extended supporting now any kind of logical expression. Moreover, PH-Helper adds new loop statements, such as *foreach*, for traversing complex data structures like dictionaries and lists.

All the programming activity above described is carried out using an intuitive and simple graphical interface. This interface clearly shows: i) The type of variables that the user can define, and ii) For each visual statement, its parameters and available operations.

To finish this section, it is important to notice that the extensions described are already implemented in the current version of PH-Helper. But if the user needs more functions and properties that are present in the Project Hoshimi but not in the editor, he can extend it[11]. The steps needed for extending functions are:

1. Define the new function in C#.
2. Modify the Function Configuration File (FCF).

Listing 1 DTD for FCF descriptions.

```
<!ELEMENT FUNCTIONS (FUNCTION*)>
<!ELEMENT FUNCTION (PARAMETERS, CODE)>
<!ELEMENT PARAMETERS (PARAMETER*)>
<!ELEMENT PARAMETER EMPTY>
<!ELEMENT CODE (#PCDATA)>
<!ATTLIST FUNCTION name CDATA type CDATA>
<!ATTLIST PARAMETER name CDATA
                    type CDATA>
```

This configuration file (FCF) contains a description of all functions supported by PH-Helper. This description is written in a dialect of XML (also referred to as FCF); the respective DTD is shown in listing 1. In order to add a function, the user only needs to specify a `FUNCTION` element by providing the following elements: `parameters`, `function C# code`, `name` and `type`.

New properties can also be added to the kernel; the following steps describe what is necessary to do that:

1. Define the property or variable in C#.
2. Modify the Properties Configuration File (PVCF).

Similar to the previous case, this configuration file (PVCF) contains a description of all properties and variables supported by PH-Helper. The description is again written in a dialect of XML (named PVCF) with the DTD depicted in listing 2. In order to add a property, the user only needs to specify a `VARIABLE` element by providing the following elements: `property C# code`, `name` , `scope`, `readOnly` and `type`.

[11] The student can define new extensions, but it is advisable that the teacher carries out this task because it is too complex.

Listing 2 DTD for PVCF descriptions.

```
<!ELEMENT VARIABLES (VARIABLE*)>
<!ELEMENT VARIABLE (CODE)>
<!ELEMENT CODE (#PCDATA)>
<!ATTLIST VARIABLE name CDATA
                   scope (global | local)
                   readOnly (yes | no)
                   type CDATA>
```

Listing 3 Examples of Function and Variable extensions to HL.

```
<FUNCTION name="Length" type="Int32">
 <PARAMETERS>
 <PARAMETER name="str" type="String"/>
 </PARAMETERS>
 <CODE>
  return str.Length;
 </CODE>
</FUNCTION>
```

(a)

```
<VARIABLE name="IsAlive"
 scope="local" type="Bool"
 readOnly="yes">
 <CODE>
 {get{return HitPoint > 0;}}
 </CODE>
</VARIABLE>
```

(b)

Listing 3.a shows an example of a function extension. This extension consist in adding the *Length* function. This function has a string parameter called *str* and it returns an integer that represents the length of *str*.

Listing 3.b shows an example of a variable extension. This extension adds a new property called *IsAlive*. This property has a local scope, it is a readonly property and its type is bool.

4 Editing Hoshimi Programs with PH-Helper

To define a strategy using the PH-Helper Main Window, three steps must be carried out (see figure 1):

- *Create a New Program*
- *Create NanoBots Types*
- *Define Strategies*

Create a New Program, is used for creating a new *Working Space*. A *Work Space* is composed by several sub-spaces. Each sub-space represents a NanoBot and it is composed by:

