ESPranto: a Framework for Developing Applications for Adaptive Hardware

Robert van Herk & Leszek Holenderski
Human Interaction & Experiences
Philips Research
Eindhoven, the Netherlands
Email: robert.van.herk@philips.com, leszek.holenderski@philips.com

***Abstract*—Evermore highly adaptive hardware toolkits become available, of which the applications are configured, fine-tuned or even created altogether by their end-users. To support these users, we need to think about flexible, yet accessible frameworks with which these end-users can adapt applications to their personal needs. We present a programming framework, called ESPranto, which strives exactly to do so. ESPranto is basically Esterel extended with funcros. Esterel is a reactive programming language suited for programming control-dominated applications. Commands can be executed in parallel and parallel commands can communicate. Contrary to most language, the Esterel compiler automatically proves that programs will never cause run time problems such as deadlocks, race conditions or crashes. Because Esterel does not have very strong abstraction mechanisms we added funcros. Funcros are functional macros. Like macros, they are statically expanded when applied to actual arguments. Unlike macros, funcros are type safe and hygienic (local declarations are renamed during expansion, to guarantee that there are no collisions with existing identifiers). Funcros can be recursive and higher-order in the sense that they take funcros as parameters and return funcros as results. One can also apply them partially, by providing less actual parameters than formal parameters. In addition, we added a polymorphic type system to Esterel, to allow funcros to be polymorphic and hence make ESPranto more expressive, while maintaining type safety. These extensions allowed us to use ESPranto as a host language for embedding several domain specific languages. The languages were specifically designed for developing reactive applications for a storytelling environment called StoryToy and for a tangible interaction tablet called TagTiles. The macro-like properties allowed us to keep the useful features of Esterel described above. The functional properties allowed us to use ESPranto to facilitate end-user programming: our end-users use ESPranto to adapt and extend applications to their own needs.**

***Keywords*-adaptive systems; computer languages; parallel programming; functional programming.**

I. INTRODUCTION

Recently the world has seen a huge rise in commercially available, highly adaptive hardware toolkits, such as the iCat [1], Arduino [2], Phidgets [3] and Lego MindStorms [4]. Whereas in classical computing a rather fixed number of applications suffices for most users, for these toolkits the applications are configured, adapted or even created altogether by end-users. These application creators may be non-technical domain experts – for instance experts in human robot interaction that use these products for their research [1] – or laymen, such as technical hobbyists, children, teachers or parents [2].

Figure 1. StoryToy

Enabling all these users to easily adapt such hardware to their own needs, whist giving them access to all functionality, is a daunting task, but some attempts are being made [1]–[3]. However, we still face the issue that adaptation frameworks that are powerful and truly accessible to non-technical users hardly exist for such hardware toolkits.

This paper describes the development and usage of the programming language ESPranto. We apply ESPranto especially to embed domain specific languages (DSLs) used by end-users to fine-tune or create applications for adaptive hardware toolkits. We called the language ESPranto because it is a language for "Edutainment Sensor Platforms (ESPs)" and because it strives to be applicable to a diverse class of users.

One toolkit for which the language is used is StoryToy [5] (Fig. 1). A second system is TagTiles [6] (Fig. 2), marketed by Serious Toys [7].

StoryToy is a toolkit for creating storytelling applications. These applications can entertain and promote the development of young children (2-6). The child implicitly provides input to the system by shaking fluffy farm animals, thus

 ISBN: 978-1-61208-109-0

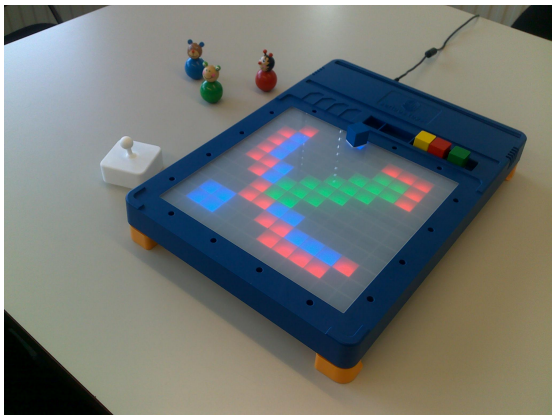

Figure 2. TagTiles

guiding the plot of the story. Lighting support is available for drama. For example, one created application was set in a thunderstorm, featured lightning effects. We have found that laymen users, such as parents or teachers, enjoy adapting StoryToy to specific children by making personal applications (stories) for them [8].