- Toolbar: It is used for doing editing operations, such as: cut, copy and paste, change node's position, etc.
- Language Command Bar: This bar contains all instructions available in the language. These instructions are separated by three logical sections.
  The first one holds all flow control instructions. For example: *state* (an equivalent instruction to case statement), *decision* (an equivalent instruction to if-then-else statement), *repetition* (an equivalent instruction to while statement), etc.
  The second one provides all Project Hoshimi primitive Actions, like: *move*, *stop*, *collect*, etc.
  The last one contains all new commands added by PH-Helper. The most important is the one that allow to save states. It is the variable assignment.
- Program Window: This window visualize the current program. The Hoshimi Programs are naturally represented as a *r-tree*. The tree nodes are classified in:

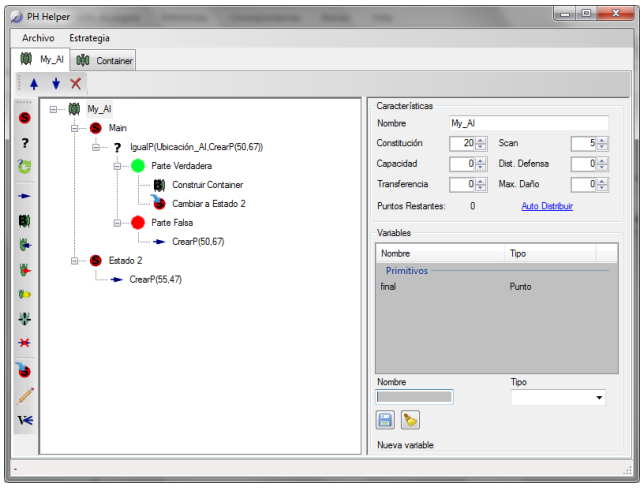

Figure 1 PH-Helper Main Window.

- Leaf Node: This kind of node represents an atomic instruction. For example: *write* (for writing information on the log console), etc.
- Parent Node: This kind of node represents a group of actions. For example: the instruction *if-then-else* has two child node, they are *true* and *false* branches. They in turn are composed by actions. It is important to remark that these actions can be in turn composed by actions or group of actions.

- Parametrization Window: This window is aimed at providing the parameters needed by the selected instruction in the program window. Generally, these instructions are functions with n parameters. In this context, PH-Helper restricts the construction of expressions taking into account the parameter type and the value type returned by the expression. An example is shown in figure 2. This figure shows all data needed for specifying an assignment operation.

  In this case, the Parametrization Window is divided in two parts:

  - Variable: It is the *lvalue* of assignment instruction, and it has two lists:
    1. *Type* is used for filtering the variables according to the selected type.
    2. *Names* is employed for selecting the variable.
  - Value: It is the *rvalue* of assignment instruction, and it has three tabs:
    1. Variable: It shows a lists of variables. The type of these variables is the same that the one selected in the Variable section.
    2. Functions: This tab shows a set of functions whose return type is the same that the one selected in the Variable section.
    3. Constants: This tab allows to specify a literal.

    It is important to notice that if a function is selected in the tab *Function*, the parameter selection process (previously explained) is repeated for each parameter. Otherwise, the process is ended.

Create NanoBots Types is used for creating several roles. A role describes the NanoBot behavior, the number of roles is user-dependent. When a role is created two main activities must be specified:

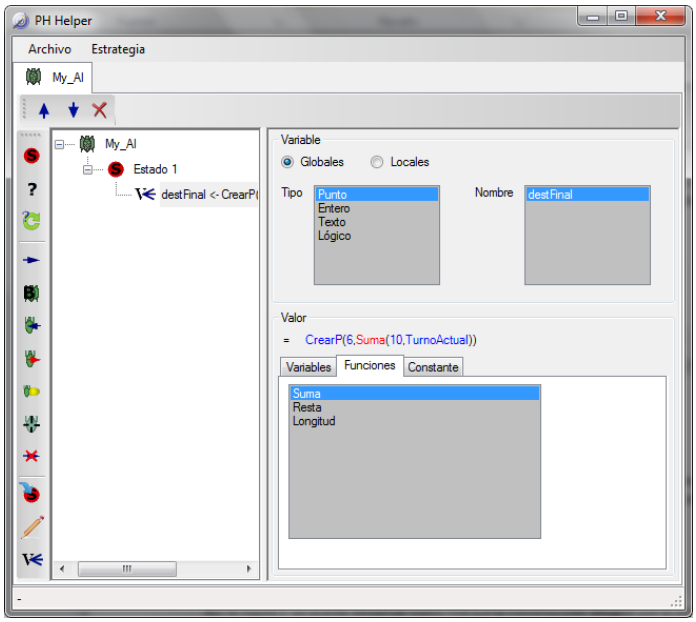

Figure 2 PH-Helper Parametrization Window – Variable Assignment Operation.

- The specification of NanoBot Type: In this activity, the user selects the NanoBot type[12] which can not be changed during the game. The NanoBot type is used for two important goals. The first one sets the available commands, and the second one limits the characteristic's maximum values.
- The specification of NanoBot Characteristics: In this activity, the user gives a value to each NanoBot characteristics[13], which can not be changed during the game.

The activities previously mentioned are easily carried out with PH-Helper. When a NanoBot is created its type is directly established.

The user only needs to select the option *Strategy* and then the option *Add*.

In this moment, a NanoBot type list is shown.

The last task consists in selecting one NanoBot type from the list.

These operations can be observed on the top of figure 3.

After doing these tasks, a new tab is created for the NanoBot (it is a sub-space as described at the beginning of this section). In the Program Window a root node is inserted. This root represents the action for specifying the NanoBot characteristics. The user can establish the characteristic values by following two steps. The first one consists in selecting the root action. The last one consists in changing the characteristic values displayed in the Parametrization Window. The process is illustrated in figure 3, at bottom.

Define Strategies is used to build the action tree. This task is simple to do, the user just needs to add the wished actions by clicking on the *Command Language Bar*. At any moment the user can save the project. This task is achieved by applying the following steps:

1. For each root node r do

[12] Remeber that the available types were described in section 2.
[13] Remember that the characteristics were described in section 2.

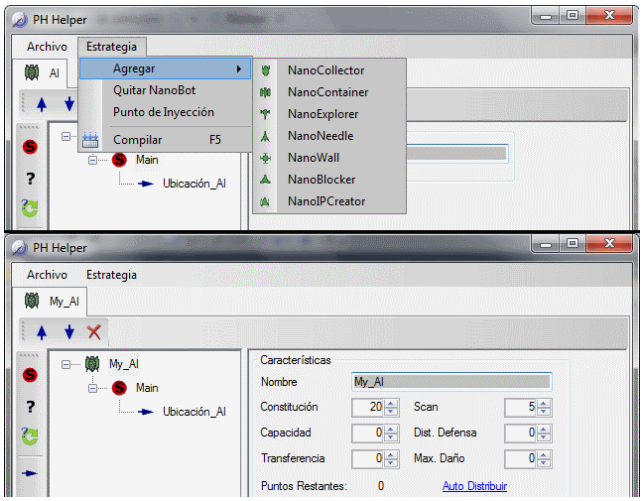

Figure 3 Types and Characteristics available in PH-Helper.

a. For each action A in r do

i. Create a XML element for A. This element contains the A parametrization.

ii. Append the XML code generated in previous step in the XML document.

2. Save the XML document into a file.

The flow chart of this procedure is shown in figure 4.

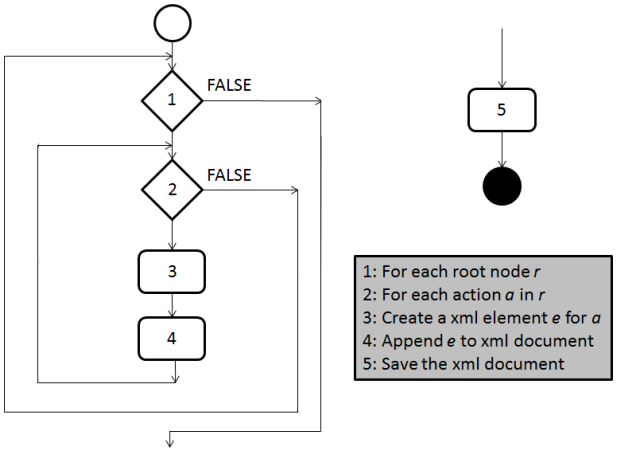

Figure 4 Save Procedure provided by PH-Helper.