TagTiles is a tangible interface console designed for educational board games (Fig. 2). It features an antenna grid that identifies and localizes objects tagged with RFID tags on the two-dimensional surface. For providing feedback, every grid cell is equipped with a full color LED and the board can produce audio like speech, sound effects or music. Also, the console is equipped with an embedded computer. Non-technical domain experts, e.g., psychologists, are using TagTiles in their research [6], [9]. Also TagTiles applications can be adapted by teachers for usage in their classrooms [8].

Before creating the ESPranto language, we conducted a series of studies in which domain experts and laymen were asked to design a range of applications for adaptive hardware such as StoryToy and TagTiles [8]. We found that participants tend to describe these applications as 'story lines' that develop in parallel. Each story line is described sequentially and in some cases must be preempted (i.e., aborted). Furthermore, the application domains the different users came up with were very diverse [6], [9], [10]. Domain experts and laymen indicated that they want to spend as little time as possible addressing technical issues. From these sessions, we concluded that ESPranto needed the following features:

1) To keep the language close to the users' mental model, parallelism, sequencing and preemption must be supported.
2) ESPranto must enable embedding of a multitude of domains.
3) Errors must be given in terms of the end-user's domain.
4) Run time problems such as crashes, non deterministic behavior, or deadlocks must be prevented.

Implementing ESPranto as an Esterel-like language seemed an obvious choice: Esterel supports parallelism and preemption [11], meeting requirement 1. Also, Esterel applications will never cause run time errors, meeting requirement 4. However, we argue that Esterel *per se* is not suitable for implementing DSLs because it lacks crucial abstraction mechanisms, needed for requirement 2 and 3. Many DSLs have been successfully implemented in modern functional programming languages that do provide those crucial abstraction mechanisms.

To meet all requirements, we designed ESPranto as a subset of Esterel, extended with the required features from functional programming. ESPranto is based on the concept of 'funcros'. Funcros are like functions in functional programming and at the same time like macros. Because of the power of funcros, ESPranto maintains the core benefits of Esterel but introduces the benefits of functional programming, thus making it suitable for embedding DSLs for tangible applications. We have embedded a series of DSLs in ESPranto and these DSLs are successfully used by domain experts to create tangible applications.

The remainder of this paper is structured as follows. In Section II, we go deeper into DSLs and explain how they are embedded into functional programming languages. Section III gives a basic introduction into reactive programming in Esterel. Section IV describes previous work on combining reactive programming with functional programming. In Section V, we explain how ESPranto can be translated into Esterel, yet unleashes the power of functional programming through funcros. In Section VI, we provide examples of DSLs that are embedded in ESPranto and show how ESPranto's features are applied by non-technical end-users to create and adapt applications. We will use examples of TagTiles applications, because they are most advanced and are indicative of ESPranto's strength. In Section VII, we present our conclusions.

II. Domain Specific Languages

Mainstream programming languages (like C and Java) do not enable domain experts who are not professional programmers to adapt applications. Therefore, DSLs have been designed, which enable domain experts to develop applications for their particular domain quickly and effectively, yielding programs that are easy to understand, reason about, and maintain by the domain experts themselves [12].

Designing a programming language from scratch is difficult and time-consuming. Effort is wasted, since large parts of many DSLs are not domain specific at all. For instance, many require a type system and a sub-language for integer and Boolean expressions. For these reasons, language designers often *embed* a DSL as a library into an existing *host* language. Because a Domain Specific Embedded Language (DSEL) inherits features from its host language, the chosen

 ISBN: 978-1-61208-109-0

host language must be flexible enough to allow for a natural embedding of the DSL, and its features must be suitable for, or at least not at odds with, the properties of the domain.

Modern functional programming languages are often a sensible choice when crafting DSELs for creating desktop computing applications; examples are abundant [12]. Functional languages provide good abstraction mechanisms due to polymorphic type systems and, as an alternative to statements, the flexible concept of functions. For example, polymorphic functions allow to define generic statements. Higher order functions allow to pass statements as parameters, allowing for yet stronger abstractions. Partial function application allows to use specialization, by defining a function in terms of another function with only some of its parameters provided. For instance, a function `inc` can be defined as `(+) 1`.

III. ESTEREL

Although it is generally easy to embed DSLs in functional programming languages, functional programming *per se* is not suitable for embedding DSLs for *tangible* computing: functional programs cannot be checked for causal correctness, are not real-time and not optimal for resource constrained systems. Reactive programming languages provide these features, but lack flexibility. Before describing how we combined both paradigms, we will first give a basic introduction into Esterel by discussing some examples. For a full explanation of Esterel we refer to [11].

Esterel is based on the two basic concepts of *commands* and *signals*. Commands describe control-flow and signals are used for communication with hardware or between commands.

An external control loop is in place that decides when the Esterel program is allowed to execute; usually when the underlying hardware has detected some physical event or when a predetermined amount of time has passed. Then, the Esterel program computes a reaction, typically in a very short amount of time, and pauses again. One such an iteration is called an *instant*. When the control loop decides to run the next instant, the Esterel program will resume execution at the point at which it stopped in the previous instant.

In every instant, for every signal it holds that that signal is either present or absent. With this feature Esterel provides *causal correctness*, which means that race conditions cannot occur because all communication within an instant appears to be instantaneous.

Esterel is crafted such that an instant always runs in a finite amount of time. Because of this feature, Esterel is a *real-time* programming language. This requires that all commands that *wait* for some event to happen end the current instant and pass control back to the control loop.

After giving a few introductory examples, we will briefly discuss the implications of causal correctness and real-timeness.

```
//Declaration of signals
input ButtonPress
output SoundBell

//The control code itself:
await ButtonPress;
emit SoundBell
```

Figure 3. Esterel program controlling a doorbell

A. Controlling a simple machine

A trivial example of an Esterel program is given in Figure 3. This program controls a simple doorbell. The program will make the bell sound once when the button is pressed and then terminate.

The program has two signals. `ButtonPress` is an *input* signal that describes the state of a sensor: when the button is pressed the control loop sets the signal to present and then makes the Esterel program run for one instant. `SoundBell` is an *output* signal: after the Esterel program has run for one instant, the control loop decides to sound the bell or not depending on the presence of this signal.

When the external control loop makes this program react for the first time, it will immediately end the instant because control reaches the `await` command. Hence, in the first instant the output signal `SoundBell` is absent, meaning that the external control loop should not sound the bell.

When the control loop makes the program react for the second time, it will resume at the point where it stopped in the first instant; i.e., at the `await` command. Two things can happen, depending on the presence of the input signal `ButtonPress`. If `ButtonPress` is absent, the program will immediately end the instant again, as it is waiting for an instant in which `ButtonPress` is present. When an instant is run in which `ButtonPress` is present, the `await` command terminates and the program continues and emits `SoundBell`. Since that is the last command of the program, it will then terminate. The control loop reacts to the presence of `SoundBell` by ringing the bell.

B. Parallelism and communication

The example given in Figure 4 controls a more advanced doorbell and uses parallelism and communication. In this example, the doorbell has two buttons and has two different melodies. The program turn-wise plays one of these melodies when button 1 and/or button 2 is pressed, but it will never interrupt a playing melody.

We have introduced an input signal `MelodyDone` that the control loop will set to be present in every instant in which a melody has just stopped playing. The program has three loops. The three loops will run in parallel because

 ISBN: 978-1-61208-109-0

```
input Button1Press
input Button2Press
input MelodyDone
output PlayMelody1
output PlayMelody2

signal GenerateOuput in
  loop
    await Button1Press;
    emit GenerateOutput
  end loop
  |
  loop
    await Button2Press;
    emit GenerateOutput
  end loop
  |
  loop
    await GenerateOutput;
    emit PlayMelody1;
    await MelodyDone;
    await GenerateOutput;
    emit PlayMelody2;
    await MelodyDone
  end loop
end signal
```

Figure 4. A more advanced doorbell

they are combined with the parallel operator `|`. The loops communicate through the *local* signal `GenerateOutput`.

The upper loop waits for an instant in which `Button1Press` is present. In such an instant, it will emit `GenerateOutput` and then restart the loop again. Restarting the loop will cause the upper loop to end for this instant, because of the `await` command. The second loop reacts likewise on the presence of `Button2Press`.

The third loop waits for an instant in which the local signal `GenerateOutput` is present. In the first instant in which that happens, it will react by emitting `PlayMelody1`. The program will then wait until the melody has stopped playing. This prevents the program from ever starting a new melody when a melody is already playing. Only after the melody has stopped, it will wait for an instant in which `GenerateOutput` is present again and then react by emitting `PlayMelody2`. When that melody has finished, the third loop will be restarted and wait again for `GenerateOutput`.