The DTD that defines the XML dialect used by PH-Helper to save the edited code is shown in listing 4.

Basically, this DTD allows to define a strategy ia an action set. The actions are specified taking into consideration their parameters. Each action can have sub-actions. In this way a tree structure is built.

Listing 4 XML dialect used to save edited code.

```
<!ELEMENT STRATEGY (ACTION*)>
<!ELEMENT ACTION (VARIABLES, ACTION*)>
<!ELEMENT VARIABLES (VARIABLE*)>
<!ELEMENT VARIABLE EMPTY>
<!ATTLIST ACTION name CDATA
                 tag CDATA>
<!ATTLIST VARIABLE name CDATA
                   type CDATA
                   keyType CDATA
                   valueType CDATA
                   isGlobal CDATA
                   isSystem CDATA
                   isReadOnly CDATA>
```

5 Compiling Hoshimi Programs with PH-Helper

The process for compiling Hoshimi programs is based on the following tasks (a complete view is shown in figure 5):

1. Error Verifying: PH-Helper provides syntax-directed assistance for avoiding errors. However, some semantic errors can even occur. For example, the variable type used in an assignment can be changed. If this happens, a type error arise. Other example is presented when a state referenced in change state operation is deleted. The error sub-system works as follows:
   a. For each action A do
      i. Apply the method *CheckErrors*. In this method, A verifies its parametrization. If it is not correct then an error collection is returned. In other case, an empty collection is returned.
      ii. For each error E recovered in previous step do
         A. Print a message error on *Error Console*.
      iii. If an error is detected, then finish the process.
2. Source Code Storage: This phase is carried out applying the algorithm described below.
   a. For the root action R do
      i. Apply the method *GetCode*. This method is recursively invoked in R sub-actions with the goal of gathering the source code. Each action is responsible of building its proper source code.
      ii. When the strategy source code is collected, it is stored into a file.

The steps described above are applied for each root node in the program. When all the source code components are recovered, they are compiled and the corresponding IL (.Net Intermediate Language) is generated. It is important to remark that some library files are always included in the compilation process, as is the case of:

- MyUtils: this file contains common functions like *DistanceCalculation*, *GetNearestPoint*, *Barycenter*, etc.
- AISystem: this file contains the scheduler and other administration structures.

Furthermore, a project file is also generated. This task is achieved to visualize all files in Visual Studio .Net.

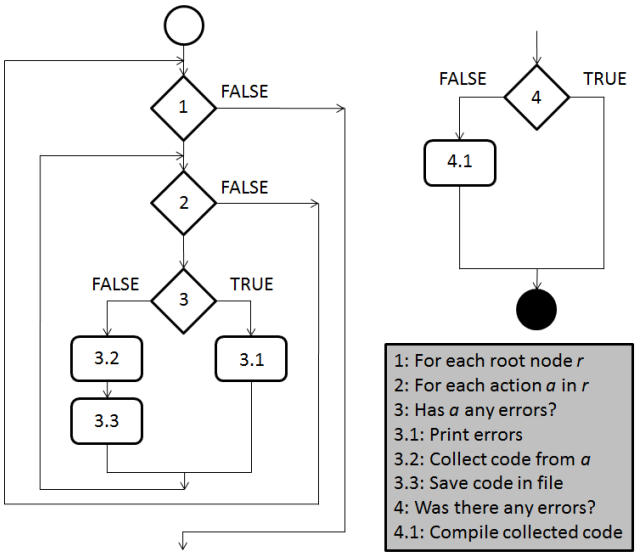

Figure 5 PH-Helper Compilation Workflow.