C. Implications of causal correctness and real-timeness

In Esterel writing such control-dominated programs is easy: separate parts of the program can be defined separately and composed in parallel, parallel commands can communicate through signals but race conditions will not occur, no variables are needed to keep track of the state of the program and the program will never deadlock because all instants will always end after a finite amount of time.

In imperative languages such as C, or functional languages such as Haskell, writing a similar program correctly is more difficult.

One approach would be to use asynchronous multi-threading and shared memory. This would require the programmer to *synchronize* when accessing the shared memory to prevent race-conditions, and to prove that deadlocks will not occur due to this synchronization.

Another approach would be to use event-handling procedures that are called when the buttons are pressed or when a melody has finished playing. In that case, one would need to introduce variables to keep track of which melody is currently playing, if any. The programmer would need to think about all possible interleavings of events by himself, which can be difficult. Furthermore, the programmer would have to construct a proof that the event-handling procedures always terminate such that new events can continue to be accepted from the hardware.

```
signal S in
  if S
    then nothing
    else emit S
  end if
end signal
```

Figure 5. Causally incorrect Esterel program

D. Incorrect programs

The Esterel compiler rejects programs of which causal correctness cannot be proved. An example of such an erroneous program is given in Figure 5. The program is erroneous, because we cannot decide on the presence of `S`: if `S` is absent in the instant in which this program is started, `S` has to be emitted which means that it is present in that instant. Conversely, when `S` is present in the instant in which this program is started, it is never emitted, which means that it is absent in that instant.

Another example of an incorrect Esterel program is given in Figure 6. This program is causally correct, but still erroneous because the body of the loop does not contain any command that ends the instant. This means that we cannot establish a finite run time for the instant in which the loop starts, and therefore the Esterel compiler will reject this program. A correct version of this program pauses after emitting `S`. The command `pause` ends the current instant and terminates immediately in the next instant.

 ISBN: 978-1-61208-109-0

```
//Erroneous program
signal S in
  loop
    emit S
  end loop
end signal

-----------------------------------------

//Correct program
signal S in
  loop
    emit S;
    pause
  end loop
end signal
```

Figure 6. Esterel program with instantaneous loop

E. Limitations

Although Esterel provides very strong correctness properties, it provides few abstraction mechanisms. It is not possible to define functions or procedures. If such mechanisms were in place, the Esterel compiler could not automatically check for causal correctness, nor check that every instant has finite run times, because this is undecidable for Turing complete languages [13].

Some modularity can be achieved through a mechanism called *modules*, which are textually expanded. Modules can only be parametrized with signals and not with commands or other modules, nor can they be applied recursively or partially.

IV. Combining functional and reactive programming

As functional programming provides flexibility, and reactive programming provides causal correctness and real-timeness, ESPranto strives to combine best of both paradigms such that it can be used to embed DSLs for tangible computing.

A combination of functional programming with reactive programming was first proposed in Lucid Synchrone [14], a language that extends OCaml with some concepts borrowed from a data-flow synchronous language Lustre [15]. There are two main differences between Lucid Synchrone and ESPranto.

The first main difference is that our combination uses a different order in which the one programming paradigm extends the other. Unlike Lucid Synchrone, we extend a synchronous language with some concepts borrowed from functional languages. This difference has two important consequences. First, for our class of users, the reactive programming paradigm is much more natural than the functional one (requirement 1). Therefore ESPranto allows novices to start with reactive programming, and later on learn to fully use the additional functional features funcros provide for their convenience. This is of the utmost importance when dealing with application developers who are not professional programmers. Second, we can reduce substantially the runtime resources required to execute ESPranto programs, compared with Lucid Synchrone. We first compile ESPranto programs to Esterel, by expanding funcro applications, and then compile Esterel to assembly code. In the approach taken by Lucid Synchrone, programs are first expanded to OCaml, and then compiled by the OCaml compiler. This approach requires a much larger execution footprint, especially in terms of memory due to the need for garbage collection, which is inherent to functional programs.

```
/* ESPranto program */
main = (switch true)
switch c = (if c then pause else halt)

-----------------------------------------

/* Esterel translation */
if true then pause else halt
```

Figure 7. A trivial ESPranto program and its Esterel translation

The second main difference is that ESPranto incorporates an imperative synchronous language while Lucid Synchrone incorporates a declarative one. Again, this is easier to understand for our users, as sequential composition must be a basic combinator in our language (requirement 1). The data-flow paradigm, on which Lustre is based, is not well suited to describe imperative control flow.