6 Case Study: The AZN Way

As it was explained in section 2, one of the game objectives is to cure diseases. In order to do this, a NanoNeedle (hereinafter called *needle*) must be created in the diseased zone (Hoshimi Points – HPo). Once created, the needle must be filled with the only resource available in the game: the AZN. Each needle can contain up to 100 units of AZN. It is important to mention that it is convenient to fill the needles until the top, because they provide a score proportional to the AZN stock. Needles can not move or gather AZN by themselves. Because of this, the collaboration between a NanoAI (hereinafter denoted as AI) and a nanobot with gathering capabilities is needed. NanoCollectors (hereinafter called *collectors*) and NanoContainers (also known as *containers*) are two kinds of nanobots that meet this property.

The AIs job is to go to the desired destination (HPo) and build there a needle [14] [15]. Furthermore, the AI must build collectors or containers too. These tasks can be carried out in any order, however it is encouraged to build the gatherers and then the needles. In this way, the firsts can go and gather AZN, while the AI moves to the HPo. Any number of collectors can be built. Nevertheless, it is best to choose a multiple of the amount needed to fill a needle, considering its capacity. If collectors are chosen, is wise to build 5 or 10 of them, because they have a capacity of 20 units of AZN. If containers are chosen instead, is advisable to build a pair number of them (generally between 4 and 10) because they can transport 50 units of AZN.

[14] On the one hand, nanobots with movement capabilities appear in the team injection point when built. On the other hand, nanobots without movement capabilities emerge in the current location of the NanoAI.

[15] Several nanobots with movement capabilities can be at the same point, but just one of them can be active. Because of that, only one needle can be built in a HPo.

6.1 The Problem Statement

In this section, a simple problem is given as exercise with the goal of illustrating the operation of the tool described in this article.

The problem statement is: *Elaborate a strategy for creating and filling 3 needles in the following points: <147, 22>, <159, 26> and <170, 38>*. Tip: It should be noticed that the nearest AZN is at point <154, 54>.

6.2 The Solution

To solve the problem, two nanobots must be added to the AI. One of them must be of NanoNeedle type, and the other one must possess gathering capabilities. Containers will be used in this solution, but the user can choose another type like collectors. The default values of the characteristics are recommended for each nanobot, nevertheless the student can experiment with other values. If these steps were carried out successfully, three tabs will be displayed on the screen. Each one of them is labeled with the name of the represented nanobot. Figure 6 shows the screen previously described.

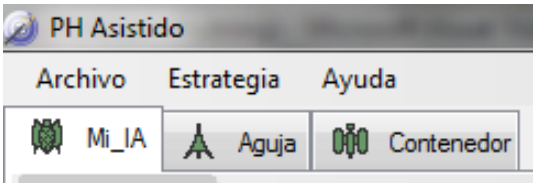

Figure 6 PH-Helper Tabs corresponding to the 3 necessary nanobots created.

In this case, the tabs are labeled Mi_IA, Aguja and Contenedor which are the Spanish words for "My AI", "Needle" and "Container".

Two steps must be specified to define an AI strategy for building a needle at the desired HPo.

First, to save time, it is convenient to build the containers. Generally, the best approach for solving problems is to separate each task into states. In this case, the first state consists in creating the containers. This task is carried out using a conditional statement and a counter. If the counter is less than the number of desired nanobots then a new nanobot will be created; otherwise, the process will finish its execution. It is important to remark that: i) Before adding the actions, the counter must be declared, and ii) The conditional action is executed several times by the Hoshimi Project engine. The complete state and the parametrization of the main action are shown in figure 7.

The next step is to specify the task of moving the AI to the desired location (HPo) and build there a needle. The procedure consists in asking if the current AI location equals the HPo location. If that is the case, the needle is created, and then the AI changes to the next state; otherwise, the AI keeps moving towards the HPo location. The state for building a needle and the predicate parametrization is shown in figure 8.

Two similar states must be specified for the remaining points, with the corresponding HPo locations. The AI responsibilities are performed by all the tasks previously specified.