V. ESPranto

A. Funcros

Every funcro is a tuple of a name, a series of zero or more formal parameters and a body. Funcros can be applied by providing them with actual parameters. The ESPranto compiler expands a funcro with all its actual arguments into basic Esterel statements. A trivial example of how this works is shown in Figure 7. Expansion always starts at the funcro called `main`. For the reader unfamiliar with Esterel, an overview of the most important basic statements is given in Appendix A.

When expanding a funcro application, formal parameters are substituted for their values. From this perspective, funcros are like macros as available in languages like C. ESPranto uses expansion (like macros) instead of dynamic application (like functions) to substantially simplify the causal correctness problem. If we used functions, but still wanted to keep Esterel's strong correctness properties, we would be faced with the problem of compositional verification of causal correctness. We are not aware of any method that solves that problem.

 ISBN: 978-1-61208-109-0

```
#define INC(i) {int b = 17; ++i;}
void weird () {
    int a = 0; int b = 0;
    INC(a); // a becomes 1.
    INC(b); // b remains 0.
}
```

Figure 8. Unhygienic macro expansion in C

Unlike C macros, funcros are type safe, hygienic, and ESPranto supports recursion, higher order funcros, and partial funcro application.

B. Type safety

The ESPranto compiler automatically infers the type of each funcro, using the Hindley-Milner type inference algorithm [16]. Before expanding a program into Esterel code, it checks if all funcro applications are type correct. If the program is type incorrect, the compiler will provide feedback on this to the user. Because types are checked *before* funcro expansion, the compiler gives feedback in terms of the domain specific funcros, and not in terms of the Esterel code. This is needed to meet requirement 3. Only if it is type correct, the compiler will produce the corresponding Esterel program that can be compiled into machine code in the next step. The `main` funcro must always have type `Command`, i.e., it must expand to a valid Esterel command.

For the code segment of Fig. 7, the compiler infers that `switch :: Condition → Command` (`::` should be read as *has type*, in other words `switch` takes a `Condition` as a parameter and then unfolds into a `Command`) because `c` is used as a condition in its body and because its body unfolds into an if-statement. Likewise the compiler infers that `main :: Command`, making the example program type correct.

C. Hygienic expansion

A known problem with 'naive' macro expansion systems is that identifiers may collide. Languages featuring such macro expansion systems are called 'unhygienic' [17]. An example is the C programming language (see Fig. 8). Programmers typically try to solve hygiene problems by *obfuscation*: use only unusual variable names in macros and hope that the same names will never be used in the rest of the program.

Obfuscation is not a suitable solution for ESPranto. It is a delicate and error prone method that requires the domain expert to inspect all used funcros. Instead, ESPranto has two features that together prevent hygiene problems. The first feature is scope: identifiers such as signals or traps are *scoped* within the funcro in which they are declared. They can be used outside their declaring funcros only by passing them explicitly as parameters to other funcros.

```
/* ESPranto code */
main = (
  trap T (
      myFuncro T //Explicitly pass T
  )
)
myFuncro TrapLabel = (
  trap T (
    exit TrapLabel
  )
)

---

/* Esterel translation */
trap T_0 in
  trap T_1 in
     exit T_0
  end trap
end trap
```

Figure 9. Example showing ESPranto's hygienic transformation

The second feature is the manner of transformation into Esterel code, in which Esterel identifier clashes are prevented by augmenting signal and trap labels with their *call depth*. The call depth is defined as the number of funcro applications in which the currently expanded funcro application is nested. The call depth of `main` is 0, and subsequently increases when calls are nested. Examples of how this works are given in Fig. 9. Note how the trap label in the exit statement is augmented with call depth 0 in the Esterel transformation, correctly referring to the outer trap.

D. Recursive funcro definitions

Most programming languages do not support recursively defined macros, because naively substituting macro applications with their expanded bodies would lead to an infinite amount of expanded code for recursive definitions. However, recursive functions is one of the essential features of functional programming that we wanted to include in ESPranto. We resolved this issue by allowing programmers to define alternative expansions of a funcro, guarded by *patterns*. For each funcro application, the compiler matches actual parameters against the patterns of the alternatives. For instance, the pattern i, where i is an identifier, matches against any parameter, the pattern `{}` matches only against an empty list, and the pattern $p_1 : p_2$, where p_1 and p_2 are patterns, matches only against a non-empty list, if the first element of that list matches against p_1 and the remainder of the list against p_2.

It is still possible to define recursive funcros that generate an infinite amount of code when they are expanded. An example is given in Fig. 10. We are currently investigating

 ISBN: 978-1-61208-109-0

```
/* ESPranto code */
length (h:t) =
  (1 + length (h:t)) // Wrong!
length {}    =
  (0)   // Non-recursive alternative

//This expansion will not terminate:
main = (
  if length {1,2,3} = 3
    then nothing
    else nothing
)
```

```
/* ESPranto code */
length (h:t) =
  (1 + length t) // Right
length {}    =
  (0)    // Non-recursive alternative

//This expansion will terminate:
main = (
  if length {1,2,3} = 3
    then nothing
    else nothing
)
```

Figure 10. Example showing recursive funcros

```
<funcrodef>  ::=  <identifier> <patterns>
                  = (<expr>)
  <pattern>  ::=  <identifier>
               |  <pattern> : <pattern>
               |  {}
               |  etc
     <expr>  ::=  <esterel>
               |  <identifier>
               |  <funcroapp>
               |  etc
  <esterel>  ::=  nothing
               |  pause
               |  true
               |  false
               |  <number>
               |  <expr> ; <expr>
               |  <expr> | <expr>
               |  <expr> && <expr>
               |  <expr> + <expr>
               |  etc
<funcroapp>  ::=  <identifier> <exprs>
```

Figure 11. Simplified syntax of ESPranto

if we should restrict the class of allowed recursive funcros. A possible restricted class can be those funcros of which the compiler can prove that all their possible applications will generate a finite amount of code. A first idea is to allow catamorphic [18] recursive funcros only, as applying such funcros always terminates. Catamorphic funcros break down a data structure in their recursion, and have a non-recursive alternative for $\perp$. With such a restriction, the compiler would indeed reject the first program in Fig. 10, and allow the second. Similar restrictions have been successfully applied before, for instance in the Agda programming language [19].

E. Syntax of ESPranto

For clarity, we present an excerpt of the grammar of ESPranto in Fig. 11. For clarity, the excerpt is heavily simplified. Important to note is that in ESPranto, an expression can be an Esterel literal, a funcro application, or an identifier bound through a pattern of a formal parameter. A complete description of the Esterel syntax can be found in [11].

VI. Embedding Domain Specific Languages in ESPranto

We have successfully embedded multiple DSLs in ESPranto, each for intrinsically different domains, including physio-therapy, [9] and games that use concepts from geometry [10].

To demonstrate how ESPranto's features facilitate end-users to adapt and extend applications, we will go deeper into one DSEL. This DSEL is created for a set of related exercises for the TagTiles console, that are, somewhat confusingly, called TagTiles Classic [6].

A. The domain of TagTiles Classic

The goal of TagTiles Classic was to create fun, educational exercises for children, which address cognitive and motor skills. In the vocabulary of the domain expert (psychologist), TagTiles Classic consists of a series of tasks and each task consists of a series of levels. All levels in TagTiles Classic start with displaying an *assignment pattern* by lighting up cells in particular colors on the TagTiles board. They then require the child to indicate a *solution pattern* by *tagging* cells with colored game pieces. The tasks differ in how the child has to work out the solution pattern from the assignment pattern. A particular set of tasks targets training spatial insight. For instance, in the *translation task* the child has to translate assignment patterns

 ISBN: 978-1-61208-109-0

```
//Global signals
signal LevelFinished
signal RestartLevel
level body = (
  abort SkipLevel (
    trap LevelFinished (
      loop (
        abort RestartLevel (
          body;
          exit LevelFinished
  ) ) ) )
  ;
  putBackThePieces
)
putBackThePieces = (
  ... /*some code*/
)
```

Figure 12. Example of parametrization in the TagTiles DSEL

```
drawTiles {} = (nothing)
drawTiles (coord:cs) = (
  drawTile coord;
  drawTiles cs //catamorphic recursion
)
awaitAndDrawTiles {} = (nothing)
awaitAndDrawTiles (coord:cs) = (
  ( awaitTile coord;
    drawTile coord
  )
  | awaitAndDrawTiles cs // catamorphic
)
memoryLevel coords = (
  level (
    drawTiles coords;
    waitTime 2000;
    clearBoard;
    emitSound "pling.wav";
    awaitAndDrawTiles coords
) )
```

Figure 13. Using recursive definitions for the TagTiles DSEL

4 cells horizontally. In the *mirror task*, the child has to create the symmetric counterpart of assignment patterns.