The needles strategies are very simple because they can not move. The needle can defend by itself but the corresponding strategy will not be implemented because it was not included in the problem statement.

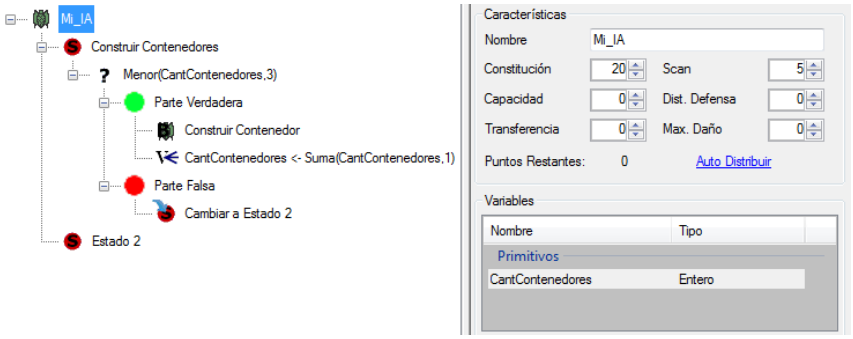

Figure 7 PH-Helper Building a Container.

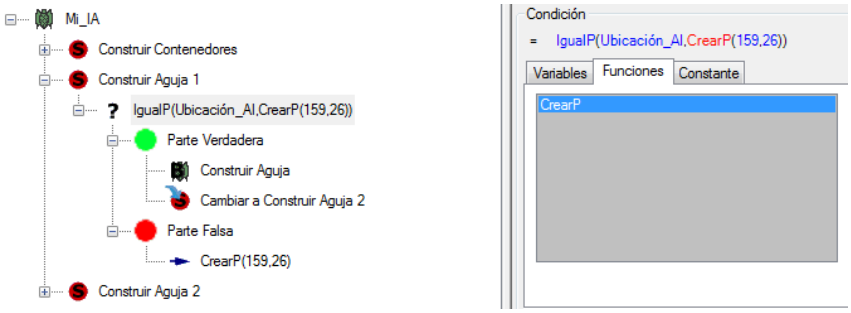

Figure 8 PH-Helper Predicate Parametrization.

The containers strategy has the following tasks:

1. Go to the AZN point.
2. Gather AZN.
3. Go to a HPo with needle.
4. Transfer the AZN.

Clearly, this is translated into four states. The first one simply compares the nanobot location with AZN location. If they are the same, then it changes to the next state; otherwise, the containers keep moving towards the AZN location. The second state commands the container to gather AZN for 10 turns at a rate of 5 units loaded by turn, this action will fill the stock of the container. Then it changes to the next state. The third state is similar to the first, but it uses the HPo location instead of the AZN location. The final state is similar to the second, but it transfers the AZN to the needle.

With all strategies ready, only remains to compile. This process generates an assembly (dll file) that will be used by the Project Hoshimi. An equivalent C# source code is also generated. Part of the source code corresponding to the AI strategy is shown in figure 9, and in figure 10 a snapshot of the strategy execution is displayed.

7 Conclusion

In this paper, a tool aimed at helping to teach programming was presented. This tool is intended to keep students away from those harmful issues that distract them from learning programming concepts (algorithms/strategies and data structures), that is the actual focus

```
switch (___ESTADO)
{
    case "Construir Contenedores":
        if (Menor(CantContenedores, 3))
        {
            AI.Build(typeof(Contenedor));
            CantContenedores = Suma(CantContenedores, 1);
        }
        else
        {
            ___ESTADO = "Construir Aguja 1";
        }
        break;
    case "Construir Aguja 1":
        if (IgualP(Ubicación_AI, CrearP(159, 26)))
        {
            AI.Build(typeof(Aguja));
            ___ESTADO = "Construir Aguja 2";
        }
        else
        {
            MoveTo(CrearP(159, 26));
        }
        break;
```

Figure 9 PH-Helper generated C# Source Code.