A programmer has embedded a DSL for these exercises into ESPranto and created a basic set of exemplary exercises. The psychologist then adapted this to her own needs, by fine-tuning the way the different tasks work, by adding extra tasks and by working out many levels for each task.

B. Capturing the essence of levels with parametrization

The domain expert wanted all levels to have some commonalities, specifically:

1) when playing a certain level, it must be possible to skip that level;
2) when playing a certain level, it must be possible to restart that level;
3) at the end of each level, the child must put the pieces back into a starting position.

The programmer captured these commonalities in the DSEL by introducing a funcro `level` (Fig. 12), which is parametrized with a command `body` that contains the control code for a specific level. The compiler infers that `level :: Command → Command`: if the domain expert applies this funcro and passes it a command, `level` will augment it with the common functionality of levels.

C. Using recursion in TagTiles

In the memory task every level first displays the assignment pattern for two seconds. Then the board is cleared and the child has to reconstruct the pattern from memory by tagging the correct cells. The child can reconstruct the pattern in any order. As he reconstructs the pattern, the cells light up again. Both the code for displaying the assignment pattern and for lighting up the solution pattern as the child reconstructs it can be neatly defined in ESPranto through *recursion* (see Fig. 13). Both funcros are defined recursively over a list. The funcro `drawTiles` takes a (static) list and creates a sequence of commands that each draw one cell. The funcro `awaitAndDrawTiles` takes a (static) list and waits for the child to put the right game piece on each coordinate and then illuminates that coordinate. Note how parallelism is used in `awaitAndDrawTiles` to allow the child to tag the cells in the solution pattern in any order.

D. Higher order funcros and partial application

The levels of the tasks addressing spatial insight share further commonalities. E.g., both in the mirror task and in the translation task, in every level the assignment pattern is displayed for two seconds, then the TagTiles board is cleared and the child has to recreate the solution pattern by applying the correct mathematical transformation; e.g., mirroring or translation. Figure 14 shows how we managed to capture these commonalities in the `spatialInsightLevel` funcro, that takes the mathematical transformation as its parameter `f`, and takes a list of coordinates `coords`. Hence,

```
spatialInsightLevel :: (Coord→ Coord) →
                       {Coord} →
                       Command
```

where `{Coord}` should be read as: list of which the elements have type `Coord`.

Since `spatialInsightLevel` takes a funcro as its first parameter, it is of *higher order*. Likewise, `map` is a higher order funcro. The funcros `mirrorLevel` and

```
/* Defined in a library */
map f {} = {}
map f (e:l) = {f e : map f l}

/* Defined for the DSEL of TagTiles */
spatialInsightLevel f coords = (
  level (
    drawTiles coords;
    waitTime 2000;
    emitSound "pling.wav";
    awaitAndDrawTiles (map f coords)
) )

mirror coord = (...)
  /* takes a coord and mirrors it
     point-symmetrically  w.r.t
     the center of the TagTiles board */
translate coord dx dy = (...)

/* Defined by the domain expert */
mirrorLevel = (
  spatialInsightLevel mirror
)
translationLevel = (
  spatialInsightLevel (translate 4 0)
)
mirrorLevel1 = (
  mirrorLevel coords1
)
coords1 = (...)
          //Some list of coordinates
```

Figure 14. Using higher order funcros and partial application for the TagTiles DSEL

`translationLevel` are partially applied funcros: they feed a first parameter to `spatialInsightLevel` and hence both have type `{Coord}` → `Command`. The funcro `mirrorLevel1` is exemplary for how the domain expert finally defines a level by feeding a list of actual coordinates to `mirrorLevel`, resulting in a command.

VII. Conclusions

We created ESPranto to alleviate the problems domain experts face when adapting and creating applications for adaptable hardware. ESPranto has proven to be a valuable framework to host DSLs for various adaptable hardware toolkits, for instance StoryToy and TagTiles, which is a tangible interface for educational board games. ESPranto enables the creation of a new domain specific language (DSL) when needed (for example, for a new type of game), with relatively low effort.

We have embedded several DSLs in ESPranto. Domain experts and laymen with no prior programming experience use these DSLs successfully to adapt and extend applications to their own needs.

We designed and implemented ESPranto as a programming language that combines reactive and functional programming by extending the reactive programming language Esterel with funcros and polymorphic types. Funcros are statically expanded similarly to macros. However, like functions, funcros are strongly typed, can be recursive, higher-order, and applied partially.