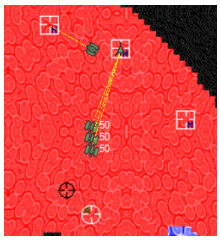

Figure 10 PH-Helper Execution Snapshot.

when solving exercises using the computer. The main difficulties that the tool should keep clear arise from the following facts: i) Programming Languages use idioms different from Programmers' native language, and ii) the syntax of Programming Language is full of tricky details that disturb the quiet writing of programs.

In order to overcome these problems, PH-Helper allows the use of a simple visual language and then generates C# source code. On one hand, the visual language is composed of intuitive icons that can be easily related among them. By intuitive we mean icons that any student, aware of the problem domain, can immediately understand. On the other hand, PH-Helper offers an editing environment that is directed (or guided) by HL syntax, avoiding in this way annoying syntactic errors.

PH-Helper features described above allow the student to be concentrated in solving problems, understanding programming essentials, instead of being bother with technical details.

In order to facilitate programming, HL, the Hoshimi Language, was extended: i) Some statements were enhanced, such as, *loop* and *decision* statements; ii) *Variable definitions* and *Assignment* statement were added; iii) New *functions* and *properties* can also be defined by the programmer.

For each strategy PH-Helper generates C# code that can then be executed in the appropriate .Net environment to test the ideas implemented. Moreover, notice that the generated code is saved into a file and so it can be used to teach object-oriented programming. In this sense, our project also contributes for teaching this important topic, not as easy as we

could forecast in the beginning [9]. Old code generators, mainly those depending directly on the text being freely edit by the programmer (like HTML generators, etc,) produced awful, unreadable code. However we know that recent code generators for formal languages are delivering nice code, plenty of comments, that can actually be used for students to learn with its inspection; we did many experiments in that direction when using compiler generators.

To sum up, we can say that the work here reported was concerned with three challenging and complementary research directions: programming languages design and programming languages implementation towards the improvement of programming courses.

As future work, the research team has scheduled the following tasks: i) To deploy as soon as possible a first PH-Helper stable release to start the practical experiments with real users in real class environments to assess the user-friendliness of the tool and its effective pedagogical aid; ii) To apply the strategies used in PH-Helper development to build similar tools for other domain specific languages. For instance, at UNSL (National University of San Luis), in the first programming course an algorithmic language, similar to natural language, is used for teaching. With this approach, the student can not execute programs. Therefore, the student can not verify his solutions. For the reasons previously mentioned, we plan to develop a visual language with an editor similar to the one described in this paper.

Another interesting topic, for future research, is to plan a set of experiments that can let us to measure and understand how much that approach, of starting programming with DSL s, can effectively help in learning to program properly with GPL s.

References

1 Francisco P. Andrés, Juan de Lara, and Esther Guerra. Domain Specific Languages with Graphical and Textual Views. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *AGTIVE*, volume 5088 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2007.

2 F. Arefi, C.E. Hughes, and D.A. Workman. The object-oriented design of a visual syntax-directed editor generator. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 389 –396, sep 1989.

3 Farah Arefi, Charles E. Hughes, and David A. Workman. Automatically generating visual syntax-directed editors. *Commun. ACM*, 33:349–360, March 1990.

4 W. F. Clocksin and Chris Mellish. *Programming in Prolog*. Springer, 1981.

5 M.J. Conway. *Alice: easy-to-learn 3D scripting for novices*. University of Virginia, 1998.

6 Daniela da Cruz, Ruben Fonseca, Maria Joao Varanda Pereira, Mario Beron, and Pedro Rangel Henriques. Comparing generators for language-based tools. In *CoRTA-07 - Compiler Related Techonlogies and Applications, Covilhã, Portugual*, July 2007.

7 Saumya K. Debray. *The SB-Prolog System, version 3.1: A User Manual*. Dep. of Computer Science / Univ. of Arizona, 1.st edition, Dec. 1989.