During compilation, an ESPranto program is expanded to an Esterel program which can then be checked for causal correctness using standard Esterel techniques. Using funcros (expanded during compile time) instead of functions (called at run time) allows us to verify causal correctness.

As such, ESPranto combines the prevention of runtime problems such as deadlocks or race-conditions that reactive programming offers, with the strong abstraction features that functional programming offers, making it very suitable for novice programming.

Appendix

Statement	Meaning
`nothing`	Terminates immediately
`pause`	Terminates in the next instant
`emit` S	Signal S is present in this instant. Terminates immediately.
`loop` p	Starts p. In the instant in which p terminates p is started again.
`halt`	Equal to `loop pause`
`await` c	Terminates in the next instant in which c holds.
`if` c `then` p `else` q	Starts p if c holds, q otherwise.
`trap` T p	Starts p. Terminates in the instant in which p terminates, or in the first instant in which p executes `exit` T.
`abort` c p	Starts executing p. Terminates in the instant in which p terminates, or in the next instant in which c holds.
p`;` q	Sequential composition. Starts executing p. In the instant in which p terminates, starts executing q. Terminates in the instant in which q terminates.
p `|` q	Parallel composition. Starts executing p and q. Terminates when both p and q have terminated.

Acknowledgments

The authors would like to thank Panos Markopoulos, Willem Fontijn, Janneke Verhaegh, Alexander Sinitsyn and Andrew Tokmakoff for their valuable input for this paper.

 ISBN: 978-1-61208-109-0

REFERENCES

[1] M. Saerbeck, "Software architecture for social robots," Ph.D. dissertation, Technical University of Eindhoven, 2010.

[2] E. Rosenbaum, E. Eastmond, and D. Mellis, "Empowering programmability for tangibles," in *TEI '10: Proceedings of the fourth international conference on Tangible, embedded, and embodied interaction*. New York, NY, USA: ACM, 2010, pp. 357–360.

[3] S. Greenberg and C. Fitchett, "Phidgets: easy development of physical interfaces through physical widgets," in *User Interface Software and Technology*, 2001, pp. 209–218.

[4] D. Benedettelli, *Creating Cool MINDSTORMS NXT Robots (Technology in Action)*. APress, 2008.

[5] W. Fontijn and P. Mendels, "Storytoy the interactive storytelling toy," in *PerGames workshop, International Conference on Pervasive Computing*, 2005.

[6] J. Verhaegh, W. Fontijn, and H. Hoonhout, "Tagtiles: optimal challenge in educational electronics," in *TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction*. New York, NY, USA: ACM, 2007, pp. 187–190.

[7] "Serious Toys," http://www.serioustoys.com (last access: August 2010).

[8] R. Van Herk, J. Verhaegh, and W. F. Fontijn, "ESPranto SDK: an adaptive programming environment for tangible applications," in *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*. New York, NY, USA: ACM, 2009, pp. 849–858.

[9] Y. Li, W. Fontijn, and P. Markopoulos, "A tangible tabletop game supporting therapy of children with cerebral palsy," in *Proceedings of the 2nd International Conference on Fun and Games*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 182–193.

[10] K. Hendrix, R. Van Herk, J. Verhaegh, and P. Markopoulos, "Increasing children's social competence through games, an exploratory study," in *IDC '09: Proceedings of the 8th International Conference on Interaction Design and Children*. New York, NY, USA: ACM, 2009, pp. 182–185.

[11] G. Berry, "The Esterel v5 Language Primer," http://www-sop.inria.fr/meije/esterel/esterel-eng.html (last access: August 2010), 1999.

[12] P. Hudak, "Building domain-specific embedded languages," *ACM COMPUTING SURVEYS*, vol. 28, 1996.

[13] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936.

[14] P. Caspi and M. Pouzet, "Lucid Synchrone, a functional extension of Lustre," Université Pierre et Marie Curie, Laboratoire LIP6, Tech. Rep., 2000.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," in *Proceedings of the IEEE*, vol. 79, Issue 9, 1991, pp. 1305–1320.

[16] B. Pierce, *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002, ch. 22.

[17] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*. New York, NY, USA: ACM, 1986, pp. 151–161.

[18] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*. New York, NY, USA: Springer-Verlag New York, Inc., 1991, pp. 124–144.

[19] "The Agda Wiki," http://wiki.portal.chalmers.se/agda (last access: August 2010).

 ISBN: 978-1-61208-109-0