8 P. Deransart and G. Ferrand. Initiation a prolog: Concepts de base. Support de Cours 86-2, Université d'Orleans, Dep. de Mathématiques et Informatique, Jun. 1986.

9 Stavroula Georgantaki and Symeon Retalis. Using educational tools for teaching object oriented design and programming. *Journal of Information Technology Impact (Jiti)*, 7(2):111–130, 2007.

10 Javier Gonzalez Sanchez, Ramiro A. Berrelleza Perez, and Maria Elena Chavez Echeagaray. Introducing computer science with project hoshimi. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 908–914, New York, NY, USA, 2007. ACM.

11 Pedro Henriques, Maria Joao Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using lisa system. *IEE Software Journal*, 152(2):54–70, April 2005.

12 Christopher D. Hundhausen, Sean F. Farley, and Jonathan L. Brown. Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Trans. Comput.-Hum. Interact.*, 16(3):13:1–13:40, Sep. 2009.

13 A. Hutchinson, B. Moskal, W. Dann, and S. Cooper. The alice curriculum and its impact on women in programming courses. In *Annual Meeting of the American Society for Engineering Education (ASEE06)*, 2006.

14 A. Hutchinson, B. Moskal, W. Dann, S. Cooper, and W. Navidi. The alice curricular approach: A community college intervention in introductory programming courses. In *Innovations 2008, International Network for Engineering Education Research*, pages 157–176, 2008.

15 Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers text editing. In *CHI '05: HUMAN FACTORS IN COMPUTING*, pages 1557–1560. Press, 2005.

16 Andrew Jensen Ko. Designing a flexible and supportive direct-manipulation programming environment. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 277–278, Washington, DC, USA, 2004. IEEE Computer Society.

17 J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):1–15, 2010.

18 Raul Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1982.

19 MI-students, Daniela da Cruz, and Pedro Rangel Henriques. Agile - a structured-editor, analyzer, metric-evaluator and transformer for attribute grammars. In Luis S. Barbosa and Miguel P. Correia, editors, *INForum'10 — Simposio de Informatica (CoRTA'10 track)*, pages 197–200, Braga, Portugal, September 2010. Universidade do Minho.

20 Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 75–79, New York, NY, USA, 2004. ACM.

21 Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog*. John Wiley & Sons, 1st edition, 1990.

22 K.A. Olsen, P. Harnes, B. Pedersen, and O.-J. Tosse. The dsp system-a visual system to support teaching of programming. In *Visual Languages, 1988., IEEE Workshop on*, pages 199 –206, oct 1988.

23 Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980.

24 John Peterson. A Language for Mathematical Visualization. In *Proceedings of FPDE'02: Functional and Declarative Languages in Education*, 2002.

25 M. Resnick, Y. Kafai, and J. Maeda. A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities. MIT Media Laboratory, Proposal to National Science Foundation (Information Technology Research), 2003.

26 P. Roussel. *Prolog: Manual de reference et d'utilisation*. Groupe d'Intelligence Artificielle, Marseille–Luminy, 1.st edition, Sep. 1975.

27 Leon Sterling and Ehud Shapiro. *The Art of Prolog*, chapter 16. Series in logic programming. MIT Press, 1986.

28 Tim Teitelbaum. The cornell program synthesizer: a syntax-directed programming environment. *SIGPLAN Not.*, 14(10):75–75, Oct. 1979.

29 Ian Utting, Stephen Cooper, Michael Kölling, John Maloney, and Mitchel Resnick. Alice, greenfoot, and scratch – a discussion. *ACM Transactions on Compututer Education*, 10(4):17:1–17:11, Nov. 2010.

30 Maria Joao Varanda and Pedro Rangel Henriques. Visualization / animation of programs based on abstract representations and formal mappings. In *HCC'01 - 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*. IEEE, September 2001.

31 Daniel Watt. *Learning With Logo*. McGraw Hill, 1983.

32 Molly Watt and Daniel Watt. *Teaching With Logo: Building Blocks For Learning*. Addison-Wesley Pub, 1986.